

Copyright 1981 by Zilog, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Zilog.

The information contained herein is subject to change without notice. Zilog assumes no responsibility for the use of any circuitry other than circuitry embodied in a Zilog product. No other circuit patent licenses are implied.

Introduction

Zilog's name has become synonymous with logic innovation and advanced microprocessor architecture since the introduction of the Z80™ CPU in 1975. The Zilog Family of microprocessors and microcomputers has grown to include the products listed in the table below. Each product exhibits special features that make it stand above similar products in the semiconductor marketplace. These special features have proven to be of substantial aid in the solution of microprocessor design problems.

This reference book contains a collection of application information about Zilog microprocessor products. It includes technical articles, application notes, concept papers, and benchmarks. The reference book is intended as the first of several such volumes. We at Zilog believe that designing innovative microprocessor integrated circuit

products is only half the key that unlocks the future of microprocessor-based end products: the other half is the creative application of those products. Advanced microprocessor products and their creative application lead to end product designs with more features, more simply implemented, at a lower system cost. It is hoped this reference book will stimulate new product design ideas as well as fresh approaches to the design of traditional microprocessor-based products.

The material in this book is believed to be accurate and up-to-date. If you do find errors, or would like to offer suggestions for future application notes, we would appreciate hearing from you. Correction inputs should be directed to Components Division Technical Publications, and application suggestions should be directed to Components Division Application Engineering.

Z8600 FAMILY	8-bit Single Chip Microcomputer, 2K Bytes ROM and 144 Bytes RAM
Z8601	Mask Programmed
Z8602	Development Package
Z8603	Protopack
Z8671	Basic/Debug
Z8681	ROMless
Z8610 FAMILY	8-bit Single Chip Microcomputer, 4K Bytes ROM and 144 Bytes RAM
Z8610	Mask Programmed
Z8612	Development Package
Z8613	Protopack
Z80 FAMILY	8-bit General Purpose Microprocessor
Z8400 CPU	CPU
Z8410 DMA	Direct Memory Access
Z8420 PIO	Parallel I/O Controller
Z8430 CTC	Counter Timer Circuit
Z8440 SIO	Serial I/O Controller
Z8449 SIO/9	Serial I/O Controller
Z8470 DART	Dual Asynchronous Receiver/Transmitter

Z8000 FAMILY	16-bit General Purpose Microprocessor
Z8001 CPU	Segmented CPU
Z8002 CPU	Non-Segmented CPU
Z8003 VMPU	Segmented Virtual Memory Processing Unit
Z8010 MMU	Memory Management Unit
Z8015 PMMU	Paged Memory Management Unit
Z8030 Z-SCC	Serial Communications Controller
Z8036 Z-CIO	Counter Timer and I/O
Z8038 Z-FIO	FIFO I/O Interface
Z8052 Z-CRTC	CRT Controller
Z8060 Z-FIFO	FIFO Buffer and FIO Expander
Z8065 Z-BEP	Burst Error Processor
Z8068 Z-DCP	Data Ciphering Processor
Z8090 Z-UPC	Universal Peripheral Controller
Z8500 FAMILY	Universal Peripherals
Z8530 SCC	Serial Communications Controller
Z8536 CIO	Counter Timer and I/O
Z8590 UPC	Universal Peripheral Controller
Z6000 FAMILY	Microprocessor Memories
Z6132	Quasi-Static RAM

Table of Contents

Z8 Single-Chip Microcomputer Family	1
Advanced Architecture of the Z8 Microcomputer	1-3
A Comparison of MCU Units	1-15
Z8600 Interrupt Request Register	1-29
12-Bit Addressing with the Z8600 Family	1-31
Z8 Family Software Framing Error Detection	1-33
A Programmer's Guide to the Z8 Microcomputer	1-35
<hr/>	
Z80 8-Bit Microprocessor Family	2
Get Powerful Microprocessor Performance	2-2
Designing a Microprocessor Driven Multipurpose Peripheral Controller	2-13
Z80 Family Interrupt Structure	2-23
A Z80-Based System Using the DMA with SIO	2-35
Using the Z80 SIO in Asynchronous Communications	2-47
Using the Z80 with SDLC	2-71
Binary Synchronous Communication Using the Z80 SIO	2-83
Serial Communication with the Z80A DART	2-87
Interfacing 8500 Peripherals to the Z80	2-97
Serial Clock Generation Using the Z8536 CIO	2-103
Timing in an Interrupt-Based System with the Z80 CTC	2-107
Interfacing 16-Pin Dynamic RAMs to the Z80A Microprocessor	2-119
Controlling Z80 Microcomputer I/O? An Interrupt-Driven Program Could Help	2-147
<hr/>	
Z8000 16-Bit Microprocessor	3
Advanced Architectural Features of the Z8000 CPU	3-3
A Small Z8000 System	3-17
An Introduction to the Z8010 MMU Memory Management Unit	3-29
An Introduction to Memory Management	3-49
The Z8000 and the 68000 Concept Papers: Introduction	3-57
Z8000 vs 68000 Register Architecture	3-59
Memory-Mapped vs Explicit I/O	3-61
Address/Data Bus	3-63
Segmented vs Linear Addressing	3-65
Operating System Support Features	3-75
A Tale of Four Microprocessors: Benchmarks Quantify Performance	3-85
Architectural Concepts for Microprocessor Peripheral Families	3-105
<hr/>	
Z-Family Memories	4
Interfacing the Z6132 Intelligent Memory	4-3
<hr/>	
Z-BUS	5
Z-BUS Component Interconnect	5-3
Z-BUS and Peripheral Support Packages Tie Distributed Computer Systems Together	5-19
<hr/>	
General	6
An Optimizing Driver for NEC Spinwriters and Diablo Printers	6-3
ZRTS 8000 Zilog Real-Time Software for the Z8000 Microprocessor	6-61
Peripheral Controller Chip Ties into 8- and 16-Bit Systems	6-65
Adapting Unix to a 16-Bit Microcomputer	6-71
Major Firms Join Unix Parade	6-77

Z8™ Single Chip Microcomputer Family 1

Z8™
Z8™
Z8™
Z8™
Zilog

The Advanced Architectural Features of the Z8 Microcomputer

Stephen Walters
Manager of Component Applications
Zilog, Inc.
10460 Bubb Road
Cupertino, CA 95014

INTRODUCTION

The semiconductor industry accomplished dramatic technological advances in the area of MOS integrated circuit microprocessors during the 1970's, and as the next decade begins two trends are very clear. The first is the continued increased capability of the high-end general purpose microprocessors. Sixteen bit microprocessors will mature with additional "big machine" features, and 32-bit microprocessors will develop.

The second trend is in the area of single chip microcomputers. Single chip microcomputers are offering substantially greater processing power than when they were first introduced. Microcomputers are no longer limited to low end applications where unit cost and power dissipation are the primary design considerations.

Zilog is applying classical computer architecture concepts to the design of its microcomputer products. Upon close examination of the Zilog Z8 Microcomputer, one notices features that once were available only on general purpose bus oriented microprocessor products such as;

- separate program and data space
- the stack pointer and the PUSH and POP instructions
- 126K byte total memory address space
- vectored interrupts
- the CALL and RET (Return) instructions for procedure calls.

The trend in high-end single chip microcomputer architecture is clear and the consequences are obvious. The multi-chip solutions of today that employ 8-bit general purpose microprocessors will be replaced by more powerful 8-bit or 16-bit single chip microcomputers in the future.

This paper will discuss the architectural features of the Z8 Microcomputer and describe an application of the Z8 that takes advantage of the off chip expansion capability.

ARCHITECTURAL OVERVIEW

The architecture of the Z8 microcomputer offers many advanced processing features not previously available with single chip microcom-

puters. The Z8 combines a powerful instruction set, simplified system expansion off chip, and flexible serial and parallel I/O capabilities to provide design solutions for a wide range of application problems.

The Z8 has a 16-bit Program Counter and a separate 16-bit Stack Pointer. The memory space may be extended beyond the 2K bytes of ROM and 124 bytes of RAM on chip, up to 126K bytes of program and data memory. There are 32 bits of I/O which can be configured into a variety of bit, nibble, and byte organizations, and the serial I/O port is a complete full duplex asynchronous receiver/transmitter. The Z8 interrupt structure allows the user to mask and prioritize the interrupt functions under program control, and the interrupts are directed to the appropriate service routine through 16-bit vectors in the first 12 locations of program memory. Two counter/timers are provided to off load time base generation and interval detection tasks from the Z8. The Z8 will operate with an 8 MHz clock and the exact frequency up to 8 MHz may be set with an external crystal, an external RC, or an external clock source. The Z8 operates from a single 5 volt power supply and offers a power down mode that allows the 124 general purpose registers on chip to operate from a back up battery. A Block Diagram of the Z8 is given in Figure 1.

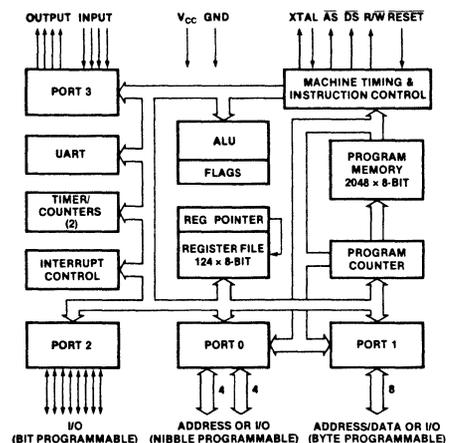


Figure 1 Z8 Block Diagram

MEMORY SPACE AND REGISTER

ORGANIZATION

Memory Space

The Z8 can address up to 126K bytes of program and data memory separately from the on chip registers. The 16-bit program counter provides for 64K bytes of program memory, the first 2K bytes of which are internal to the Z8. The remaining 62K bytes of program memory are located externally and can be implemented with ROM, EPROM, or RAM.

The 62K bytes of data memory are also located external to the Z8 and begin with location 2048. The two address spaces, program memory and data memory, are individually selected by the Data Memory Select output (\overline{DM}) which is available from Port 3.

The Program Memory Map and the Data Memory Map are shown in Figure 2.

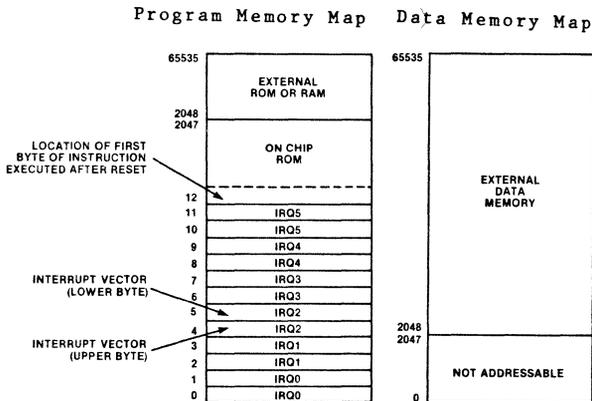


Figure 2 Program Memory Map And Data Memory Map

External memory access is accomplished by the Z8 through its I/O Ports. When less than 256 bytes of external memory are required, Port 1 is programmed for the multiplexed address/data mode ($\overline{AD0-AD7}$). In this configuration 8-bits of address and 8-bits of data are time multiplexed on the 8 I/O lines for memory transfers. The memory "handshake" control lines are provided by the Address Strobe (\overline{AS}), Data Strobe (\overline{DS}), and the Read/Write (R/\overline{W}) pins on the Z8. If program and data are included in the external memory space, the Data Memory Select (\overline{DM}) function may be programmed into the Port 3 Mode register. When this is done, the \overline{DM} signal is available on

line 4 of the Port 3 (P34) to select between program and data memory for external memory operations.

Port 0 is used to provide the additional address bits for external memory beyond the first 256 locations up to a full 16-bits of external memory address. It becomes immediately obvious that the first 8-bits of external memory address from Port 1 must be latched externally to the Z8 so that program or data may be transferred over the same 8 lines during the external memory transaction machine cycle. The \overline{AS} , \overline{DS} , and R/\overline{W} control lines simplify the required interface logic. The timing for external memory transactions is given in Figure 3.

Registers

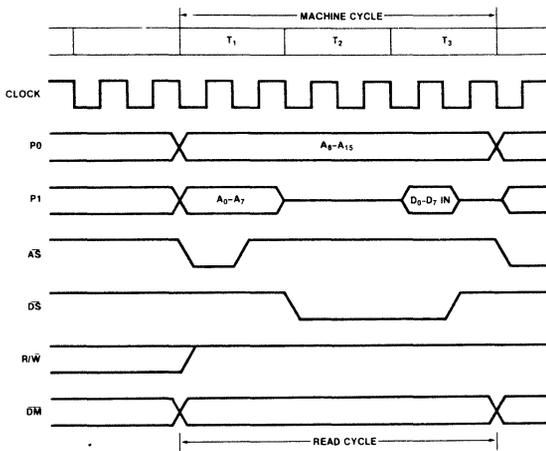
The Z8 has 144 8-bit registers including four Port registers (R0-R3), 124 general purpose registers (R4-R127), and 16 control and status register (R240-R255). The 144 registers are all located in the same 8-bit address space to allow any Z8 instruction to operate on them. The 124 general purpose registers can function as accumulators, address pointers, or index registers. The registers are read when they are referenced as source registers, and written when they are referenced as destination registers. Registers may be addressed directly with an 8-bit address, or indirectly through another register with an 8-bit address, or with a 4-bit address and Register Pointer.

The entire Z8 register space may be divided into 16 contiguous Working Register Areas, each having 16 registers. A control register, called the Register Pointer, may be loaded with the most significant nibble of a Working Register Area address. The Register Pointer provides for the selection of the Working Register Area, and allows registers within that area to be selected with a 4-bit address.

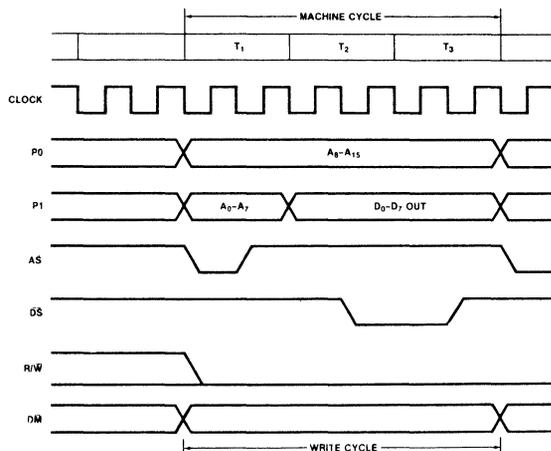
The Z8 register organization is shown in Figure 4.

Stacks

The Z8 provides for stack operations through the use of a stack pointer, and the stack may be located in the internal register space or in the external data memory space. The "stack selection" bit (D2) in the Port 0-1 Mode control register selects an internal or external stack. When the stack is located internally, register 255 contains an 8-bit stack pointer and register 254 is available as a general purpose register. If an external stack is used, register 255 or registers 254 and 255 may be used as the stack pointer depending on the anticipated "depth" of the stack. When registers 254 and 255 are both used, the stack pointer is a full 16-bits wide. The CALL, IRET, RET, PUSH, and



Memory Read Cycle



Memory Write Cycle

External Memory Transaction Cycle

Figure 3

LOCATION		IDENTIFIERS
255	STACK POINTER (BITS 7-0)	SPL
254	STACK POINTER (BITS 15-8)	SPH
253	REGISTER POINTER	RP
252	PROGRAM CONTROL FLAGS	FLAGS
251	INTERRUPT MASK REGISTER	IMR
250	INTERRUPT REQUEST REGISTER	IRQ
249	INTERRUPT PRIORITY REGISTER	IPR
248	PORTS 0-1 MODE	P01M
247	PORT 3 MODE	P3M
246	PORT 2 MODE	P2M
245	T0 PRESCALER	PRE0
244	TIMER/COUNTER 0	T0
243	T1 PRESCALER	PRE1
242	TIMER/COUNTER 1	T1
241	TIMER MODE	TMR
240	SERIAL I/O	SIO
	NOT IMPLEMENTED	
127	GENERAL-PURPOSE REGISTERS	
4		
3	PORT 3	P3
2	PORT 2	P2
1	PORT 1	P1
0	PORT 0	P0

Figure 4 Register File Organization

POP instructions are Z8 instructions which include implicit stack operations.

I/O STRUCTURE

Parallel I/O

The Z8 microcomputer has 32 lines of I/O arranged as four 8-bit ports. All of the I/O ports are TTL compatible and are configurable as input, output, input/output, or address/data. The handshake control lines for Ports 0, 1, and 2 are bits from Port 3 that have been programmed through a Mode control register, except for \overline{AS} , \overline{DS} , and R/W which are available as separate Z8 pins. The I/O ports are accessed as separate internal registers by the Z8. Ports 0 and 1 share one Mode control register, and Ports 2 and 3 each have a Mode control register for configuring the port.

Port 0 can be programmed to be an I/O port or as an address output port. More specifically Port 0 can be configured to be an 8-bit I/O port, or a 4-bit address output port (A8-A11) for external memory and one 4-bit I/O port, or an 8-bit address output port (A8-A15) for external memory.

Port 1 can be programmed as an I/O port (with or without handshake), or an address/data port (AD $\overline{0}$ -AD7) for interfacing with external memory. If Port 1 is programmed to be an address/data port, it cannot be accessed as a register.

Port 2 can be configured as individual input or output bits, and Port 3 can be programmed to be parallel I/O bits, and/or serial I/O bits, and/or handshake control lines for the other ports. Figure 5 shows the port Mode registers.

The off chip expansion capability using Ports 0 and 1 offers the added feature of being Z-Bus compatible. All Z-Bus compatible peripheral chips that are available now, and will be available in the future, will interface directly with the Z8 multiplexed address/data bus.

Serial I/O

As mentioned in the last section, Port 3 can be programmed to be a serial I/O port with bits 0 and 7, the serial input and serial output lines respectively. The serial I/O capability provides for full duplex asynchronous serial data at rates up to 62.5K bits per second. The transmitted format is one start bit, eight data bits including odd parity (if parity is enabled), and two stop bits. The received data format is one start bit, eight data bits and at least one stop bit. If parity is enabled, the eighth data bit received (bit 7) is replaced by

a parity error flag which indicates a parity error if it is set to a ONE.

Timer/Counter T_0 is the baud rate generator and runs at 16 times the serial data bit rate. The receiver is double duffered and an internal interrupt (IRQ3) is generated when a character is loaded into the receive buffer register. A different internal interrupt (IRQ4) is generated when a character is transmitted.

COUNTER/TIMERS

The Z8 has two 8-bit programmable counter/timers, each of which is driven by a programmable 6-bit prescaler. The T_1 prescaler can be driven by internal or external clock sources, and the T_0 prescaler is driven by the internal clock only. The two prescalers and the two counters are loaded through four control registers (see Figure 4) and when a counter/timer reaches the "end of count" a timer interrupt is generated (IRQ4 for T_0 , and IRQ5 for T_1). The counter/timers can be programmed to stop upon reaching the end of count, or to reload and continue counting. Since either counter (one at a time) can have its output available external to the Z8, and Counter/Timer T_1 can have an external input, the two counters can be cascaded.

Port 3 can be programmed to provide timer outputs for external time base generation or trigger pulses.

INTERRUPT STRUCTURE

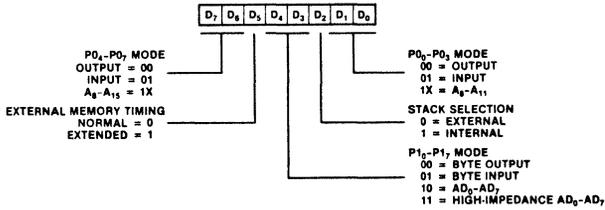
The Z8 provides for six interrupts from eight different sources including four Port 3 lines (P30-P33), serial in, serial out, and two counter/timers. These interrupts can be masked and prioritized using the Interrupt Mask Register (register 251) and the Interrupt Priority Register (register 249). All interrupts can be disabled with the master interrupt enable bit in the Interrupt Mask Register.

Each of the six interrupts has a 16-bit interrupt vector that points to its interrupt service routine. These six 2-byte vectors are placed in the first twelve locations in the program memory space (see Figure 2).

When simultaneous interrupts occur for enabled interrupt sources, the Interrupt Priority Register determines which interrupt is serviced first. The priority is programmable in a way that is described by Figure 6.

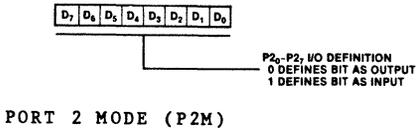
When an interrupt is recognized by the Z8, all other interrupts are disabled, the program counter and program control flags are saved, and the program counter is loaded with the corresponding interrupt vector. Interrupts must be re-enabled by the user upon entering the service

R248 (F8_n) Ports 0 and 1 Mode Register (P01M; Write Only)



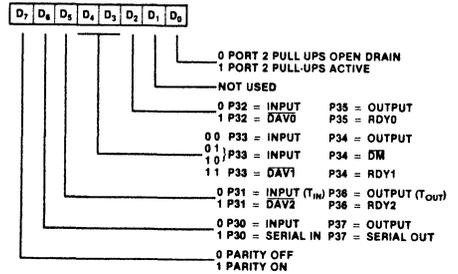
PORTS 0 AND 1 MODES (P01M)

R246 (F6_n) Port 2 Mode Register (P2M; Write Only)



PORT 2 MODE (P2M)

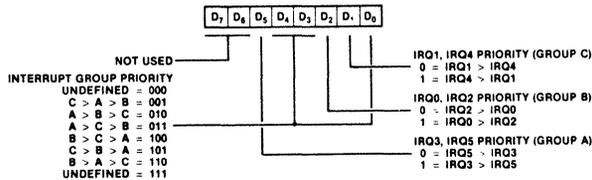
R247 (F7_n) Port 3 Mode Register (P3M; Write Only)



PORT 3 MODE (P3M)

Figure 5 Port Mode Registers

R249 (F9_n) Interrupt Priority Register (IPR; Write Only)



R249 INTERRUPT PRIORITY REGISTER (IPR)

Figure 6

routine (for nested interrupts), or upon returning from the interrupt service routine using the IRET instruction. The interrupt cycle process is shown in Figure 7.

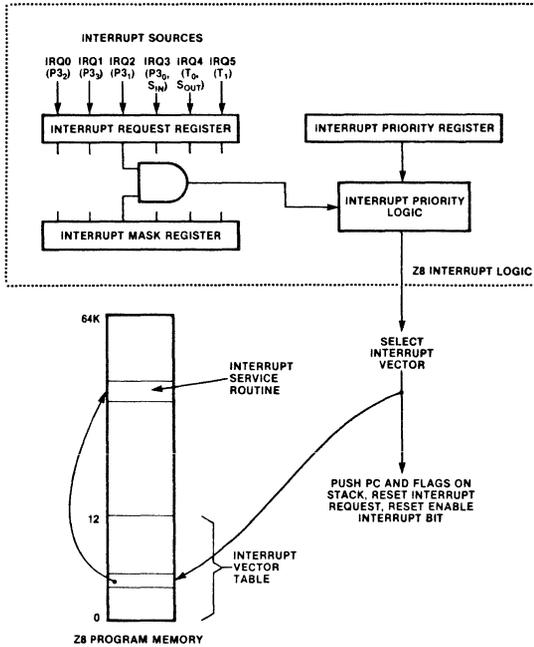


Figure 7 Interrupt Cycle Process

INSTRUCTION SET

The Z8 uses six address modes; Register, Indirect Register, Indexed, Direct, Relative, and Immediate. The Register mode refers to a register for operands. The Indirect Register mode refers indirectly through a register to an operand in a second register; indirectly through a register pair to an operand in program memory; or indirectly through a register pair to an operand in data memory. The Indexed mode is used only by the Load (LD) instruction and provides a method for generating an effective address which is the sum of a register address (contained in the instruction) and the content of the index register. The Indexed mode employs Working Register area shortened notation for specifying the Index register. The Direct mode provides for a transfer of control to anywhere in program memory with a two byte address in the instruction. The Relative mode is used only with the Jump Relative, and Decrement And Jump instructions. The relative offset contained in the instruction allows a jump to an address

which is -128 locations or +127 locations from the address of the instruction following the jump instruction. The Immediate mode provides the operand in the instruction.

There are eight instruction functional groups;

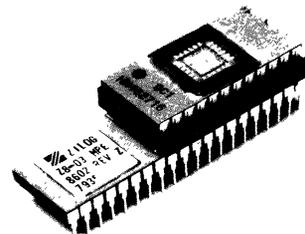
Load	Bit Manipulation
Arithmetic	Block Transfer
Logical	Rotate and Shift
Program Control	CPU Control

A summary of the Z8 instructions by function is given in Appendix A.

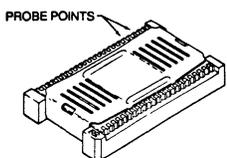
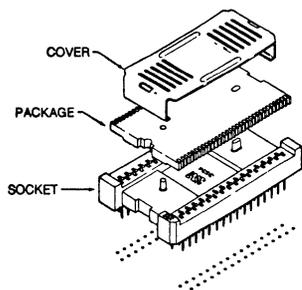
The Z8 addressing modes are optimized for using the internal ROM and RAM memories. Two of the reasons why this was done were; to improve code density (fewer bytes per instruction), and to reduce execution time.

THE Z8 FAMILY

The Z8 family emerged with three versions of the basic microcomputer; the 40-pin ROM version, the 40-pin EPROM version, and the 64-pin version. The 40-pin EPROM version is offered in a Zilog proprietary package called a Protopak (see Figure 8), that has a socket for an EPROM mounted permanently on top of a 40-pin DIP. This device will plug directly into the socket of a product designed for the ROM version, or can be the initial production component for a product that may ultimately be converted to the ROM version of the Z8. The 64-pin version has no internal ROM and comes in a 64-pin leadless chip carrier (see figure 9). The eleven ROM address lines and eight ROM data lines are brought to pins on this version of the Z8. A 4K ROM version of the Z8 is planned for release toward the end of 1980.



Z8 Protopak
Figure 8



Z8/64 Package

Figure 9

Z8 APPLICATION

Typeset Innovations, Inc. is a company based in Austin, Texas, that has designed a graphics computing system based on the Z8 micro-computer. The ProGrafix is a specialized electronic computational aid for use by graphics arts professionals in the sizing and pricing of graphic elements. Graphics arts professionals are exemplified by typesetting job estimators, typographers, graphic designers, printers, advertising layout artists, and book and magazine designers.

Graphic artists have in the past performed copyfitting through trial and error. With the increasing costs of graphics materials, the trial and error method has become a noticeable expense that can be minimized with a more accurate copyfitting technique.

The ProGrafix performs the following functions;

- Entry and display of values in the units of measurement which are commonly used in the typographical arts, including:

picas
points
picas and points
inches

ciceros
didots
ciceros and didots
centimeters

relative units

- Instantaneous, one keystroke conversion of values between any of the above units of measurement.
- Arithmetic operations using values which are expressed in any of the above units of measurement.
- One keystroke execution of an extremely accurate copyfitting algorithm which finds any unknown copyfitting value after the five known ones have been entered, from among the following copy descriptors:

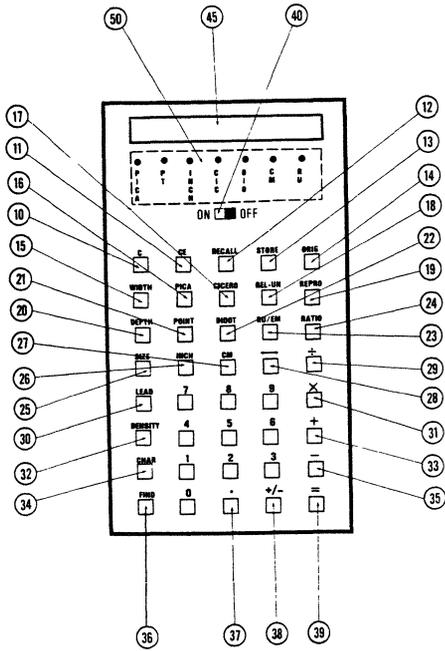
width
depth
size
leading
type style density
character count

- Graphic proportional enlargement/reduction computations by finding any one of the following values after the other two are entered:
- original size
reproduction size
enlargement/reduction ratio
- Memory storage and recall of intermediate results, pricing constants, and other user-created values.

When final packaging is complete the ProGrafix will appear similar to the drawing shown in Figure 10. The computational aid will have an 8 digit display, 7 annunciator LED's to indicate measurement units, an on/off switch, and a 40 key keyboard matrix. The key functions are defined in Table 1.

When Typeset Innovations began the design of the ProGrafix early in 1980, they looked for a microcomputer that had the following characteristics;

- A real (available) microcomputer powerful enough to do the job
- Compact coding
- Fast
- Easy to program
- External expansion of memory and I/O



ProGrafix Computational Aid

Figure 10

The Z8 offered all of these characteristics and more. By the second quarter of 1980, the ProGrafix prototypes were working.

The prototype implementation is shown in Figure 11. External ROM and RAM were added using Port 1 and half of Port 0 (A8-A11). The ability to add more than 2K bytes of external memory with only 12 address lines (A0-A11) is possible because the Data Strobe (DS) line is only active when locations above the first 2K bytes are accessed. Memory locations from 0 to 2K bytes are internal to the Z8; locations from 2K bytes to 4K bytes are external to the Z8 and selected by address line A11=1 and DS; and locations from 4K bytes to 6K bytes (RAM) are external to the Z8 and selected by address line A11 = 0 and DS active.

The remaining four bits of Port 0 were used to drive the Unit of Measure LED's and the "sign" for the numeric display.

Four of the I/O lines available from Port 3 were used to select one of eight digits on the numeric display through a 4 to 16 decoder and to scan the rows of the keypad. The other four I/O lines were used to read back the columns from the keypad.

One line from Port 2 was used for the fifth column input to the Z8 from the 40 key keypad. The remaining 7 I/O lines available from Port 2 were used for segment select on the numeric display.

The numeric display is "scan refreshed" by the Z8 at a rate that is approximately 100 times per second. As the digits of the display are being refreshed the keypad is scanned as a matrix of 8 by 5 keys. The counter/timers on the Z8 are both used; one to time the display refresh, and the other as a timer for keypad debounce. An external stack is used for temporary variable storage and during the servicing of interrupts. Only two Z8 interrupts are used by the ProGrafix, one for the display refresh counter and the other for the key debounce timer.

The development of the software for the ProGrafix, which included a BCD Floating Point package, was done on a Zilog development system with the Z8 PLZ/ASM assembler. The object code was down loaded to a Z8 Development Module (DM) where the hardware was initially debugged. The external memory was added to the Z8DM in the space provided for wire wrap. When the system was 90% - 95% debugged, a prototype circuit board was built and the Z8 in a Protopak package with an EPROM was used for final system debug.

The production version of the ProGrafix will use an LCD numeric display instead of the LED display. This will make additional address lines available for expanding off chip memory. In addition, a printer option is planned that will connect to the serial port of the Z8.

The ProGrafix is expected to sell for under \$500 without a printer and under \$750 with a printer. The availability of the ProGrafix has been targeted for May of 1981.

The configuration of the ProGrafix computational aid around the Z8 provided a very flexible and powerful microcomputer system that can be expanded to accommodate a wide variety of applications by simply changing the software. Typeset Innovations is currently looking for other products that can be implemented with the hardware that was developed for the ProGrafix.

CONCLUSION

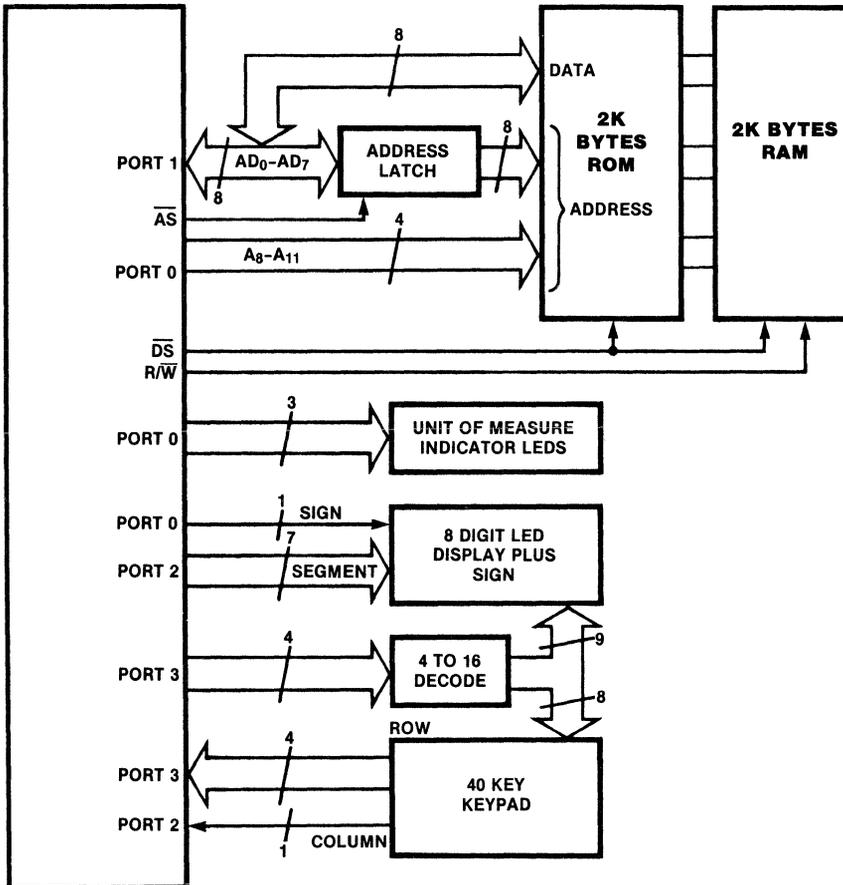
The Z8 represents the coming of age of the more powerful microcomputers. While the Z8 can be a cost effective design solution for low end applications, it can also be expanded to attack much more sophisticated design problems. The architecture of the Z8 was designed in a forward looking manner, and the integration of more capability onto the same chip is now limited only by the constraints of the integrated circuit technology.

TABLE 1

ProGrafix Key Functions

- Key 10 designated as C clears display register X and the operand register Y.
- Key 11 designated as CE, the clear-entry function, clears only the last value entered into display register X.
- Key 12 designated as RECALL enters the contents of a designated memory register 0 through 9 into register X and into the display.
- Key 13 designated STORE stores display register X into memory registers 0 through 9, and into the parameter registers.
- Key 14 designated as ORIG which is used to store or recall the value of the original size parameter in the proportional sizing algorithm.
- Key 15 designated as WIDTH which is used to store or recall the value of the width (line length) parameter of the copyfitting algorithm.
- Key 16 designated as PICA which provides for the function of entering information in picas and points and for converting information on the display into either picas and points or decimal picas.
- Key 17 designated as CICERO which provides for the function of entering information in ciceros and for converting information on the display into either ciceros and didots or decimal ciceros.
- Key 18 designated as REL-UN which provides for the function of entering information in relative units and for converting information on the display into relative units.
- Key 19 designated as REPRO which is used to store or recall the value of the reproduction size parameter in the proportional sizing algorithm.
- Key 20 designated as DEPTH which is used to store or recall the value of the depth (vertical measure) parameter of the copyfitting algorithm.
- Key 21 designated as POINT which provides for the function of entering information in points and for converting the information on the display into points.
- Key 22 designated as DIDOT which provides for the function of entering information in didot points and for converting the information on the display into didot points.
- Key 23 designated as RU/EM which is used to store or recall of the relative units per em space parameter.
- Key 24 designated as RATIO which is used to store or recall the value of the ratio parameter in the proportional sizing algorithm.
- Key 25 designated as SIZE which is used to store or recall the value of the type size parameter for the copyfitting algorithm and for the relative units conversion algorithm.
- Key 26 designated as INCH which provides for the function of entering information in inches and for converting the information on the display into inches.
- Key 27 designated as CM which provides for the dual function of entering information in centimeters, and for converting the information on the display into centimeters.
- Key 28 designated as "double arrow" interchanges the contents of display register X and operand register Y.
- Key 29 designated as \div . (the divide sign) divides operand register Y by display register X.
- Key 30 designated as LEAD which is used to store or recall the value of the leading (line spacing) parameter of the copyfitting algorithm.
- Key 31 designated as x multiples display register X by operand register Y.
- Key 32 designated as DENSITY which is used to store or recall the value of the type style density parameter of the copyfitting algorithm.
- Key 33 designated as + adds display register X to operand register Y.
- Key 34 designated as CHAR which is used to store or recall the value of the character count parameter of the copyfitting algorithm.
- Key 35 designated as - subtracts display register X from operand register Y.
- Key 36 designated as FIND invokes the calculation of an unknown copyfitting parameter given five known copyfitting parameters. This key is also used to solve for an unknown proportional sizing parameter given two known proportional sizing parameters.
- Key 37 designated . is used to enter the decimal point of a floating-point number.
- Key 38 designated as +/- reverses the sign of the value in display register X.
- Key 39 designated as = invokes the last entered arithmetic operation using the X and Y registers as operands and places the result in display register X.
- Key 40 designated as ON/OFF powers the micro-computer system on and off.

**Z8
MICROCOMPUTER**



PROGRAFIX BLOCK DIAGRAM

Figure 11

APPENDIX A

Z8 Instruction Set: Functional Groups

Load Instructions

Instruction	Operand (s)	Name of Instruction
CLR	dst	Clear
LD	dst, src	Load
LDC	dst, src	Load Constant
LDE	dst, src	Load External Data
POP	dst	Pop
PUSH	src	Push

Arithmetic Instructions

Instruction	Operand (s)	Name of Instruction
ADC	dst, src	Add With Carry
ADD	dst, src	Add
CP	dst, src	Compare
DA	dst	Decimal Adjust
DEC	dst	Decrement
DECW	dst	Decrement Word
INC	dst	Increment
INCW	dst	Increment Word
SBC	dst, src	Subtract With Carry
SUB	dst, src	Subtract

Logical Instructions

Instruction	Operand (s)	Name of Instruction
AND	dst, src	Logical And
COM	dst	Complement
OR	dst, src	Logical Or
XOR	dst, src	Logical Exclusive Or

Program-Control Instructions

Instruction	Operand (s)	Name of Instruction
CALL	dst	Call
DJNZ	r, dst	Decrement and Jump If Nonzero
IRET		Interrupt Return
JP	cc, dst	Jump
JR	cc, dst	Jump Relative
RET		Return

APPENDIX A (cont.)

Bit-Manipulation Instructions

Instruction	Operands	Name of Instruction
TCM	dst, src	Test Complement Under Mask
TM	dst, src	Test Under Mask
AND	dst, src	Logical And
OR	dst, src	Logical Or
XOR	dst, src	Logical Exclusive Or

Block-Transfer Instructions

Instruction	Operands	Name of Instruction
LDCI	dst, src	Load Constant Autoincrement
LDEI	dst, src	Load External Data Auto-increment

Rotate and Shift Instructions

Instruction	Operand	Name of Instruction
RL	dst	Rotate Left
RLC	dst	Rotate Left Through Carry
RR	dst	Rotate Right
RRC	dst	Rotate Right Through Carry
SRA	dst	Shift Right Arithmetic
SWAP	dst	Swap Nibbles

CPU Control Instructions

Instruction	Operand	Name of Instruction
CCF		Complement Carry Flag
DI		Disable Interrupts
EI		Enable Interrupts
NOP		No Operation
RCF		Reset Carry Flag
SCF		Set Carry Flag
SRP	src	Set Register Pointer

A Comparison of Microcomputer Units



Benchmark Report

MAY 1981

INTRODUCTION

The microcomputer industry has recently developed single-chip microcomputers that incorporate on one chip functions previously performed by peripherals. These microcomputer units (MCUs) are aimed

at markets requiring a dedicated computer. This report describes and compares the most powerful MCUs in today's market: the Zilog Z8611, the Intel 8051, and the Motorola MC6801. Table 1 lists facts that should be considered when comparing these MCUs.

Table 1. MCU Comparison

FEATURES	Zilog Z8611	Intel 8051	Motorola MC6801
On-Chip ROM	4Kx8	4Kx8	2Kx8
General-Purpose Registers	124	128	128
Special-Function Registers			
Status/Control	16	16	17
I/O ports	4	4	4
I/O Parallel lines	32	32	29
Ports	Four 8-bit	Four 8-bit	Three 8-bit, one 5-bit
Handshake	Hardware on three ports	None	Hardware on one port
Interrupts			
Source	8	5	7
External source	4	2	2
Vector	6	5	7
Priority	48 Programmable orders	2 Programmable orders	Nonprogrammable
Maskable	6	5	6
External Memory	120K bytes	124K bytes	64K bytes
Stack			
Stack pointer	16-Bit	8-Bit	16-Bit
Internal stack	Yes, uses 8-bits	Yes	Yes
External stack	Yes	No	Yes

**Table 1. MCU Comparison
(Continued)**

FEATURES	Zilog Z8611	Intel 8051	Motorola MC6801
Counter/ Timers Counters Prescalers	Two 8-bit Two 6-bit	Two 16-bit or two 8-bit No prescale with 16-bits; 5-bit prescale with 8-bits	One 16-bit None
Addressing Modes Register Indirect Register Indexed Direct Relative Immediate Implied	Yes Yes Yes Yes Yes Yes Yes	Yes Yes Yes Yes Yes Yes Yes	No No Yes Yes Yes Yes Yes
Index Registers	124, Any general- purpose register	1, Uses the accumulator for 8-bit offset	1, Uses 16-bit index register
Serial Communication Interface Full duplex UART Interrupts for transmit and receive Registers Double buffer Serial Data Rate	Yes One for each Receiver 62.5K b/s @8 MHz 93.5K b/s @12 MHz	Yes One for both Receiver 187.5K b/s @12 MHz	Yes One for both Transmitter/Receiver 62.5K b/s @4 MHz
Speed Instruction execution average Longest instruction	2.2 Usec 1.5 Usec @12 MHz 4.25 Usec 2.8 Usec @12 MHz	1.5 Usec 4 Usec	3.9 Usec 10 Usec
Clock Frequency	8 and 12 MHz	12 MHz	4 MHz
Power Down Mode	Saves first 124 registers	Saves first 128 registers	Saves first 64 registers
Context Switching	Saves PC and flags	Saves PC; programmer must save all registers	Saves PC, PSW, accumulators, and Index register

Table 1. MCU Comparison
(Continued)

FEATURES	Zilog Z8611	Intel 8051	Motorola MC6801
Development	40-Pin Protopack (8613) 64-Pin (8612) 40-Pin ROMless (Z8681)	40-Pin (8751)	40-Pin (68701)
Eprom	4K bytes (2732) 2K bytes (2716)	4K bytes	2K bytes
Availability	Now	TBA	Now

ARCHITECTURAL OVERVIEW

This section examines three chips: the on-chip functions and data areas manipulated by the Zilog, Intel and Motorola MCUs. The three chips have somewhat similar architectures. There are, however, fundamental differences in design criteria. The 8051 and the MC6801 were designed to maintain compatibility with older products, whereas the Z8611 design was free from such restrictions and could experiment with new ideas. Because of this, the accumulator architectures of the MC6801 and the 8051 are not as flexible as that of the Z8611, which allows any register to be used as an accumulator.

Memory Spaces

The Z8611 CPU manipulates data in four memory spaces:

- 60K bytes of external data memory
- 60K bytes of external program memory
- 4K bytes of internal program memory (ROM)
- 144-byte register file

The 8051 CPU manipulates data in four memory spaces:

- 64K bytes of external data memory
- 60K bytes of external program memory
- 4K bytes of internal program memory
- 148-byte register file

The MC6801 manipulates data in three memory spaces:

- 62K bytes of external memory
- 2K bytes of internal program memory
- 149-byte register file

On-Chip ROM. All three chips have internal ROM for program memory. The Z8611 and the 8051 have 4K bytes of internal ROM, and the MC6801 has 2K bytes. In some cases, external memory may be

required with the MC6801 that is not necessary with the Z8611 or the 8051.

On Chip RAM. All three chips use internal RAM as registers. These registers are divided into two categories: general-purpose registers and special function registers (SFRs).

The 124 general-purpose registers in the Z8611 are divided into eight groups of 16 registers each. In the first group, the lowest four registers are the I/O port registers. The other registers are general purpose and can be accessed with an 8-bit address or a short 4-bit address. Using the 4-bit address saves bytes and execution time. Four-bit short addresses are discussed later. The general-purpose registers can be used as accumulators, address pointers, or Index registers.

The 128 general-purpose registers in the 8051 are grouped into two sets. The lower 32 bytes are allocated as four 8-register banks, and the upper registers are used for the stack or for general purpose. The registers cannot be used for indexing or as address pointers.

The MC6801 also has a 128-byte, general-purpose register bank, which can be used as a stack or as address pointers, but not as Index registers.

As pointed out in Table 1, any of the Z8611 general-purpose registers can be used for indexing; the MC6801 and the 8051 cannot use registers this way. The Z8611 can use any register as an accumulator; the MC6801 and the 8051 have fixed accumulators. The use of registers as memory pointers is very valuable, and only the Z8611 can use its registers in this way.

The number of general-purpose registers on each chip is comparable. However, because of its flexible design, the Z8611 clearly has a more powerful register architecture.

The Z8611 has 20 special function registers used for status, control, and I/O. These registers include:

- Two registers for a 16-bit Stack Pointer (SPH, SPL)
- One register used as Register Pointer for working registers (RP)
- One register for the status flags (FLAGS)
- One register for interrupt priority (IPR)
- One register for interrupt mask (IMR)
- One register for interrupt request (IRQ)
- Three mode registers for the four ports (P01M, P2M, P3M)
- Serial communications port used like a register (SIO)
- Two counter/timer registers (T0, T1)
- One Timer Mode Register (TMR)
- Two prescaler registers (PRE0, PRE1)
- Four I/O ports accessed as registers (PORT0, PORT1, PORT2, PORT3)

The 8051 also has 20 special function registers used for status, control, and I/O. They include:

- One register for the Stack Pointer (SP)
- Two accumulators (A,B)
- One register for the Program Status Word (PSW)
- Two registers for pointing to data memory (DPH, DPL)
- Four registers that serve as two 16-bit counter/timers (TH0, TH1, TL0, TL1)
- One mode register for the counter/timers (TMOD)
- One control register for the counter/timers (TCON)
- One register for interrupt enable (IEC)
- One register for interrupt priority (IPC)
- One register for serial communications buffer (SBUF)
- One register for serial communications control (SCON)
- Four registers used as the four I/O ports (P0, P1, P2, P3)

The MC6801 has 21 special function registers used for status, control, and I/O. These include:

- One register for RAM/EROM control
- One serial receive register
- One serial transmit register
- One register for serial control and status
- One serial rate and mode register
- One register for status and control of port 3
- One register for status and control of the timer
- Two registers for the 16-bit timer
- Two registers for 16-bit input capture used with timer
- Two registers for 16-bit output compare used with timer
- Four data direction registers associated with the four I/O ports
- Four I/O ports

The special function registers in the three chips seem comparable in number and function. However, upon closer examination, the SFRs of the MC6801 prove less efficient than those of the Z8611. The MC6801 has five registers associated with the I/O ports, whereas the Z8611 uses only three registers for the same functions. The MC6801 uses four registers to perform the serial communication function, whereas the Z8611 uses only one register and part of another.

The 8051 uses two registers for the accumulators; the Z8611 is not limited by this restriction. The 8051 also uses two registers for the serial communication interface, whereas the Z8611 accomplishes the same job with one register. Another two registers in the 8051 are used for data pointers; these are not necessary in the Z8611 since any register can be used as an address pointer.

The Z8611 uses registers more efficiently than either the MC6801 or the 8051. The registers saved by this optimal design are used to perform the functions needed for enhanced interrupt handling and for register pointing with short addresses. The Z8611 also supplies the extra register required for the external stack. These features are not available on the 8051 or the MC6801.

External Memory. All three chips can access external memory. The Z8611 and the 8051 can generate signals used for selecting either program or data memory. The Data Memory strobe (the signal used for selecting data or program memory) gives the Z8611 access to 120K bytes of external memory (60K bytes in both program and data memory). The 8051 can use 124K bytes of external memory (64K bytes of external data memory and 60K bytes of external program memory). The MC6801 can access only 62K bytes of external memory and does not distinguish between program and data memory. Thus, the Z8611 and the 8051 are clearly able to access more external memory than the MC6801.

On-Chip Peripheral Functions

In addition to the CPU and memory spaces, all chips provide an interrupt system and extensive I/O facilities including I/O pins, parallel I/O ports, a bidirectional address/ data bus, and a serial port for I/O expansion.

Interrupts. The Z8611 acknowledges interrupts from eight sources, four are external from pins IRQ₀-IRQ₃, and four are internal from serial-in, serial-out, and the two counter/timers. All interrupts are maskable, and a wide variety of priorities are realized with the Interrupt Mask Register and the Interrupt Priority Registers (see Table 1). All Z8611 interrupts are vectored, with six vectors located in the on-chip ROM. The vectors are fixed locations, two bytes long, that contain the memory address of the service routine.

The 8051 acknowledges interrupts from five sources: two external sources (from INTO and INT1) and three internal sources (one from each of the internal counters and one from the serial I/O port). All interrupts can be disabled individually or globally. Each of the five sources can be assigned one of two priorities: high or low. All 8051 interrupts are vectored. There are five fixed locations in memory, each eight bytes long, allocated to servicing the interrupt.

The MC6801 has one external interrupt, one non-maskable interrupt, an internal interrupt request, and a software interrupt. The internal interrupts are caused by the serial I/O port, timer overflow, timer output compare, and timer input capture. The priority of each interrupt is preset and cannot be changed. The external interrupt can be masked in the Condition Code register. The MC6801 vectors the interrupts to seven fixed addresses in ROM where the 16-bit address of the service routine is located.

When an interrupt occurs in the 8051, only the Program Counter is saved; the user must save the flags, accumulator, and any registers that the interrupt service routine might affect. The MC6801 saves the Program Counter, accumulators, Index register, and the PSW; the user must save all registers that the interrupt service routine might affect. The Z8611 saves the Program Counter and the Flags register. To save the 16 working registers, only the Register Pointer register need be pushed onto the stack and another set of working registers is used for the service routine. For more detail on working registers and interrupt context switching, see the Z8 Technical Manual (03-3047-02).

With regard to interrupts, the Z8611 is clearly superior. The Z8611 requires only one command to save all the working registers, which greatly increases the efficiency of context switching.

I/O Facilities. The Z8611 has 32 lines dedicated to I/O functions. These lines are grouped into four ports with eight lines per port. The ports can be configured individually under software control to provide input, output, multiplexed address/data lines, timing, and status. Input and output can be serial or parallel, with or without handshake. One port can be configured for serial transmission and four ports can be configured for parallel transmission. With parallel transmission, ports 0, 1, and 2 can transmit data with the handshake provided by port 3.

The 8051 also has 32 I/O lines grouped together into four ports of eight lines each. The ports can be configured under program control for parallel or serial I/O. The ports can also be configured for multiplexed address/data lines, timing, and status. Handshake is provided by user software.

The MC6801 has 29 lines for I/O (three 8-bit ports and one 5-bit port). One port has two lines for

handshake. The ports provide all the signals needed to control input and output either serially or in parallel, with or without multiplexed address/data lines. They can be used to interface with external memory.

The main differences in I/O facilities are the number of 8-bit ports and the hardware handshake. The Z8611 and the 8051 have four 8-bit ports, whereas the MC6801 has three 8-bit ports and an additional 5-bit port. The Z8611 has hardware handshake on three ports, the MC6801 has hardware handshake on only one port, and the 8051 has no hardware handshake.

Counter/timers. The Z8611 has two 8-bit counters and two 6-bit programmable prescalers. One prescaler can be driven internally or externally; the other prescaler is driven internally only. Both timers can interrupt the CPU when counting is completed. The counters can operate in one of two modes: they can count down until interrupted, or they can count down, reload the initial value, and start counting down again (continuously). The counters for the Z8611 can be used for measuring time intervals and pulse widths, generating variable pulse widths, counting events, or generating periodic interrupts.

The 8051 has two 16-bit counter/timers for measuring time intervals and pulse widths, generating pulse widths, counting events, and generating periodic interrupts. The counter/timers have several modes of operation. They can be used as 8-bit counters or timers with two 5-bit programmable prescalers. They can also be used as 16-bit counter/timers. Finally, they can be set as 8-bit modulo-n counters with the reload value held in the high byte of the 16-bit register. An interrupt is generated when the counter/timer has completed counting.

The MC6801 has one 16-bit counter which can be used for pulse-width measurement and generation. The counter/timer actually consists of three 16-bit registers and an 8-bit control/status register. The timer has an input capture register, an output compare register, and a free-running counter. All three 16-bit registers can generate interrupts.

Serial Communications Interface. The Z8611 has a programmable serial communication interface. The chip contains a UART for full-duplex, asynchronous, serial receiver/transmitter operation. The bit rate is controlled by counter/timer 0 and has a maximum bit rate of 93,500 b/s. An interrupt is generated when an assembled character is transferred to the receive buffer. The transmitted character generates a separate interrupt. The receive register is double-buffered. A hardware parity generator and detector are optional.

The 8051 handles serial I/O using one of its parallel ports. The 8051 bit rate is controlled

by counter/timer 1 and has a maximum bit rate of 187,500 b/s. The 8051 generates one interrupt for both transmission and receipt. The receive register is double-buffered.

The MC6801 contains a full-duplex, asynchronous, serial communication interface. The bit rate is controlled by a rate register and by the MCU's clock or an external clock. The maximum bit rate is 62,500 b/s. Both the transmit and the receive registers are double-buffered. The MC6801 generates only one interrupt for both transmit and receive operations. No hardware parity generation or detection is available, although it does have automatic detection of framing errors and overrun conditions.

The 8051 and the MC6801 generate only one interrupt for both transmit and receive, whereas the Z8611 has a separate interrupt for each. The ability to generate separate interrupts greatly enhances the use of serial communications, since separate service routines are often required for transmitting and receiving.

Other differences between the Z8611, MC6801, and the 8051 occur in the hardware parity detector, the double-buffering of registers, framing error detectors and overrun conditions. The 8051 has a faster data rate than either the Z8611 or the MC6801. The MC6801 has the advantage of a hardware framing error detector and automatic detection of overrun conditions. The MC6801 also has both its transmit and receive registers double-buffered. The Z8611 has a hardware parity detector. For detection of framing errors and overrun conditions, a simple, low-overhead software check is available that uses only two instructions. See Z8600 Software Framing Error Detection Application Brief (document #617-1881-0004).

INSTRUCTION ARCHITECTURE

The architecture of the Z8611 is designed specifically for microcomputer applications. This fact is manifest in the instruction composition. The arduous task of programming the MC6801 and the 8051 starkly contrasts that of programming the Z8611.

Addressing Modes

The Z8611 and the 8051 both have six addressing modes: Register, Indirect Register, Indexed, Direct, Relative, and Immediate. The MC6801 has five addressing modes: Accumulator, Indexed, Direct, Relative, and Immediate. A quick comparison of these addressing modes reveals the versatility of the Z8611 and the 8051. The addressing modes of the MC6801 have several restrictions, as shown in Table 1. While the 8051 has all the addressing modes of the Z8611, its use of them is restricted. The Z8611 allows many more combina-

tions of addressing modes per instruction, because any of its registers can be used as an accumulator. For example, the instructions to clear, complement, rotate, and swap nibbles are all accumulator oriented in the 8051 and operate on the accumulator only. These same commands in the Z8611 can use any register and access it either directly, with register addressing, or with indirect register addressing.

Indexed Addressing. All three chips differ in their handling of indexing. The Z8611 can use any register for indexing. The 8051 can use only the accumulator as an Index register in conjunction with the data pointer or the Program Counter. The MC6801 has one 16-bit Index register. The address located in the second byte of an instruction is added to the lower byte of the Index register. The carry is added to the upper byte for the complete address. The MC6801 requires the index value to be an immediate value.

The MC6801 has only one 16-bit Index register and an immediate 8-bit value from the second byte of the instruction. Hence, the Indexed mode of the MC6801 is much more restrictive than that of the Z8611. The 8051 must use the accumulator as its only Index register, loading the accumulator with the register address each time a reference is made. Then, using indexing, the data is moved into the accumulator, eradicating the previous index. This forces a stream of data through the accumulator and requires a reload of the index before access can be made again. The Z8611 is clearly superior to both the MC6801 and the 8051 in the flexibility of its indexed addressing mode.

Short and Long Addressing. Short addressing helps to optimize memory space and execution speed. In sample applications of short register addressing, an eight percent decrease in the number of bytes used was recorded.

All three chips have short addressing modes, but the Z8611 has short addressing for both external memory and register memory. The 8051 has short addressing for the lowest 32 registers only.

The Z8611 has two different modes for register addressing. The full-byte address can be used to provide the address, or a 4-bit address can be used with the Register Pointer. To use the working registers, the Register Pointer is set for a particular bank of 16 registers, and then one of the 16 registers is addressed with four bits. Another feature for addressing external memory is the use of a 12-bit address in place of a full 16-bit address. To use the 12-bit address, one port supplies the eight multiplexed address/data lines and another port supplies four bits for the address. The remaining four bits of the second port can be used for I/O. This feature allows access to a maximum of 10K bytes of memory.

The 8051 uses short addresses by organizing its lowest 32 registers into four banks. The bank select is located in a 2-bit field in the PSW, with three bits addressing the register in the bank.

The MC6801 uses extended addressing for addressing external memory. With a special, nonmultiplexed expansion mode, 256 bytes of external memory can be accessed without the need for an external address latch. The MC6801 uses one 8-bit port for the address and another port for the data.

Stacks

The Z8611 and the MC6801 provide for external stacks, which require a 16-bit Stack Pointer. Internal stacks use only an 8-bit Stack Pointer. The 8051 uses only a limited internal stack requiring an 8-bit Stack Pointer. Using an external stack saves the internal RAM registers for general-purpose use.

Summary

The stack structure of the Z8611 and the MC6801 is better than that of the 8051. In most applications, the 8051 is more flexible and easier to program than the MC6801. The Z8611 is easier to use than either the 8051 or the MC6801 because of its register flexibility and its numerous combinations of addressing modes. The 8051 features a unique $4\mu n$ multiply and divide command. The MC6801 has a multiply, but it takes $10\mu s$ to perform it.

In summary, the Z8611 has the most flexible addressing modes, the most advanced indexing capabilities, and superior space- and time-saving abilities with respect to short addressing.

DEVELOPMENT SUPPORT

All three vendors provide development support for their products. This section discusses the different support features, including development chips, software, and modules.

Chips

Zilog offers an entire family of microcomputer chips for product development and final product. The Z8611 is a single-chip microcomputer with 4K bytes of mask-programmed ROM. For development, two other chips are offered. The Z8612 is a 64-pin, development version with full interface to external memory. The Z8613 is a prototype version that uses a functional, piggy-back, EPROM protopak. The Z8613 can use either a 4K EPROM (2732) or a 2K EPROM (2716). Zilog also offers a ROMless version in a 40-pin package that has all the features of the Z8611 except on-board ROM (Z8681).

Intel offers a similar line of development chips

with its 8051 family. The 8031 has no internal ROM and the 8751 has 4K of internal EPROM.

Motorola offers the MC6801, MC6803, MC6803NR, and MC68701. These are all similar except the MC68701 has 2K bytes of EPROM and the MC6801 has 2K bytes of ROM. The MC6803 has no internal ROM and the MC6803NR has neither ROM nor RAM on board.

The Z8613 and the MC68701 are both available now, but the 8751 is still unavailable (as of April 1981).

Software

Development software includes assemblers, and conversion programs. All manufacturers offer some or all of these features.

Since the MC6801 is compatible with the 6800, there is no need for a new assembler. The Z8611 and the 8051 both offer assemblers for their products. The Zilog PLZ/ASM assembler generates relocatable and absolute object code. PLZ/ASM also supports high-level control and data statements, such as IF... THEN... ELSE. Intel offers an absolute macroassembler, ASM51, with their product. They also offer a program for converting 8048 code to 8051 code.

Modules

The Z8611 development module has two 64-pin development versions of the 40-pin, ROM-masked Z8611. Intel offers the EM-51 emulation board, which contains a modified 8051 and PROM or EPROM in place of memory. Motorola has the MEX6801EVM evaluation board for program development. All three development boards are available now.

ADDITIONAL FEATURES

Additional features include Power Down mode, self-testing, and family-compatibility.

Power Down Mode

All three microcomputers offer a Power Down mode. The Z8611 and the 8051 save all of their registers with an auxiliary power supply. The MC6801 uses an auxiliary power supply to save only the first 64 bytes of its register file.

The Z8611 uses one of the crystal input pins for the external power supply to power the registers in Power Down mode. Since the XTAL2 input must be used, an external clock generator is necessary and is input via XTAL1. The 8051 and the MC6801 both have an input reserved for this function. The MC6801 uses the V_{CC} standby pin, and the 8051 uses the V_{pd} pin.

Family Compatibility

Another strength of the Z8611 is its expansion bus, which is completely compatible with the Zilog Z-BUS™. This means that all Z-BUS peripherals can be used directly with the Z8611.

The MC6801 is fully compatible with all MC6800 family products. The 8051 is software compatible with the older 8048 series and all others in that family.

BENCHMARKS

The following benchmark tests were used in this report to compare the Z8611, 8051, and MC6801:

- Generate CRC check for 16-bit word.
- Search for a character in a block of memory.
- Execute a computed GOTO - jump to one of eight locations depending on which of the eight bits is set.
- Shift a 16-word five places to the right.
- Move a 64-byte block of data from external memory to the register file.
- Toggle a single bit on a port.
- Measure the subroutine overhead time.

These programs were selected because of their importance in microcomputer applications. Algorithms that reflect a unique function or feature were excluded for the sake of comparison. Although programs can be optimized for a particular chip and for a particular attribute (code density or speed) these programs were not.

The figures cited in this text are taken directly from the vendor's documentation. Therefore, the cycles given below for the MC6801 and the 8051 are in machine cycles and the Z8611 figures are given in clock cycles. The Z8611 clock cycles should be divided by six to give the instruction time in microseconds. The 8051 and MC6801 machine cycle is 1 μs, and the Z8611 clock cycle is .166 μs at 12 MHz.

Because of the lack of availability of the MC6801 and the 8051, the benchmark programs listed here have not yet been run. When these products are readily available, the programs will be run and later editions of this document will reflect any changes in the findings.

Program Listings

CRC Generation

8051		Machine Cycles	Bytes
	MOV INDEX, #8	1	2
LOOP:	MOV A, DATA	1	2
	XRL A, HCHECK	1	2
	RLC A	1	1
	MOV A, LCHECK	1	2
	XRL A, LPOLY	1	2
	RLC A	1	1
	MOV LCHECK, A	1	2
	MOV A, HCHECK	1	2
	XRL A, HPOLY	1	2
	RLC A	1	1
	MOV HCHECK, A	1	2
	CLR C	1	1
	MOV A, DATA	1	2
	RLC A	1	1
	MOV DATA, A	1	2
	DJNZ INDEX, LOOP	2	3
	RET	2	1
N = 3+17X8 = 139 cycles			
@12 MHz = 139 μs			
Instructions = 18			
Bytes = 31			

MC6801		Machine Cycles	Bytes
	LDAA #\$08	2	2
LOOP:	STAA COUNT	3	2
	LDAA HCHECK	3	2
	EORA DATA	3	2
	ROLA	2	1
	LDAD POLY	4	2
	EORA HCHECK	3	2
	EORB LCHECK	3	2
	ROLB	2	1
	ROLA	2	1
	STAD LCHECK	4	2
	ASL DATA	6	3
	DEC COUNT	6	3
	BNE LOOP	4	2
	RTS	5	1
N = 45X8+7 = 367 cycles			
@4 MHz = 367 μs			
Instructions = 15			
Bytes = 28			

Z8611		Clock Cycles	Bytes
	LD INDEX, #8	6	2
LOOP:	LD R6, DATA	6	2
	XOR R6, HCHECK	6	2
	RLC R6	6	2
	XOR LCHECK, LPOLY	6	2
	RLC LCHECK	6	2
	XOR HCHECK, HPOLY	6	2
	RLC HCHECK	6	2
	RCF	6	1
	RLC DATA	6	2
	DJNZ INDEX, LOOP	12 or 10	2
	RET	14	1
N = 20+66X7+64 = 546 cycles			
@12 MHz = 91 μs			
Instructions = 12			
Bytes = 22			

Character Search Through Block of 40 Bytes

Shift 16-Bit Word to Right 5-Bits

8051	Machine	Cycles	Bytes
	MOV INDEX, #41	1	2
	MOV DPTR, #TABLE	2	3
LOOP1:	DJNZ INDEX, LOOP 2	2	2
	SJMP OUT	2	2
LOOP2:	MOV A, INDEX	1	2
	MOVC A, @A+DPTR	2	1
	CJNE A, CHARAC, LOOP1	2	3
OUT:			
	N = 3+39X7+4 = 280 cycles		
	@12 MHz = 280μs		
	Instructions = 7		
	Bytes = 15		

8051	Machine	Cycles	Bytes
	MOV INDEX #5	1	2
LOOP:	CLR C	1	1
	MOV A, WORD + 1	1	2
	RRC A	1	1
	MOV WORD + 1, A	1	2
	MOV A, WORK	1	2
	RRC A	1	1
	MOV WORD, A	1	2
	DJNZ INDEX, LOOP	2	2
	N = 1+9X5 = 46 Cycles		
	@12 MHz = 46μs		
	Instructions = 9		
	Bytes = 15		

MC6801	Machine	Cycles	Bytes
	LDAB #\$40	2	2
	LDA A #CHARAC	2	2
	LDX #TABLE	3	3
LOOP:	CMPA \$0, X	4	2
	BEQ OUT	4	2
	INX	3	1
	DECB	2	1
	BNE LOOP	4	2
OUT:	-		
	-		
	-		
	N = 7+40X17 = 687 cycles		
	@4 MHz = 687μs		
	Instructions = 8		
	Bytes = 15		

MC6801	Machine	Cycles	Bytes
	LDX #5	6	3
	LDAD WORK	4	2
LOOP:	LSRD	3	1
	DEX	3	1
	BNE LOOP	4	2
	STAD WORD	4	2
	N = 10X5+11 = 61 Cycles		
	@4 MHz = 61μs		
	Instructions = 6		
	Bytes = 11		

Z8611	Clock	Cycles	Bytes
	LD INDEX, #40	6	2
LOOP:	LD DATA, TABLE (INDEX)	10	3
	CP DATA, CHARAC	6	2
	JR Z, OUT	12 or 10	2
	DJNZ INDEX, LOOP	12 or 10	2
OUT:	-		
	-		
	N = 6+38X40 = 1524 cycles		
	@12 MHz = 254μs		
	Instructions = 5		
	Bytes = 11		

Z8611	Clock	Cycles	Bytes
	LD INDEX, #5	6	2
LOOP:	CCF	6	1
	RRC WORD + 1	6	2
	RRC WORD	6	2
	DJNZ INDEX, LOOP	12 or 10	2
	N = 6+4X30+28 = 154 Cycles		
	@12 MHz = 26μs		
	Instructions = 5		
	Bytes = 9		

Computed GOTO

Move 64-Byte Block

8051		Machine	
		Cycles	Bytes
	MOV INDEX, #40	1	2
LOOP:	MOV A, DATA	1	2
	RLC A	1	1
	JC OUT	2	2
	MOV A, INDEX	1	1
	ADD A, #3	1	2
	MOV INDEX, A	1	1
	SJMP LOOP	2	2
OUT:	MOV DPTR, #TABLE	2	3
	MOV A, INDEX	1	1
	JMP @A+DPTR	2	1
TABLE:	LCALL ADDR1		3
	-		
	-		
	LCALL ADDR2	2	
	N = 1+9X7+11 = 75 Cycles		
	@12 MHz = 75µs		
	Instructions = 12		
	Bytes = 21		

MC6801		Machine	
		Cycles	Bytes
	LDAB #2	2	2
	LDX TABLE	3	3
LOOP:	RORA	2	1
	BCS OUT	4	2
	ABX	3	1
	JMP LOOP	3	2
OUT:	LDX 0, X	5	3
	JMP 0, X	4	3
	N = 8X12+14 = 110 Cycles		
	@4 MHz = 110µs		
	Instructions = 8		
	Bytes = 17		

Z8611		Clock	
		Cycles	Bytes
	CLR INDEX	6	2
LOOP:	INC INDEX	6	1
	RLC DATA	6	2
	JR NC, LOOP	12 or 10	2
	LD ADDR, TABLE 1, (INDEX)	10	3
	LD ADDR+1, TABLE 2, (INDEX)	10	3
	JP @ADDR	12	2
	N = 6+24X7+54 = 228 Cycles		
	@12 MHz = 38µs		
	Instructions = 7		
	Bytes = 15		

8051		Machine	
		Cycles	Bytes
	MOV INDEX, #COUNT	1	2
LOOP:	MOV DPTR, #ADDR1	2	3
	MOVX A, @DPTR	2	1
	INC #ADDR1	1	1
	MOV @ADDR2,A	1	1
	INC ADDR2	1	1
	DJNZ INDEX, LOOP	2	1
	N = 1+9X64 = 577 Cycles		
	@12 MHz = 577µs		
	Instructions = 7		
	Bytes = 10		

MC6801		Machine	
		Cycles	Bytes
	LDAB #COUNT	2	2
LOOP:	LDX ADDR1	4	3
	LDAA 0, X	4	2
	INX	3	1
	STAA ADDR1	4	2
	LDX ADDR2	4	3
	STAA 0, X	4	2
	INX	3	1
	STX ADDR2	4	2
	DECB	2	1
	BNE LOOP	4	2
	N = 64X36+2 = 2306 Cycles		
	@4 MHz = 2306µs		
	Instructions = 11		
	Bytes = 21		

Z8611		Clock	
		Cycles	Bytes
	LD INDEX, #COUNT	6	2
LOOP:	LDEI @ADDR2, @ADDR1	18	2
	DJNZ INDEX, LOOP	12 or 10	2
	N = 6+63X30+28 = 1924 Cycles		
	@12 MHz = 321µs		
	Instructions = 3		
	Bytes = 6		

Toggle a Port Bit

Subroutine Call/Return Overhead

8051

	Machine Cycles	Bytes
XRL PO, #YY	2	3
N = 2 Cycles		
@12 MHz = 2 μs		
Instructions = 1		
Bytes = 3		

MC6801

	Machine Cycles	Bytes
LDA PORTO	3	2
EORA #YY	2	2
STAA PORTO	3	2
N = 8 Cycles		
@4 MHz = 8 μs		
Instructions = 3		
Bytes = 6		

Z8611

	Clock Cycles	Bytes
XOR PORTO, #YY	10	2
N = 10 Cycles		
@12 MHz = 1.7 μs		
Instructions = 1		
Byte = 2		

8051

	Machine Cycles	Bytes
LCALL SUBR	2	3
-		
-		
SUBR: -		
-		
-		
RET	2	1
N = 4 Cycles		
@12 MHz = 4 μs		
Instructions = 2		
Bytes = 4		

MC6801

	Machine Cycles	Bytes
JSR SUBR	9	2
-		
-		
SUBR: -		
-		
-		
RTS	5	1
N = 14 Cycles		
@4 MHz = 14 μs		
Instructions = 2		
Bytes = 3		

Z8611

	Clock Cycles	Bytes
CALL @SUBR	20	2
-		
-		
SUBR: -		
-		
-		
RET	14	1
N = 34 Cycles		
@12 MHz = 5.7 μs		
Instructions = 2		
Bytes = 3		

Results

Table 2 summarizes the results of this comparison. The relative performance column lists the speeds of the MC6801 and 8051 divided by the Z8611 speeds (12 MHz). The overall performance averages the separate relative performances. The higher the number, the faster the Z8611 as compared to the MC6801 and the 8051.

The relative performance figures show that the Z8611 runs 50 percent faster than the 8051 and 250 percent faster than the MC6801. Although speed is not necessarily the most important criterion for selecting a particular product, the Z8611 proves to be an undeniably superior product when speed is added to the advantages of programming ease, code density, and flexibility.

Table 2. Benchmark Program Results

Benchmark Test	MC6801 (4 MHz) cycles time		8051 (12 MHz) cycles time		Z8 (8 MHz) cycles time		Z8 (12 MHz) cycles time		Relative Performance	
	MC6801	8051	Z8	Z8	MC6801	8051				
CRC Generation	367	139	546	137	546	91	4.03	1.53		
Character Search	687	280	1524	382	1524	254	2.70	1.10		
Computed GOTO	110	75	228	57	228	38	2.89	1.97		
Shift Right 5 Bits	61	46	154	38	154	26	2.35	1.78		
Move 64-byte block	2306	577	1924	481	1924	321	7.18	1.80		
Subroutine Overhead	14	4	34	8.5	34	5.7	2.46	0.70		
Toggle a Port Bit	8	2	10	2.5	10	1.7	4.71	1.18		
			Overall Performance				3.76	1.44		

Note: All times are given in microseconds.

Table 3. Byte/Instruction/Time Comparison

	Bytes				Instructions				Time (microseconds)		
	MC6801	8051	Z8611		MC6801	8051	Z8611		MC6801	8051	Z8611
	CRC Generation	28	31		22	15	18		12	367	139
Character Search	15	15	11	8	7	5	687	280	254		
Shift Right 5 Bits	11	15	9	6	9	5	61	46	26		
Computed GOTO	17	21	15	8	12	7	110	75	38		
Move Block	21	10	6	11	7	3	2306	577	321		
Toggle Port Bit	6	3	2	3	1	1	8	2	1.7		
Subroutine Call	3	4	3	2	2	2	14	4	5.7		

SUMMARY

The hardware of the three chips compared is very similar. The Z8611, however, has several advantages, the most important of which is its interrupt structure. It is more advanced than the interrupt structures of both the 8051 and the MC6801. Other advantages of the Z8611 over either the MC6801 or the 8051 include I/O facilities with parity detection and hardware handshake and a larger amount of internal ROM (the MC6801 has only 2K bytes).

Substantial differences are apparent with regard to software architecture. The addressing modes of

the Z8611 are more flexible than those of either the MC6801 or the 8051. The Z8611 can use byte-saving addressing with working registers, and it has short external addresses for saving I/O lines. It can also provide for an external stack. The register architecture (as opposed to the accumulator architecture) of the Z8611 saves execution time and enhances programming speed by reducing the byte count.

The Z8611 microcomputer stands out as the most powerful chip of the three, and concurrently, it is the easiest to program and configure.

Z8600 Interrupt Request Register



Application Brief

October 1980

The Interrupt Request Register (IRQ, R250) stores requests from the six possible interrupt sources (IRQ⁰-IRQ⁵) in the Z8600 series microcomputer. In addition to other functions, a hardware reset to the Z8600 disables the IRQ register and resets its request bits. Before the IRQ will register requests, it must first be enabled by executing an Enable Interrupts (EI) instruction. Setting the Enable Interrupt bit in the Interrupt Mask Register (IMR, R251) is not an equivalent operation for this purpose; to enable the IRQ, an EI instruction is required. The function of this EI instruction is distinct from its task of globally enabling the interrupt system. Even in a polled system where IRQ bits are tested in software, it is necessary to execute the EI.

The designer must ensure that unexpected and undesirable interrupt requests will not occur after the EI is executed. One method of doing this is to reset all interrupt enable bits in the IMR for levels that are possible interrupt sources; the EI instruction may then be safely executed. Once EI is executed, the program may immediately execute a Disable Interrupts (DI) instruction. The code necessary to perform these operations is as follows:

```
RESET: LD  IMR, #%XX !SET INTERRUPT MASK!  
      EI                      !ENABLE GLOBAL INTER-  
                               RUPT, ENABLE IRQ!
```

where XX has a 0 in each bit position corresponding to the interrupt level to be disabled. If all IMR bits are to be reset, a CLR IMR instruction may be used.

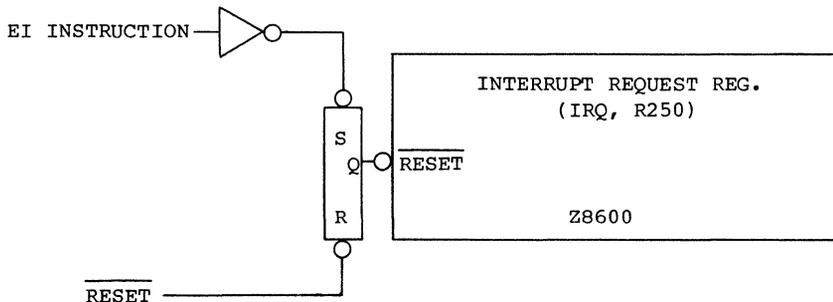


Figure 1 - IRQ Reset Functional Logic Diagram

12-Bit Addressing with the Z8 Family



Application Brief

January 1981

12-BIT ADDRESSING WITH THE Z8600 SERIES FAMILY

The Z8601 can manipulate data in four memory spaces: internal program memory, internal register file, external program memory, and external data memory. The internal register file is not discussed in this paper. Port 3 may be configured optionally to provide a Data Memory (\overline{DM}) strobe that is used to select program and data memory. The Z8601 generates another signal, Data Strobe (\overline{DS}), that signals an external memory operation. \overline{DS}

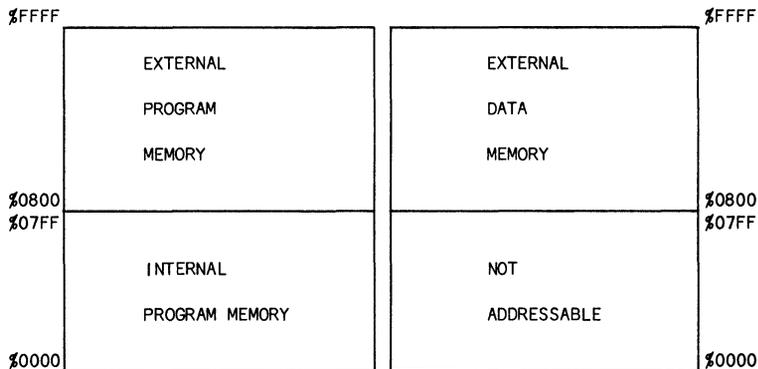
is generated each time an address greater than 2047 is used.

The Z8601 has 2K bytes of on-chip program memory. The user cannot directly access external memory in the address range of 0 to 2K since this address range is decoded as an internal address. The Z8600 accesses external memory in the following manner:

Table 1. Port 0 Configured to Output A_8-A_{15}

USER ADDRESS	PHYSICAL MEMORY		LOCATION	\overline{DS}	ADDRESSES ON PORTS 0 & 1
	DATA	PROGRAM			
%0000-%07FF	NONE	%0000-%07FF	INTERNAL	INACTIVE	0000-07FF
%0800-%FFFF	%0800-%FFFF	%0800-%FFFF	EXTERNAL	ACTIVE	0800-FFFF

NOTE: The external physical addresses %0000-%07FF cannot be accessed.



With Port 0 giving the high byte of address and Port 1 giving the low byte of address, a total of 126K bytes of memory can be accessed: 2K bytes of on-chip ROM, 62K bytes of external data memory, and 62K bytes of external program memory.

This scheme does not provide access to the external memory in the address range of 0 to 2K. To access memory in the 0 to 2K range of external memory, the upper address nibble of Port 0 is truncated and address locations 4K to 6K are mapped into the 0 to 2K external memory range as follows:

Table 2. Port 0 Configured to Output A₈-A₁₁

USER ADDRESS	PHYSICAL MEMORY		LOCATION	DS	ADDRESSES ON PORTS 0 & 1
	DATA	PROGRAM			
0000-07FF	NONE	0000-07FF	INTERNAL	INACTIVE	0000-07FF
0800-0FFF	0800-0FFF	0800-0FFF	EXTERNAL	ACTIVE	0800-0FFF
1000-17FF	0000-07FF	0000-07FF	EXTERNAL	ACTIVE	0000-07FF

Using the above configuration, memory is accessible in the address range of 0 to 6K. Higher addresses are indistinguishable from the 0 to 6K address space, because the upper four address bits have not been programmed to appear on Port 0.

The Z8600 can access up to 10K of memory using only 12 address lines. It can access 2K of program memory on-chip, 4K of external data memory, and 4K of external program memory for a total of 10K. With only 12 address lines, four lines are released in Port 0 for I/O.

To configure Port 3 to provide the Data Memory (DM) signal the following command is used:

```
LD P3M,#(2)XXX10XXX
```

The following instruction specifies Port 0 as address lines A₈-A₁₁ and Port 1 as address/data multiplexed lines AD₀-AD₇.

```
LD P01M,#(2)0XX10X1X
```

The above Xs do not represent "don't care" states. These bits must be set or reset depending on the particular configuration in which the Z8600 is set.

For medium-sized memory applications, the Z8600 can be configured to output address lines A₈-A₁₁ on Port 0, address/data multiplexed lines AD₀-AD₇ on Port 1, and DM on Port 3. In addition, the Z8600 can access a total of 10K bytes of memory.

Z8 Family Software Framing Error Detection



Application Brief

October 1980

INTRODUCTION The Zilog Z8600 UART microcomputer is a high-performance, single-chip device that incorporates on-chip ROM, RAM, parallel I/O, serial I/O, and a baud rate generator. The UART is capable of full-duplex, asynchronous serial communication at nine standard software-selectable baud rates from 110 to 19.2K baud; other nonstandard rates can also be obtained under software control. Odd parity generation and checking can also be selected.

Three possible error conditions can occur during reception of serial data: framing error, parity error, and overrun error. A framing error condition occurs when a stop bit is not received at the proper time (Figure 1). This can result from noise in the data channel, causing erroneous detection of the previous start bit or lack of detection of a properly transmitted stop bit. The Z8600 UART does not incorporate hardware framing error detection but does facilitate a simple, low-overhead software detection method.

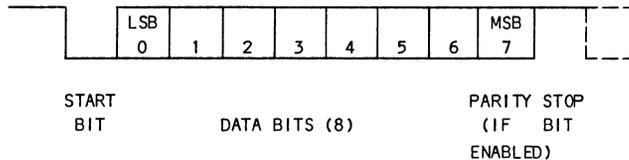


Fig. 1 - Asynchronous Data Format

METHOD In the middle of the stop bit time, the Z8600 UART automatically posts a serial input interrupt request on IRQ_3 . The serial input can also be tested by reading Port 3 bit 0 ($P3_0$) as shown in Figure 2. Thus, within the interrupt service routine or polling loop, it is only necessary to test $P3_0$ in order to identify a framing error. If $P3_0$ is Low when IRQ_3 goes High, a framing error con-

dition exists and the following code is used to test this:

```
TM P3, #%01 ! TEST FOR P30 = 1 !  
JR Z, FERR ! ELSE FRAMING ERROR !
```

The execution time of this framing error test is only 5.5 μ s at 8 MHz. In the worst case (19.2K baud), this would result in 1% overhead. Only five program bytes are required.

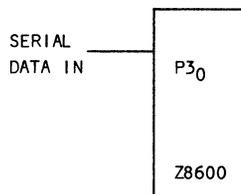


Fig. 2 - Z8600 Serial Input Connection

CONCLUSION While the Z8600 UART does not incorporate maximum penalty of 1% at 19.2K baud using no
hardware framing error detection, this additional hardware and only five bytes of
feature can be implemented in software with a program memory.



Application Note

Doll Freund

October 1980

SECTION 1

Introduction

The Z8 is the first microcomputer to offer both a highly integrated microcomputer on a single chip and a fully expandable microprocessor for I/O-and memory-intensive applications. The Z8 has two timer/counters, a UART, 2K bytes internal ROM, and a 144-byte internal register file including 124 bytes of RAM, 32 bits of I/O, and 16 control and status registers. In addition, the Z8 can address up to 124K bytes of external program and data memory, which can provide full, memory-mapped I/O capability.

This application note describes the important features of the Z8, with software examples that illustrate its power and ease of use. It is divided into sections by topic; the reader need not read each section sequentially, but may skip around to the sections of current interest.

It is assumed that the reader is familiar with the Z8 and its assembly language, as described in the following documents:

- *Z8 Technical Manual* (03-3047-02)
- *Z8 PLZ/ASM Assembly Language Programming Manual* (03-3023-02)

SECTION 2

Accessing Register Memory

The Z8 register space consists of four I/O ports, 16 control and status registers, and 124 general-purpose registers. The general-purpose registers are RAM areas typically used for accumulators, pointers, and stack area. This section describes these registers and how they are used. Bit manipulation and stack operations affecting the register space are discussed in Sections 4 and 5, respectively.

2.1 Registers and Register Pairs. The Z8 supports 8-bit registers and 16-bit register pairs. A register pair consists of an even-numbered register concatenated with the next higher numbered register (%00 and %01, %02 and %03, ... %7E and %7F, %F0 and %F1, ... %FE and %FF). A register pair must be addressed by reference to the even-numbered register. For example,

%F1 and %F2 is not a valid register pair;
%F0 and %F1 is a valid register pair,
addressed by reference to %F0.

Register pairs may be incremented (INCW) and decremented (DECW) and are useful as pointers for accessing program and external data memory. Section 3 discusses the use of register pairs for this purpose.

Any instruction which can reference or modify an 8-bit register can do so to any of the 144 registers in the Z8, regardless of the inherent nature of that register. Thus, I/O ports, control, status, and general-purpose registers may all be accessed and manipulated without the need for special-purpose instructions. Similarly, instructions which reference or modify a 16-bit register pair can do so to any of the valid 72 register pairs. The only exceptions to this rule are:

- The DJNZ (decrement and jump if non-zero) instruction may successfully operate on the general-purpose RAM registers (%04-%7F) only.
- Six control registers are write-only registers and therefore, may be modified only by such instructions as LOAD, POP, and CLEAR. Instructions such as OR and AND require that the current contents of the operand be readable and therefore will not function properly on the write-only registers. These registers are the following: *the timer/counter prescaler registers PRE0 and PRE1, the port mode registers P01M, P2M, and P3M, the interrupt priority register IPR.*

2. Accessing Register Memory

(Continued)

2.2 Register Pointer. Within the register addressing modes provided by the Z8, a register may be specified by its full 8-bit address (0-%7F, %F0-%FF) or by a short 4-bit address. In the latter case, the register is viewed as one of 16 working registers within a working register group. Such a group must be aligned on a 16-byte boundary and is addressed by Register Pointer RP (%FD). As an example, assume the Register Pointer contains %70, thus pointing to the working register group from %70 to %7F. The LD instruction may be used to initialize register %76 to an immediate value in one of two ways:

LD %76,#1 !8-bit register address is given by instruction (3 byte instruction)!

or

LD R6,#1 !4-bit working register address is given by instruction; 4-bit working register group address is given by Register Pointer (2 byte instruction)!

The address calculation for the latter case is illustrated in Figure 1. Notice that 4-bit working-register addressing offers code compactness and fast execution compared to its 8-bit counterpart.

To modify the contents of the Register Pointer, the Z8 provides the instruction

```
SRP #value
```

Execution of this instruction will load the upper four bits of the Register Pointer; the lower four bits are always set to zero. Although a load instruction such as

```
LD RP,#value
```

could be used to perform the same function, SRP provides execution speed (six vs. ten cycles) and code space (two vs. three bytes) advantages over the LD instruction. The instruction

```
SRP #%70
```

is used to set the Register Pointer for the above example.

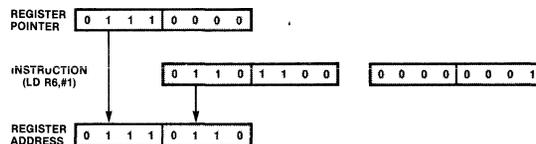


Figure 1. Address Calculation Using the Register Pointer

2.3 Context Switching. A typical function performed during an interrupt service routine is context switching. Context switching refers to the saving and subsequent restoring of the program counter, status, and registers of the interrupted task. During an interrupt machine cycle, the Z8 automatically saves the Program Counter and status flags on the stack. It is the responsibility of the interrupt service routine to preserve the register space. The recommended means to this end is to allocate a specific portion of the register file for use by the service routine. The service routine thus preserves the register space of the interrupted task by avoiding modification of registers not allocated as its own. The most efficient scheme with which to implement this function in the Z8 is to allocate a working register group (or portion thereof) to the interrupt service routine. In this way, the preservation of the interrupted task's registers is solely a matter of saving the Register Pointer on entry to the service routine, setting the Register Pointer to its own working register group, and restoring the Register Pointer prior to exiting the service routine. For example,

assume such a register allocation scheme has been implemented in which the interrupt service routine for IRQ0 may access only working register Group 4 (registers %40-%4F). The service routine for IRQ0 should be headed by the code sequence:

```
PUSH RP !preserve Register Pointer of
interrupted task!
SRP #%40 !address working register
group 4!
```

Before exiting, the service routine should execute the instruction

```
POP RP
```

to restore the Register Pointer to its entry value.

It should be noted that the technique described above need not be restricted to interrupt service routines. Such a technique might prove efficient for use by a subroutine requiring intermediate registers to produce its outputs. In this way, the calling task can assume that its environment is intact upon return from the subroutine.

2. Accessing Register Memory

(Continued)

2.4 Addressing Mode. The Z8 provides three addressing modes for accessing the register space: Direct Register, Indirect Register, and Indexed.

2.4.1 Direct Register Addressing. This addressing mode is used when the target register address is known at assembly time. Both long (8-bit) register addressing and short (4-bit) working register addressing are supported in this mode. Most instructions supporting this mode provide access to single 8-bit registers. For example:

```
LD    %FE,#HI STACK
      !load register %FE (SPH) with
      !the upper 8-bits of the label
      !STACK!
AND  0,MASK_REG
      !AND register 0 with register
      !named MASK_REG!
OR   1,R5
      !OR register 1 with working
      !register 5!
```

Increment word (INCW) and decrement word (DECW) are the only two Z8 instructions which access 16-bit operands. These instructions are illustrated below for the direct register addressing mode.

```
INCW RR0
      !increment working register
      !pair R0, R1:
      !R1 ← R1 + 1
      !R0 ← R0 + carry!
DECW %7E
      !decrement working register
      !pair %7E, %7F:
      !%7F ← %7F - 1
      !%7E ← %7E - carry!
```

Note that the instruction

```
INCW RR5
```

will be flagged as an error by the assembler (RR5 not even-numbered).

2.4.2 Indirect Register Addressing. In this addressing mode, the operand is pointed to by the register whose 8-bit register address or 4-bit working register address is given by the instruction. This mode is used when the target register address is not known at assembly time and must be calculated during program execution. For example, assume registers %60-%7F contain a buffer for output to the serial line via repetitive calls to procedure SERIAL_OUT. SERIAL_OUT expects working register 0 to hold the output character. The following instructions illustrate the use of the indirect addressing mode to accomplish this task:

```
LD    R1,#%20
      !working register 1 is the byte
      !counter: output %20 bytes!
```

```
LD    R2,#%60
      !working register 2 is the buf-
      !fer pointer register!
out_again:
LD    R0,@R2
      !load into working register 0
      !the byte pointed to by working
      !register 2!
INC   R2
      !increment pointer!
CALL  SERIAL_OUT
      !output the byte!
DJNZ  R1,out_again
      !loop till done!
```

Indirect addressing may also be used for accessing a 16-bit register pair via the INCW and DECW instructions. For example,

```
INCW @R0
      !increment the register pair
      !whose address is contained in
      !working register 0!
DECW @%7F
      !decrement the register pair
      !whose address is contained in
      !register %7F!
```

The contents of registers R0 and %7F should be even numbers for proper access; when referencing a register pair, the least significant address bit is forced to the appropriate value by the Z8. However, the register used to point to the register pair need not be an even-numbered register.

Since the indirect addressing mode permits calculation of a target address prior to the desired register access, this mode may be used to simulate other, more complex addressing modes. For example, the instruction

```
SUB  4,BASE(R5)
```

requires the indexed addressing mode which is not directly supported by the Z8 SUBtract instruction. This instruction can be simulated as follows:

```
LD    R6,#BASE
      !working register 6 has the
      !base address!
ADD   R6,R5
      !calculate the target address!
SUB   4,@R6
      !now use indirect addressing to
      !perform the actual subtract!
```

Any available register or working register may be used in place of R6 in the above example.

2.4.3 Indexed Addressing. The indexed addressing mode is supported by the load instruction (LD) for the transference of bytes between a working register and another register. The effective address of the latter register is given by the instruction which is offset by the contents of a designated working (index)

2. Accessing Register Memory
(Continued)

register. This addressing mode provides efficient memory usage when addressing consecutive bytes in a block of register memory, such as a table or a buffer. The working register used as the index in the effective address calculation can serve the additional role of counter for control of a loop's duration.

For example, assume an ASCII character buffer exists in register memory starting at address BUF for LENGTH bytes. In order to determine the logical length of the character string, the buffer should be scanned backward until the first nonoccurrence of a blank character. The following code sequence may be used to accomplish this task:

```

LD    R0,#LENGTH
      !length of buffer!
      !starting at buffer end, look for
      !1st non-blank!

loop:
LD    R1,BUF-1(R0)
CP    R1,#' '
JR    ne,found
      !found non-blank!
DJNZ R0,loop
      !look at next!

all_blanks: !length = 0!
found:
5 instructions
12 bytes
1.5 μs overhead
10.5 μs (average) per character tested

```

At labels "all_blanks" and "found," R0 contains the length of the character string. These labels may refer to the same location, but they are shown separately for an application where special processing is required for a string of zero length. To perform this task without indexed addressing would require a code sequence such as:

```

LD    R1,#BUF + LENGTH - 1
LD    R0,#LENGTH
      !starting at buffer end, look for
      !1st non-blank!

loop1:
CP    @R1,#' '
JR    ne,found1
      !found non-blank!
DEC   R1    !dec pointer!
DJNZ R0,loop1
      !are we done?!

all_blanks1: !length = 0!
found1:
6 instructions
13 bytes
3 μs overhead
9.5 μs (average) per character tested

```

The latter method requires one more byte of program memory than the former, but is faster by four execution cycles (1 μs) per character tested.

As an alternate example, assume a buffer exists as described above, but it is desired to scan this buffer forward for the first occurrence of an ASCII carriage return. The following illustrates the code to do this:

```

LD    R0,# - LENGTH
      !starting at buffer start, look for
      !1st carriage return (= %0D)!

next:
LD    r1,BUF + LENGTH(R0)
CP    R1,#%0D
JR    eq,cr !found it!
INC   R0    !update counter/index!
JR    nz,next
      !try again!

cr:
ADD   R0,#LENGTH
      !R0 has length to CR!

7 instructions
16 bytes
1.5 μs overhead
12 μs (average) per character tested

```

SECTION 3 Accessing Program and External Data Memory

In a single instruction, the Z8 can transfer a byte between register memory and either program or external data memory. Load Constant (LDC) and Load Constant and Increment (LDCI) reference program memory; Load External (LDE) and Load External and Increment (LDEI) reference external data memory. These instructions require that a working register pair contain the address of the byte in either program or external data memory to be accessed by the instruction (indirect working register pair addressing mode). The register byte operand is specified by using the direct working register addressing mode in LDC and

LDE or the indirect working register addressing mode in LDCI and LDEI. In addition to performing the designated byte transfer, LDCI and LDEI automatically increment both the indirect registers specified by the instruction. These instructions are therefore efficient for performing block moves between register and either program or external data memory. Since the indirect addressing mode is used to specify the operand address within program or external data memory, more complex addressing modes may be simulated as discussed earlier in Section 2.4.2. For example, the instruction

```
LDC R3,BASE(R2)
```

requires the indexed addressing mode, where

3. Accessing Program and External Data Memory

(Continued)

BASE is the base address of a table in program memory and R2 contains the offset from table start to the desired table entry. The following code sequence simulates this instruction with the use of two additional registers (R0 and R1 in this example).

```
LD R0,#HI BASE
LD R1,#LO BASE
      !RRO has table start address!
ADD R1,R2
ADC R0,#0
      !RRO has table entry address!
LDC R3,@RRO
      !R3 has the table entry!
```

3.1 Configuring the Z8 for I/O Applications vs. Memory Intensive Applications.

The Z8 offers a high degree of flexibility in memory and I/O intensive applications. Thirty-two port bits are provided of which 16, 12, eight, or zero may be configured as address bits to external memory. This allows for addressing of 62K, 4K or 256 bytes of external memory, which can be expanded to 124K, 8K, or 512 bytes if the Data Memory Select output (DM) is used to distinguish between program and data memory accesses. The following instructions illustrate the code sequence required to configure the Z8 with 12 external addressing lines and to enable the Data Memory Select output.

```
LD P01M,#%(2)00010010
      !bit 3-4: enable AD0-AD7;
      !bit 0-1: enable A8-A11!
LD P3M,#%(2)00001000
      !bit 3-4: enable  $\overline{DM}$ !
```

The two bytes following the mode selection of ports 0 and 1 should not reference external memory due to pipelining of instructions within the Z8. Note that the load instruction to P3M satisfies this requirement (providing that it resides within the internal 2K bytes of memory).

3.2 LDC and LDE. To illustrate the use of the Load Constant (LDC) and Load External (LDE) instructions, assume there exists a hardware configuration with external memory and Data Memory Select enabled. The following module illustrates a program for tokenizing an ASCII input buffer. The program assumes there is a list of delimiters (space, comma, tab, etc.) in program memory at address DELIM for COUNT bytes (accessed via LDC) and that an ASCII input buffer exists in external data memory (accessed via LDE). The program scans the input buffer from the current location and returns the start address of the next token (i.e. the address of the first nondelimiter found) and the length of that token (number of characters from token start to next delimiter).

```
Z8ASM      2.0
LOC      OBJ CODE      STMT SOURCE STATEMENT

          1 SCAN      MODULE
          2 CONSTANT
          3 COUNT :=      6
          4 GLOBAL
          5 $SECTION PROGRAM
          6 DELIM ARRAY [COUNT BYTE] :=

P 0000 20 3B 2C
P 0003 2E 0A 0D

          7          [' ', ';', ',', '.', '%0A', '%0D']
          8
P 0006
          9 scan      PROCEDURE
         10 !*****
         11 Purpose =      To find the next token within an
         12          ASCII buffer.
         13
         14 Input =      RR0 = address of current location
         15          within input buffer in external
         16          memory.
         17
         18 Output =      RR4 = address of start of next token
         19          RR0 = address of new token's ending
         20          delimiter
         21          R2 = length of token
         22          R3 = ending delimiter
         23          R6,R7,R8,R9 destroyed
         24
         25 *****!
         26 ENTRY
P 0006 B0 E2
         27      clr      R2          !init. length counter!
         28      DO
P 0008 82 30
         29      LDE      R3,@RR0 !get byte from input buffer!
P 000A A0 E0
         30      incw     RR0 !increment pointer!
P 000C D6 002E'
         31      call    check !look for non-delimiter!
P 000F FD 0015'
         32      IF C THEN
P 0012 8D 0018'
         33      EXIT      !found token start!
         34      FI
P 0015 8D 0008'
         35      OD
```

3. Accessing Program and External Data Memory
(Continued)

```

36
37      ld      R4,RO
38      ld      R5,R1      !RR4 = token starting addr!
39      DO
40      inc     R2          !inc. length counter!
41      LDE    R3,@RRO    !get next input byte!
42      call   check      !look for delimiter!
43      IF NC THEN
44      EXIT              !found token end!
45      FI
46      incw   RRO        !point to next byte!
47      OD
48
49      ret
50 END      scan
51
52 P 002E      check  PROCEDURE
53 !*****!
54 Purpose =   compare current character with
55             delimiter table until table
56             end or match found
57
58 input =     DELIM = start address of table
59             COUNT = length of that table
60             R3 = byte to be scrutinized
61
62 output =    Carry flag = 1 => input byte
63             is not a delimiter (no match found)
64
65             Carry flag = 0 => input byte
66             is a delimiter (match found)
67             R6,R7,R8,R9 destroyed
68
69 *****!
70 ENTRY
71 P 002E 6C 00* 71      ld      R6,#HI DELIM
72 P 0030 7C 00* 72      ld      R7,#LO DELIM      !RR6 points to
73                                     delimiter list!
74 P 0032 8C 06 74      ld      R8,#COUNT      !R8 = length of list!
75 here:
76 P 0034 C2 96 76      LDC    R9,@RR6      !get table entry!
77 P 0036 A0 E6 77      incw   RR6          !point to next entry!
78 P 0038 A2 93 78      cp     R9,R3        !R3 = delimiter?!
79 P 003A 6B 03 79      jr     eq,bye       !yes. carry = 0!
80 P 003C 8A F6 80      djnz  R8,here      !next entry!
81 P 003E DF      scf          !table done. R3
82                                     not a delimiter!
83 bye:
84 P 003F AF      ret
85 P 0040      END      check
86 END      SCAN

```

0 ERRORS
ASSEMBLY COMPLETE

27 instructions
58 bytes

Execution time is a function of the number of leading delimiters before token start (x) and the number of characters in the token (y): 123 μs overhead + 59x μs + 102y μs (average) per token

3.3 LDCI. A common function performed in Z8 applications is the initialization of the register space. The most obvious approach to this function is the coding of a sequence of "load register with immediate value" instructions (each occupying three program bytes for a

register or two program bytes for a working register). This approach is also the most efficient technique for initializing less than eight consecutive registers or 14 consecutive working registers. For a larger register block, the

3. Accessing Program and External Data Memory

(Continued)

LDCI instruction provides an economical means of initializing consecutive registers from an initialization table in program memory. The following code excerpt illustrates this technique of initializing control registers %F2 through %FF from a 14-byte array (INIT__tab) in program memory:

```
SRP  #%00          !RP not %F0!
LD   R6,#HI INIT__tab
LD   R7,#LO INIT__tab
LD   R8,#%F2
      !1st reg to be initialized!
LD   R9,#14
      !length of register block!
loop:
LDCI @R8,@RR6
      !load a register from the
      !init table!
DJNZ R9,loop
      !continue till done!

7 instructions
14 bytes
7.5 μs overhead
7.5 μs per register initialized
```

3.4 LDEI. The LDEI instruction is useful for moving blocks of data between external and register memory since auto-increment is performed on both indirect registers designated by the instruction. The following code excerpt illustrates a register buffer being saved at address %40 through %60 into external memory at address SAVE:

```
LD   R10,#HI SAVE
      !external memory!
LD   R11,#LO SAVE
      !address!
LD   R8,#%40
      !starting register!
LD   R9,#%21
      !number of registers to save in
      !external data memory!
loop:
LDEI @RR10,@R8
      !init a register!
DJNZ R9,loop
      !until done!

6 instructions
12 bytes
6 μs overhead
7.5 μs per register saved
```

SECTION 4

Bit Manipulations

Support of the test and modification of an individual bit or group of bits is required by most software applications suited to the Z8 microcomputer. Initializing and modifying the Z8 control registers, polling interrupt requests, manipulating port bits for control of or communication with attached devices, and manipulation of software flags for internal control purposes are all examples of the heavy use of bit manipulation functions. These examples illustrate the need for such functions in all areas of the Z8 register space. These functions are supported in the Z8 primarily by six instructions:

- Test under Mask (TM)
- Test Complement under Mask (TCM)
- AND
- OR
- XOR
- Complement (COM)

These instructions may access any Z8 register, regardless of its inherent type (control, I/O, or general purpose), with the exception of the six write-only control registers (PRE0, PRE1, P01M, P2M, P3M, IPR) mentioned earlier in Section 2.1. Table 1 summarizes the function performed on the destination byte by each of the above instructions. All of these instructions, with the exception of COM, require a mask operand. The "selected" bits referenced in Table 1 are those bits in the destination operand for which the corresponding mask bit is a logic 1.

Opcode	Use
TM	To test selected bits for logic 0
TCM	To test selected bits for logic 1
AND	To reset all but selected bits to logic 0
OR	To set selected bits to logic 1
XOR	To complement selected bits
COM	To complement all bits

Table 1. Bit Manipulation Instruction Usage

The instructions AND, OR, XOR, and COM have functions common to today's microprocessors and therefore are not described in depth here. However, examples of the use of these instructions are laced throughout the remainder of this document, thus giving an integrated view of their uses in common functions. Since they are unique to the Z8, the functions of Test under Mask and Test Complement under Mask, are discussed in more detail next.

4.1 Test under Mask (TM). The Test under Mask instruction is used to test selected bits for logic 0. The logical operation performed is destination AND source

Neither source nor destination operand is modified; the FLAGS control register is the only register affected by this instruction. The zero flag (Z) is set if all selected bits are logic 0; it is reset otherwise. Thus, if the selected destination bits are either all logic 1 or a combination of 1s and 0s, the zero flag would be cleared by this instruction. The sign flag (S) is either set or reset to reflect the result of the

4. Bit Manipulations
(Continued)

AND operation; the overflow flag (V) is always reset. All other flags are unaffected. Table 2 illustrates the flag settings which result from the TM instruction on a variety of source and destination operand combinations. Note that a given TM instruction will never result in both the Z and S flags being set.

4.2 Test Complement under Mask. The Test Complement under Mask instruction is used to test selected bits for logic 1. The logical operation performed is

(NOT destination) AND source.

Destination	Source	Flags		
		Z	S	V
(binary)	(binary)	Z	S	V
10001100	01110000	1	0	0
01111100	01110000	0	0	0
10001100	11110000	0	1	0
11111100	11110000	0	1	0
00011000	10100001	1	0	0
01000000	10100001	1	0	0

Table 2. Effects of the TM Instruction

As in Test under Mask, the FLAGS control register is the only register affected by this operation. The zero flag (Z) is set if all selected destination bits are 1; it is reset otherwise. The sign flag (S) is set or reset to reflect the result of the AND operation; the overflow flag (V) is always reset. Table 3 illustrates the flag settings which result from the TCM instruction on a variety of source and destination operand combinations. As with the TM instruction, a given TCM instruction will never result in both the Z and S flags being set.

Destination	Source	Flags		
		Z	S	V
(binary)	(binary)	Z	S	V
10001100	01110000	0	0	0
01111100	01110000	1	0	0
10001100	11110000	0	0	0
11111100	11110000	1	0	0
00011000	10100001	0	1	0
01000000	10100001	0	1	0

Table 3. Effects of the TCM Instruction

SECTION 5

Stack Operations

The Z8 stack resides within an area of data memory (internal or external). The current address in the stack is contained in the stack pointer, which decrements as bytes are pushed onto the stack, and increments as bytes are popped from it. The stack pointer occupies two control register bytes (%FE and %FF) in the Z8 register space and may be manipulated like any other register. The stack is useful for subroutine calls, interrupt service routines, and parameter passing and saving. Figure 2 illustrates the downward growth of a stack as bytes are pushed onto it.

5.1 Internal vs. External Stack. The location of the stack in data memory may be selected to be either internal register memory or external data memory. Bit 2 of control register P01M (%F8) controls this selection. Register pair SPH (%FE), SPL (%FF) serves as the stack pointer for an external stack. Register SPL is the stack pointer for an internal stack. In the

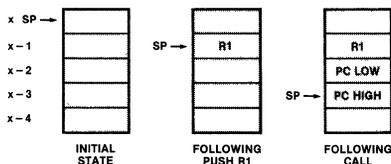


Figure 2. Growth of a Stack

latter configuration, SPH is available for use as a data register. The following illustrates a code sequence that initializes external stack operations:

```
LD P01M,#%(2)00000000
!bit 2: select external stack!
LD SPH,#HI STACK
LD SPL,#LO STACK
```

5.2 CALL. A subroutine call causes the current Program Counter (the address of the byte following the CALL instruction) to be pushed onto the stack. The Program Counter is loaded with the address specified by the CALL instruction. This address may be a direct address or an indirect register pair reference. For example,

```
LABEL 1: CALL %4F98
!direct addressing: PC is
loaded with the hex value
4F98;
address LABEL 1 + 3 is pushed
onto the stack!

LABEL 2: CALL @RR4
!indirect addressing: PC is
loaded with the contents of
working register pair R4, R5;
address LABEL 2 + 2 is pushed
onto the stack!
```

5. Stack Operations (Continued)

LABEL 3: CALL @%7E
!indirect addressing: PC is loaded with the contents of register pair %7E, %7F; address LABEL 3+2 is pushed onto the stack!

5.3 RET. The return (RET) instruction causes the top two bytes to be popped from the stack and loaded into the Program Counter. Typically, this is the last instruction of a subroutine and thus restores the PC to the address following the CALL to that subroutine.

5.4 Interrupt Machine Cycle. During an interrupt machine cycle, the PC followed by the status flags is pushed onto the stack. (A more detailed discussion of interrupt processing is provided in Section 6.)

5.5 IRET. The interrupt return (IRET) instruction causes the top byte to be popped from the stack and loaded into the status flag register, FLAGS (%FC), the next two bytes are then popped and loaded into the Program Counter. In this way, status is restored and program execution continues where it had left off when the interrupt was recognized.

5.6 PUSH and POP. The PUSH and POP instructions allow the transfer of bytes between

the stack and register memory, thus providing program access to the stack for saving and restoring needed values and passing parameters to subroutines.

Execution of a PUSH instruction causes the stack pointer to be decremented by 1; the operand byte is then loaded into the location pointed to by the decremented stack pointer. Execution of a POP instruction causes the byte addressed by the stack pointer to be loaded into the operand byte; the stack pointer is then incremented by 1. In both cases, the operand byte is designated by either a direct register address or an indirect register reference. For example:

```
PUSH R1 !direct address: push working register 1 onto the stack!  
POP 5 !direct address: pop the top stack byte into register 5!  
PUSH @R4 !indirect address: pop the top stack byte into the byte pointed to by working register 4!  
PUSH @17 !indirect address: push onto the stack the byte pointed to by register 17!
```

SECTION 6

Interrupts

The Z8 recognizes six different interrupts from four internal and four external sources, including internal timer/counters, serial I/O, and four Port 3 lines. Interrupts may be individually or globally enabled/disabled via Interrupt Mask Register IMR (%FB) and may be prioritized for simultaneous interrupt resolution via Interrupt Priority Register IPR (%F9). When enabled, interrupt request processing automatically vectors to the designated service routine. When disabled, an interrupt request may be polled to determine when processing is needed.

6.1 Interrupt Initialization. Before the Z8 can recognize interrupts following RESET, some initialization tasks must be performed. The initialization routine should configure the Z8 interrupt requests to be enabled/disabled, as required by the target application and assigned a priority (via IPR) for simultaneous enabled-interrupt resolution. An interrupt request is enabled if the corresponding bit in the IMR is set (= 1) and interrupts are globally enabled (bit 7 of IMR = 1). An interrupt request is disabled if the corresponding bit in the IMR is reset (= 0) or interrupts are globally disabled (bit 7 of IMR = 0).

A RESET of the Z8 causes the contents of the Interrupt Request Register IRQ (%FA) to be held to zero until the execution of an EI

instruction. Interrupts that occur while the Z8 is in this initial state will not be recognized, since the corresponding IRQ bit cannot be set. The EI instruction is specially decoded by the Z8 to enable the IRQ; simply setting bit 7 of IMR is therefore *not* sufficient to enable interrupt processing following RESET. However, subsequent to this initial EI instruction, interrupts may be globally enabled either by the instruction

```
EI !enable interrupts!
```

or by a register manipulation instruction such as

```
OR IMR,#%80
```

To globally disable interrupts, execute the instruction

```
DI !disable interrupts!
```

This will cause bit 7 of IMR to be reset.

Interrupts *must* be globally disabled prior to any modification of the IMR, IPR or enabled bits of the IRQ (those corresponding to enabled interrupt requests), unless it can be *guaranteed* that an enabled interrupt will not occur during the processing of such instructions. Since interrupts represent the occurrence of events asynchronous to program execution, it is highly unlikely that such a guarantee can be made reliably.

6. Interrupts
(Continued)

6.2 Vectored Interrupt Processing. Enabled interrupt requests are processed in an automatic vectored mode in which the interrupt service routine address is retrieved from within the first 12 bytes of program memory. When an enabled interrupt request is recognized by the Z8, the Program Counter is pushed onto the stack (low order 8 bits first, then high-order 8 bits) followed by the FLAGS register (#%FC). The corresponding interrupt request bit is reset in IRQ, interrupts are globally disabled (bit 7 of IMR is reset), and an indirect jump is taken on the word in location 2x, 2x + 1 (x = interrupt request number, 0 ≤ x ≤ 5). For example, if the bytes at addresses %0004 and %0005 contain %05 and %78 respectively, the interrupt machine cycle for IRQ2 will cause program execution to continue at address %0578.

When interrupts are sampled, more than one interrupt may be pending. The Interrupt Priority Register (IPR) controls the selection of the pending interrupt with highest priority. While this interrupt is being serviced, a higher-priority interrupt may occur. Such interrupts

may be allowed service within the current interrupt service routine (nested) or may be held until the current service routine is complete (non-nested).

To allow nested interrupt processing, interrupts must be selectively enabled upon entry to an interrupt service routine. Typically, only higher-priority interrupts would be allowed to nest within the current interrupt service. To do this, an interrupt routine must "know" which interrupts have a higher priority than the current interrupt request. Selection of such nesting priorities is usually a reflection of the priorities established in the Interrupt Priority Register (IPR). Given this data, the first instructions executed in the service routine should be to save the current Interrupt Mask Register, mask off all interrupts of lower and equal priority, and globally enable interrupts (EI). For example, assume that service of interrupt requests 4 and 5 are nested within the service of interrupt request 3. The following illustrates the code required to enable IRQ4 and IRQ5:

```

CONSTANT      INT_MASK_3      :=      %(2) 00110000
GLOBAL
IRQ3__service  PROCEDURE      ENTRY
!service routine for IRQ3!
      PUSH IMR                !save Interrupt Mask Register!
                                !interrupts were globally disabled during the interrupt
                                machine cycle - no DI is needed prior to modification of IMR!
      AND  IMR,#INT_MASK_3    !disable all but IRQ4 & 5!
      EI
      !...!                   !service interrupt!
                                !interrupts are globally enabled now — must disable them prior to
                                modification of IMR!
      DI
      POP  IMR                !restore entry IMR!
      IRET
END IRQ3__service

```

Note that IRQ4 and IRQ5 are enabled by the above sequence only if their respective IMR bits = 1 on entry to IRQ3__service.

The service routine for an interrupt whose processing is to be completed without interruption should not allow interrupts to be nested within it. Therefore, it need not modify the IMR, since interrupts are disabled automatically during the interrupt machine cycle.

The service routine for an enabled interrupt is typically concluded with an IRET instruction, which restores the FLAGS register and Program Counter from the top of the stack and globally enables interrupts. To return from an interrupt service routine without re-enabling

interrupts, the following code sequence could be used:

```

      POP  FLAGS
                                !FLAGS ← @SP!
      RET      !PC ← @SP!

```

This accomplishes all the functions of IRET, except that IMR is not affected.

6.3 Polled Interrupt Processing Disabled interrupt requests may be processed in a polled mode, in which the corresponding bits of the Interrupt Request Register (IRQ) are examined by the software. When an interrupt request bit is found to be a logic 1, the interrupt should be processed by the appropriate

6. Interrupts
(Continued)

service routine. During such processing, the interrupt request bit in the IRQ must be cleared by the software in order for subsequent interrupts on that line to be distinguished from the current one. If more than one interrupt request is to be processed in a polled mode, polling should occur in the order of estab-

lished priorities. For example, assume that IRQ0, IRQ1, and IRQ4 are to be polled and that established priorities are, from high to low, IRQ4, IRQ0, IRQ1. An instruction sequence like the following should be used to poll and service the interrupts:

```
!...!
!poll interrupt inputs here!
      TCM      IRQ, #%(2)00010000      !IRQ4 need service?!
      JR       NZ, TEST0                !no!
      CALL    IRQ4__service             !yes!
TEST0: TCM      IRQ, #%(2)00000001      !IRQ0 need service?!
      JR       NZ, TEST1                !no!
      CALL    IRQ0__service             !yes!
TEST1: TCM      IRQ, #%(2)00000010      !IRQ1 need service?!
      JR       NZ, DONE                 !no!
      CALL    IRQ1__service             !yes!
DONE:  !...!

IRQ4__service      PROCEDURE          ENTRY
      !...!
      AND     IRQ, #%(2)11101111      !clear IRQ4!
      !...!
      RET
END IRQ4__service

IRQ0__service      PROCEDURE          ENTRY
      !...!
      AND     IRQ, #%(2)11111110      !clear IRQ0!
      !...!
      RET
END IRQ0__service

IRQ1__service      PROCEDURE          ENTRY
      !...!
      AND     IRQ, #%(2)11111101      !clear IRQ1!
      !...!
      RET
END IRQ1__service
!...!
```

SECTION 7

Timer/Counter Functions

The Z8 provides two 8-bit timer/counters, T₀ and T₁, which are adaptable to a variety of application needs and thus allow the software (and external hardware) to be relieved of the bulk of such tasks. Included in the set of such uses are:

- Interval delay timer
- Maintenance of a time-of-day clock
- Watch-dog timer
- External event counting
- Variable pulse train output
- Duration measurement of external event
- Automatic delay following external event detection

Each timer/counter is driven by its own 6-bit prescaler, which is in turn driven by the internal Z8 clock divided by four. For T₁, the internal clock may be gated or triggered by an external event or may be replaced by an external clock input. Each timer/counter may operate in either single-pass or continuous mode where, at end-of-count, either counting stops or the counter reloads and continues counting. The counter and prescaler registers may be altered individually while the timer/counter is running; the software controls whether the new values are loaded immediately or when end-of-count (EOC) is reached.

Although the timer/counter prescaler registers (PRE0 and PRE1) are write-only, there is a technique by which the timer/

7. Timer/Counter Functions (Continued)

counters may simulate a readable prescaler. This capability is a requirement for high resolution measurement of an event's duration. The basic approach requires that one timer/counter be initialized with the desired counter and prescaler values. The second timer/counter is initialized with a counter equal to the prescaler of the first timer/counter and a prescaler of 1. The second timer/counter must be programmed for continuous mode. With both timer/counters driven by the internal clock and started and stopped simultaneously, they will run synchronous to one another; thus, the value read from the second counter will always be equivalent to the prescaler of the first.

7.1 Time/Count Interval Calculation To determine the time interval (i) until EOC, the equation

$$i = t \times p \times v$$

characterizes the relation between the prescaler (p), counter (v), and clock input period (t); t is given by

$$1/(XTAL/8)$$

where XTAL is the Z8 input clock frequency; p is in the range 1–64; v is in the range 1–256. When programming the prescaler and counter registers, the maximum load value is truncated to six and eight bits, respectively, and is therefore programmed as zero. For an input clock frequency of 8 MHz, the prescaler and counter register values may be programmed to time an interval in the range

$$1 \mu s \times 1 \times 1 \leq i \leq 1 \mu s \times 64 \times 256$$

$$1 \mu s \leq i \leq 16.384 \text{ ms}$$

To determine the count (c) until EOC for T_1 with external clock input, the equation

$$c = p \times v$$

characterizes the relation between the T_1 prescaler (p) and the T_1 counter (v). The divide-by-8 on the input frequency is bypassed in this mode. The count range is

$$1 \times 1 \leq c \leq 64 \times 256$$

$$1 \leq c \leq 16,384$$

7.2 T_{OUT} Modes. Port 3, bit 5 (P3₆) may be configured as an output (T_{OUT}) which is dynamically controlled by one of the following:

- T₀
- T₁
- Internal clock

When driven by T₀ or T₁, T_{OUT} is reset to a logic 1 when the corresponding load bit is set in timer control register TMR (%F1) and toggles on EOC from the corresponding counter.

When T_{OUT} is driven by the internal clock, that clock is directly output on P3₆.

While programmed as T_{OUT}, P3₆ is disabled from being modified by a write to port register %03; however, its current output may be examined by the Z8 software by a read to port register %03.

7.3 T_{IN} Modes. Port 3, bit 1 (P3₁) may be configured as an input (T_{IN}) which is used in conjunction with T₁ in one of four modes:

- External clock input
- Gate input for internal clock
- Nonretriggerrable input for internal clock
- Retriggerable input for internal clock

For the latter two modes, it should be noted that the existence of a synchronizing circuit within the Z8 causes a delay of two to three internal clock periods following an external trigger before clocking of the counter actually begins.

Each High-to-Low transition on T_{IN} will generate interrupt request IRQ2, regardless of the selected T_{IN} mode or the enabled/disabled state of T₁. IRQ2 must therefore be masked or enabled according to the needs of the application.

The "external clock input" T_{IN} mode supports the counting of external events, where an event is seen as a High-to-Low transition on T_{IN}. Interrupt request IRQ5 is generated on the n^{th} occurrence (single-pass mode) or on every n^{th} occurrence (continuous mode) of that event.

The "gate input for internal clock" T_{IN} mode provides for duration measurement of an external event. In this mode, the T₁ prescaler is driven by the Z8 internal clock, gated by a High level on T_{IN}. In other words, T₁ will count while T_{IN} is High and stop counting while T_{IN} is Low. Interrupt request IRQ2 is generated on the High-to-Low transition on T_{IN}. Interrupt request IRQ5 is generated on T₁ EOC. This mode may be used when the width of a High-going pulse needs to be measured. In this mode, IRQ2 is typically the interrupt request of most importance, since it signals the end of the pulse being measured. If IRQ5 is generated prior to IRQ2 in this mode, the pulse width on T_{IN} is too large for T₁ to measure in a single pass.

The "nonretriggerable input" T_{IN} mode provides for automatic delay timing following an external event. In this mode, T₁ is loaded and clocked by the Z8 internal clock following the first High-to-Low transition on T_{IN} after T₁ is enabled. T_{IN} transitions that occur after this point do not affect T₁. In single-pass mode, the

7. Timer/Counter Functions
(Continued)

enable bit is reset on EOC; further T_{IN} transitions will not cause T_1 to load and begin counting until the software sets the enable bit again. In continuous mode, EOC does not modify the enable bit, but the counter is reloaded and counting continues immediately; IRQ5 is generated every EOC until software resets the enable bit. This T_{IN} mode may be used, for example, to time the line feed delay following end of line detection on a printer or to delay data sampling for some length of time following a sample strobe.

The "retriggerable input" T_{IN} mode will load and clock T_1 with the Z8 internal clock on every occurrence of a High-to-Low transition on T_{IN} . T_1 will time-out and generate interrupt request IRQ5 when the programmed time interval (determined by T_1 prescaler and load register values) has elapsed since the last High-to-Low transition on T_{IN} . In single-pass mode, the enable bit is reset on EOC; further T_{IN} transitions will not cause T_1 to load and begin counting until the software sets the enable bit again. In continuous mode, EOC does not modify the enable bit, but the counter is reloaded and counting continues immedi-

ately; IRQ5 is generated at every EOC until the software resets the enable bit. This T_{IN} mode may provide such functions as watch-dog timer (e.g., interrupt if conveyor belt stopped or clock pulse missed), or keyboard time-out (e.g., interrupt if no input in x ms).

7.4 Examples. Several possible uses of the timer/counters are given in the following four examples.

7.4.1 Time of Day Clock. The following module illustrates the use of T_1 for maintenance of a time of day clock, which is kept in binary format in terms of hours, minutes, seconds, and hundredths of a second. It is desired that the clock be updated once every hundredth of a second; therefore, T_1 is programmed in continuous mode to interrupt 100 times a second. Although T_1 is used for this example, T_0 is equally suited for the task.

The procedure for initializing the timer (TOD_INIT), the interrupt service routine (TOD) which updates the clock, and the interrupt vector for T_1 end-of-count (IRQ_5) are illustrated below. XTAL = 7.3728 MHz is assumed.

```

Z8ASM      2.0
LOC      OBJ CODE      STMT SOURCE STATEMENT

          1  TIMER1  MODULE
          2  CONSTANT
          3  HOUR   :=      R12
          4  MINUTE :=      R13
          5  SECOND :=      R14
          6  HUND   :=      R15
          7  $SECTION PROGRAM
          8  GLOBAL
          9  !IRQ5 interrupt vector!
         10  $ABS      10
P 0000 000F' 11  IRQ_5  ARRAY  [1 WORD] := [TOD]
         12
         13  $REL
P 000C      14  TOD_INIT  PROCEDURE
         15  ENTRY
P 0000 E6 F3 93 16  LD      PRE1,%%(2)10010011
         17                      !bit 2-7: prescaler = 36;
         18                      !bit 1: internal clock;
         19                      !bit 0: continuous mode!
P 0003 E6 F2 00 20  LD      T1,#0    !(256) time-out =
         21                      1/100 second!
P 0006 46 F1 0C 22  OR      TMR,%%0C  !load, enable T1!
P 0009 8F      23  DI
P 000A 46 FB 20 24  OR      IMR,%%20  !enable T1 interrupt!
P 000D 9F      25  EI
P 000E AF      26  RET
P 000F      27  END      TOD_INIT
         28
P 000F      29  TOD      PROCEDURE
         30  ENTRY
P 000F 70 FD    31  PUSH     RP
         32  !Working register file %10 to %1F contains
         33  the time of day clock!
P 0011 31 10    34  SRP      #%10
P 0013 FE      35  INC      HUND      !1 more .01 sec!
P 0014 A6 EF 64 36  CP      HUND,#100 !full second yet?!
P 0017 EB 13    37  JR      NE,TOD_EXIT !jump if no!
P 0019 B0 EF    38  CLR      HUND
P 001B EE      39  INC      SECOND     !1 more second!
P 001C A6 EE 3C 40  CP      SECOND,#60 !full minute yet?!
P 001F EB 0B    41  JR      NE,TOD_EXIT !jump if no!

```

7. Timer/Counter Functions
(Continued)

```

P 0021 B0 EE 42 CLR SECOND
P 0023 DE EE 43 INC MINUTE !1 more minute!
P 0024 A6 ED 3C 44 CP MINUTE,#60 !full hour yet?!
P 0027 EB 03 45 JR NE,TOD_EXIT !jump if no!
P 0029 B0 ED 46 CLR MINUTE
P 002B CE 47 INC HOUR
48 TOD_EXIT:
P 002C 50 FD 49 POP RP !restore entry RP!
P 002E BF 50 IRET
P 002F 51 END TOD
52 END TIMER1

```

0 ERRORS
ASSEMBLY COMPLETE

<i>TOD_INIT:</i>	<i>TOD:</i>
7 instructions	17 instructions
15 bytes	32 bytes
16 μ s	19.5 μ s (average) including interrupt response time

7.4.2 Variable Frequency, Variable Pulse Width Output. The following module illustrates one possible use of T_{OUT}. Assume it is necessary to generate a pulse train with a 10% duty cycle, where the output is repetitively high for 1.6 ms and then low for 14.4 ms. To do this, T_{OUT} is controlled by end-of-count from T₁, although T₀ could alternately be chosen. This example makes use of the Z8 feature that allows a timer's counter register to be modified without disturbing the count in progress. In continuous mode, the new value is loaded when T₁ reaches EOC. T₁ is first loaded and enabled with values to generate the short interval. The counter register is then immediately modified with the value to generate the long interval; this value is loaded into the counter automatically on T₁ EOC. The prescaler selected value must be the same for both long and short intervals. Note that the

initial loading of the T₁ counter register is followed by setting the T₁ load bit of timer control register TMR (%F1); this action causes T_{OUT} to be reset to a logic 1 output. Each subsequent modification of the T₁ counter register does not affect the current T_{OUT} level, since the T₁ load bit is NOT altered by the software. The new value is loaded on EOC, and T_{OUT} will toggle at that time. The T₁ interrupt service routine should simply modify the T₁ counter register with the new value, alternating between the long and short interval values.

In the example which follows, bit 0 of register %04 is used as a software flag to indicate which value was loaded last. This module illustrates the procedure for T₁/T_{OUT} initialization (PULSE_INIT), the T₁ interrupt service routine (PULSE), and the interrupt vector for T₁ EOC (IRQ_5). XTAL = 8 MHz is assumed.

```

Z8ASM 2.0
LOC OBJ CODE STMT SOURCE STATEMENT
1 TIMER2 MODULE
2 $SECTION PROGRAM
3 GLOBAL
4 !IRQ5 interrupt vector!
5 $ABS 10
P 0000 0017' 6 IRQ_5 ARRAY [1 WORD] := [PULSE]
7
8 $REL
P 000C 9 PULSE_INIT PROCEDURE
10 ENTRY
P 0000 E6 F3 03 11 LD PRE1,#%(2)00000011
12 !bit 2-7: prescaler = 64;
13 !bit 1: internal clock;
14 !bit 0: continuous mode!
P 0003 E6 F7 00 15 LD P3M,#00 !bit 5: let P36 be Tout!
P 0006 E6 F2 19 16 LD T1,#25 !for short interval!
P 0009 8F 17 DI
P 000A 46 FB 20 18 OR IMR,#%(2)00100000 !enable T1 interrupt!
P 000D E6 F1 8C 19 LD TMR,#%(2)10001100
20 !bit 6-7: Tout controlled
21 !by T1;
22 !bit 3: enable T1;
23 !bit 2: load T1 !
P 0010 E6 F2 E1 24 !Set long interval counter, to be loaded on T1 EOC!
25 LD T1,#225
26 !Clear alternating flag for PULSE!

```

7. Timer/Counter Functions
(Continued)

```

P 0013 B0 04      27      CLR      %04      != 0 : 25 next;
                  28                        = 1 : 225 next !
P 0015 9F        29      EI
P 0016 AF        30      RET
P 0017           31      END      PULSE_INIT
                  32
                  33
P 0017           34      PULSE   PROCEDURE
                  35      ENTRY
P 0017 E6 F2 E1   36      LD      T1,#225      !new load value!
P 001A B6 04 01   37      XOR      %04,#1      !which value next?!
P 001D 6B 03     38      JR      Z,PULSE_EXIT !should be 225!
P 001F E6 F2 19   39      LD      T1,#25      !should be 25!
                  40      PULSE_EXIT:
P 0022 BF        41      IRET
P 0023           42      END      PULSE
                  43      END      TIMER2

```

0 ERRORS
ASSEMBLY COMPLETE

PULSE_INIT:
10 instructions
23 bytes
23 μs

PULSE:
5 instructions
12 bytes
25 μs (average) including interrupt response time

7.4.3 Cascaded Timer/Counters. For some applications it may be necessary to measure a greater time interval than a single timer/counter can measure (16.384 ms). In this case, T_{1N} and T_{OUT} may be used to cascade T₀ and

T₁ to function as a single unit. T_{OUT}, programmed to toggle on T₀ end-of-count, should be wired back to T_{1N}, which is selected as the external clock input for T₁. With T₀ programmed for continuous mode, T_{OUT} (and therefore T_{1N}) goes through a High-to-Low transition (causing T₁ to count) on every other T₀ EOC. Interrupt request IRQ5 is generated when the programmed time interval has elapsed. Interrupt requests IRQ2 (generated on every T_{1N} High-to-Low transition) and IRQ4 (generated on T₀ EOC) are of no importance in this application and are therefore disabled.

To determine the time interval (i) until EOC, the equation

$$i = t \times p_0 \times v_0 \times (2 \times p_1 \times v_1 - 1)$$

characterizes the relation between the T₀ prescaler (p₀) and counter (v₀), the T₁ prescaler (p₁) and counter (v₁), and the clock input period (t); t is defined in Section 7.1. Assuming XTAL = 8 MHz, the measurable time interval range is

$$1 \mu s \times 1 \times 1 \times (2 \times 1 - 1) \leq i \leq 1 \mu s \times 64 \times 256 \times (2 \times 64 \times 256 - 1)$$

$$1 \mu s \leq i \leq 536.854528 \text{ s}$$

Figure 3 illustrates the interconnection between T₀ and T₁. The following module illustrates the procedure required to initialize the timers for a 1.998 second delay interval:

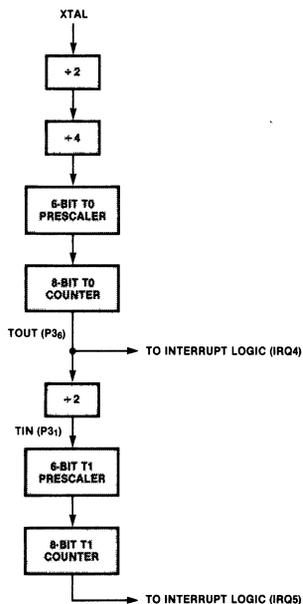


Figure 3. Cascaded Timer/Counters

**7. Timer/
Counter
Functions**
(Continued)

```

Z8ASM      2.0
LOC      OBJ CODE      STMT SOURCE STATEMENT
                                1 TIMER3  MODULE
                                2 GLOBAL
                                3 TIMER_16  PROCEDURE
P 0000                                4 ENTRY
P 0000 E6 F3 28      5          LD      PRE1,%%(2)00101000
                                6
                                7          !bit 2-7: prescaler = 10;
                                8          !bit 1: external clock;
                                9          !bit 0: single-pass mode!
P 0003 E6 F7 00      9          LD      P3M,#00 !bit 5: let P36 be Tout!
P 0006 E6 F2 64     10         LD      T1,#100 !T1 counter register!
P 0009 E6 F5 29     11         LD      PRE0,%%(2)00101001
                                12          !bit 2-7: prescaler = 10;
                                13          !bit 0: continuous mode!
P 000C E6 F4 64     14         LD      T0,#100 !T0 counter register!
P 000F 8F          15         DI
P 0010 56 FB 2B     16         AND     IMR,%%(2)00101011 !disable IRQ2 (Tin);
                                17          and IRQ4 (T0) !
P 0013 46 FB 20     18         OR      IMR,%%(2)00100000 !enable IRQ5 (T1)!
P 0016 9F          19         EI
P 0017 E6 F1 4F     20         LD      TMR,%%(2)01001111
                                21          !bit 6-7: Tout controlled
                                22          by T0;
                                23          bit 4-5: Tin mode is ext.
                                24          clock input;
                                25          bit 3: enable T1;
                                26          bit 2: load T1;
                                27          bit 1: enable T0;
                                28          bit 0: load T0 !
P 001A AF          29         RET
P 001B          30 END     TIMER_16
                                31 END     TIMER3

```

0 ERRORS
ASSEMBLY COMPLETE

11 instructions
27 bytes
26.5 μ s

7.4.4 Clock Monitor. T_1 and T_{IN} may be used to monitor a clock line (in a diskette drive, for example) and generate an interrupt request when a clock pulse is missed. To accomplish this, the clock line to be monitored is wired to $P3_1$ (T_{IN}). T_{IN} should be programmed as a retriggerable input to T_1 , such that each falling edge on T_{IN} will cause T_1 to reload and continue counting. If T_1 is programmed to time-out after an interval of one-and-a-half times the clock period being monitored, T_1 will time-out and generate interrupt request IRQ5 only if a clock pulse is missed.

The following module illustrates the procedure for initializing T_1 and T_{IN} (MONITOR__INIT) to monitor a clock with a period of 2 μ s. XTAL = 8 MHz is assumed. Note that this example selects single-pass rather than continuous mode for T_1 . This is to prevent a continuous stream of IRQ5 interrupt requests in the event that the monitored clock fails completely. Rather, the interrupt service routine (CLK__ERR) is left with the choice of whether or not to re-enable the monitoring. Also shown is the T_1 interrupt vector (IRQ__5).

```

Z8ASM      2.0
LOC      OBJ CODE      STMT SOURCE STATEMENT
                                1 TIMER4  MODULE
                                2 $SECTION PROGRAM
                                3 GLOBAL
                                4 !IRQ5 interrupt vector!
P 0000 0015'      5          $ABS 10
                                6 IRQ_5  ARRAY  [1 WORD] := [CLK_ERR]
                                7
                                8          $REL
P 000C          9 MONITOR_INIT  PROCEDURE
                                10 ENTRY
P 0000 E6 F3 04     11         LD      PRE1,%%(2)00000100
                                12          !bit 2-7: prescaler = 1;
                                13          !bit 1: external clock;
                                14          !bit 0: single-pass mode!
P 0003 E6 F7 00     15         LD      P3M,#00 !bit 5: let P36 be Tout!
P 0006 E6 F2 03     16         LD      T1,#3 !T1 load register,
                                17          = 1.5 * 2 usec !

```

```

7. Timer/Counter Functions
(Continued)
P 0009 8F          18      DI
P 000A 56 FB 3B   19      AND      IMR, #%(2)00111011 !disable IRQ2 (Tin)!
P 000D 46 FB 20   20      OR       IMR, #%(2)00100000 !enable IRQ5 (T1)!
P 0010 9F          21      EI
                22
P 0011 E6 F1 38   23      LD       TMR, #%(2)00111000
                24                      !bit 4-5: Tin mode is
                25                      retrig. input;
                26                      bit 3: enable T1 !
P 0014 AF          27      RET
P 0015             28 END     MONITOR_INIT
                29
                30
P 0015             31 CLK_ERR PROCEDURE
                32 ENTRY
                33             !...!             !handle the missed clock!
                34
                35 !if clock monitoring should continue...!
P 0015 46 F1 08   36      OR       TMR, #%(2)00001000
                37                      !bit 3: enable T1 !
P 0018 BF          38      IRET
P 0019             39 END     CLK_ERR
                40 END     TIMER4

        0 ERRORS
ASSEMBLY COMPLETE

MONITOR_INIT:                CLK_ERR:
    9 instructions                2 + instructions
    21 bytes                    4 + bytes
    21.5  $\mu$ s                18.5 +  $\mu$ s including interrupt response time

```

SECTION 8

I/O Functions

The Z8 provides 32 I/O lines mapped into registers 0-3 of the internal register file. Each nibble of port 0 is individually programmable as input, output, or address/data lines (A_{15} - A_{12} , A_{11} - A_8). Port 1 is programmable as a single entity to provide input, output, or address/data lines (AD_7 - AD_0). The operating modes for the bits of Ports 0 and 1 are selected by control register P01M (%F8). Selection of I/O lines as address/data lines supports access to external program and data memory; this is discussed in Section 3. Each bit of Port 2 is individually programmable as an input or an

output bit. Port 2 bits programmed as outputs may also be programmed (via bit 0 of P3M) to all have active pull-ups or all be open-drain (active pull-ups inhibited). In Port 3, four bits (P_{30} - P_{33}) are fixed as inputs, and four bits (P_{34} - P_{37}) are fixed as outputs, but their functions are programmable. Special functions provided by Port 3 bits are listed in Table 4. Use of the Data Memory select output is discussed in Section 3; uses of T_{IN} and T_{OUT} are discussed in Section 7.

8.1 Asynchronous Receiver/Transmitter

Operation. Full-duplex, serial asynchronous receiver/transmitter operation is provided by the Z8 via P_{37} (output) and P_{30} (input) in conjunction with control register SIO (%F0), which is actually two registers: receiver buffer and transmitter buffer. Counter/Timer T_0 provides the clock for control of the bit rate.

The Z8 always receives and transmits eight bits between start and stop bits. However, if parity is enabled, the eighth bit (D_7) is replaced by the odd-parity bit when transmitted and a parity-error flag (= 1 if error) when received. Table 5 illustrates the state of the parity bit/parity error flag during serial I/O with parity enabled.

Although the Z8 directly supports either odd parity or no parity for serial I/O operation, even parity may also be provided with additional software support. To receive and transmit with even parity, the Z8 should be configured for serial I/O with odd parity disabled. The Z8 software must calculate parity

Function	Bit	Signal
Handshake	P_{31}	$\overline{DAV2}/RDY2$
	P_{32}	$\overline{DAV0}/RDY0$
	P_{33}	$\overline{DAV1}/RDY1$
	P_{34}	$RDY1/\overline{DAV1}$
	P_{35}	$RDY0/\overline{DAV0}$
	P_{36}	$RDY2/\overline{DAV2}$
Interrupt Request	P_{30}	IRQ3
	P_{31}	IRQ2
	P_{32}	IRQ0
	P_{33}	IRQ1
Counter/Timer	P_{31}	T_{IN}
	P_{36}	T_{OUT}
Data Memory Select	P_{34}	\overline{DM}
Status Out		
Serial I/O	P_{30}	Serial In
	P_{37}	Serial Out

Table 4. Port 3 Special Functions

8. I/O Functions

(Continued)

Character Loaded Into SIO	Transmitted To Serial Line	Received From Serial Line	Character Transferred To SIO	Note*
11000011	01000011	01000011	01000011	no error
11000011	01000011	01000111	11000111	error
01111000	11111000	11111000	01111000	no error
01111000	11111000	01111000	11111000	error

Table 5. Serial I/O With Odd Parity

* Left-most bit is D7

and modify the eighth bit prior to the load of a character into SIO and then modify a parity error flag following the load of a character from SIO. All other processing required for serial I/O (e.g. buffer management, error handling, etc.) is the same as that for odd parity operations.

To configure the Z8 for Serial I/O, it is necessary to:

- Enable P3₀ and P3₇ for serial I/O and select parity,
- Set up T₀ for the desired bit rate,
- Configure IRQ3 and IRQ4 for polled or automatic interrupt mode,
- Load and enable T₀.

To enable P3₀ and P3₇ for serial I/O, bit 6 of P3M (R247) is set. To enable odd parity, bit 7 of P3M is set; to disable it, the bit is reset. For example, the instruction

```
LD P3M,#%40
```

will enable serial I/O, but disable parity. The instruction

```
LD P3M,#%C0
```

will enable serial I/O, and enable odd parity.

In the following discussions, bit rate refers to all transmitted bits, including start, stop, and parity (if enabled). The serial bit rate is given by the equation:

$$\text{bit rate} = \frac{\text{input clock frequency}}{(2 \times 4 \times T_0 \text{ prescaler} \times T_0 \text{ counter} \times 16)}$$

The final divide-by-16 is incurred for serial communications, since in this mode T₀ runs at 16 times the bit rate in order to synchronize the data stream. To configure the Z8 for a specific bit rate, appropriate values must first be selected for T₀ prescaler and T₀ counter by the above equation; these values are then programmed into registers T₀ (%F4) and PRE0 (%F5) respectively. Note that PRE0 also controls the continuous vs. single-pass mode for T₀; continuous mode should be selected for serial I/O. For example, given an input clock frequency of 7.3728 MHz and a selected bit rate of 9600 bits per second, the equation is

satisfied by T₀ counter = 2 and prescaler = 3. The following code sequence will configure the T₀ counter and T₀ prescaler registers:

```
LD T0,#2 !T0 counter = 2!
LD PRE0,#%(2)00001101
!bit 2-7: prescaler = 3; bit 0:
continuous mode!
```

Interrupt request 3 (IRQ3) is generated whenever a character is transferred into the receive buffer; interrupt request 4 (IRQ4) is generated whenever a character is transferred out of the transmit buffer. Before accepting such interrupt requests, the Interrupt Mask, Request, and Priority Registers (IMR, IRQ, and IPR) must be programmed to configure the mode of interrupt response. The section on Interrupt Processing provides a discussion of interrupt configurations.

To load and enable T₀, set bits 0 and 1 of the timer mode register (TMR) via an instruction such as

```
OR TMR,#%03
```

This will cause the T₀ prescaler and counter registers (PRE0 and T₀) to be transferred to the T₀ prescaler and counter. In addition, T₀ is enabled to count, and serial I/O operations will commence.

Characters to be output to the serial line should be written to serial I/O register SIO (%F0). IRQ4 will be generated when all bits have been transferred out.

Characters input from the serial line may be read from SIO. IRQ3 will be generated when a full character has been transferred into SIO.

The following module illustrates the receipt of a character and its immediate echo back to the serial line. It is assumed that the Z8 has been configured for serial I/O as described above, with IRQ3 (receive) enabled to interrupt, and IRQ4 (transmit) configured to be polled. The received character is stored in a circular buffer in register memory from address %42 to %5F. Register %41 contains the address of the next available buffer position and should have been initialized by some earlier routine to #%42.

8. I/O Functions
(Continued)

```

Z8ASM      2.0
LOC      OBJ CODE      STMT SOURCE STATEMENT
          1 SERIAL_IO      MODULE
          2 CONSTANT
          3 next_addr      :=      %41
          4 start         :=      %42
          5 length        :=      %1E
          6 $SECTION PROGRAM
          7 GLOBAL
          8 !IRQ3 vector!
          9 $ABS          6
P 0006 0000' 10 IRQ_3      ARRAY [1 WORD] := [GET_CHARACTER]
          11
          12 $REL          0
P 0000      13 GET_CHARACTER PROCEDURE      ENTRY
          14
          15 !Serial I/O receive interrupt service!
          16 !Echo received character and wait for
          17 echo completion!
P 0000 E4 F0 F0 18 ld      SIO,SIO      !echo!
          19
          20 !save it in circular buffer!
P 0003 F5 F0 41 21 ld      @next_addr,SIO !save in buffer!
P 0006 20 41 22 inc     next_addr      !point to next position!
P 0008 A6 41 60 23 cp      next_addr,#start+length
          24 !wrap-around yet?!
P 000B EB 03 25 jr      ne,echo_wait !no.!
P 000D E6 41 42 26 ld      next_addr,#start !yes. point to start!
          27 !now, wait for echo complete!
          28 echo_wait:
P 0010 66 FA 10 29 tcm     IRQ,##10      !transmitted yet?!
P 0013 5B FB 30 jr      nz,echo_wait !not yet!
          31
P 0015 56 FA EF 32 and     IRQ,##EF      !clear IRQ4!
P 0018 BF 33 33 IRET
P 0019 34 END GET_CHARACTER
          35 END SERIAL_IO

          0 ERRORS
          ASSEMBLY COMPLETE
    
```

10 instructions
25 bytes
35.5 μ s + 5.5 μ s for each additional pass through the echo_wait loop,
including interrupt response time

8.2 Automatic Bit Rate Detection. In a typical system, where serial communication is required (e.g. system with a terminal), the desired bit rate is either user-selectable via a switch bank or nonvariable and "hard-coded" in the software. As an alternate method of bit-rate detection, it is possible to automatically determine the bit rate of serial data received by measuring the length of a start bit. The advantage of this method is that it places no requirements on the hardware design for this function and provides a convenient (automatic) operator interface.

In the technique described here, the serial channel of the Z8 is initialized to expect a bit rate of 19,200 bits per second. The number of bits (n) received through Port pin P30 for each bit transmitted is expressed by

$$n = 19,200/b$$

where b = transmission bit rate. For example, if the transmission bit rate were 1200 bits per second, each incoming bit would appear to the receiving serial line as 19,200/1200 or 16 bits.

The following example is capable of disting-

uishing between the bit rates shown in Table 6 and assumes an input clock frequency of 7.3728 MHz, a T₀ prescaler of 3, and serial I/O enabled with parity disabled. This example requires that a character with its low order bit = 1 (such as a carriage return) be sent to the serial channel. The start bit of this character can be measured by counting the number of zero bits collected before the low order 1 bit. The number of zero bits actually collected into data bits by the serial channel is less than n (as given in the above equation), due to the detection of start and stop bits. Figure 4 illustrates the collection (at 19,200

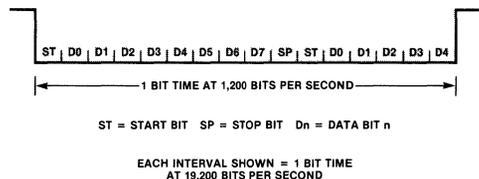


Figure 4. Collection of a Start Bit Transmitted at 1,200 BPS and Received at 19,200 BPS

8. I/O Functions
(Continued)

Bit Rate	Number of Bits Received Per Bit Transmitted	Number of 0 Bits Collected as Data Bits		T ₀ Counter	
		dec	binary	dec	binary
19200	1	0	00000000	1	00000001
9600	2	1	00000001	2	00000010
4800	4	3	00000011	4	00000100
2400	8	7	00000111	8	00001000
1200	16	13	00001101	16	00010000
600	32	25	00011001	32	00100000
300	64	49	00110001	64	01000000
150	128	97	01100001	128	10000000

Table 6. Inputs to the Automatic Bit Rate Detection Algorithm

bits per second) of a zero bit transmitted to the Z8 at 1,200 bits per second. Notice that only 13 of the 16 zero bits received are collected as data bits.

Once the number of zero bits in the start bit has been collected and counted, it remains to translate this count into the appropriate T₀ counter value and program that value into T₀ (%F4). The patterns shown in the two binary columns of Table 6 are utilized in the algorithm for this translation.

As a final step, if incoming data is to commence immediately, it is advisable to wait until the remainder of the current "elongated"

character has been received, thus "flushing" the serial line. This can be accomplished either via a software loop, or by programming T₁ to generate an interrupt request after the appropriate amount of time has elapsed. Since a character is composed of eight bits plus a minimum of one stop bit following the start bit, the length of time to delay may be expressed as

$$(9 \times n)/b$$

where n and b are as defined above. The following module illustrates a sample program for automatic bit rate detection.

```

Z8ASM      2.0
LOC      OBJ CODE      STMT SOURCE STATEMENT

1 bit_rate          MODULE
2 EXTERNAL
3 DELAY      PROCEDURE
4 GLOBAL
P 0000      5 main      PROCEDURE
6          6          ENTRY
P 0000 8F      7          di          !disable interrupts!
P 0001 56 FB 77 8          and      IMR,%%77 !IRQ3 polled mode!
P 0004 56 FA F7 9          and      IRQ,%%F7 !clear IRQ3!
P 0007 E6 F7 40 10         ld      P3M,%%40 !enable serial I/O!
P 0C0A E6 F4 01 11         ld      T0,#1
P 000D E6 F5 0D 12         ld      PRE0,#(3 SHL 2)+1 !bit rate = 19,200;
13                                     continuous count mode!
P 0010 B0 E0 14          clr      R0          !init. zero byte counter!
P 0012 E6 F1 03 15         ld      TMR,#3      !load and enable T0!
16
17 !collect input bytes by counting the number of null
18 characters received. Stop when non-zero byte received!
19 collect:
P 0015 76 FA 08 20         TM      IRQ,%%08 !character received?!
P 0018 6B FB 21          jr      z,collect !not yet!
P 001A 18 F0 22          ld      R1,SIO !get the character!
P 001C 56 FA F7 23         and      IRQ,%%F7 !clear interrupt request!
P 001F 1E 24          inc      R1          !compare to 0 ...!
P 0020 1A 05 25         djnz   R1,bitloop !...(in 3 bytes of code)!
P 0022 06 E0 08 26         add     R0,#8      !update count of 0 bits!
P 0025 8B EE 27          jr      collect
28 bitloop:
29                                     !add in zero bits from low
30                                     end of 1st non-zero byte!
P 0027 E0 E1 30          RR      R1
P 0029 7B 03 31          jr      c,count_done
P 002B 0E 32          inc     R0
P 002C 8B F9 33          jr      bitloop
34
35 !R0 has number of zero bits collected!
36 !translate R0 to the appropriate T0 counter value!
37 count_done:
38                                     !R0 has count of zero bits!
39          ld      R1,#7
P 0030 2C 80 39          ld      R2,%%80 !R2 will have T0 counter value!
P 0032 90 E0 40          RL      R0
41
P 0034 90 E0 42 loop:    RL      R0

```

8. I/O Functions
(Continued)

```

P 0036 7B 04      43      jr      c,done
P 0038 E0 E2      44      RR      R2
P 003A 1A F8      45      djnz   r1,loop
                  46
P 003C 29 F4      47 done:  ld      T0,R2      !load value for detected
                  48                          bit rate!
                  49 !Delay long enough to clear serial line of bit stream!
P 003E D6 0000*   50      call   DELAY
                  51 !clear receive interrupt request!
P 0041 56 FA F7   52      and    IRQ,##F7
                  53
P 0044            54 END    main
                  55 END    bit_rate

      0 ERRORS
ASSEMBLY COMPLETE

```

30 instructions
68 bytes
Execution time is variable based on transmission bit rate.

8.3 Port Handshake. Each of Ports 0, 1 and 2 may be programmed to function under input or output handshake control. Table 7 defines the port bits used for the handshaking and the mode bit settings required to select handshaking. To input data under handshake control, the Z8 should read the input port when the \overline{DAV} input goes Low (signifying that data is available from the attached device). To output data under handshake control, the Z8 should write the output port when the RDY input goes Low (signifying that the previously output data has been accepted by the attached device). Interrupt requests IRQ0, IRQ1, and IRQ2 are generated by the falling edge of the handshake signal input to the Z8 for Port 0, Port 1, and Port 2 respectively. Port handshake operations may therefore be processed under interrupt control.

Consider a system that requires communication of eight parallel bits of data under handshake control from the Z8 to a peripheral device and that Port 2 is selected as the output port. The following assembly code illustrates the proper sequence for initializing Port 2 for output handshake.

```

CLR  P2M  !Port 2 mode register: all Port
        2 bits are outputs!
OR   %03,##%40
        !set  $\overline{DAV}2$ : data not available!
LD   P3M,##%20
        !Port 3 mode register: enable
        Port 2 handshake!
LD   %02,DATA
        !output first data byte;  $\overline{DAV}2$ 
        will be cleared by the Z8 to
        indicate data available to
        the peripheral device!

```

Note that following the initialization of the output sequence, the software outputs the first data byte without regard to the state of the RDY2 input; the Z8 will automatically hold $\overline{DAV}2$ High until the RDY2 input is High. The peripheral device should force the Z8 RDY2 input line Low after it has latched the data in response to a Low on $\overline{DAV}2$. The Low on RDY2 will cause the Z8 to automatically force $\overline{DAV}2$ High until the next byte is output. Subsequent bytes should be output in response to interrupt request IRQ2 (caused by the High-to-Low transition on RDY2) in either a polled or an enabled interrupt mode.

	Port 0	Port 1	Port 2
Input handshake lines	$\left\{ \begin{array}{l} P3_2 = \overline{DAV} \\ P3_5 = RDY \end{array} \right.$	$\left\{ \begin{array}{l} P3_3 = \overline{DAV} \\ P3_4 = RDY \end{array} \right.$	$\left\{ \begin{array}{l} P3_1 = \overline{DAV} \\ P3_6 = RDY \end{array} \right.$
Output handshake lines	$\left\{ \begin{array}{l} P3_2 = RDY \\ P3_5 = \overline{DAV} \end{array} \right.$	$\left\{ \begin{array}{l} P3_3 = RDY \\ P3_4 = \overline{DAV} \end{array} \right.$	$\left\{ \begin{array}{l} P3_1 = RDY \\ P3_6 = \overline{DAV} \end{array} \right.$
To select input handshake:	$\left\{ \begin{array}{l} \text{set bit 6 \& reset bit 7 of} \\ \text{P01M (program high} \\ \text{nibble as input)} \end{array} \right.$	set bit 3 & reset bit 4 of P01M (program byte as input)	set bit 7 of P2M (program high bit as input)
To select output handshake:	$\left\{ \begin{array}{l} \text{reset bits 6, 7 of P01M} \\ \text{(program high nibble as} \\ \text{output)} \end{array} \right.$	reset bits 3, 4 of P01M (program byte as output)	reset bit 7 of P2M (program high bit as output)
To enable handshake.	$\left\{ \begin{array}{l} \text{set bit 5 of Port 3 (P3}_5\text{);} \\ \text{set bit 2 of P3M} \end{array} \right.$	set bit 4 of Port 3 (P3 ₄), set bits 3, 4 of P3M	set bit 6 of Port 3 (P3 ₆); set bit 5 of P3M

Table 7. Port Handshake Selection

SECTION 9

Arithmetic Routines

This section gives examples of the arithmetic and rotate instructions for use in multiplication, division, conversion, and BCD arithmetic algorithms.

9.1 Binary to Hex ASCII. The following module illustrates the use of the ADD and SWAP arithmetic instructions in the conversion of a 16-bit binary number to its hexadecimal ASCII representation. The 16-bit number is viewed as a string of four nibbles and is pro-

cessed one nibble at a time from left to right, beginning with the high-order nibble of the lower memory address. %30 is added to each nibble if it is in the range 0 to 9; otherwise %37 is added. In this way, %0 is converted to %30, %1 to %31, . . . %A to %41, . . . %F to %46. Figure 5 illustrates the conversion of RRO (contents = %F2BE) to its hex ASCII equivalent; the destination buffer is pointed to by RR4.

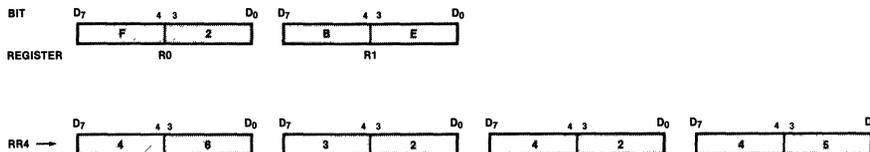


Figure 5. Conversion of (RRO) to Hex ASCII

```

Z8ASM      2.99      INTERNAL RELEASE
LOC      OBJ CODE      STMT SOURCE STATEMENT
P 0000
1 ARITH  MODULE
2 GLOBAL
3 BINASC  PROCEDURE
4 !*****
5 Purpose =      To convert a 16-bit binary
6                number to Hex ASCII
7
8 Input =        RRO = 16-bit binary number.
9                RR4 = pointer to destination
10               buffer in external memory.
11
12 Output =      Resulting ASCII string (4 bytes)
13               in destination buffer.
14               RR4 incremented by 4 .
15               R0,R2,R6 destroyed.
16 *****!
17 ENTRY
18
19         ld      R6,#%04 !nibble count!
20 again:  SWAP   R0      !look at next nibble!
21         ld      R2,R0
22         and    R2,#%0F !isolate 4 bits!
23 !convert to ASCII : R2 + #%30 if R0 in range 0 to 9
24                else R2 + #%37 (in range 0A to 0F)
25 !
26         ADD    R2,#%30
27         cp     R2,#%3A
28         jr     ult,skip
29         ADD    R2,#%07
30 skip:   lde    @RR4,R2      !save ASCII in buffer!
31         incw   RR4         !point to next
32                               buffer position!
33         cp     R6,#%03 !time for second byte?!
34         jr     ne,same_byte !no.!
35         ld     R0,R1      !2nd byte!
36 same_byte:
37         djnz  R6,again
38         ret
39 END      BINASC
40 END      ARITH

```

0 errors
Assembly complete

15 instructions
34 bytes
120.5 μs (average)

9. Arithmetic Routines
(Continued)

9.2 BCD Addition. The following module illustrates the use of the add with carry (ADC) and decimal adjust (DA) instructions for the addition of two unsigned BCD strings of equal length. Within a BCD string, each nibble represents a decimal digit (0-9). Two such digits are packed per byte with the most

significant digit in bits 7-4. Bytes within a BCD string are arranged in memory with the most significant digits stored in the lowest memory location. Figure 6 illustrates the representation of 5970 in a 6-digit BCD string, starting in register %33.

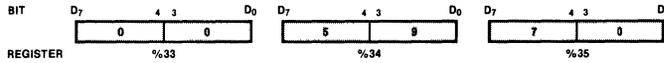


Figure 6. Unsigned BCD Representation

```

Z8ASM      2.0
LOC      OBJ CODE      STMT SOURCE STATEMENT
          1 ARITH  MODULE
          2 CONSTANT
          3 BCD_SRC := R1
          4 BCD_DST := R0
          5 BCD_LEN := R2
          6 GLOBAL
P 0000    7 BCDADD  PROCEDURE
          8 !*****
          9 Purpose =      To add two packed BCD strings of
         10 equal length.
         11 dst <-- dst + src
         12
         13 Input =        R0 = pointer to dst BCD string.
         14                  R1 = pointer to src BCD string.
         15                  R2 = byte count in BCD string
         16                      (digit count = (R2)*2 ).
         17
         18 Output =       BCD string pointed to by R0 is
         19 the sum.
         20 Carry FLAG = 1 if overflow.
         21 R0 , R1 as on entry.
         22 R2 = 0
         23 *****!
         24 ENTRY
         25
P 0000 02 12      26          add      BCD_SRC,BCD_LEN !start at least... !
P 0002 02 02      27          add      BCD_DST,BCD_LEN !significant digits!
P 0004 CF         28          rcf          !carry = 0!
         29 add_again:
P 0005 00 E1      30          dec      BCD_SRC      !point to next two
         31                      src digits!
P 0007 00 E0      32          dec      BCD_DST      !point to next two
         33                      dst digits!
P 0009 E3 31      34          ld       R3,@BCD_SRC  !get src digits!
P 000B 13 30      35          ADC      R3,@BCD_DST  !add dst digits!
P 000D 40 E3      36          DA       R3          !decimal adjust!
P 000F F3 03      37          ld       @BCD_DST,R3    !move to dst!
P 0011 2A F2      38          djnz    BCD_LEN,add_again !loop for next
         39                      digits!
P 0013 AF         40          ret          !all done!
         41
P 0014           42 END      BCDADD
         43 END      ARITH

          0 ERRORS
          ASSEMBLY COMPLETE
    
```

11 instructions

20 bytes

Execution time is a function of the number of bytes (n) in input BCD string:

$$20 \mu s + 12.5 (n - 1) \mu s$$

9. Arithmetic Routines

(Continued)

9.3 Multiply. The following module illustrates an efficient algorithm for the multiplication of two unsigned 8-bit values, resulting in a 16-bit product. The algorithm repetitively shifts the multiplicand right (using RRC), with the low-order bit being shifted out (into the carry flag). If a one is shifted out, the multiplier is added

to the high-order byte of the partial product. As the high-order bits of the multiplicand are vacated by the shift, the resulting partial-product bits are rotated in. Thus, the multiplicand and the low byte of the product occupy the same byte, which saves register space, code, and execution time.

```
Z8ASM      2.99      INTERNAL RELEASE
LOC      OBJ CODE      STMT SOURCE STATEMENT

          1 ARITH      MODULE
          2 CONSTANT
          3 MULTIPLIER      :=      R1
          4 PRODUCT_LO      :=      R3
          5 PRODUCT_HI      :=      R2
          6 COUNT           :=      R0
          7 GLOBAL
P 0000    8 MULT       PROCEDURE
          9 !*****
         10 Purpose =      To perform an 8-bit by 8-bit unsigned
         11                binary multiplication.
         12
         13 Input =        R1 = multiplier
         14                R3 = multiplicand
         15
         16 Output =       RR2 = product
         17                R0 destroyed
         18 *****!
         19 ENTRY
P 0000 OC 09      20                ld      COUNT,#9          !8 BITS + 1!
P 0002 B0 E2      21                clr      PRODUCT_HI      !INIT HIGH RESULT BYTE!
P 0004 CF          22                RCF
P 0005 C0 E2      23 LOOP:       RRC      PRODUCT_HI
P 0007 C0 E3      24                RRC      PRODUCT_LO
P 0009 FB 02      25                jr      NC,NEXT
P 000B 02 21      26                ADD      PRODUCT_HI,MULTIPLIER
P 000D 0A F6      27 NEXT:       djnz     COUNT,LOOP
P 000F AF          28                ret
P 0010            29 END      MULT
          30 END      ARITH
```

0 errors
Assembly complete

9 instructions
16 bytes
92.5 μ s (average)

9.4 Divide. The following module illustrates an efficient algorithm for the division of a 16-bit unsigned value by an 8-bit unsigned value, resulting in an 8-bit unsigned quotient. The algorithm repetitively shifts the dividend left (via RLC). If the high-order bit shifted out is a one or if the resulting high-order dividend byte is greater than or equal to the divisor, the

divisor is subtracted from the high byte of the dividend. As the low-order bits of the dividend are vacated by the shift left, the resulting partial-quotient bits are rotated in. Thus, the quotient and the low byte of the dividend occupy the same byte, which saves register space, code, and execution time.

9. Arithmetic Routines
(Continued)

```

Z8ASM      2.0
LOC      OBJ CODE      STMT SOURCE STATEMENT

1 ARITH      MODULE
2 CONSTANT
3 COUNT      :=      R0
4 DIVISOR    :=      R1
5 DIVIDEND_HI :=      R2
6 DIVIDEND_LO :=      R3
7 GLOBAL
P 0000      8 DIVIDE      PROCEDURE
9 *****
10 Purpose =      To perform a 16-bit by 8-bit unsigned
11                binary division.
12
13 Input =      R1 = 8-bit divisor
14              RR2 = 16-bit dividend
15
16 Output =      R3 = 8-bit quotient
17              R2 = 8-bit remainder
18              Carry flag = 1 if overflow
19              = 0 if no overflow
20 *****
21 ENTRY
P 0000 OC 08 22          ld      COUNT,#8          !LOOP COUNTER!
23
24 !CHECK IF RESULT WILL FIT IN 8 BITS!
P 0002 A2 12 25          cp      DIVISOR,DIVIDEND_HI
P 0004 BB 02 26          jr      UGT,LOOP          !CARRY = 0 (FOR RLC)!
27 !WON'T FIT.  OVERFLOW!
P 0006 DF 28          SCF                      !CARRY = 1!
P 0007 AF 29          ret
30
31 LOOP:      !RESULT WILL FIT.  GO AHEAD WITH DIVISION!
32          RLC      DIVIDEND_LO          !DIVIDEND * 2!
33          RLC      DIVIDEND_HI
34          jr      c,subt
35          cp      DIVISOR,DIVIDEND_HI
36          jr      UGT,next          !CARRY = 0!
37 subt:      SUB      DIVIDEND_HI,DIVISOR
38          SCF                      !TO BE SHIFTED INTO RESULT!
P 0015 0A F1 39 next:      djnz     COUNT,LOOP          !no flags affected!
40
41 !ALL      DONE!
42          RLC      DIVIDEND_LO
43
44          ret
P 0019 AF 44          ret
P 001A 45          END DIVIDE
46          END ARITH

0 ERRORS
ASSEMBLY COMPLETE

15 instructions
26 bytes
124.5 µs (average)

```

SECTION 10

Conclusion

This Application Note has focused on ways in which the Z8 microcomputer can easily yet effectively solve various application problems. In particular, the many sample routines

illustrated in this document should aid the reader in using the Z8 to greater advantage. The major features of the Z8 have been described so that the user can continue to expand and explore the Z8's repertoire of uses.

Z80[®] 8-Bit Microprocessor Family 2

Zilog
Zilog
Zilog
Zilog
Zilog

Get powerful microprocessor performance by using the Z80. With 158 instructions it offers more flexibility than other μ Ps, plus 8080 code compatibility.

The Z80 8-bit microprocessor combines all the processing power of the 8080 with 80 additional instructions. And to keep chip count to a minimum, many of the peripheral circuits necessary for 8080 systems have been built into the Z80. All members of the Z80 family are built with n-channel, silicon-gate, depletion-load technology; function at single-phase clock rates of 4 MHz; require just a 5-V supply; and have TTL-compatible inputs and outputs.

The circuit family consists of the Z80-CPU and the following peripherals: a counter-timer circuit (CTC), a parallel input/output circuit (PIO), a direct-memory-access controller (DMA), and a serial input/output circuit (SIO), as well as a group of support boards (Table 1). All the circuits are available in 2.5 or 4-MHz versions, ceramic packages, and extended temperature ranges. All are housed in 40-pin DIPs, except the CTC, which comes in a 28-pin DIP.

All peripheral circuits can be daisy-chained for priority interrupt control. Since most peripheral circuits necessary for system operation are built into the Z80, a minimum system consists of the Z80, a system clock, a power-on reset circuit and any memory and peripheral circuits desired (Fig. 1). At the system level, the μ P supports vectored priority-interrupt structures without any extra hardware.

Interfaces to the Z80 are simple

Although the Z80 maintains timing and control-signal compatibility with the 8080, it is not pin-compatible. All output lines can sink 1.8 mA at 0.4 V—the equivalent of one standard TTL load.

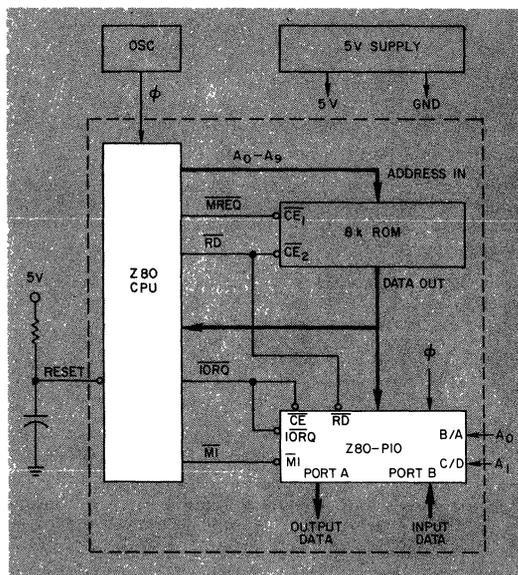
Three major buses from the chip—the 16-bit address bus, the 8-bit bidirectional data bus and a 13-line control bus—account for 37 of the Z80's 40 pins (Fig. 2). The other three pins are for power, ground and the single-phase clock. Unlike the 8080, the Z80 needs no status latch or clock, and interrupt vectoring and dynamic-memory refresh are completely supported within the μ P itself.

The 13 control lines are actually subdivided into three control buses: system control (six lines), μ P

Table 1. Z80 system components

Part #	Description	Price ** 100 qty
Z80-CPU	8-bit CPU, 2.5 MHz *	\$26.50
Z80-CTC	Counter/timer, 2.5 MHz *	\$17.00
Z80-PIO	Parallel I/O, 2.5 MHz *	\$11.00
Z80-DMA	Direct mem. access, 2.5 MHz *	\$38.00
Z80-SIO	Serial I/O, 2.5 MHz *	N.A.
Support boards		unit qty
MCB	Microcomputer board—kit	\$435.
	—assembled	\$495.
MDC	Memory/floppy-disc controller	\$795.
RMB	RAM memory board	\$750.
IOB	Input/output board	\$350.
PMB	PROM/ROM memory board	\$395.
PPB/ EPROM	EPROM programmer (for 2708)	\$475.
PPB/ PROM	PROM programmer (for 7620, 7640)	\$475.
CPB/ ROM	Combination programmer	\$575.
VDB	Video-display board	\$475.

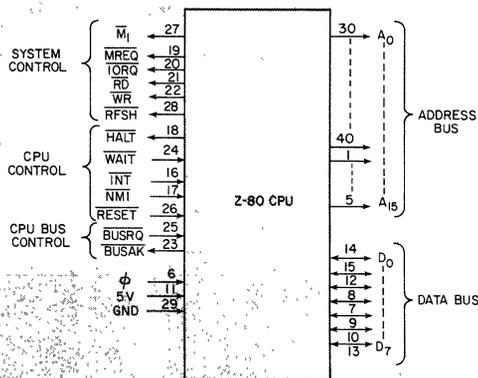
* 4-MHz versions of these parts are available.
** 0 to 70-C ratings in plastic packages



1. A minimal Z80 system can be built with the μ P, an oscillator, some memory and an I/O port such as the PIO. Just a power supply and reset circuit must be added.

Authors: Ralph Ungerman, (former Zilog Vice President); Bernard Peuto (Director of Engineering).

Reprinted with permission of Electronic Design.



2. The three major buses on the Z80 are an address bus, a data bus and a control bus. The control bus can be split into three smaller buses—one for system control, one for processor control and one for bus control.

control (five lines), and μ P-bus control (two lines). One bus-control line functions as a bus-request line (\overline{BUSRQ}), which is an input that requests not only the μ P's address and data buses, but also the memory-request, I/O-request, read-data and write-data lines of the system-control bus to go to a high-impedance state so that other devices can use the bus. The other bus-control line, an output signal called bus-acknowledge (\overline{BUSAK}), goes high to indicate when the lines go into a high-impedance third state.

All six system-control signals are outputs from the μ P. An \overline{M}_1 line (machine cycle 1) goes Low to indicate when the μ P is in the op-code-fetch part of an instruction. The memory-request line (\overline{MREQ}) goes Low when the address bus holds a valid address for a memory-read or write operation. An I/O-request line (\overline{IORQ}) goes Low to indicate that the lower byte of the address bus holds a valid I/O-port address for an I/O-read or write operation.

Memory-read and memory-write lines (\overline{RD} and \overline{WR}) are also active when Low. \overline{RD} indicates that the μ P wants to read data from a memory or I/O device, while \overline{WR} indicates that the data bus holds data to be stored in the addressed location. When the sixth system-control line, a refresh signal (\overline{RFSH}), goes Low, it

indicates that the lower seven bits of the address bus contain a refresh address for dynamic memories, so the current \overline{MREQ} signal should be used to do a refresh read to all dynamic memory.

The five μ P-control lines consist of one output signal and four input lines. All lines are active when Low. The only output is the halt line, which indicates when the μ P has executed a software HALT instruction and is waiting for either a nonmaskable or maskable interrupt. While halted, the μ P automatically executes NOP instructions to maintain the memory refresh. The wait input (\overline{WAIT}) indicates to the μ P that the addressed memory or I/O device isn't ready for a data transfer (the μ P will enter wait states for as long as this line is Low). This line allows memory or peripheral of any speed to be synchronized with the Z80.

To reset the μ P or initialize it once it is on, the \overline{RESET} line can be pulled Low. When pulled Low, it forces the Z80's program counter to 00_{16} , disables the interrupt-enable flip-flop, sets register I to 00_{16} , sets register R to 00_{16} , and sets the interrupt node to \emptyset .

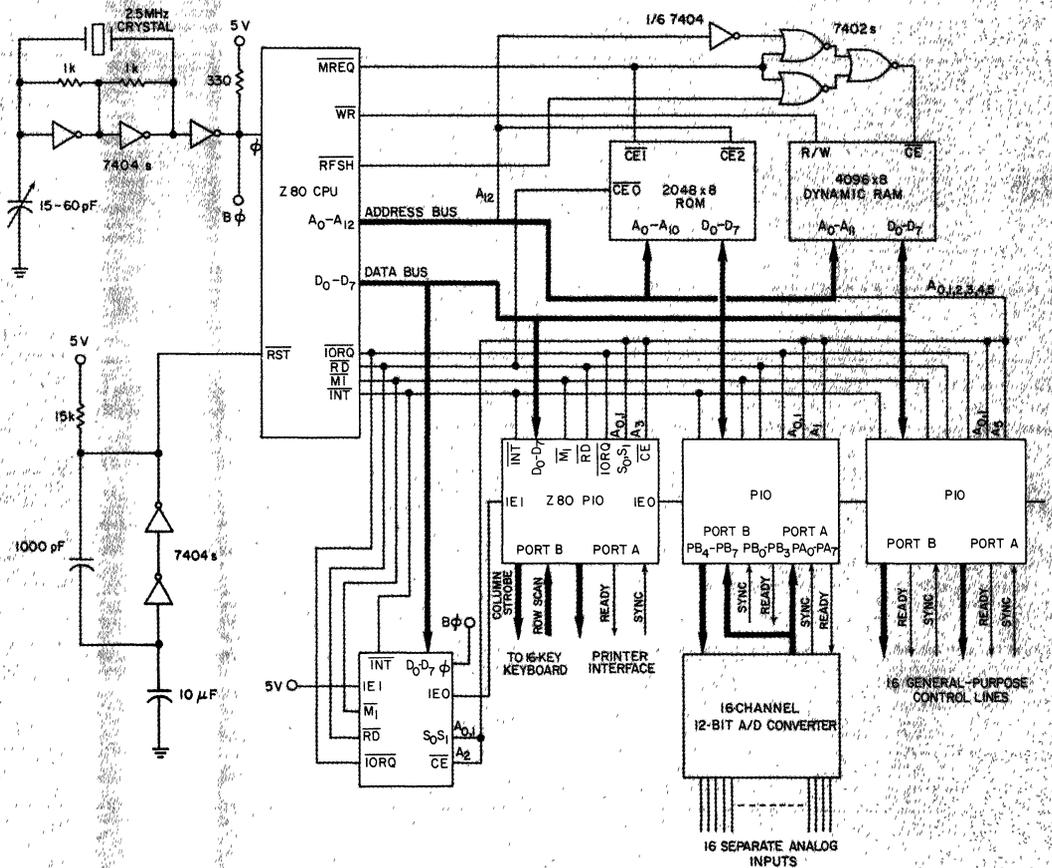
The last two lines are the interrupt-request (\overline{INT}) and nonmaskable-interrupt (\overline{NMI}) inputs. When pulled Low, the \overline{INT} line interrupts the processor at the end of the current instruction if the software-controlled interrupt-enable flip-flop (IFF) is enabled, and if the \overline{BUSRQ} line is High. Each time the μ P accepts an interrupt, an acknowledge signal (\overline{IORQ} during an \overline{M}_1 time) is sent out at the beginning of the next instruction cycle.

The \overline{NMI} line is a negative-edge triggered input, has a higher priority than the \overline{INT} line, and is recognized at the end of the current instruction regardless of the IFF state. When triggered, it forces the Z80 to begin execution at location 0066_{16} after saving the current contents of the program counter in an external stack.

Interrupts and flags add flexibility

Three interrupt modes are available to the programmer. Mode 0 permits the interrupting device to insert any instruction on the data bus and have the μ P execute it. Mode 1 has the μ P automatically execute a restart to location 0038_{16} —no external hardware is required (the contents of the program counter are pushed onto the internal stack).

Mode 2, the most powerful, permits an indirect call



3. By daisy-chaining the peripheral support circuits, any number of peripheral chips can be added to this Z80-based process-control system. The device closest to the μ P has

the highest priority interrupt. Just 16 IC packages are needed to build this data-acquisition subsystem; and of the 16, nine are memories.

to any memory location. In this mode, the μ P forms the indirect address from the upper byte of the I register and eight bits that are supplied by the interrupting device.

Two identical 8-bit flag registers (F and F') are part of the Z80. Six of the bits in each register can be used as conditions for jump, call or return instructions; they are set or reset by various μ P operations. Both the F and F' registers have four testable flag bits and two nontestable bits. The four testable bits are the Carry flag, Zero flag, Negative-sign flag and Parity/overflow flag.

The Carry flag contains carry from the highest-order accumulator bit—add, subtract, shift and rotate instructions can alter its state. If an operation loads a zero into the accumulator, the Zero flag gets set. Otherwise, it is reset. Used with signed numbers, the Negative-sign flag gets set if the result of an operation is negative (bit 7 of the accumulator is the sign bit). The dual-purpose Parity/overflow bit gets set when the parity of the result in the accumulator for a logic

operation is even, or is used to indicate overflow when signed 2's complement arithmetic is performed.

The two nontestable bits are Half-carry and Subtract flags. The Half-carry flag is a BCD-carry or borrow result from the least-significant four bits of the operation. (When a DAA instruction is used, this flag corrects the result of a previously packed decimal-add or subtract operation.) The Subtract flag corrects BCD operations by helping identify the previous instruction; The correction differs for addition and subtraction.

Shifting operations can be performed on any register or memory location rather than just on the accumulator. What's more, I/O operations can also be done with any register, rather than just the accumulator. Sixteen-bit direct loads and stores can be sent to the BC-register pair, the DE pair or the IX or IY registers—instead of just the HL as in the 8080. Consequently, the number of exchange and register-move operations is reduced considerably. Also, 16-bit arithmetic operations using the HL pair

Z80 microprocessor architecture

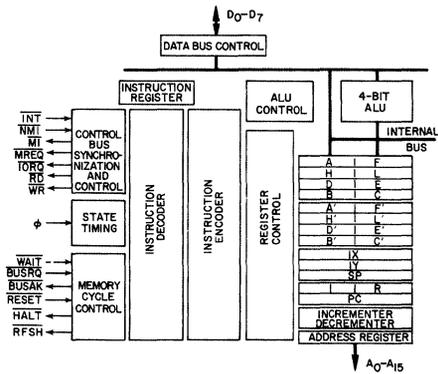
Built into the Z80 microprocessor are all bus-control, memory-control, and timing signals in addition to eight general-purpose 16-bit registers and an arithmetic-and-logic unit (ALU). The Z80 is upward-compatible with the Intel 8080A¹ and 8085 μ Ps.

All the 8080 registers are duplicated within the Z80 and, in addition to the eight 8-bit registers (A, F, B, C, D, E, H and L) of the 8080, there is an alternate set (A', F', B', C', D', E', H' and L') and several other special-purpose registers. The additional registers include two 16-bit index registers (IX and IY), an 8-bit interrupt-vector register (I) and an 8-bit memory-refresh register (R). Also carried forward from the 8080 register set are the 16-bit stack pointer and the 16-bit program counter (PC).

Normally, all instructions reference the main register set, and alternate registers are accessed via two exchange commands that swap register contents in the banks. One command, exchanges the accumulator and register flags, while another instruction, exchanges the other six general-purpose registers. Since both instructions are single-byte, minimum-execution-time instructions, a complete swap can be done in four clock cycles (1 μ s for a 4-MHz clock). These commands and registers are very handy for rapid single-level interrupt handling.

The Z80's two index registers have no direct corollary in the 8080 architecture, but in operation they resemble the single index register in the 6800 μ P.² Instructions using this mode such as the accumulator-load command [LD A, (IX + 7)] contain a single-byte offset field (+7, in this case). The effective address of the operand is the sum of the offset and the IX-register contents. This addressing mode is particularly convenient for table references, multibyte entries or for passing a pointer to a group of subroutine parameters. The offset byte is interpreted by the Z80 as a 2's complement number, so both positive and negative indexing is possible.

A special feature of the Z80 is its ability to refresh dynamic memory automatically. Its memory-refresh register acts as a 7-bit counter that is incremented after every op-code fetch. After the fetch, the R-



register contents are loaded onto the low-order seven bits of the address bus, and a status line on the processor goes low to indicate the presence of a valid refresh count. Because this entire process takes place while the op code is decoded internally, it never interferes with any other μ P activity on the bus.

The I register forms the high-order eight bits of an address. When an interrupt occurs and the Z80 is in the vectored mode, the lower order eight bits are supplied by an interrupting peripheral. In response to the interrupt, the μ P does an Indirect Call instruction with the composite address. All the support chips have corresponding registers that store the low-order eight bits and supply them to the Z80 when the interrupt is acknowledged.

able to perform 12 basic operations—add, subtract, AND, OR, Ex-OR, compare, test-bit, reset-bit, set-bit, increment, decrement, and left or right-shift and rotate (arithmetic or logic)—the ALU communicates with the registers and external-data bus by means of a buffered internal bus. As each instruction is fetched from memory, it is loaded into the instruction register and decoded by the control section, which supplies all the control signals for the Z80's subsystems.

have been expanded over the 8080's to include add with carry and subtract with borrow.

Software gives the Z80 horsepower

Many of the instructions available only in the Z80 support the manipulation of multibyte blocks of data—a great plus in data communications and text manipulation. For instance, a block-move instruction takes data from the memory location specified by the HL-register pair, deposits them in the location specified by the DE pair, increments the HL and DE registers and then decrements the BC pair, which is assumed to hold a byte counter for the operation. This

instruction can be executed in a single cycle or repeat sequence. Decrementing the HL and DE addresses is also possible.

By using the block move command, the μ P can transfer bytes of data at 5.25 μ s/byte (for a 4-MHz clock). Block operations are also available for memory searches and I/O operations. And shift and rotate operations have been enhanced. For decimal arithmetic, 4-bit shifts through the accumulator can greatly speed up BCD multiplication and division, and bit-manipulation instructions permit fast access to any bit in either the external memory or an internal register.

Other enhancements of the instruction set include

Software capabilities of the Z80

Able to execute over 150 different instructions, including all 78 of the 8080A command set, the Z80 features seven basic families of instructions: load-and-exchange, block-transfer-and-search, arithmetic and logic, bit-manipulation (set, reset and test), jump, call-and-return, input/output, and basic μ P-control commands. In all, the Z80 can recognize 696 op codes—244 are the codes of the 8080A.

Load instructions move data internally between μ P registers or between the registers and external memory. All these instructions must specify a source location, from which data are to be moved, and a destination location. Block-transfer instructions permit any block of memory to be moved to any other location. Search commands let any block of external memory be examined for any 8-bit character. Once the character is found, the instruction is terminated.

The ALU instructions operate on data held in the accumulator and other general-purpose registers or external memory. Results are held in the accumulator, and appropriate flags are set. Bit-manipulation commands allow any bit in the accumulator, any general-purpose register or any external memory location to be set, reset or tested with a single instruction. Jump, Call and Return instructions are used to transfer between various locations in the program.

I/O instructions permit a wide range of transfers between external memory locations or general-purpose Z80 registers and external I/O devices. In either case, the port number is provided on the lower eight bits of the address bus during any I/O operation. Also, the basic μ P-control commands include such instructions as setting or resetting the interrupt-enable flip-flop or setting the mode of interrupt response.

In addition to the seven addressing modes of the 8080—direct, register, register indirect, modified page \emptyset , extended, implied and immediate—the Z80 has three more addressing modes: relative, indexed, and bit addressing—that can be used.

A special byte-call instruction lets the Z80 program proceed to any of eight locations in page \emptyset of the memory. This modified page \emptyset addressing allows a single byte to specify a complete 16-bit address, which saves memory space.

Relative addressing lets the Z80 use the byte following the op code to specify a displacement from the current program-counter value. The displacement value is in 2's-complement form, which permits up to a +127 or -128 byte displacement. Extended addressing includes two bytes of address in the instruction.

Index registers can also be used as part of the address. In the indexed addressing mode, a byte of data following the op code is a displacement value that must be added to the specified index register (the op code indicates which register) to form a memory pointer. Also available is an implied addressing mode in which the op code uses the contents of one Z80 register or more as the operands. The last addressing mode lets the Z80 access any memory location or μ P register and permits any bit to be set, reset or tested.

Mnemonic	Description
8-bit load instructions	
LD r, r'	Load register r with r'
LD r, n	Load register r with n
LD r, (HL)	Load r with location (HL)
LD r, (IX+d)	Load r with location (IX+d)
LD r, (IY+d)	Load r with location (IY+d)
LD (HL), r	Load location HL with r
LD (IX+d), r	Load location IX+d from register r
LD (IY+d), r	Load location IY+d from register r
LD (HL), n	Load location HL with value n
LD (IX+d), n	Load location IX+d with n
LD (IY+d), n	Load location IY+d with n
LD A, (BC)	Load AC with location BC
LD A, (DE)	Load AC with location DE
LD A, (nn)	Load AC with location nn
LD (BC), A	Load location BC with AC
LD (DE), A	Load location DE with AC
LD (nn), A	Load location nn with AC
LD A, I	Load register A from I
LD A, R	Load AC with register R
LD I, A	Load register I with AC
LD R, A	Load register R with AC
16-bit load instructions	
LD dd, nn	Load registers dd with nn
LD IX, nn	Load register IX with nn
LD IY, nn	Load register IY with nn
LD HL, (nn)	Load L with contents of location nn and H with (nn+1)
LD dd, (nn)	Load registers dd with location nn
LD IX, (nn)	Load IX with location nn
LD IY, (nn)	Same but for IY
LD (nn), HL	Load location nn with HL
LD (nn), dd	Load location (nn) with register pair dd
LD (nn), IX	Same but for IX
LD (nn), IY	Same but for IY
LD SP, HL	Load stack pointer from HL
LD SP, IX	Load stack pointer from IX
LD SP, IY	Load stack pointer from IY
PUSH qq	Load register pair qq onto stack
PUSH IX	Load IX onto stack
PUSH IY	Load IY onto stack
POP qq	Load register pair qq with top of stack
POP IX	Load IX with top of stack
POP IY	Load IY with top of stack
Exchange, transfer and search instructions	
EX DE, HL	Exchange contents of DE & HL
EX AF, A' F'	Exchange contents of AF & A' F'
EXX	Exchange all six general purpose registers with alternates
EX (SP), HL	Exchange stack pointer contents with HL contents
EX (SP), IX	Same but use IX register
EX (SP), IY	Same but use IY register
LDI	Load (HL) into DE, increment DE and HL, decrement BC
LDIR	Same but loop until (BC) = 0
LDD	Load location (PE) with location (HL) and decrement DE, HL and BC
LDDR	Same but loop until (BC) = 0
CPI	Compare contents of AC with (HL), set Z flag if =, increment HL and decrement BC
CPDR	Same but repeat until BC = 0
CP s	Compare operand s with AC
CPD	Same as CPI but decrement HL
CPDR	Same as CPDR but decrement HL
8-bit arithmetic and logic instructions	
ADD A, r	Add contents of r to AC
ADD A, n	Add byte n to AC
ADD A, (HL)	Add contents of HL to AC

ADD A, (IX+d)	Add location (IX+d) to AC
ADD A, (IY+d)	Same but (IY+d)
ADC A, s	Add with carry operand s to AC
SUB s	Subtract contents of r, n, HL, IX+d or IY+d from AC
SBC s	Same but also subtract carry flag
AND s	Logic AND of operand s and AC
OR s	Same but OR with AC
XOR s	Same but EX-OR with AC
INC r	Increment register r
INC (HL)	Increment location (HL)
INC (IX+d)	Same but use (IX+d)
INC (IY+d)	Same but use (IY+d)
DEC m	Decrement operand m

16-bit Arithmetic instructions

ADD HL, ss	Add register pair ss to HL
ADC HL, ss	Same but include carry flag
SBC HL, ss	From HL subtract contents of ss and carry flag
ADD IX, pp	Add register pair pp to IX
ADD IY, rr	Same but use rr and IY
INC ss	Increment register pair ss
INC IX	Increment IX register
INC IY	Same but IY register
DEC ss	Decrement register pair ss
DEC IX	Same but IX register
DEC IY	Same but IY register

General purpose arithmetic & control instructions

DAA	Decimal adjust accumulator
CPL	Complement (AC)
NEG	Complement (AC) and add 1
CCF	Complement carry flag
SCF	Set carry flag = 1
NOP	No operation
HALT	Halt, wait for interrupt or reset
DI	Disable interrupts
EI	Enable interrupts
IM0	Set μP to interrupt mode 0
IM1	Set μP to interrupt mode 1
IM2	Set μP to interrupt mode 2

Rotate and shift instructions

RLCA	Rotate AC left
RLA	Same but include carry flag
RRCA	Rotate AC right
RRA	Same but include carry flag
RLC r	Rotate register r left
RLC (HL)	Rotate location (HL) left
RLC (IX+d)	Same but location (IX+d)
RLC (IY+d)	Same but location (IY+d)
RL m	Same as any RLC but include carry flag
RRC m	Same as RLC but shift right
RR m	Same as RL m but shift right
SLA s	Shift left (any RLC register)
SRA s	Same but shift right and keep MSB
SRL s	Same as SLA but shift right
RLD	Simultaneous 4-bit rotate from AC_L to L, L to H and H to AC_L
RRD	Simultaneous 4-bit rotate from AC_L to H, H to L and L to AC_L

Bit set, reset and test instructions

BIT b, r	Test bit b of register r
BIT b, (HL)	Test bit b of location (HL)
BIT b, (IX+d)	Test bit b of location (IX+d)
BIT b, (IY+d)	Test bit b of location (IY+d)
SET b, r	Set bit b in register r to 1
SET b, (HL)	Same but use contents of location HL
SET b, (IX+d)	Same but use contents of location IX+d
SET b, (IY+d)	Same but use contents of location IY+d
RES b, s	Reset bit b of operand m

Jump, c all and return instructions

JP nn	Unconditional jump to location nn
JP cc, nn	If condition cc True, do a JP nn otherwise continue
JR e	Unconditional jump to PC+e
JR C, e	If C = 0 continue. If C = 1 do JR e
JR NC, e	Reverse of JR c, e
JR Z, e	If Z = 0 continue. If Z = 1 do JR e
JR NZ, e	Reverse of JR Z, e
JP (HL)	Load PC from (HL)
JP (IX)	Load PC from (IX)
JP (IY)	Load PC from (IY)
DJNZ, e	Decrement register B and jump relative if B = 0
CALL nn	Unconditional call subroutine at location nn
CALL cc, nn	Call subroutine at location nn if condition cc is True
RET	Return from subroutine
RET cc	If cc false continue, otherwise do RET
RETI	Return from interrupt
RETN	Return from nonmaskable interrupt
RST p	Store PC in stack, load 0 in PC_H and restart vector in PC_L

Input/output instructions

IN A, n	Load AC with input from device n
IN r, (C)	Load r with input from device C
INI	Store contents of location specified by C in address specified by HL, decrement B and increment HL
INIR	Same but repeat until B = 0
IND	Same as INI but decrement HL too
INDR	Same as INIR but decrement HL too
OUT n, A	Load output port (n) with AC
OUT (C), r	Load output port (C) with register r
OUTI	Load output port (C) with location (HL) and increment HL and decrement B
OTIR	Same but repeat until B = 0
OUTD	Same as OUTI but decrement HL
OTDR	Same as OTIR but decrement HL

Notes

b represents a 3-bit code that indicates position of the bit to be modified

cc represents a 3-bit code that indicates which of eight condition codes are to be used

d is an 8-bit offset value

dd refers to register pairs BC, DC, HL or the stack pointer

e represents a signed two's complement number between -126 and +129

m is an 8-bit number

n is an 8-bit number

nn refers to two 8-bit bytes

p represents one of eight restart vector locations on page 0

pp refers to register pairs BC, DE, the IX register or the stack pointer.

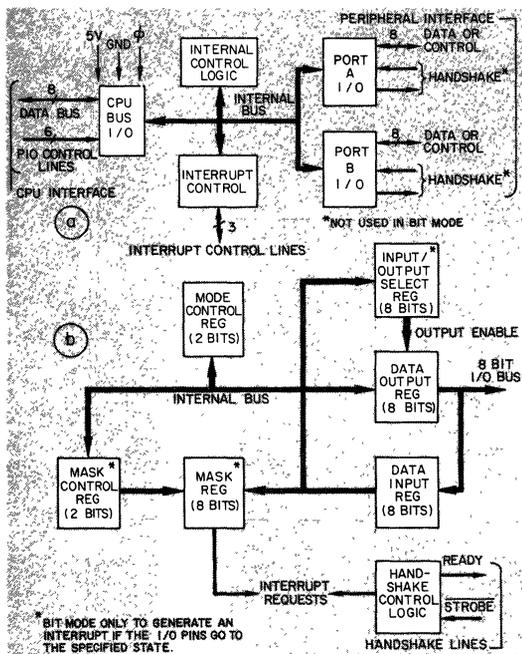
qq refers to register pairs AF, BC, DE or HL

r or r' refers to registers A, B, C, D, E, H or L or their alternates

rr refers to register pairs BC, DE, the IY register or the stack pointer

s refers to either the r registers, the n data word or the contents of locations specified by the contents of the HL, IX+d or IY+d registers

ss refers to register pairs BC, DE, HL or the stack pointer



4. With two parallel, 8-bit I/O ports, the PIO circuit (a) can use either of the ports in a parallel system or on a line-by-line basis for 16 separate I/O lines. Inside each port, five control registers are loaded by the Z80 before operation to initialize the port (b).

the decimal-adjust command, which now works after subtract as well as add operations. Negate-instructions and looping commands are also part of the set. The looping instruction decrements the B register and takes a relative branch if that register has not reached zero. Other operations are shown in the box on Z80 software (see page 58).

Put the Z80 to work

With the four basic Z80 peripheral circuits described virtually any high-performance microcomputer can be constructed. For example, a process-control system can be built around the Z80, as shown in Fig. 3. The peripherals handled by the Z80 controller include three parallel input/output circuits and one counter/timer. The PIOs handle a 16-key keyboard, a printer, a multichannel a/d converter and 16 control lines. Because the peripheral chips can be daisy-chained, a priority interrupt structure can be formed with little or no software or hardware overhead. Using the interrupt mode, the requesting PIO causes the μP to go to a service routine, and, after the routine, a special instruction—return-from-interrupt—goes back to the PIO and allows the μP to service lower-priority interrupts.

All support chips have two lines for daisy-chaining—the Interrupt-enable-in (IEI) and Interrupt-enable-out (IEO). Since a CTC is used in the controller to

relieve the Z80 from doing timing loops, software overhead is minimized. For the controller of Fig. 3, 14 ICs are needed—and nine of them are memories (2048 bytes of ROM and 4096 bytes of RAM).

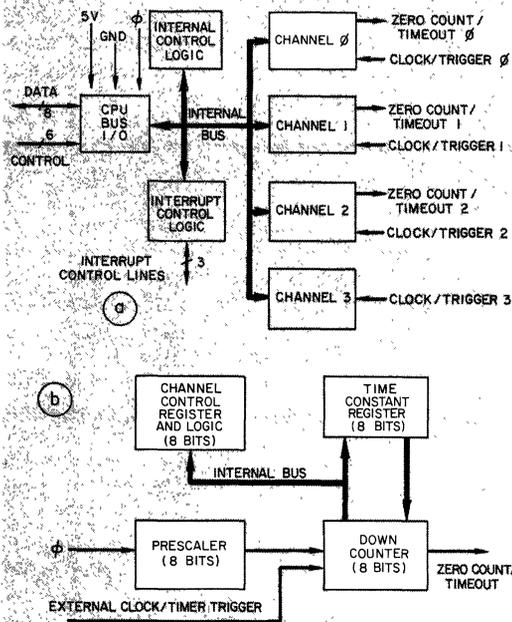
The Z80-PIO, a parallel-interface controller, has two 8-bit ports and provides TTL-compatible interfaces (Fig. 4a). Port A has four possible modes of operation: byte output, byte input, byte bidirectional bus and bit. Port B has all the modes except byte bidirectional. The port I/O logic consists of handshake control and six registers (Fig. 4b): an 8-bit input register, an 8-bit output register, a 2-bit mode-control register, an 8-bit mask register, an 8-bit I/O-select register and a 2-bit mask-control register. The last three are used only when the port is programmed to operate in the bit mode. Of the 40 pins on the PIO, 24 are required by the port and CPU buses, six more for μP interfacing, three for interrupt control, four for handshaking the I/O ports and three for power, ground and the single-phase clock.

Four of the six internal registers are loaded by the Z80 for characteristic programming. The contents of the 2-bit mode-control register determine which of the four PIO operating modes is to be used. Similarly, the 2-bit mask-control register specifies the active state (High or Low) of any peripheral-interface lines which are to be monitored. It also permits an interrupt to be generated when all unmasked pins are active (AND condition) or when any unmasked pin is active (OR condition). The code loaded into the mask register determines which peripheral-device interface pins are to be monitored for the specified status condition. And the code held in the I/O-select register determines which pins are inputs or outputs during bit-mode operation. The other two registers hold incoming or outgoing data.

To relieve some software overhead in timing situations, the CTC provides four channels of programmable timing and counting functions that can be set with software (Fig. 5). Each channel operates in either a timer or counter mode, and programmable interrupts can occur on counter or timer states. Other features include a readable down counter, a selectable 16 or 256 clock prescaler for each timer, a selectable positive or negative trigger for timer initiation and automatic reload of counter or timer constants. In addition three channels have zero count/timeout outputs capable of driving Darlington transistors.

Each channel has two registers, both eight bits long and loaded by the μP . One register, the time-constant register, loads the preset value into the down counter. The other, called a channel-control register, contains the mode and condition information for channel operation. Also included in each channel are an 8-bit down counter and an 8-bit prescaler. The counter is decremented by the prescaler in the timer mode and by the clock-trigger input in the counter mode.

Of the 28 pins on the CTC, eight connect to the data bus, seven to the control lines, three handle interrupt control and three are required for power, ground and



5. Each CTC provides four channels of counting/timing capability with an 8-bit counter on each channel (a). There is a control register for each channel and a programmable 8-bit prescaler (b).

the single-phase clock. Three of the four input channels have one input and one output line and the fourth channel has only an input line.

Speed up data transfer with DMA

One of the interface circuits, a direct-memory-access controller, is designed to effect the high-speed transfer of a block of data between any two ports in a Z80 system and can also be used with other μ Ps. The circuit is a programmable, single-channel device that provides all address, timing and control signals for the data transfer (Fig. 6). Also, the DMA circuit can search a block of data for a particular, bit-maskable byte, with or without transferring the data. Capable of transfer-only, search-only or search-and-transfer operations at up to 1.2 Mbyte/s, the circuit can automatically increment or decrement the port address from a programmed starting address.

Four communications modes are available on the chip—a byte-at-a-time mode that transfers one byte per request, a burst mode that lets the transfer continue as long as ports are ready, a continuous mode that locks out the μ P until the operation is completed, and a transparent mode that steals refresh cycles. When the circuit finds a match or finishes a transfer, it can be programmed to generate an interrupt. Or a complete repeat cycle can be programmed for automatic repeat or repeat on command. A built-in block counter can generate a signal when a certain

number of bytes has been transferred—without halting the transfer.

Inside the DMA controller are bus-interface circuits for both the data and address buses, logic and registers to control parameters of the circuit, and address and byte-count circuitry to generate port addresses. There are also provisions for incrementing or decrementing the address, timing circuitry for adjusting the read/write timing of both ports being addressed, and compare logic that permits a byte-matching operation (if a match is encountered, a flag is set in the DMA's status register). Also built-in is the interrupt and $\overline{\text{BUSRQ}}$ logic, which includes a control register that specifies conditions for the chip to generate an interrupt, all the priority-encoding logic to select between generation of an $\overline{\text{INT}}$ or $\overline{\text{BUSRQ}}$ output, and an interrupt-vector register for automatic vectoring to an interrupt-service routine.

Of the 40 pins on the DMA controller, 24 are needed for the address and data bus, and five are needed for the μ P control bus. Eight more handle the interrupt control and timing, and three more are necessary for power, ground and clock inputs.

For serial communications, the serial-input/output circuit (SIO) provides two full duplex programmable channels capable of handling asynchronous, synchronous, and synchronous-bit protocols (IBM Bisync, HDLC and SDLC). It can also generate cyclic-redundancy check codes in any synchronous mode. The SIO has four independent serial ports—two for transmitting and two for receiving (Fig. 7). Asynchronous data with 5, 6, 7 or 8 bits and 1, 1- $\frac{1}{2}$ or 2-stop bits as well as even, odd or no-parity generation or checking can be handled.

The circuit has $\times 1, 16, 32$ and 64 clock modes and data rates from 0 to 600 kHz. The transmitter sections have eight modem-control lines, quadruple buffers on receiver data and error registers, and double buffers on the transmitter sections. The bus-I/O control block includes the logic for selecting channels and registers, read/write control, and control of special timing for interrupt-acknowledge cycles. Interrupt logic includes the daisy-chain provision as well as two special 8-bit control registers to handle the various interrupt options, as well as an 8-bit vector register for interrupt response.

Three receive buffers allow enough time for interrupt servicing of fast data rates. The receiver-shift register is controlled by the receive-control logic, which includes two 8-bit registers for receive-mode selection and options. There are two more 8-bit registers for programmable-sync characters. The external-status register is an 8-bit, read-only register that indicates the state of the modem-control pins as well as several internal-status conditions. An internal-status register also indicates the state of the SIO. Each channel has its own receive, transmit and status-register banks.

Now that you are familiar with all the basic system-building blocks, you can mold them with software into

a working system. Because of the Z80's rich instruction set, assembling software programs by hand can be too complicated for most applications; you should use either a dedicated development system or time-sharing service.

Development systems speed software

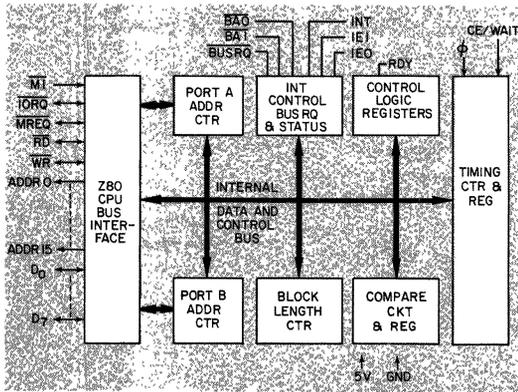
The Z80 development systems and the software available from Zilog include several large dedicated units that permit hardware or software development, or both (Table 2). Also available are assemblers, compilers and time-sharing services as well as Basic and PLZ. (Cobol and Fortran will be available soon.) All program statements in the development systems

are handled by a text editor and stored in a dual floppy-disc file management system. Once filed, the program is ready for testing and can be translated by an assembler or compiler into code for the Z80. The code can be tested by a hardware/software debug package that provides interrogation, control and tracing capabilities.

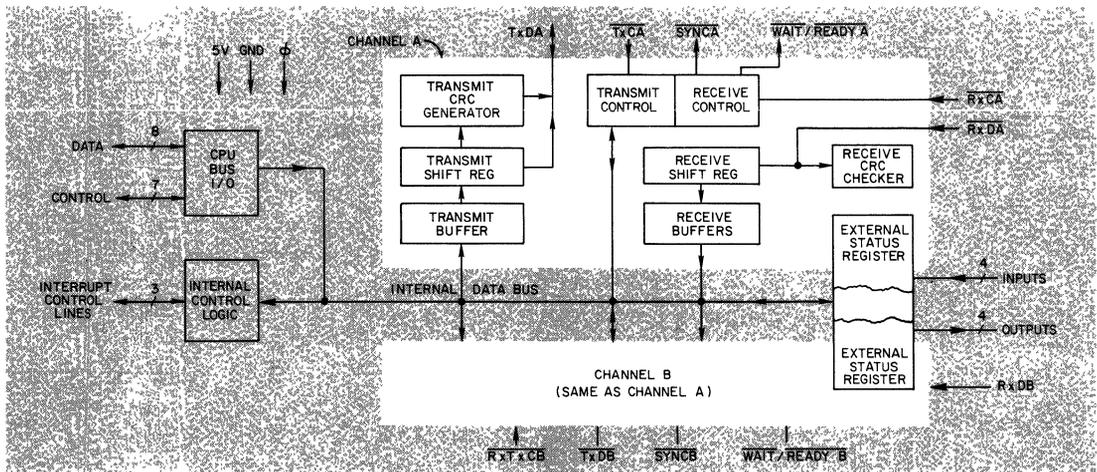
In the monitor mode the system has four operating environments: file, edit, debug and assemble. The file capabilities are pretty standard types of features—storing records on disc, pulling records from disc, changing records and saving the new results. The debug and assembler features of the development system offer some pretty powerful capabilities. With the debug commands, you can set up breakpoints, compare blocks of memory and trace an operation.

In the debug mode, for instance, system transactions can be loaded into a special memory as the program executes in real time. And, once any user-defined condition has occurred (such as the setting of bit 6 of port 8B₁₆ or reading from address 21C8₁₆), the program execution can be suspended and the system can re-enter the monitor mode. A complete record of the last 256 transactions just prior to program termination is in the system memory and available to the user.

The main assembler in the development system supports the following features: macros, conditional assembly, the ability to assemble a large file and a sorted-symbol table with cross reference. All these options as well as the printing and listing options are available by setting parameters at the time of assembly. A relocatable assembler with I/O management provides relocatable code and has a linking loader. These permit you also to specify other files that should be included within the current file being



6. The direct-memory-access controller has three classes of operation: transfer-only, search-only or search-and-transfer. Any device on the system bus can be controlled by the DMA; internal counters keep track of source and destination addresses.



7. Two independent full-duplex serial I/O channels are built into the SIO. Either channel can be programmed to

operate in asynchronous or synchronous modes, including BiSync and HDLC/SDLC.

Table 2. Hardware and software support

Type	Price unit qty.	Name	Description
Systems	\$8990	Z80-hardware & software development system	3 kbytes ROM, 1 kbyte RAM for system monitor; 16 kbyte RAM; real-time debug module; dual floppy discs; in-circuit emulator; RS-232 or current loop interface; software and user's manuals; extra card slots; 2 chassis system. Universal interface to printers, PROM programmers, etc.
	\$6990	Z80-software development system	Same as above, except no in-circuit emulation capability.
	\$6990	Z80-hardware development system	Same as first system, except no universal interface.
	\$5990	Z80-microcomputer system	Dual floppy disc system in single chassis containing any combination of Z80 board products (MCB, MDC, etc.)
Resident software	N.A.	OSZ80-operating system for Z80 development systems and MCB family	Assembler: translates assembly language mnemonics into machine language. Includes macro's, conditional assembly, the ability to assemble programs of virtually any length and sorted symbol tables with complete cross-reference listings. Relocating assembler and linking loader: Facility for linking programs which have been assembled independently and executing Editor environment: allows the user to input and modify texts, such as, assembly language source programs. File environment: controls and manipulates disc files that the user creates while writing, debugging and executing programs. Debug environment: allows the user to load, test and save programs using an assortment of debugging aids.
	N.A.	BASIC interpreter	This program supports an interpretive language that allows translation into machine code at execution time on a statement-by-statement basis.
	N.A.	PLZ-Zilog resident programming language	From relocatable assembly to high-level system programming: <ul style="list-style-type: none"> • allows access to architecture of Z80 • compiles efficient code • easy to translate to machine language Two levels of the language allow tailoring to programming task needs. ANSII 16-Bit Fortran and PLI version available.
	Cross software	N.A.	Z80 cross assembler Z80-PLM language compiler

assembled so you can combine programs.

The text editor in the system includes many commands (for more than many full minicomputer editors) to help you manipulate the source files. Although it is a line editor (the pointer always indicates the beginning of a line), some string-oriented commands are available. Automatic paging permits you to edit files that are larger than available memory work space. Put and Get commands help you copy sections from one disc file to another or insert them into a program. Over 20 commands in the editor permit text repeats, alterations, storage, line-number printing and macro capabilities.

To develop higher-level language programs, you can use a Basic interpreter. This permits programs to be written and debugged interactively. Also made for resident use is PLZ, a procedure-oriented language

with a syntactic and semantic style that blends Algol, PL/I and Pascal. It permits access to the Z80 architecture, can compile efficient code and is easy to translate into machine code. Two levels are available: PLZ Level I combines assembly language with statements necessary to create relocatable program modules; Level II is similar to a high-level systems language in which single statements can substitute for sequences of assembly-language statements. ■■

DESIGNING A MICROPROCESSOR DRIVEN MULTIPURPOSE PERIPHERAL CONTROLLER

Requisites of adaptability to mix/match combinations of I/O devices, operation with existing software, and intelligence formulated the design of a microprocessor based multifunction controller architecture

Richard F. Binder Modular Computer Systems, Incorporated, Fort Lauderdale, Florida

Requirements for a revised generation of peripheral controllers became apparent while the ModComp CLASSIC computer series was still in the conceptual stage of design. System packaging was based on card-edge pluggable wirewrapped boards for modularity and ease of maintenance. To devote a full board space (approximately 550 integrated circuits) to a single card reader or line printer controller seemed unreasonable; this configuration would waste space and entail extra cost. The decision therefore was made to package several such low performance controllers on one board. Specifying that the design approach would be toward a multiported controller adaptable to many different devices in mix/match configuration avoided the problem of choosing which controllers to conjoin. Also, the new controller had to operate with existing software and would therefore require some intelligence. For example, the existing card reader controller is fully buffered and can transfer data in a direct 12-bit card image; in a transitional 8-bit code called "half-ASCII," packed either one or two bytes per word; or in any 8-bit code downloaded by the host mini-computer, again packed one or two bytes per word. It performs multipunch detection while translating to 8-bit codes. Other controllers to be reimplemented are similarly sophisticated.

Clearly, a microprocessor is the way to package the requisite intelligence on a single board. This approach relieves the designer of complex hardware and/or custom microcode design; a microprocessor's firmware is generally more maintainable than microcode fitted to custom logic. Also, interfacing to future devices should be easier.

General Architecture

Since a microprocessor based controller is extremely slow in relation to a controller implemented with discrete logic, the designer must take into consideration the microprocessor's response time. This response deficiency can be concealed for the most part under the overhead of the host's interrupt-driven input/output (I/O) bus without slowing down the overall system. Several nearly instant system responses are still required, however, such as the setting of controller busy status for the addressed port in response to a transfer-initiate command. These responses are generated by hardware in the form of a programmed logic array (PLA) to set status latches. A Z80A microprocessor computes all other status which are stored as 16-

Reprinted with permission from *Computer Design*—April, 1979 issue
Copyright 1979 by Computer Design Publishing Corp.

bit words in four 4-word by 4-bit register files for access by the host's software. Fig 1 is a simplified block diagram of the multifunction controller's final design.

Actual execution of commanded operations is of course carried out by the microprocessor; all commands and data are loaded by the host into the command/data (C/D) first in, first out (FIFO) buffer. This buffer allows the host to issue several commands in rapid sequence. The microprocessor fetches the commands from the buffer one at a time and processes each as required. Even though four independent devices can be controlled by this design, the C/D FIFO buffer need not be very deep in storage capacity; the interrupt-driven I/O bus makes it possible for the microprocessor to control to some extent the rate at which it receives commands by controlling the rate at which it generates interrupts. The C/D FIFO buffer in the controller is 16 words deep by 21 bits wide (16 bits for the data and 5 bits to identify the command's function and destination within the controller).

Similarly, since four independent devices can be controlled, the handling of one device cannot wait for the I/O's response to an interrupt for another device. Therefore, three request FIFO buffers are loaded by the

microprocessor for the host: service interrupt (SI), data interrupt (DI), and direct memory processor (DMP). The first two requests are vectored in the host for software processing, while the third activates concurrent hardware in the host's I/O processor itself. As each request is needed, the microprocessor loads the request's source identification word into the appropriate request FIFO buffer. The request FIFO buffers are unloaded by the host at its own rate, and the microprocessor is thereby freed to attend to other functions. Another use of the request FIFO buffers is made by device firmware sets (tasks) which must be able to "stack" more than one request of the same type; a single register for each request type prohibits such stacking.

The microprocessor selected had to be fast enough to support the required system throughput. Tentative short benchmark routines were coded for the 8080A, Z80, 9900, and 6800. One of the coded benchmarks was a routine to fetch the 21-bit contents of the C/D FIFO buffer and transfer control to the appropriate task. The following table gives an approximate comparison of various microprocessors' performance derived from a sample routine based on the controller's firmware.

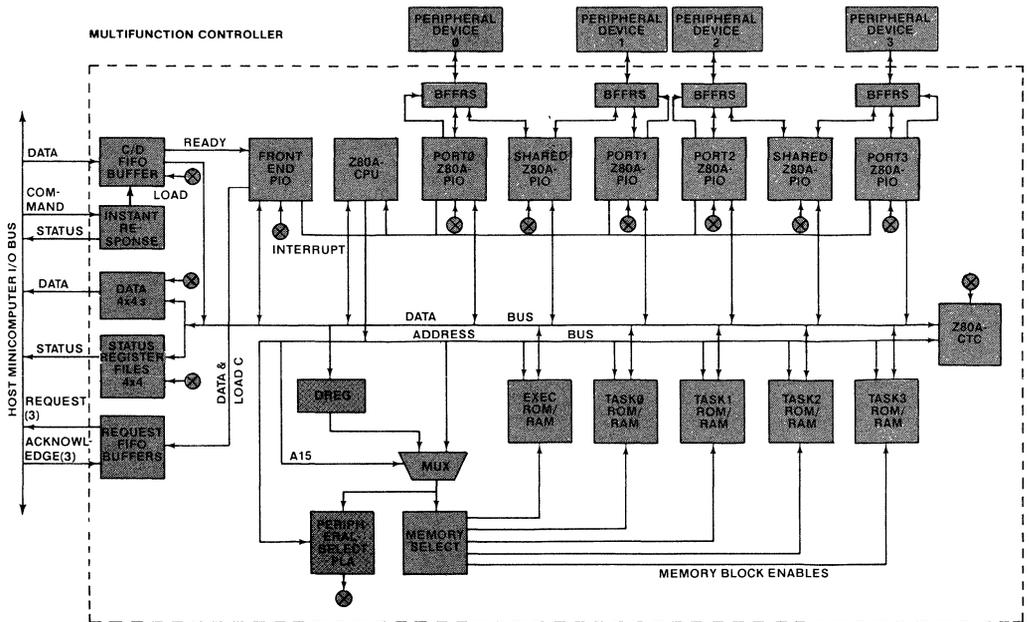


Fig 1 Controller block diagram. Layout exhibits straightforward bus architecture. Distinguishing feature is addressing scheme consisting of displacement register (DREG) and peripheral-select PLA. This hardware makes possible firmware-transparent bank switching

Microprocessor	Clock Periods	Time
8080A-2	167 at 320 ns	53.4 μ s
Z80A	92 at 250 ns	23.0 μ s
9900	114 at 300 ns	34.2 μ s
68B00	58 at 500 ns	29.0 μ s

Calculations based upon these short routines indicated that of the machines coded for, only the Z80A would be adequate. All further design was tailored explicitly for the Z80A; no detailed hardware or firmware design was produced for the other machines. (These values were attained by a designer most familiar with the Z80. Greater familiarity with other microprocessors might lessen the disparity in performance, but the Z80's powerful instruction set, vectored interrupt scheme, and twin register sets made it the undisputed choice for this application.)

The four device ports (numbered 0 to 3) must be adaptable to both serial and parallel devices. Originally, the multifunction controller specification called for support of a card reader, three types of line printers (two parallel and one serial), a paper tape punch, a paper tape reader, a serial console terminal, and a full-duplex RS-232-C asynchronous channel with full modem control and fully programmable parameters. A typical configuration might include a card reader

in port 0, a line printer in port 1, and an asynchronous channel in ports 2 and 3. Packaging requirements specified a total of 80 signal pins for all four ports. This constraint, together with an analysis of all the parallel devices, led to a 20-bit port configured as eight bidirectional bits for data transfer, four bidirectional bits for status or control (handshaking, etc), seven input bits for status or control, and one output bit for control (Fig 2).

Of the seven input bits, two can be programmed online for signal inversion, and one of these two can be connected to either a pullup or pulldown resistor for device power sensing. The two groups of bidirectional bits, including control of their buffers, can be reprogrammed online. (For a card reader, all bits are input; for a line printer, all bits are output.) This interface configuration can be made to handle most common 8-bit devices. For serial devices, the 20-pin limitation requires that the parallel buffers be removed and replaced with a universal synchronous/asynchronous receiver/transmitter (USART), as well as appropriate line drivers and receivers.

The Z80A-PIO parallel I/O controller chip provides the required bit-programmable port capability (Fig 2), but it has only two 8-bit ports. Six PIO chips are

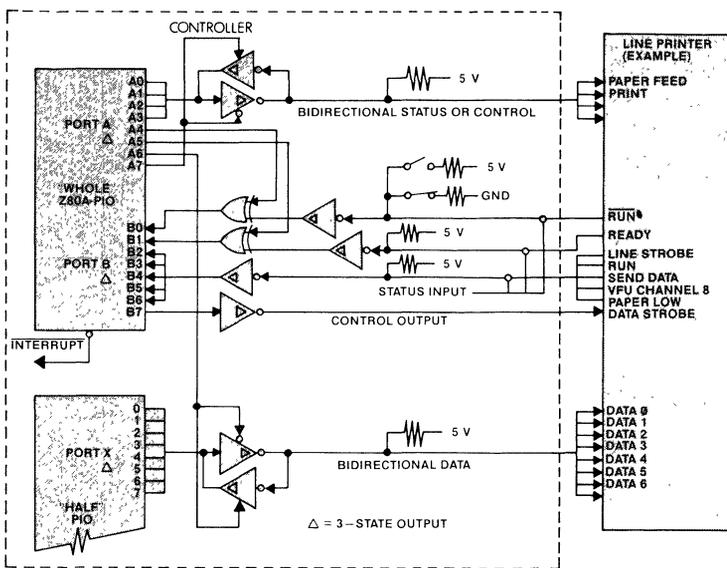


Fig 2 Parallel interface port. Each parallel port interface consists of 3-state buffers connected to port's PIOs to provide each task with ability to program interface to suit its own requirements. For uniformity, all buffers are Intel 8226 even if used only in one direction

needed to drive four 20-bit controller ports. Since one and one-half PIO chips provide 24 bits, the extra four bits control the buffers connected to the programmable bits. The two shared PIO controllers handle only data paths, and therefore are not connected to the microprocessor's interrupt system. All six chips are configured to operate in bit control mode; hence, their handshake lines are not used. Handshaking is accomplished by addressing various port bits. Each controller port has one complete PIO chip that can generate any needed interrupt.

For serial applications, all 24 bits are available to be programmed as required to best support the specialized serial hardware. To minimize serial hardware, the decision was made to restrict console tasks to port 0 or 1, and the channel task to ports 2 and 3 together. A serial line printer uses the console hardware. A USART is connected so that it is handled as though it were an external device. Serial handling may seem somewhat clumsy, but the hardware involved in the microprocessor's bus structure is simplified since there is no need to interface directly to a specific chip. This approach also helps to standardize the tasks in their port handling. The Z80-s10 serial I/O chip was not yet available when this controller was designed. Examination of the preliminary s10 specification, however, indicated that use of the s10 would seriously complicate the controller's internal structure; even if the IC had been available, it probably would not have been used. (The area in question is the displacement register, which will be discussed later.)

Some of the devices to be controlled require either timeouts or cyclic testing of status. These timing functions are triggered by a Z80A-CTC (counter-timer circuit); its four channels are allocated one to each controller port, and are used as timers for intervals up to 16.4 ms (the longest timeout possible with the 4-MHz clock). Longer timeouts are made by firmware counting of CTC interrupts.

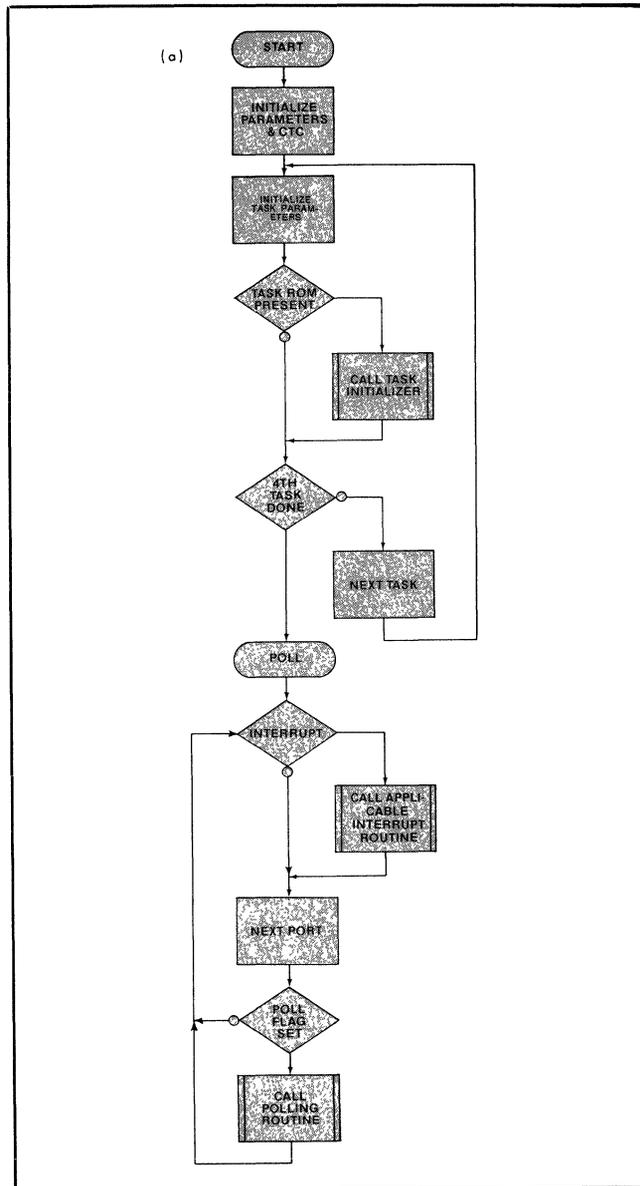
A seventh, or frontend, PIO is used between the microprocessor and the host's I/O to load the various requests into the appropriate FIFO buffers and to provide a vectored interrupt signal to the microprocessor when the C/D FIFO contains information to be processed. Sixteen-bit status and data words for the host are stored in separate 4 x 4 register files whose inputs are I/O mapped for loading by the microprocessor.

Firmware Considerations

In order to be able to switch among several concurrent activities, the firmware is designed as a multitasking operating system consisting of an executive program and the various device handlers, or tasks. The executive is always present, while tasks are added as needed by plugging in read-only memory (ROM) sets.

Executive Program

The executive occupies 768 bytes of ROM and 256 bytes of random-access memory (RAM), and has three primary functions: to initialize the system, control time-



sharing, and provide executive services available to all tasks. System initialization is performed at power-up [Fig 3(a)]. The first routine executed sets up the parameters required for the controller as a whole and initializes the CTC since the latter function is needed only once for all four ports. A loop is then entered which executes four times, once for each port. Task-not-present status is loaded into the status register file, interrupt entry vectors are loaded into the PIO assigned to the port represented by the pass count of the loop (port 0 on the first pass, port 1 on the second, etc), and a test is made to determine whether the port's task ROM is present. If not, its command entry dedi-

```

152 ; *****
153 ; * POLL *
154 ; *****
155 ;
156 ; POLLING ROUTINE - TEST EACH TASK SEQUENTIALLY FOR POLLING
157 ; SERVICE REQUESTS, CALL TASK IF POLL FLAG IS NON-ZERO.
158 ; OPEN INTERRUPT WINDOW ONCE EACH PASS.
159 ;
9071 110603 160 LD DE,POLF ;FETCH POLL FLAG ADDRESS
9074 210098 161 LD HL,DADR ;FETCH DISPL REG ADDRESS
9077 FB 162 POLL EI ;ENABLE INTERRUPTS
9078 04 163 INC B
9079 04 164 INC B
907A F3 165 DI ;DISABLE INTERRUPTS FOR POLL SERVICE
907B 70 166 LD (HL),B ;WRITE DISPLACEMENT
907C 1A 167 LD A,(DE) ;THIS PORT NEED POLLING SERVICE?
907D B7 168 OR A
907E CA7790 169 JP Z,POLL ;NO,TRY NEXT PORT
9081 D9 170 EXX ;YES, SAVE CURRENT PARAMETERS
9082 2A0403 171 LD HL,(POLE) ;FETCH TASK POLL SERVICE ENTRY
9085 C08C90 172 CALL ICALL ;CALL POLL ROUTINE INDIRECT
9088 D9 173 EXX ;GET OWN REG SET
9089 C37790 174 JP POLL ;NOW GO POLL NEXT PORT
175 ; THE Z80 DOES NOT HAVE A CALL-INDIRECT INSTRUCTION -
176 ; THE FOLLOWING JUMP SERVES THE PURPOSE.
908C E9 177 ICALL JP (HL)

```

(b)

Fig 3 Simplified main controller flow. Controller and four tasks are initialized under control of executive program (a). Program then enters polling loop (b), which provides for priority interrupt service and for one task's round-robin polling service on each pass. In idle condition, loop executes in 11 μ s/pass, ensuring reasonably rapid interrupt response

cated location in RAM is loaded with a common ignore-this-command return. If it is present, the first 10 ROM locations—containing PIO interrupt, CTC interrupt, command interrupt, data transfer interrupt, and polling service entry addresses—are transferred to dedicated locations in either executive or task RAM. Control is then transferred to an initializer within the task itself; this routine sets up the port PIOS and CTC as required for the particular task, and generates and loads valid status to replace the initial status loaded by the executive. Control is then returned to the executive initializer, which processes all four ports in this manner before enabling the hardware to respond to the I/O.

Once initialized, the system enters an idle loop whose function is to control timesharing among the tasks present. This idle loop, called the polling loop [Fig 3(b)], enables a task in two ways: interrupt service (priority enabling) and polling service (round-robin enabling). Any activity must begin with an interrupt, either from a task's CTC port or from the outside world (the host or the device connected to the particular port). A CTC or device PIO interrupt is vectored to the relevant task routine, which takes appropriate action. An interrupt from the host's I/O, through the frontend PIO, is vectored to an executive routine which extracts the contents of the current c/d

FIFO buffer location, decides whether it is a command or data, and transfers control to the task routine whose address is in the pertinent dedicated location. Whichever task routine is activated completes its action and returns control to the polling loop. The task activity in question may need service of a type which cannot be triggered by further interrupts (such as emptying a buffer asynchronously with its filling, to a device that does not handshake). Such service is activated by the setting of a dedicated location, called the polling flag, to any nonzero value.

Each task has its own polling flag and an associated polling entry dedicated location. During each pass of the polling loop, an interrupt window is opened for 2 μ s. If no interrupt is pending, or upon return from the servicing of an interrupt, the loop tests one port's polling flag. If the flag is zero, the port number is incremented and the polling loop restarts, opening the interrupt window. Each port is tested once every four passes through the loop. If the polling flag is nonzero, the loop fetches the address of the task polling routine from the dedicated location and calls that routine. The task routine takes the action for which it has been set up and resets the polling flag if no further polling service is required, and then returns to the polling loop, which continues as before. Note that interrupt service always receives priority over polling service; this arrangement provides the fastest possible response to the outside world, and is guaranteed by specifying that all interrupt routines must enable the interrupt before returning to the polling loop. If another interrupt is pending, it is serviced immediately.

To minimize both interrupt and polling service times, the system takes advantage of the Z80's two sets of working registers. One set contains registers A, B, C, D, E, H, and L; the second set is a duplicate of the first. A single instruction (EXX) will exchange all but A with their duplicates, and another instruction (EX, AF, AF') will exchange A and the machine's flag register. The latter instruction is not used in the multi-function controller—A is considered volatile by each routine. The polling loop does the context swap for polling routines, but interrupt routines must do the swap themselves. One set is dedicated to the polling loop; register B contains the number of the next port whose polling flag will be tested, register pair DE contains the address of the polling flag in memory, and register pair HL contains the address of the polling routine being called. The second register set is available for use by any task or executive service routine. The Z80 also has two index registers, IX and IY, but these registers are not used in the controller because indexed instructions suffer a 1- μ s/instruction time penalty.

The executive provides several services to any task in the form of callable subroutines. These services perform the functions of

(1) Decoding commands that a task has determined to be of a control nature, such as controller interrupt connection, data transfer termination, etc. Appropriate action is taken and control is returned to the calling routine if required.

(2) Generating one request for a data transfer either to or from the I/O. This request may be either a DI or a DMP request; the executive service routine tests current controller parameters to decide which type is proper.

(3) Initializing or terminating the host's DMP hardware by generating specialized DMP requests for these functions.

(4) Requesting startup or shutdown service of the host's software by generating an SI, and optionally resetting controller busy when setting the SI.

(5) Reinitializing the calling task exactly as is done at power-up. Primarily a diagnostic tool, this function is essentially free—the same routines are used in both cases.

Primary value of the executive services is to reduce the size of the tasks, since each task is limited to 768 bytes of ROM and 256 bytes of RAM. An added advantage lies in the fact that a task designer need not reinvent the wheel by designing all the common functions again for each new task; the effort required to implement new tasks is thereby minimized.

As mentioned above, tasks are limited in size. A more serious problem, however, is the necessity that any task (with certain specific exceptions) be installed into any port position. It is clear that the various port memory areas will have different starting addresses. A conventional software program designed to be loaded into various areas of memory (relocatable software) is accompanied by a list of locations within the program which must be modified upon loading to reflect the program's starting address. Once programmed, however, a ROM set cannot be changed; so it would seem that each task must come in four versions, one for each port. This constraint was considered unacceptable; stocking of all the different ROM sets would create problems for both manufacturer and user. The solution to this problem lies in relocatable firmware, which can be implemented by memory mapping, of which bank switching is a simplified form. Two address bits (A10 and A11) are used to select one of the four tasks, and the most significant address bit (A15) is used to control whether the bank switch is invoked [Fig 4(a)]. All tasks, then, can originate at memory address zero. It is possible to address any memory location in absolute mode (A15 = 1), but only the selected task is accessible in relative mode (A15 = 0). The executive is always addressed absolutely to make its services available to any task. The addresses of those services are assembled with each task as "external" equates.

Located in executive RAM, the push-pop stack is addressed absolutely. PIO and CTC interrupt dedicated locations are also in executive RAM, but these locations are addressed relatively so that accesses to the same relative address in each task will be routed to the proper absolute address by the bank switching control hardware. The interrupts themselves are routed through the same absolute addresses by the vectors loaded into the hardware.


```

468 ; *****
469 ; * INTH *
470 ; *****
471 ;
472 ; ENTER ON INTERRUPT FROM NOT HOLD
473 ;
022E D9 474 INTH EXX ;NOT-HOLD INTRPT IF GET HERE
022F 210C03 475 INTH1 LD HL,CSTAT
0232 7E 476 LD A,(HL) ;FETCH CSTAT
0233 0F 477 RRCA ;D0, D1 = D1
0234 0F 478 RRCA
0235 07 479 RLCA
0236 17 480 RLA
0237 77 481 LD (HL),A ;SAVE NEW CSTAT
0238 CD0A02 482 CALL LPSTA ;LOAD MOST-RECENT STATUS TO 4x4'S
023B 3E97 483 LD A,097H
023D 0383 484 OUT (PIOBC),A ;SET UP FOR INTRPT ON LINE STROBE
023F 3EFB 485 LD A,0FBH
0241 0383 486 OUT (PIOBC),A
487 ; STRIP PAGE PORTION OF BOF INTRPT HANDLER ADDRESS
0243 3E4F 488 LD A,INTB-INTB/256*256
489 ; THE ABOVE MATHEMATICAL TECHNIQUE TAKES ADVANTAGE OF THE
490 ; FACT THAT ALL INTRPT ROUTINES ARE LOCATED IN THE SAME
491 ; MEMORY PAGE - ONLY THE LOWER ORDER ADDRESS BYTE NEEDS
492 ; TO BE LOADED. THIS TECHNIQUE IS USED THROUGHOUT THE
493 ; TASK IN ORDER TO CONSERVE EXECUTION TIME AND MEMORY
494 ; SPACE.
0245 320040 495 LD (PIOBV),A ;CHANGE PENTV BACK TO "BOF"--ROUTINE
0248 320603 496 LD (POLF),A ;SET POLLING FLAG
024B D9 497 EXX ;EXIT
024C FB 498 EI
024D ED4D 499 RETI

```

Fig 5 Typical interrupt routine. Routine monitors controller status change from HOLD to READY when operator depresses RUN switch. It reports new status to host, sets PIO to interrupt when next line feed occurs, and loads interrupt dedicated location in executive RAM with address of routine which tests for bottom-of-form status. It sets polling flag—if controller is busy, data transfer commences (polling vector will have been set to address data-to-printer routine); if not, service interrupt is generated to notify host that printer is available (polling vector will address SI-generation routine). Manipulation of D0 in internal controller status word (CSTAT) copies enable bit stored in D1 into status that will be read by host if SI is made

as though it were an output. Upon recognizing this input transfer, the firmware ignores the FIFO data and proceeds to ready the next transfer.

Data requests may be generated by several mechanisms. An interrupt routine servicing a device whose data rate is controlled by the device (eg, a terminal, through a USART) generates a request when it has data for input or when the device requires output. A polling routine emptying an input buffer generates requests as long as there are data in the buffer. Finally, an output data transfer interrupt routine filling a buffer generates a request every time it is triggered by the receipt of a transfer, after loading the just-received data into its buffer.

Data are transferred to an output device by writing the data to the half PIO and then writing a one followed by a zero to another output bit assigned as the strobe

line. If a handshake is required, the strobe is set true and allowed to remain set until an acceptance is signalled by the device. Data from an input device are read from the half PIO and then accepted, if the device requires a response, by strobing in the same manner as for output. The CTC is used for two functions: cyclic activity and single-shot timeouts. Most cyclic activity tests and updates status for devices whose status can change during periods of controller inactivity. Such changes are often due to operator intervention. Single-shot timeouts are required for devices which take long periods to execute some function or functions and do not signal the completion of such functions. A currently supported paper tape punch, for example, takes a full second to run up to speed when started; it is left running for 10 s after the completion of a transfer to avoid repeated up and down cycles

and the consequent startup delays. Several concurrent timeouts may be controlled by a common clock handler routine, and this activity by no means precludes cyclic functions as well.

Hardware Architecture

The memory bank switching function is the central capability of the hardware, and is implemented with a single 2-bit register called the displacement register (DREG). Input to DREG is data bus bits D1 and D2 [Fig 4(a)]. This register is loaded either by an executive routine or hardware interrupt routine. The executive routine which fetches the C/D FIFO contents loads two of the extra FIFO bits into DREG by a mapped memory write. The register is addressed as though it were a memory location; hence, any firmware has the ability to load it, but tasks normally do not do so. The two loaded bits are a binary encode of the port selected by the host's controller address bus, and when used as A10 and A11, they select the specified task's memory area. Hardware interrupt response loads D1 and D2 into DREG using the interrupting device's vector to select the task whose device made the interrupt. Dedicated interrupt entry locations are allocated to provide the proper vectors. It is this function which precluded use of the sio. The sio generates a series of vectors for a given port, so that bits 1 and 2 cannot be used for port selection.

DREG outputs are multiplexed with A10 and A11 from the microprocessor, and the multiplexer is steered by A15. When A15 is a zero (relative mode), the multiplexer gates DREG's outputs to the controller's internal address bus, and any one of the four task areas can be accessed. When A15 is a one (absolute mode), the microprocessor's actual address is used, and any area of memory can be addressed. The executive is always addressed absolutely; certain tasks, which occupy more than one port and are always installed in the same port location, are also addressed absolutely to avoid the necessity of constantly reloading DREG when executing different subroutines.

DREG addresses not only memory but also most other port oriented hardware in the controller. This scheme is necessary to speed execution times; if a task were required to recognize its port address, and compute and load the addresses of all its devices, most routines would become unreasonably long. To avoid this problem, all PIOS and the CTC are selected by a peripheral-select PLA, which is steered by a combination of address bits 0 to 7 and the DREG outputs. DREG steers both data and status register files and most of the port oriented hardware in the front end. This hardware includes a multiplexer whose inputs are the controller's option-selection switches, and several registers used to control interrupt generation to the host.

In addition, DREG supplies a port selection function in addressing the executive RAM, but in this case DREG's outputs are multiplexed with address bits A1 and A2.

Vectors are loaded into the various ports' interrupting peripherals, two locations apart, and these two address bits select which port's dedicated location is addressed when the firmware uses relative mode. For example, the firmware addresses location 4000 (hexadecimal), and any one of the four locations—4000 (equivalent to C000), 4002, 4004, or 4006—is accessed as controlled by DREG [Fig 4(b)]. The firmware cannot address these locations directly in relative mode since DREG overlays the programmed address. During a hardware interrupt response, location C0XX is addressed with the XX being supplied by the interrupting peripheral [Fig 4(c)]. Port 0's PIO supplies 00 to address C000, port 1's PIO addresses C002, etc, with A15 forcing absolute addressing to one of four locations which all appear as 4000 to the firmware. This method (Fig 5) is used for all interrupt vectoring. Extended use of DREG makes it unnecessary for a task ever to know in which port it is installed, thereby significantly increasing the overall throughput of the controller.

Summary

Although the multifunction controller is limited to an aggregate throughput of from 4000 to 8000 bytes/s, depending upon configuration, this performance exceeds the requirements of the peripheral devices it is designed to handle. The microprocessor based design offers satisfactory solutions to most problems and objectives of a multipurpose intelligent peripheral controller: it allows reasonably fast response to the host, enables the system designer to mix or match peripherals, and provides an adaptable interface for additional peripherals. It can easily be configured for installation into a system, and is relatively inexpensive to manufacture and simple to service.

Bibliography

- T. Dollhoff, "μP Software: How to Optimize Timing & Memory Usage," *Digital Design*, Feb 1977, pp 44-51
- L. Teschler, "Interface Software for Microcomputers," *Machine Design*, Aug 10, 1978, pp 105-109
- J. G. Wester and W. D. Simpson, *Software Design for Microprocessors*, Texas Instruments, Inc, Dallas, Tex, 1976
- Zilog, Inc, *Z80-CPU Technical Manual, Z80-PIO Technical Manual, Z80-CTC Technical Manual*, Cupertino, Calif, 1976



Currently a design engineer and member of the technical staff with MODCOMP, Richard Binder has held various positions in the I/O development group, designing interfaces for an electrostatic printer/plotter, magnetic tape formatter, card reader, moving head discs, and bulk core memory modules. He attended Rose Polytechnic Institute, and has worked as a mechanical designer and technical illustrator.

Z80 Family Interrupt Structure



Tutorial

January 1980

INTRODUCTION

Interrupts provide a means of processing information on a random or asynchronous basis. The Z80 CPU and peripheral family support interrupts using a daisy-chain approach. As opposed to parallel priority resolution, the daisy chain uses an efficient, minimal-hardware method of prioritizing multiple interrupting devices. In addition, a parallel priority resolution scheme can be configured with the Z80 through the use of a priority encoder and other external hardware.

Coupled with the powerful vectored interrupt capabilities of the Z80, this approach allows

the system designer great flexibility in implementing an interrupt driven system.

This document describes the Z80 CPU interrupt process and evaluates the design of the daisy-chain interrupt scheme. The reader can refer to the following documents for additional information:

- Z80 Assembly Language Programming Manual (03-0002-01)
- Z80/Z80A CPU Technical Manual (03-0029-01)
- Z80/Z80A SIO Technical Manual (03-3033-01)
- Z80/Z80A PIO Technical Manual (03-0008-01)
- Microcomputer Components Data Book (03-8032-01)

Z80 CPU INTERRUPT PROCESSING

The Z80 uses two types of interrupts: maskable (\overline{INT} input) and non-maskable (\overline{NMI} input). Maskable interrupts may be nested. The simplest maskable interrupt implementation does not provide for the nesting of interrupts, thereby obligating an interrupt service routine to complete its processing and return to the main program before another interrupt can be serviced. With nested interrupts, an interrupt service routine can be interrupted either by an interrupt that invokes the same routine (reentrant type) or by a higher priority interrupt that invokes a different interrupt service routine. The Z80 family components allow the user to implement a powerful interrupt-driven system utilizing these concepts.

When both types of interrupts are employed, the Z80 CPU will service them in a specific sequence. Both the \overline{INT} and \overline{NMI} inputs are sampled by the CPU on the rising edge of CLK in the last T state of the last Machine (M) cycle of any instruction. However, if \overline{BUSRQ} is active at the same time, it will be processed before any interrupts. Figure 1 illustrates the Z80 interrupt service sequence.

Non-Maskable Interrupts

The non-maskable interrupt (\overline{NMI}) is different from the maskable interrupt in several respects. \overline{NMI} is always enabled and cannot be disabled by the programmer. It is employed when very fast response is desired independent of the maskable interrupt status

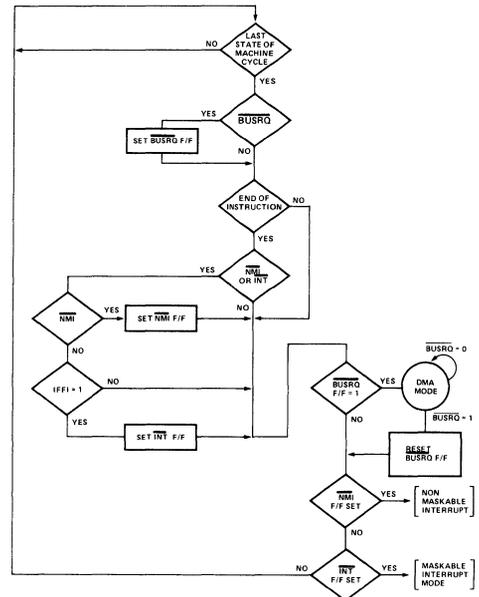


Figure 1. Z80 Flow Diagram Interrupt Sequence

and can be used for interrupt conditions like a power fail detect. \overline{NMI} is an edge-sensitive signal that has a lower priority than \overline{BUSRQ} and higher priority than \overline{INT} . When the CPU acknowledges an occurrence of \overline{NMI} , the processor begins a normal opcode fetch. How-

ever, the data read from memory is ignored and instead the CPU restarts its operation from location 66H. The restart operation involves pushing the Program Counter onto the stack, jumping to location 66H, and continuing to process there. During this time, the status of the maskable interrupt condition is

preserved and maskable interrupts are disabled, until either an EI instruction is executed or a RETN instruction is used to exit the NMI service routine.

The RETN instruction is discussed in detail in the Z80 CPU Technical Manual. Figure 2 shows the timing used for NMI interrupts.

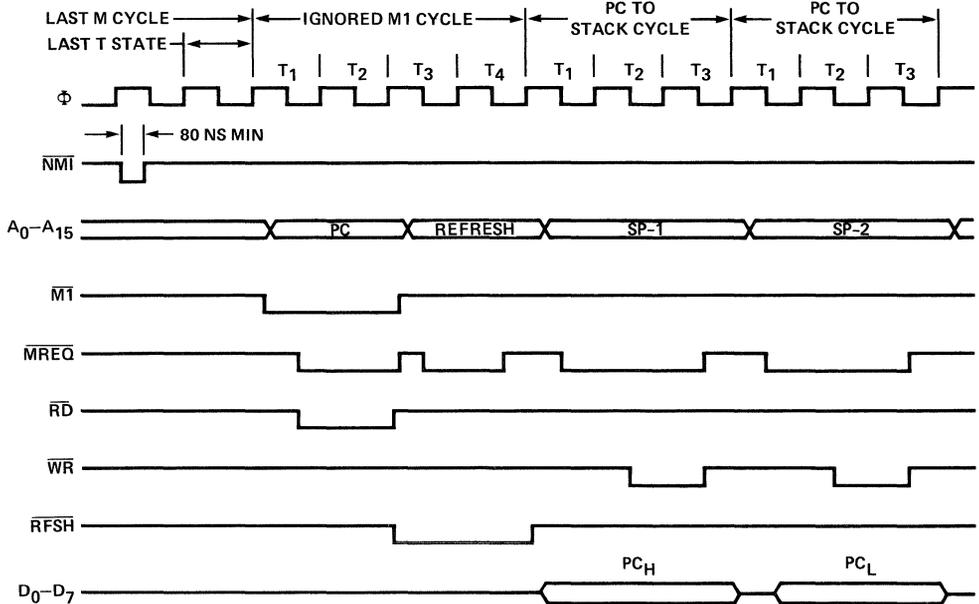


Figure 2. Non-maskable Interrupt Request Operation

Maskable Interrupts

Maskable interrupts (\overline{INT}) are acknowledged with a lower priority than the \overline{NMI} but allow the programmer more flexibility. \overline{INT} is enabled under software control by way of the EI instruction and disabled via the DI instruction. When the Z80 CPU samples \overline{INT} and it is active, the processor begins an interrupt acknowledge cycle so long as \overline{BUSRQ} and \overline{NMI} are not active. The processor does not use an interrupt acknowledge signal but instead issues the acknowledge by executing a

special $\overline{M1}$ cycle. During an interrupt acknowledge cycle, \overline{RD} is inactive, \overline{IORQ} is active, and two wait states are automatically added.

Since the Z80 peripheral devices have logic to interpret this special cycle with no additional external circuitry, a minimal amount of hardware is needed by the system and there is no loss in efficiency. Figure 3 shows the detailed timing for the Z80 CPU interrupt acknowledge cycle.

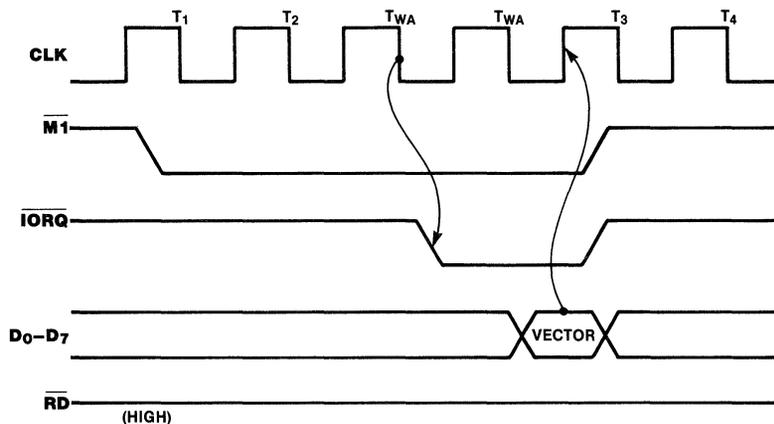


Figure 3. Interrupt Acknowledge Cycle

There are also three modes of operation for servicing maskable interrupts. These are Mode 0, Mode 1, and Mode 2. Any particular mode

is selected by the programmer using the IM instruction. Figure 4 illustrates the processing sequence for each interrupt mode.

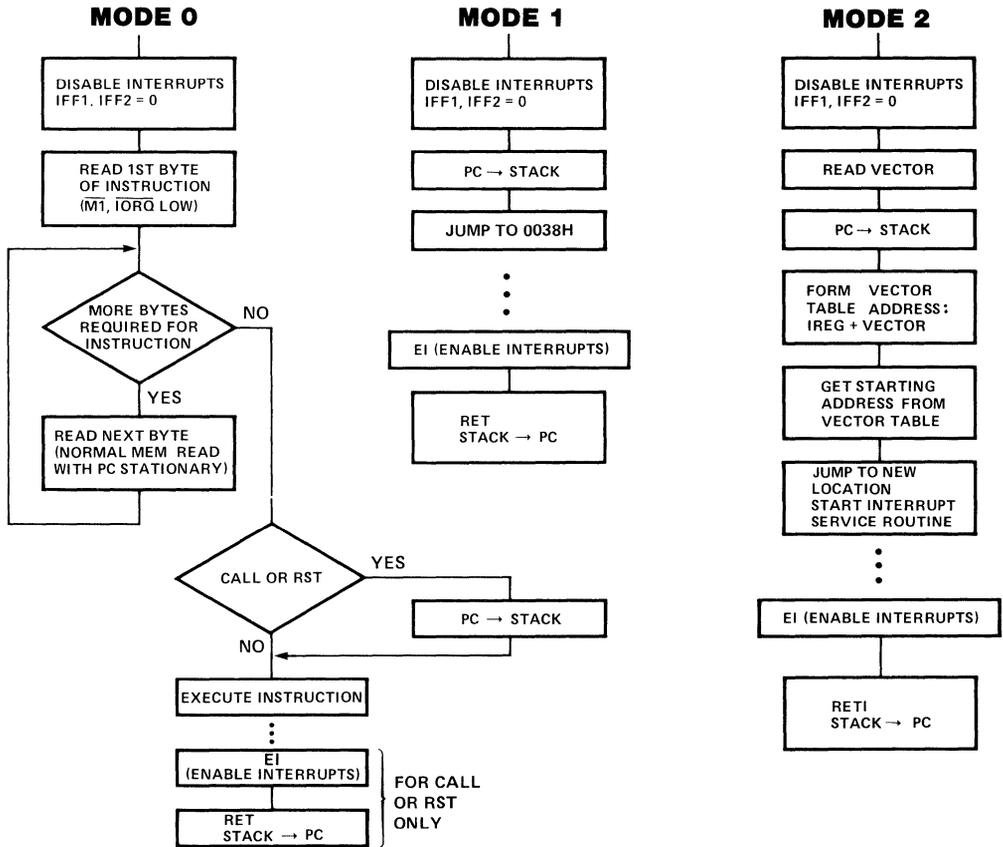


Figure 4. Maskable Interrupt Sequences

**Maskable
Interrupt
Mode 0**

In the maskable interrupt Mode 0 (as with the 8080 interrupt response mode), the interrupting device places an instruction on the data bus for execution by the Z80 CPU. The instruction used is normally a Restart (RST) instruction, since this is an efficient one-byte call to any of eight subroutines located in the first 64 bytes of memory. (Each subroutine is a maximum of eight bytes.) However, any instruction may be given to the Z80 CPU.

The first byte of a multibyte instruction is read during the interrupt acknowledge cycle. Subsequent bytes are read in by normal memory read cycles. The Program Counter remains at its preinterrupt state, and the user must insure that memory will not respond to these

read sequences, since the instruction must come from the interrupt hardware. Timing for the additional bytes of a multibyte instruction is the same as for a single byte instruction (see NMI in Figure 2).

When an interrupt is recognized by the CPU, succeeding interrupts are automatically disabled. An EI instruction can be executed anytime after the interrupt sequence begins. The subroutine can then be interrupted, allowing nested interrupts to be used. The nesting process may proceed to any level as long as all pertinent data is saved and restored correctly.

Upon $\overline{\text{RESET}}$, the CPU automatically sets interrupt Mode 0.

**Maskable
Interrupt
Mode 1**

Interrupt Mode 1 provides minimally complex peripherals access to interrupt processing. It is similar to the NMI interrupt, except that the CPU automatically CALLs to location

38H instead of 66H. As with the NMI, the CPU pushes the Program Counter onto the stack automatically (Figure 2).

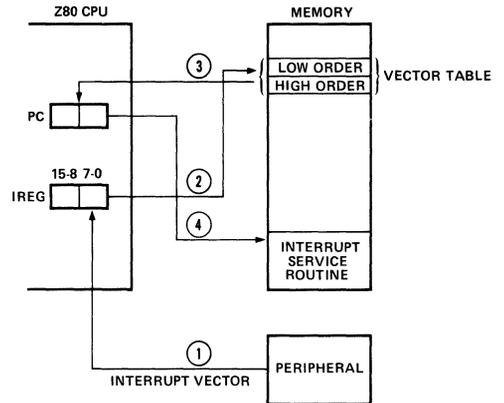
**Maskable
Interrupt
Mode 2
(Vectored
Interrupts)**

The Z80 CPU interrupt vectoring structure allows the peripheral device to identify the starting location of the interrupt service routine.

Mode 2 is the most powerful of the three maskable interrupt modes. It allows an indirect call to any memory location by a single 8-bit vector supplied by the peripheral. In this mode, the peripheral generating the interrupt places the vector onto the data bus in response to an interrupt acknowledge. The vector then becomes the least significant eight bits of the 16-bit indirect pointer, whereas the I register in the CPU forms the most significant eight bits. This address points to an even address in the vector table which then becomes the starting address of the interrupt service routine. Interrupt processing thus starts at an arbitrary 16-bit address, allowing any location in memory to begin the service routine. Since the vector is used to identify two adjacent bytes that form a 16-bit address, the CPU requires an even starting address for the vector's low byte. Figure 5 shows the sequence of events for processing vectored interrupts.

The I register is loaded by the user from the A register. There is no restriction on its

value other than its pointing to a valid memory location.



NOTES

1. Interrupt vector generated by peripheral is read by CPU during interrupt acknowledge cycle
2. Vector combined with I register contents form 16-bit memory address pointing to vector table.
3. Two bytes are read sequentially from vector table. These 2 bytes are read into PC.
4. Processor control is transferred to interrupt service routine and execution continues.

Figure 5. Vector Processing Sequence

**Return from
Maskable
Interrupt**

When execution of the interrupt service routine is complete, return to the main program (or another service routine) occurs differently in each mode. In Mode 0, the method of return depends on which instruction was executed by the CPU. If an RST instruction is used, a simple RET suffices. In Mode 1, the CPU treats the interrupt as a CALL instruction, so an RET is used. Mode 2, however, uses the vector information from the peripheral chip to identify the source of the

recognized interrupt, and a method of resetting the peripheral's interrupt condition must be found. This is accomplished by using the RETI instruction. If Mode 2 is used by the programmer, the RETI instruction must be executed in order to utilize the daisy chain properly. Figure 6 shows the RETI instruction timing for the Z80 CPU. A more complete description of how RETI affects the peripherals is given in Chapter 3.

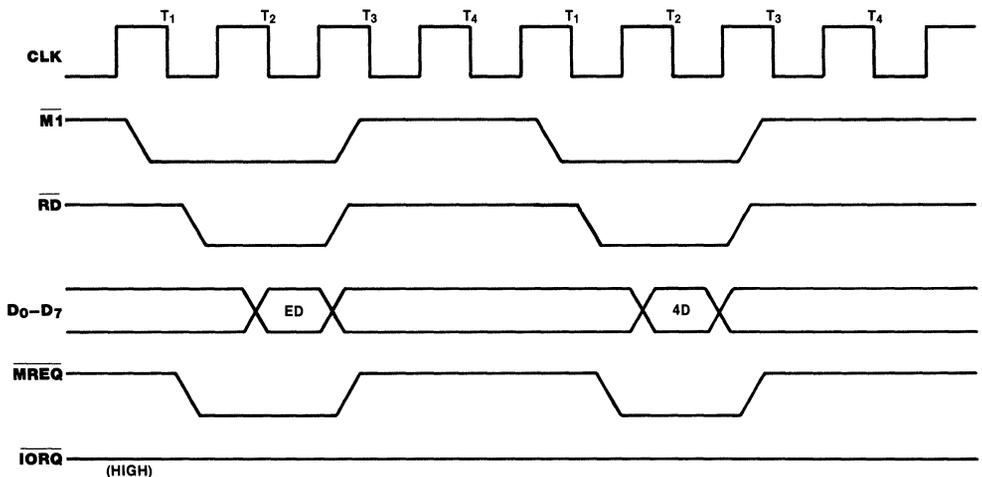


Figure 6. Return From Interrupt Timing (RETI) for Mode 2 Interrupts

**Halt Exit
Using
Interrupts**

Whenever a software halt instruction is executed, the CPU enters the Halt state by executing No-OPs (NOPs) until an interrupt or RESET is received. Each NOP consists of one M1 cycle with four T states. The CPU samples the state of the $\overline{\text{NMI}}$ and $\overline{\text{INT}}$ lines on the rising edge of each T4 clock (Figure 7).

When an interrupt exists on either line, the subsequent cycle will be either a memory read operation ($\overline{\text{NMI}}$) or an interrupt acknowledge ($\overline{\text{INT}}$). The timing in Figure 7 shows a maskable interrupt causing the CPU to exit the Halt state.

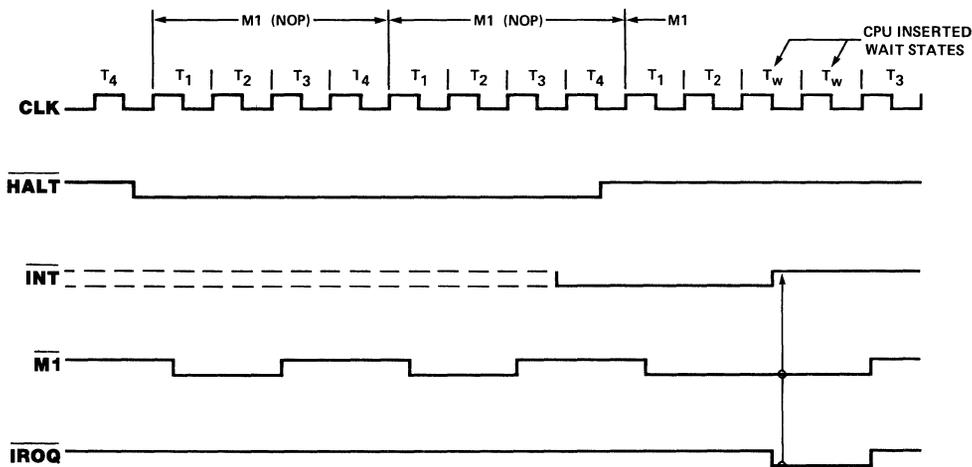


Figure 7. Exit Halt State with Maskable Interrupt

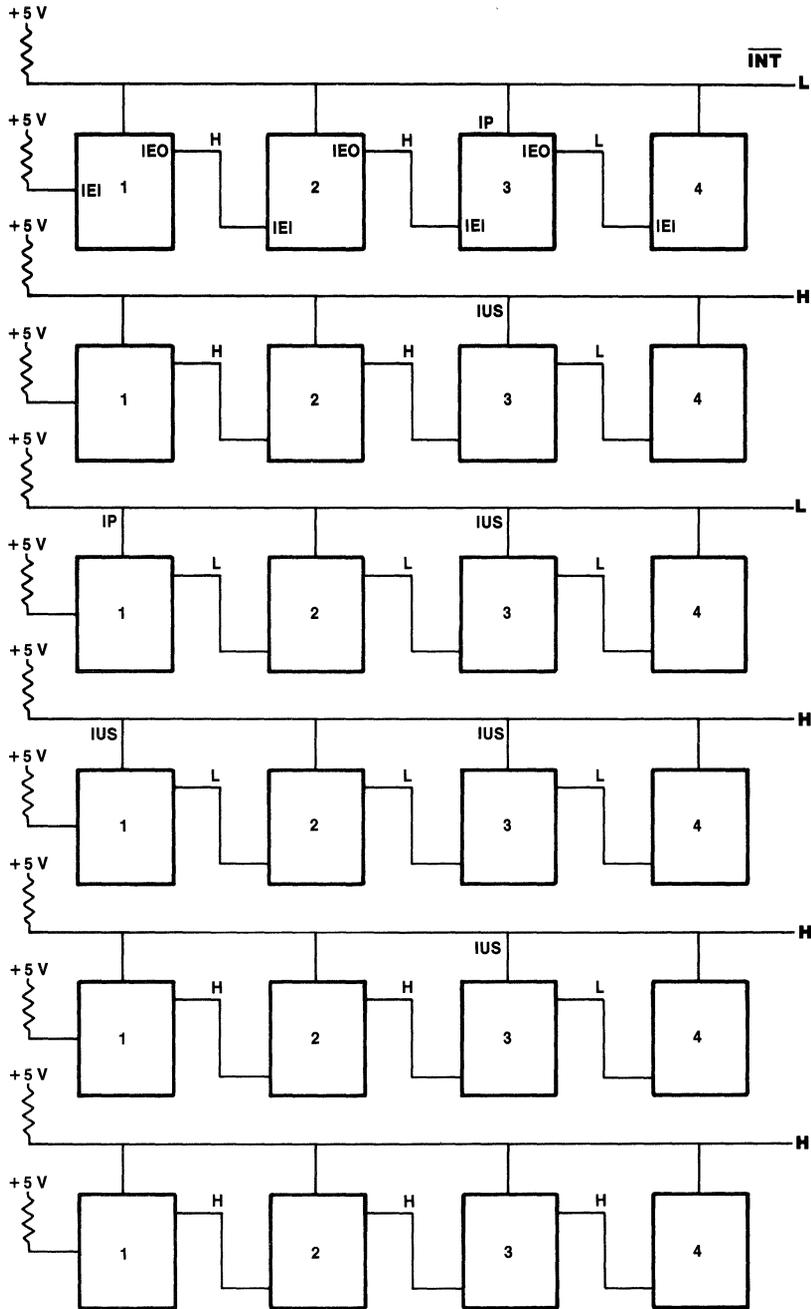
**INTERRUPT
PROCESSING
BY Z80
PERIPHERALS**

Understanding maskable interrupt processing requires a familiarity with how the Z80 peripherals respond to the CPU interrupt sequence. The Z80 family products were designed around the daisy-chain interrupt configuration, which utilizes minimal external hardware (compared to parallel contention resolution interrupt priority networks). Many devices handle interrupts via a handshake arrangement, e.g. the use of interrupt request and interrupt acknowledge signals. This is the most straightforward and probably the fastest method of implementing prioritization using more than one interrupting device. However, this method requires a separate interrupt request signal for each peripheral device and either a separate acknowledge signal for each device or a software acknowledge. Extra hardware is needed to provide contention resolution should two or more devices request an interrupt simultaneously. With the Z80 product family, however, such extra hardware is unnecessary and the software does not need to remove the interrupt request from the peripheral device. This is made possible through use of the daisy-chain priority network, which can best be visualized as a type of bucket brigade.

The Z80 peripheral products implement this daisy chain with just three extra signal lines on each chip: interrupt enable input

(IEI), interrupt enable output (IEO), and interrupt request ($\overline{\text{INT}}$). The interrupt request line is an open-drain circuit that is OR wired to the $\overline{\text{INT}}$ pins of the other devices in the chain and connected to the $\overline{\text{INT}}$ pin on the Z80 CPU. This line provides the interrupt request to the CPU.

The IEI and IEO lines provide the means for establishing priority among several requesting devices. The priority of a device is determined by its position in the chain. The IEI pin of the highest priority device in the chain is connected to +5 volts. The IEO pin of the same device is connected to the IEI pin of the next highest priority device. The IEO pin of that device goes to the IEI pin of the next lower device, as shown in Figure 8, and so on to the last device in the chain, where the IEO pin is left open. When a device has an interrupt pending, it activates its $\overline{\text{INT}}$ output which requests service from the CPU and brings its IEO pin Low, thereby preventing the lower devices in the chain from responding to further interrupt operations. When the CPU acknowledges the interrupt, the requesting device removes its interrupt request ($\overline{\text{INT}}$) signal. After the interrupt processing is completed, the peripheral will reset itself with an RETI instruction, which will bring IEO High and restore the chain to its quiescent state.



NOTES.

1. Device 3 has an interrupt pending (IP set), which causes its IEO pin to go low preventing device 4 from interrupting.
2. CPU acknowledges the interrupt and device 3 has its interrupt under service (IUS set). The device's IP is then reset.
3. Device 1 requests service, suspending device 3 processing. (Assuming interrupts were reenabled.)
4. Device 1 has its interrupt under service.
5. CPU completes processing for device 1 and returns to device 3 service routine.
6. CPU completes processing for device 3 and the daisy chain returns to quiescent state.

Figure 8. Z80 Peripheral Device Interrupt Processing Sequence

Interrupt Acknowledge Operation

The Z80 peripherals are acknowledged by the CPU and then serviced by an appropriate interrupt service routine. The acknowledge to the peripherals is accomplished by the CPU executing a special \overline{MI} cycle in which \overline{IORQ} goes active instead of \overline{MREQ} and \overline{RD} . Whenever \overline{MI} goes active, all peripheral devices are inhibited from changing their interrupt status. This allows time for IEO to propagate through the other devices in the chain before \overline{IORQ} goes active. As soon as \overline{IORQ} and \overline{MI} go active, the peripheral device that has its IEI High and an interrupt pending gates an 8-bit vector onto the data bus. (See Figure 9 for timing details.) This 8-bit vector, which was programmed into the peripheral device, is combined with the contents of the I register in the CPU to form a 16-bit address value. During the time that \overline{MI} and \overline{IORQ} are active, the requesting device removes the \overline{INT} signal (since the CPU has

acknowledged it) and waits for a return operation. If the peripheral device has its IEI pin High and has had an interrupt acknowledged, then it completes the interrupt cycle and releases IEO (when it sees an RETI instruction [ED-4D sequence] on the data bus). This restores the chain to its normal state so that lower priority interrupts can occur.

The Z80 peripherals monitor \overline{MI} and \overline{RD} for the interrupt acknowledge cycle. Since \overline{RD} goes active before \overline{IORQ} , the peripheral devices assume an interrupt acknowledge cycle if \overline{MI} is active and \overline{RD} is not. This reduces the time required for the internal device logic to respond to \overline{IORQ} when it goes active.

Thus, a very powerful interrupt-driven system can be implemented with minimal hardware, simple software, and high efficiency using the Z80 family components.

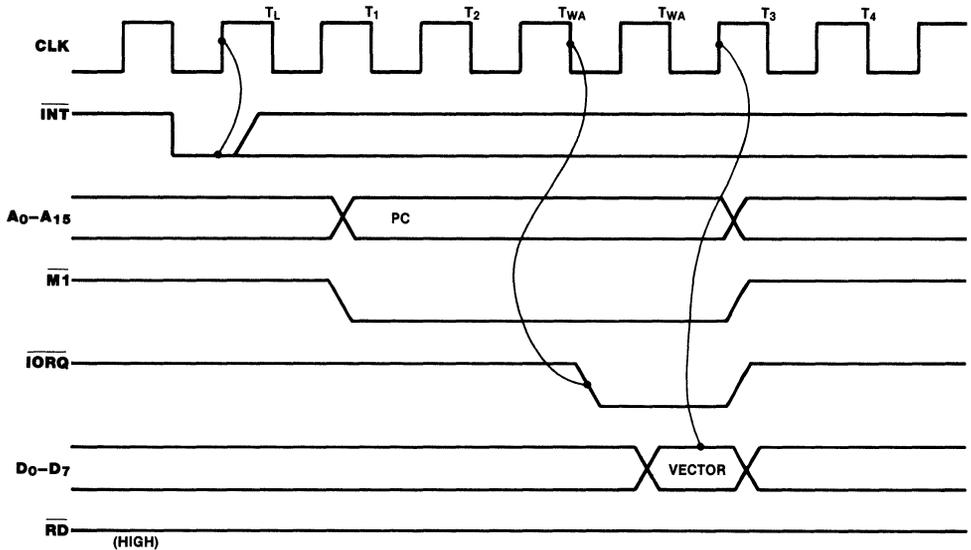


Figure 9. Peripheral Interrupt Acknowledge

Return from Interrupt Operation

When the CPU executes an RETI instruction, the device with an interrupt under service resets its interrupt condition, provided that IEI is High. All Z80 peripheral products sample the data bus for this instruction when \overline{MI} goes active along with \overline{RD} .

The RETI instruction decode by the peripheral device has certain characteristics that the designer should be aware of. Since a peripheral can request an interrupt (activate \overline{INT} and bring IEO Low) at any time, it is possible for a device whose interrupt is currently under service to have its IEI pin Low. This is undesirable, since such a condition prevents the peripheral from resetting IUS properly. To overcome this problem, all Z80 family peripherals bring IEO High momentarily

when the ED is seen during the ED-4D instruction fetch. The device whose interrupt is under service does not allow IEO to go High, but when it sees IEI High, it will reset itself when the 4D byte is fetched.

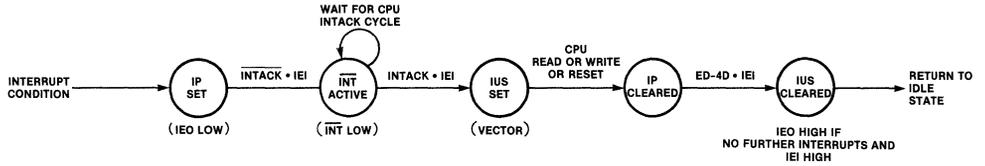
Figure 10 shows the relationship of IP and IUS to \overline{INT} , IEI, and IEO. IP is set by an interrupt condition on the peripheral (such as the transmit buffer becoming empty) whenever interrupts are enabled. However, IP being set will only cause \overline{INT} to go active (requesting an interrupt) if IUS is not set and IEI is High. IP is not necessarily cleared by the interrupt acknowledge cycle. Some specific action must be taken within the service routine, such as filling a transmit buffer. Under these conditions, IUS becomes

set and disables IEO to prevent lower priority devices in the chain from responding to an interrupt cycle. IUS is cleared when IEI is High and the peripheral decodes a valid "ED-4D" instruction. Thus,

$$IP = \overline{INTACK} * INT_COND$$

and

$$IEO = IEI * \overline{IUS} * (\overline{IP} + "ED")$$



a) State Diagram of Z80 Peripherals During Interrupt Cycle

IEI	IP	IUS	IEO
0	X	X	0
1	X	1	0
1	1	0	0
1	0	0	1

b) Truth Table of Daisy Chain During Idle or Interrupt Acknowledge Condition.

IEI	IP	IUS	IEO
0	X	X	0
1	X	1	0
1	X	0	1

c) Truth Table of Daisy Chain During "ED" Decode of Opcode Fetch
Note That IP Is Not Part of IEO Condition.

Figure 10. Z80 Peripheral Interrupt States

DAISY CHAIN DESIGN CONSIDERATIONS

There are several aspects of the Z80 family daisy chain implementation that deserve further attention.

First, since the peripheral devices must be able to monitor the data bus in order to decode the RETI instruction properly, a means of allowing them access to the data bus must be provided if buffers are used. This can be done by simply enabling the buffers from the data bus to the peripheral for all conditions except I/O read and interrupt acknowledge. Since the peripheral must assert an 8-bit

vector during interrupt acknowledge, the buffers must also accommodate this. Second, because the peripheral devices have a finite time during which IEI and IEO can stabilize within, the propagation delay of the devices must be taken into consideration. Since a device can change its interrupt status until reaching the active edge of $\overline{M1}$ during interrupt acknowledge, the time from this edge until \overline{IORQ} becomes active is the time in which the daisy chain must stabilize. Figure 11 shows the timing relationships involved in this process.

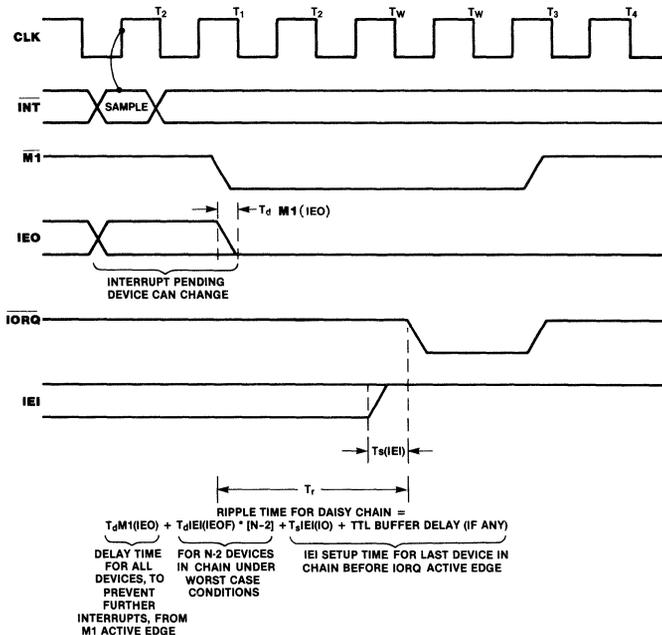


Figure 11. Interrupt Acknowledge Peripheral Propagation Delay

The Z80 CPU automatically inserts two wait states during $\overline{\text{INTACK}}$, allowing a worst-case time for a chain of four devices to become settled (when using Z80A CPU and peripherals at 4MHz). If more devices are in the chain, some other means of stabilizing the chain must be provided. This can be done either by adding additional wait states to the $\overline{\text{INTACK}}$ cycle or by providing logic to the peripherals that allows faster propagation time down the chain. Figure 12 shows circuitry that provides both additional wait states and an interrupt look-ahead circuit when more than four peripheral devices are connected to the daisy chain.

When adding wait states to the Z80 CPU interrupt acknowledge cycle, care must be taken to insure that $\overline{\text{TORQ}}$ goes active at the proper time. Normally, the CPU activates $\overline{\text{TORQ}}$ on the falling edge of the clock during the first wait cycle. If external logic is used to insert additional wait states, these are appended to the two wait states already generated by the CPU. Because $\overline{\text{TORQ}}$ goes active during the first wait state and the peripherals assert their vectors when $\overline{\text{TORQ}}$ becomes active, $\overline{\text{TORQ}}$ must be inhibited until the daisy chain becomes stable. This can be done simply by adding a few gates to the wait logic (Figure 13). $\overline{\text{TORQ}}$ is the delayed $\overline{\text{TORQ}}$ that activates the peripheral devices.

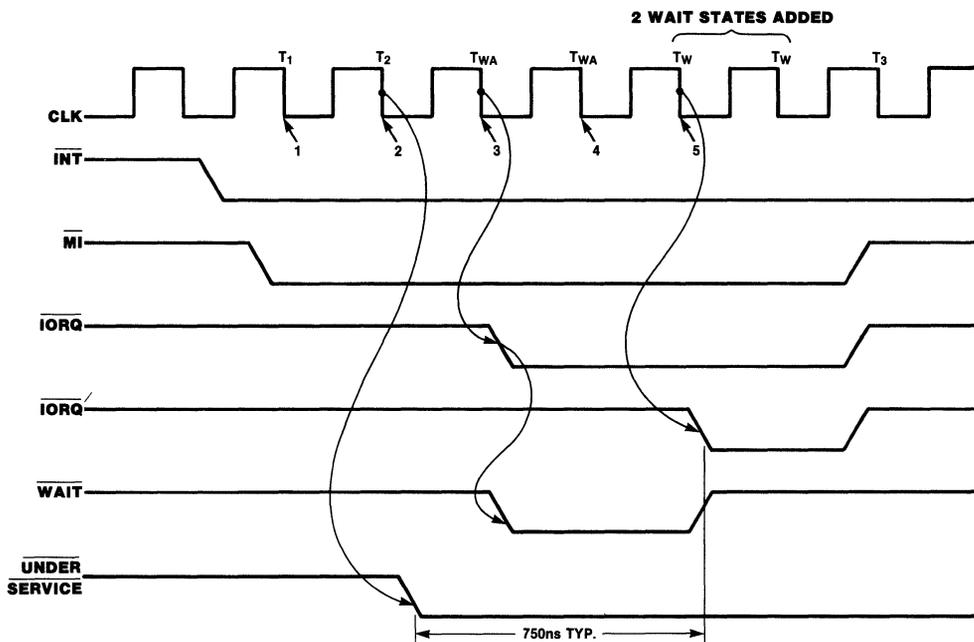
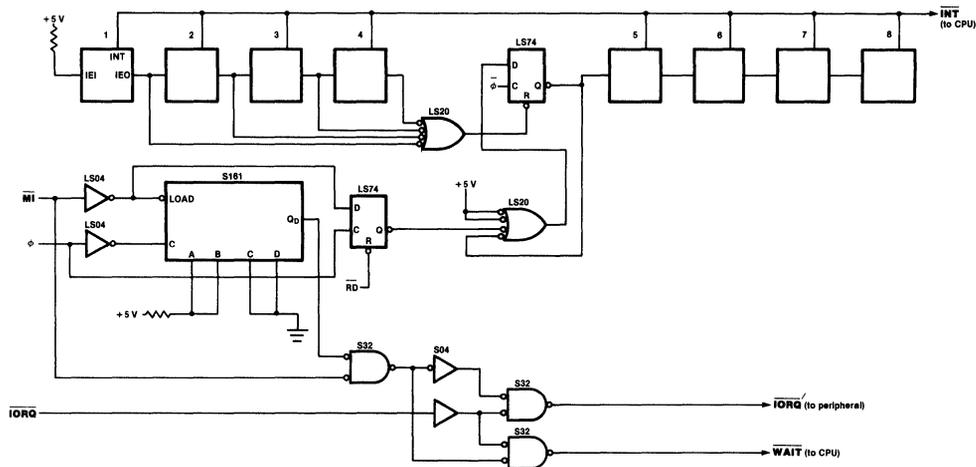


Figure 12A. Daisy Chain Look-Ahead Logic for More Than Four Peripheral Devices

The propagation delay through the peripheral devices applies during the return from interrupt condition, also. Worst-case timing involves the lowest priority device that has an interrupt under service and the highest priority device that has an interrupt pending. When the ED part of the RETI opcode is fetched, the peripheral devices must decode it, and the highest priority device must bring its IEO pin High. This IEO high signal must then propagate through the chain down to the lowest priority device

before the 4D part of RETI is decoded. Figure 14 shows the timing relationships involved. This timing is not as critical as the interrupt acknowledge timing at 4 MHz, but should be considered if wait states are being added to the INTACK cycle.

If using nested interrupts with a large daisy chain, the programmer should be careful not to place the RETI opcodes too close together. Since RETI is 14 cycles long, this is generally not a problem unless a very long chain is used.

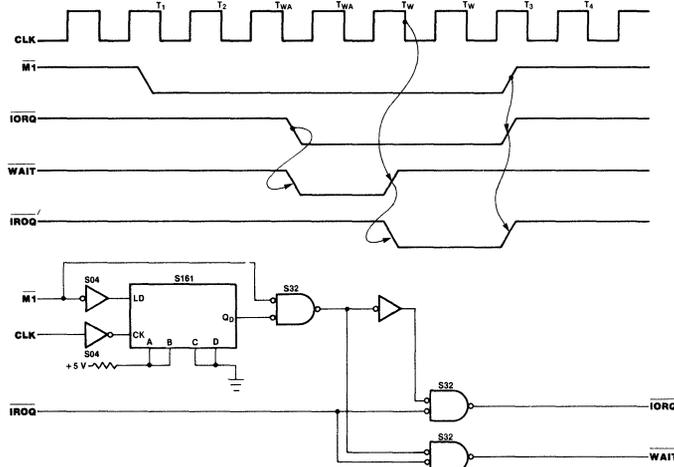
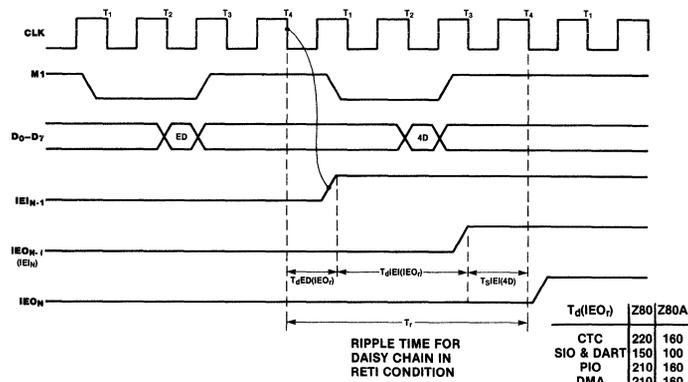


Figure 13. Wait State Logic for Interrupt Acknowledge Cycle.
Counter Preset Value Should Be 5-n. Where n = # Wait State Added



NOTES:

1. Setup time for IEI to "4D" decode \approx 200ns (4.0 MHz)
2. Must look at IEI during ED-4D because nested interrupts allow more than 1 IUS latch to be set at one time.
3. Delay time from ED decode with IP set to IEO high \approx 300ns (typ) 400ns (max) @ 2.5 MHz. This is in addition to ripple time for other devices in chain.

$$T_r \geq T_dED(IEO_r) + T_dIEI(IEO_r) \cdot [N-2] + T_sIEI(4D)$$

for N-2 devices

$T_dED(IEO_r)$ = Delay time from "ED" decode to IEO rise.

$T_dIEI(IEO_r)$ = Delay time from IEI high to IEO rise.

$T_sIEI(4D)$ = Setup time for IEI during "4D" decode.
(For last device in chain.)

Figure 14. Daisy Chain Interrupt Timing (RETI Condition)

SPECIAL CASES OF INTERRUPTS

Interfacing Zilog 8500 series peripheral products (CIO, FIO, SCC, etc.) to the Z80 CPU is a little different from interfacing the Z80 peripherals to the CPU. The primary difference between the Z80-type peripherals and the 8500-type peripherals is in the interrupt acknowledge circuitry. Functionally, they are the same, as can be seen in the timing diagrams of Figure 15. However, the 8500 peripherals do not sample $\overline{M\bar{T}}$, \overline{RD} , and \overline{IORQ} for the interrupt acknowledge, but have an explicit \overline{INTACK} pin to signal the interrupt acknowledge. Also, since the 8500 peripherals have a software reset for the interrupt under service flip-flop, these devices do not require a special return opcode to do that operation. The user need only be concerned with the interrupt

acknowledge timing when using the 8500-type peripherals.

Figure 16 shows a circuit that provides wait states for the Z80 CPU interrupt acknowledge cycle in addition to \overline{INTACK} generation. The \overline{TORQ} circuitry can be omitted if no Z80 family peripheral devices are used.

In each case, the 8500 peripheral component requires \overline{INTACK} and \overline{RD} to be active in order for the interrupt vector to be made available to the CPU. The logic shown provides for this.

This circuitry also permits extended interrupt acknowledge times to allow for the daisy chain propagation delay and the vector response delay, so that larger chains can be implemented.

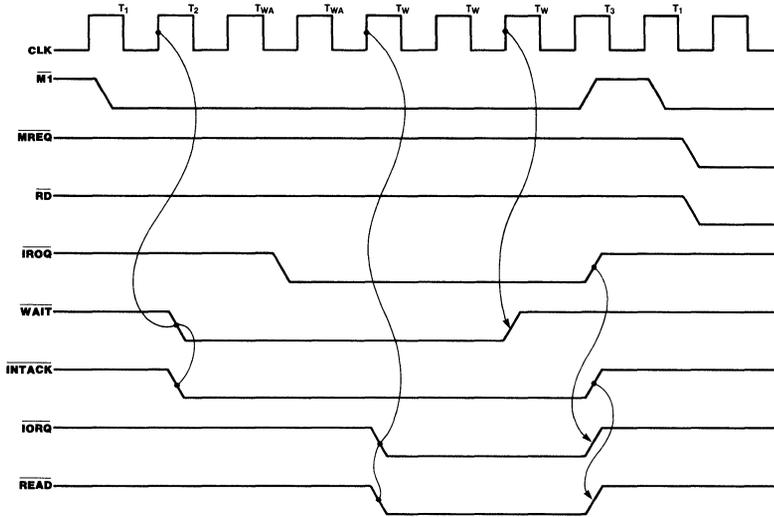


Figure 15. Timing for 8500 Peripherals During Interrupt Acknowledge.

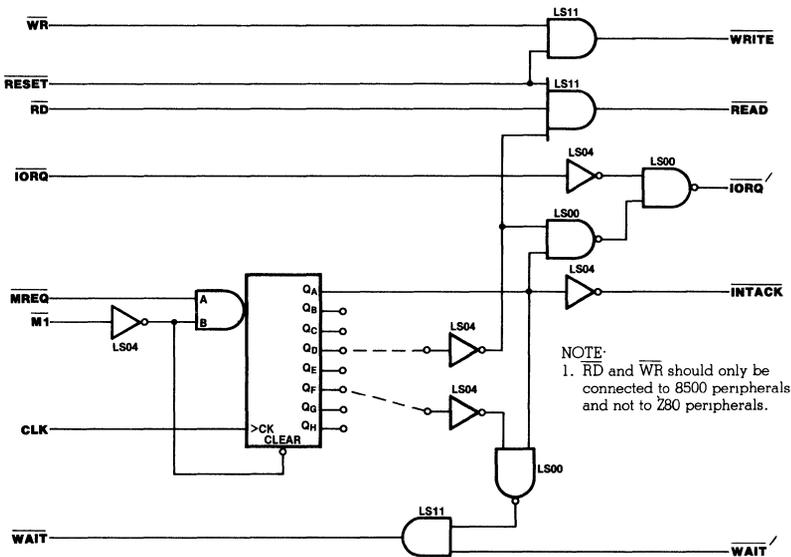


Figure 16. Interface Logic For Connecting 8500 Series Peripherals To Z80 System

**Interrupt
During RESET**

A RESET to the Z80 CPU does several things as far as interrupts are concerned. The I register, which contains the upper eight bits of the 16-bit interrupt address value, is reset to 0, and the interrupt mode is set to Mode 0. Maskable interrupts are disabled until the programmer instructs the CPU to

execute an EI instruction, just as if a DI instruction were executed. If an $\overline{\text{NMI}}$ occurs during the RESET operation, the CPU executes one instruction after the RESET condition and before acknowledging the $\overline{\text{NMI}}$. Processing then continues as usual.

A Z80-Based System Using the DMA With the SIO



Application Brief

January 1981

INTRODUCTION In certain applications, serial data communications can be handled more efficiently by using a DMA device in conjunction with a serial controller. This application brief describes the use of the Z80A SIO and Z80A DMA hardware and software in a Z80-based system to transfer data to the SIO via the DMA.

Transfers through a serial data medium are usually done with a serial controller device, often a Universal Synchronous/Asynchronous Receiver/Transmitter (USART), such as the Z80 SIO. Additionally, some sort of controlling device is required to manipulate the data on a character-by-character basis, (usually a CPU). Transferring characters can

be accomplished either by polling the USART, which forces the CPU to take time away from other activities, or by initiating an interrupt mechanism, which requires CPU time only if there is data to be moved. However, when large blocks of data need to be moved, even the interrupt mechanism becomes awkward. In these cases, a Direct Memory Access (DMA) device is especially valuable.

With DMA transfer, data is moved directly between memory and I/O (or additional memory) without CPU intervention. Once initiated by the CPU, DMA operation continues transparently to CPU operation until completed. Then the DMA device can either interrupt the CPU or restart its cycle using the previously programmed parameters.

HARDWARE DESCRIPTION The hardware used in the example for this brief consists of a Z80A CPU, a Z80A DMA controller, a Z80A SIO/2, some RAM and ROM, and some support circuitry (Figure 1).

The Z80A DMA contains a 16-bit address bus, an 8-bit data bus, and 13 control lines for external interfacing. The Z80 DMA can generate independent addresses for Port A and Port B. Each address can be variable or fixed. Variable addresses can be programmed to either increment or decrement from the programmed starting addresses, whereas fixed addressing eliminates the need for separate enabling lines for I/O ports.

Readable registers contain the current address of each port and a count of the number of bytes searched and/or transferred. Additional registers allow the DMA to perform bit-maskable data comparisons on the data that is being searched and/or transferred. The DMA has 21 writeable control registers and seven readable status registers, which together provide a high degree of programmability.

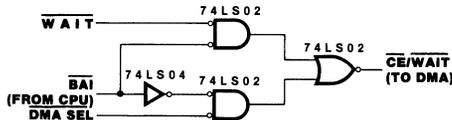
The DMA function described is for a simple test operation using memory-to-I/O transfer with no search options. The DMA is initial-

ized to transfer data from a pattern in memory to the SIO when the SIO requests a byte via the $\overline{\text{WAIT}}/\text{RDY}$ signal line. The SIO then sends the byte to a terminal, which displays it for visual inspection. After a block of bytes has been sent, the DMA restarts itself (Auto Restart mode) and the process repeats continuously. Since the data pattern in memory consists of displayable ASCII characters, data is easily verified by observing the characters displayed on the terminal.

One feature of the Z80 DMA is the ease with which it interfaces with the Z80 CPU. The DMA is designed to connect directly to the CPU, as illustrated in Figure 2. The 16 address lines, eight data lines, and seven control lines are connected directly to the corresponding lines on the Z80 CPU. These signals are then buffered by the 74LS241s and distributed to the rest of the system. The data bus is buffered by the 74LS245 bidirectional octal buffer. Other connections to the DMA include clock, $\overline{\text{CE}}/\overline{\text{WAIT}}$, $\overline{\text{INT}}$, RDY and IEI.

The clock input to the DMA is sensitive to both level and rise and fall times. The voltage should be no greater than +0.45V for a low level and no less than $V_{CC}-0.6V$ for a

The $\overline{CE}/\overline{WAIT}$ input to the DMA serves a dual purpose. When the DMA is idle [Bus Acknowledge Input (BAI) inactive], the $\overline{CE}/\overline{WAIT}$ input is used to select the DMA during a CPU access cycle, allowing the DMA to be treated as a peripheral device by the CPU. However, when the DMA takes control of the system bus, the $\overline{CE}/\overline{WAIT}$ input can be programmed as a \overline{WAIT} control line for the DMA, similar to the \overline{WAIT} input on the Z80 CPU. Figure 3 shows the gating that determines the $\overline{CE}/\overline{WAIT}$ function.



NOTES
 $\overline{CE}/\overline{WAIT} = (\overline{DMA SEL} \cdot \overline{BAI}) + (\overline{WAIT} \cdot \overline{BAI})$
 Bus Acknowledge Input (BAI) is active Low during the DMA cycle

Figure 3. $\overline{CE}/\overline{WAIT}$ Control Logic.

With the SIO, the hardware interface is slightly more complex than the DMA hardware interface. The interface to the Z80 CPU is fairly straightforward, since the SIO is accessed as an I/O peripheral device. Still, the clock input has the same requirements as the DMA; so in order to provide this signal, some sort of clock driver is needed. In addition, if the SIO is used in an interrupt environment where its internally generated vector is placed onto the data bus, the data bus buffers must allow the interrupt vector to be presented to the CPU during the interrupt acknowledge cycle. Since the data bus is buffered at the CPU, this is not a problem with the example given here; the bus is con-

trolled by the CPU circuitry. However, in larger systems, any buffers near the SIO need to be considered.

In addition, the system must supply some form of bit rate clock to the SIO for data communications. This is accomplished either by using an external clock source or by generating the clock with a device such as the CTC or CIO. Here the clock is supplied at a 1X rate for asynchronous communications from an external device such as a modem.

The $\overline{WAIT}/\overline{RDY}$ pin on the SIO is connected to the RDY input on the DMA. This provides character transfer control between the SIO and DMA. In this application, the ready function is used and the $\overline{WAIT}/\overline{RDY}$ pin is wired directly to the RDY input on the DMA with a pullup resistor. A low level initiates a DMA character transfer from memory to the SIO. The SIO drives the $\overline{WAIT}/\overline{RDY}$ line High or Low so that pullup is not strictly required. However, upon reset, the SIO $\overline{WAIT}/\overline{RDY}$ pin floats until the ready function is programmed in the SIO. Figure 4 shows the Z80 CPU-SIO interface.

Since the SIO has only one $\overline{WAIT}/\overline{RDY}$ pin per channel, it can be used with the DMA only during transmit or receive but not both simultaneously. Therefore, characters received by the SIO are transferred via interrupts with the CPU intervening. The interrupt system also handles errors detected either during reception or when the SIO notices an external or status change.

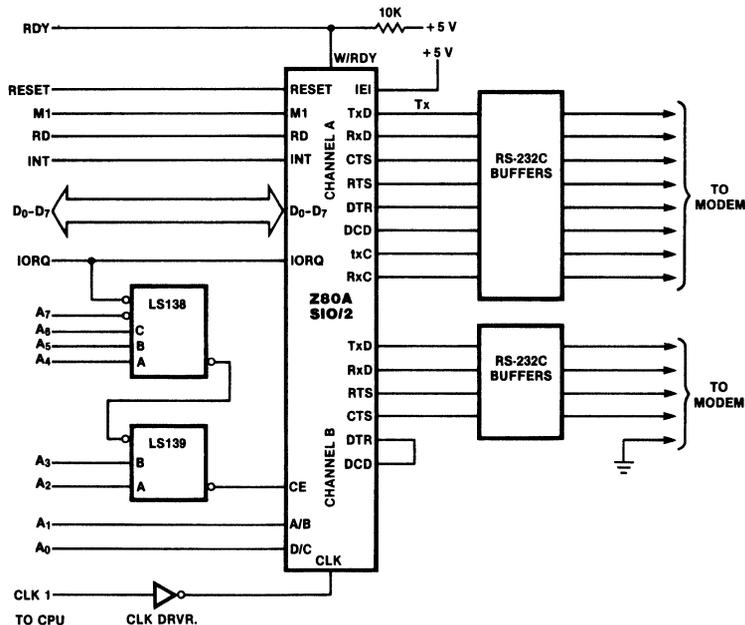


Figure 4. Z80 SIO Interface

Before any action can occur, initialization must be performed on the Z80 CPU, the DMA, and the SIO devices. Since interrupts are used in processing special SIO conditions, the Z80 CPU must be initialized for the proper interrupt mode. In the example, the CPU is set to Interrupt Mode 2 using the IM instruction. The upper eight bits of the interrupt vector are loaded into the I register via the A register in the CPU. The Stack Pointer (SP) register must be loaded by the program upon reset, because it has an undefined value. The SP register is used when processing interrupts and when the Call instruction is executed during initialization. The appendix contains a source listing for a DMA test program using the SIO.

The DMA is initialized for memory-to-I/O, byte-at-a-time transfer with the search option disabled and operates continuously until stopped by a command from the CPU. The program uses Port A of the DMA for the memory source address (SRC) and Port B for the destination address (DST) and utilizes the auto restart option on the DMA so that data can be sent to the terminal as a stream of characters. Since Port B is a fixed destination address, it must be declared as the source when the DMA is given the Load command (WR6, CFH), as stated in the programming section of the DMA Technical Manual (document number 00-2013-A). Table 1 shows the initialization sequence for the example described here.

The SIO initialization sequence is straightforward. The example uses channel A in Asynchronous Communication mode with the DMA providing data characters to the SIO on a transmit buffer empty condition. The terminal requires async format, two stop bits, and even parity. An external 1X clock is used with the SIO for the bit rate clock. The lower eight bits of the SIO interrupt vector are loaded into WR2 through channel B, and the Status Affects Vector (SAV) bit in WR1 is also set. SAV provides eight separate interrupt vectors (four for each channel), allowing easy program operation. Table 2 shows the programming sequence and mode of the SIO for DMA operation. Note that when DMA transfers are used to move data, the transmit buffer empty interrupt should not be enabled (WR1, bit 1=0).

A data test pattern is generated in the memory buffer area used for transmission to the SIO so that intelligible information can be sent to the terminal for easy verification. This is done by a short routine that fills the memory block with an incremental pattern of ASCII characters in the range of from 20H to 7FH and appends a carriage return and a linefeed to the data block. Figure 5 contains a listing of the routine involved. The block length programmed into the DMA is one less than the actual block length transferred due to the counter characteristics of the Z80 DMA.

Table 1. DMA Initialization Sequence

1. Disable DMA
2. Issue six reset commands (insures a reset if DMA in undefined state)
3. WR0 - Port A (source) characteristics
4. Port A start address - low byte
5. Port A start address - high byte
6. Port A block length - low byte
7. Port A block length - high byte
8. WR1 - Port A increment address
9. WR2 - Port B is fixed address, I/O
10. WR4 - Byte mode, Port B address (low byte) follows
11. Port B (destination) address
12. WR5 - Auto Restart mode, $\overline{CE}/\overline{WAIT}$ is multiplexed
13. Insure Port A is standard timing
14. Insure Port B is standard timing
15. Load Port B
16. WR0 - Port A is source, Port B is destination
17. Load Port A

Table 2. SIO Initialization Sequence

Channel A

1. Channel Reset
2. WR1 - $\overline{WAIT}/\overline{RDY}$ enable for TX, ready function, RX interrupt on all characters; parity affects vector
3. WR4 - X1 clock, two stop bits, even parity
4. WR5 - DTR, RTS active, TX seven bits, enable TX
5. WR3 - RX seven bits

Channel B

1. Channel Reset
2. WR1 - status affects vector
3. WR2 - lower eight bits of vector

Once the CPU, DMA, and SIO are set up, the program enables the DMA device (WR6, 87H) and the data transfer process begins. The SIO brings the WAIT/RDY output active as soon as the SIO has been initialized so that characters can be transmitted immediately. The user must insure that the DMA and data block have been set up properly before any data transfer actually occurs. DMA data transfer is different from the interrupt data transfer of the SIO, because with interrupts the SIO does not request data until it is activated by having a character sent to it.

Once operation of the DMA and SIO has begun, data transfers occur without CPU intervention unless the SIO encounters an error condition. An error causes the SIO to interrupt the CPU, thereby intervening in CPU processing. In this event, the CPU is interrupted by the device detecting the error and the DMA processing is terminated by the CPU. This termination is achieved by writing a command word to the DMA. The DMA remains disabled until given a command that enables it.

```

LD    HL, SRC      ;%HL = start address
LD    BC, BLKSIZ-2 ;%BC = length
LD    D, 20H      ;%D = data byte

LOOP:
LD    (HL), D      ;store character
INC   D            ;increment character code
LD    A,D          ;mask upper bit
AND   7FH
OR    20H          ;keep displayable character
LD    D,A          ;save in %D
INC   HL           ;Bump memory ptr.
DEC   BC           ;Bump byte count
LD    A,B          ;see if through
OR    C

JR    NZ, LOOP     ;no-loop
LD    (HL), 13     ;CR
INC   HL
LD    (HL), 10     ;LF

```

Figure 5. Data Test Pattern Generator Routine Listing.

CONCLUSION

This example shows only one aspect of using the DMA with the SIO. Use of the DMA with the SIO during receive deserves special consideration. Since the DMA operates without CPU processing, data received by the SIO does not normally indicate when the end of a message occurs. One solution to this problem is to send fixed-length data blocks so that the CPU can be interrupted when the DMA reaches terminal count. This is done by programming a fixed-length block count into the DMA and enabling it to interrupt the CPU upon End-Of-Block (EOB). As an alternative to the terminal count interrupt, the SIO can be programmed to interrupt the CPU when the closing flag is detected in SDLC mode. This allows the CPU to detect the end of a message using the SIO instead of the DMA.

Another method of detecting the end of a message is to dedicate a special EOB character used to terminate all message blocks.

The DMA can then be programmed to search for this character during data transfers and to interrupt the CPU when the character is detected. This method allows for variable-length message blocks, up to the maximum byte count the DMA will accommodate. The disadvantage with this method is that the user must dedicate one character as the special EOB character.

The unique features of the DMA and SIO combine to form a powerful and flexible data communication mechanism. Due to the designed-in compatibility of the SIO and DMA, interfacing with both in hardware and software becomes a simplified task. Programming is easy because very little CPU intervention is necessary after initialization. Thus, the user is afforded a powerful tool for implementing an efficient, cost-effective data processing system.

APPENDIX

Following is a printout of the DMA/SIO test program. This program uses the DMA to transfer data from a pattern in memory to the SIO, which then sends the data, in async format at 9600 baud, to a terminal for display. The process continues until it is externally interrupted, such as by a reset.

Interrupts are used to process error con-

ditions or to receive characters. However, no code is shown that handles the characters once they are received. Error conditions are reset by the interrupt service routine, although nothing is shown for these conditions either. The user normally sets a condition flag after resetting the error condition, so that the driver program can determine the appropriate course of action.

```

1 ; DMA/SIO TEST PROGRAM
2
3 ; BY M. PITCHER - 10/10/80
4
5 ; GENERATES BLOCK OF DATA IN RAM,
6 ; THEN OUTPUTS TO SIO VIA DMA,
7 ; THEN CONTINUES FOREVER.
8
9 RAM: EQU 2000H ;RAM START ADDR
10 RAMSIZ: EQU 1000H ;RAM SIZE
11 SIOA: EQU 0 ;SIO CH. A DATA PORT
12 SIOCA: EQU SIOA+1 ;SIO CH. A CTRL PORT
13 SIOB: EQU SIOA+2 ;SIO CH. B DATA PORT
14 SIOCB: EQU SIOB+1 ;SIO CH. B CTRL PORT
15 DMA: EQU OFOH ;DMA PORT ADDR
16 DST: EQU SIOA ;DESTINATION ADDR
17 BLKSIZ: EQU 64 ;XFER BLK SIZE
18 DMABLK: EQU BLKSIZ-1 ;DMA BLOCK SIZE VALUE
19
20
21 ; START DMA AFTER INITIALIZATION (WR6, 87H)
22 ; DMA PARAMETERS
23
24 DMAWRO: EQU 0
25 XFER: EQU 1
26 SRCH: EQU 2
27 XFRSCH: EQU 3
28 A_B: EQU 4
29 ALSTA: EQU 8
30 AHSTA: EQU 10H
31 ALBLEN: EQU 20H
32 AHBLEN: EQU 40H
33
34 DMAWR1: EQU 4
35 AIO: EQU 8
36 AINCR: EQU 10H
37 ADECR: EQU 0
38 AFIXED: EQU 20H
39 AVTIM: EQU 40H
40
41 DMAWR2: EQU 0
42 BIO: EQU 8
43 BINCR: EQU 10H
44 BDECR: EQU 0
45 BFIXED: EQU 20H
46 BVTIM: EQU 40H
47
48 DMAWR3: EQU 80H
49 DMAEN: EQU 40H
50 INTEN: EQU 20H
51 MCHBYT: EQU 10H
52 MSKBYT: EQU 8
53 SOMCH: EQU 4
54
55 DMAWR4: EQU 81H
56 BYTE: EQU 0
57 CONT: EQU 20H
58 BURST: EQU 40H
59 ICB: EQU 10H
60 INTRDY: EQU 40H
61 DMASAV: EQU 20H
62 IV: EQU 10H
63 PCB: EQU 8
64 PULSE: EQU 4
65 INTEOB: EQU 2
66 INTMCH: EQU 1
67
68 BHSTA: EQU 8
69 BLSTA: EQU 4
70
71 DMAWR5: EQU 82H

```

```

72     AUTORS: EQU    20H
73     CEWAIT: EQU    10H
74     RDYHI:  EQU     8
75
76     ,          SETUP FOR ASYNC FORMAT AS FOLLOWS
77     ,          9600 BAUD
78     ,          2 STOP BITS
79     ,          7 BIT CHARACTERS
80     ,          EVEN PARITY
81
82     ,          PROGRAM ASSUMES DMA XFER OF TX DATA
83     ,          THERE IS NO RECV DATA XFER
84     ,          STATUS IS REFLECTED IN "SIOFLG" LOC
85     ,          EXTERNAL TX AND RX CLOCK ASSUMED
86
87     ,          SIOFLG - X X 1 1 X X 1 1
88     ,          /      /      /      /
89     ,          ERROR ASLEEP ERROR ASLEEP
90     ,          CHANNEL B   CHANNEL A
91
92     SIOWR0: EQU     0
93     CHRES:  EQU    18H
94     ESCRES: EQU    10H
95     TBERES: EQU    28H
96     SRCRES: EQU    30H
97     RCRCRE: EQU    40H
98     TCRCRE: EQU    80H
99     EDMRES: EQU   0COH
100
101     SIOWR1: EQU     1
102     WREN:   EQU    80H
103     RDY:   EQU    40H
104     WRDNR: EQU    20H
105     RXIFC: EQU     8
106     RXIAP: EQU    10H
107     RXIA:  EQU    18H
108     SIO SAV: EQU     4      ; CH. B ONLY
109     TXI:   EQU     2
110     EXTI:  EQU     1
111
112     SIOWR2: EQU     2      ; CH. B ONLY
113
114     SIOWR3: EQU     3
115     RX8:   EQU    0COH
116     RX6:   EQU    80H
117     RX7:   EQU    40H
118     RX5:   EQU     0
119     AUTOEN: EQU    20H
120     HUNT:  EQU    10H
121     RXCRC: EQU     8
122     ADSRCH: EQU     4
123     SYNINH: EQU     2
124     RXEN:  EQU     1
125
126     SIOWR4: EQU     4
127     X64:   EQU    0COH
128     X32:   EQU    80H
129     X16:   EQU    40H
130     X1:    EQU     0
131     EXTSYN: EQU    30H
132     SDLC:  EQU    20H
133     SYN16: EQU    10H
134     SYN8:  EQU     0
135     STDP2: EQU    0CH
136     STDP15: EQU     8
137     STDP1: EQU     4
138     SYNCEN: EQU     0
139     EVEN:  EQU     2
140     PARITY: EQU     1
141
142     SIOWR5: EQU     5

```

LOC	OBJ CODE	M	STMT	SOURCE	STATEMENT	ASM 5.9
			143	DTR:	EQU 80H	
			144	TX8:	EQU 60H	
			145	TX6:	EQU 40H	
			146	TX7:	EQU 20H	
			147	TX5:	EQU 0	
			148	BREAK:	EQU 10H	
			149	TXEN:	EQU 8	
			150	CRC16:	EQU 4	
			151	RTS:	EQU 2	
			152	TXCRC:	EQU 1	
			153			
			154	SIQWR6:	EQU 6	
			155			
			156	SIQWR7:	EQU 7	
			157	*EJ		
			158			
			159	;;	*** MAIN PROGRAM ***	
			160			
0000			161	ORG	0	
0000	C32000		162	JP	BEGIN	
			163			
0010			164	ORG	\$. AND. OFFFOH. OR. 10H	
			165	INTVEC:		
			166	SIOVEC:		
0010	6400		167	DEFW	CHBTBE	
0012	7600		168	DEFW	CHBESC	
0014	7000		169	DEFW	CHBRCA	
0016	8A00		170	DEFW	CHBSRC	
0018	9E00		171	DEFW	CHATBE	
001A	B000		172	DEFW	CHAESC	
001C	AA00		173	DEFW	CHARCA	
001E	C400		174	DEFW	CHASRC	
			175			
			176	BEGIN:		
0020	318120		177	LD	SP, STAK ; INIT SP	
0023	ED5E		178	IM	2 ; INTERRUPT MODE 2	
0025	3E00		179	LD	A, INTVEC/256	
0027	ED47		180	LD	I, A	
0029	CD4D00		181	CALL	INIT ; INIT DMA, SIO	
002C	210120		182	LD	HL, SRC ; GENERATE DATA PATTERN	
002F	013E00		183	LD	BC, BLKSIZ-2	
0032	1620		184	LD	D, 20H	
			185	LOOP:		
0034	72		186	LD	(HL), D	
0035	14		187	INC	D	
0036	7A		188	LD	A, D	
0037	E67F		189	AND	7FH	
0039	F620		190	OR	20H	
003B	57		191	LD	D, A	
003C	23		192	INC	HL	
003D	0B		193	DEC	BC	
003E	78		194	LD	A, B	
003F	B1		195	OR	C	
0040	20F2		196	JR	NZ, LOOP	
0042	360D		197	LD	(HL), 13 ; CR	
0044	23		198	INC	HL	
0045	360A		199	LD	(HL), 10 ; LF	
0047	3E87		200	LD	A, 87H ; ENABLE DMA	
0049	D3F0		201	OUT	(DMA), A	
			202			
004B	18FE		203	JR	\$; LOOP FOREVER	
			204			
			205	INIT:		
			206	DMAINI:		
004D	0EF0		207	LD	C, DMA ; INIT DMA	
004F	21EF00		208	LD	HL, DMATAB	
0052	0616		209	LD	B, DMAEND-DMATAB	
0054	EDB3		210	OTIR		
			211	SIINI:		
0056	210501		212	LD	HL, SIOTA ; INIT SIO CH A	
0059	0E01		213	LD	C, SIOCA	
005B	060A		214	LD	B, SIOEA-SIOTA	

LOC	OBJ CODE M	STMT	SOURCE	STATEMENT	ASM 5.9
005D	EDB3	215		OTIR	
005F	AF	216		XOR	A ;CLEAR SIOFLG
0060	320020	217		LD	(SIOFLG), A
0063	C9	218		RET	
		219	*EJ		
		220			
		221	,	INTERRUPT SERVICE ROUTINES	
		222			
		223	CHBTBE:		
0064	CDD800	224		CALL	SAVE ;CH. B TX BUFFER EMPTY
0067	3E00	225		LD	A, SIOWRO
0069	D303	226		OUT	(SIOCB), A
006B	3E28	227		LD	A, TBERES
006D	D303	228		OUT	(SIOCB), A
006F	C9	229		RET	
		230			
		231	CHBRCA:		
0070	CDD800	232		CALL	SAVE ;CH. B RX CHAR AVAIL
0073	DB02	233		IN	A, (SIODB)
0075	C9	234		RET	
		235			
		236	CHBESC.		
0076	CDD800	237		CALL	SAVE ;EXTERNAL/STATUS CHG
0079	3E00	238		LD	A, SIOWRO
007B	D303	239		OUT	(SIOCB), A
007D	3E10	240		LD	A, ESCRES
007F	D303	241		OUT	(SIOCB), A
0081	3A0020	242		LD	A, (SIOFLG)
0084	CBE7	243		SET	4, A
0086	320020	244		LD	(SIOFLG), A
0089	C9	245		RET	
		246			
		247	CHBSRC:		
008A	CDD800	248		CALL	SAVE ;CH. B SPECIAL RX COND
008D	3E00	249		LD	A, SIOWRO
008F	D303	250		OUT	(SIOCB), A
0091	3E30	251		LD	A, SRCRES
0093	D303	252		OUT	(SIOCB), A
0095	3A0020	253		LD	A, (SIOFLG)
0098	CBEF	254		SET	5, A
009A	320020	255		LD	(SIOFLG), A
009D	C9	256		RET	
		257			
		258	CHATBE:		
009E	CDD800	259		CALL	SAVE ;CH. A TX BUFFER EMPTY
00A1	3E00	260		LD	A, SIOWRO
00A3	D301	261		OUT	(SIOCA), A
00A5	3E28	262		LD	A, TBERES
00A7	D301	263		OUT	(SIOCA), A
00A9	C9	264		RET	
		265			
		266	CHARCA:		
00AA	CDD800	267		CALL	SAVE ;CH. A RX CHAR AVAIL.
00AD	DB00	268		IN	A, (SIODA)
00AF	C9	269		RET	
		270			
		271	CHAESC:		
00B0	CDD800	272		CALL	SAVE ;EXTERNAL/STATUS CHG
00B3	3E00	273		LD	A, SIOWRO
00B5	D301	274		OUT	(SIOCA), A
00B7	3E10	275		LD	A, ESCRES
00B9	D301	276		OUT	(SIOCA), A
00BB	3A0020	277		LD	A, (SIOFLG)
00BE	CBC7	278		SET	0, A
00C0	320020	279		LD	(SIOFLG), A
00C3	C9	280		RET	
		281			
		282	CHASRC.		
00C4	CDD800	283		CALL	SAVE ;CH. A SPECIAL RX COND.
00C7	3E00	284		LD	A, SIOWRO
00C9	D301	285		OUT	(SIOCA), A

LOC	OBJ CODE	M	STMT	SOURCE	STATEMENT	ASM 5.9
00CB	3E30		286	LD	A, SRCRES	
00CD	D301		287	OUT	(SIOCA), A	
00CF	3A0020		288	LD	A, (SIOFLG)	
00D2	CBCF		289	SET	1, A	
00D4	320020		290	LD	(SIOFLG), A	
00D7	C9		291	RET		
			292			
			293	,	MATHEWS SAVE REGISTER ROUTINE	
			294			
			295	SAVE:		
00DB	E3		296	EX	(SP), HL ; SP = HL	
00D9	D5		297	PUSH	DE ; DE	
00DA	C5		298	PUSH	BC ; BC	
00DB	F5		299	PUSH	AF ; AF	
00DC	DDE5		300	PUSH	IX ; IX	
00DE	FDE5		301	PUSH	IY ; IY	
00E0	CDEE00		302	CALL	GO ; SAVE PC	
00E3	FDE1		303	POP	IY	
00E5	DDE1		304	POP	IX	
00E7	F1		305	POP	AF	
00E8	C1		306	POP	BC	
00E9	D1		307	POP	DE	
00EA	E1		308	POP	HL	
00EB	FB		309	EI		
00EC	ED4D		310	RETI		
			311			
			312	GO:		
00EE	E9		313	JP	(HL)	
			314	*EJ		
			315			
			316	,	CONSTANTS	
			317			
			318	DMATAB:		
00EF	83		319	DEFB	83H ; WR6, DISABLE DMA	
00F0	C3		320	DEFB	0C3H ; WR6, RESET	
00F1	C3		321	DEFB	0C3H ; WR6, RESET	
00F2	C3		322	DEFB	0C3H ; WR6, RESET	
00F3	C3		323	DEFB	0C3H ; WR6, RESET	
00F4	C3		324	DEFB	0C3H ; WR6, RESET	
00F5	C3		325	DEFB	0C3H ; WR6, RESET	
00F6	79		326	DEFB	DMAWR0+XFER+ALSTA+AHSTA+ALBLEN+AHBLEN	
00F7	01		327	DEFB	SRC.AND.255 ; PORT A ADDR (L)	
00F8	20		328	DEFB	SRC/256 ; PORT A ADDR (H)	
00F9	3F		329	DEFB	DMABLK.AND.255 ; PORT A COUNT (L)	
00FA	00		330	DEFB	DMABLK/256 ; PORT A COUNT (H)	
00FB	14		331	DEFB	DMAWR1+AINCR	
00FC	28		332	DEFB	DMAWR2+BIO+BFIXED	
00FD	85		333	DEFB	DMAWR4+BYTE+BLSTA	
00FE	00		334	DEFB	DST.AND.255 ; PORT B ADDR (L)	
00FF	B2		335	DEFB	DMAWR5+AUTORS+CEWAIT	
0100	C7		336	DEFB	0C7H ; WR6, RESET A TIMING	
0101	CB		337	DEFB	0CBH ; WR6, RESET B TIMING	
0102	CF		338	DEFB	0CFH ; WR6, LOAD PORT B	
0103	05		339	DEFB	DMAWR0+XFER+A_B ; A -> B	
0104	CF		340	DEFB	0CFH ; WR6, LOAD COUNTERS	
			341	DMAEND EQU	*	
			342			
			343	SIOTA:		
0105	00		344	DEFB	SIOWR0 ; CH. RESET	
0106	18		345	DEFB	CHRES	
0107	01		346	DEFB	SIOWR1 ; RDY/WAIT, INT. MODE	
0108	D0		347	DEFB	WREN+RDY+RXIAP	
0109	04		348	DEFB	SIOWR4 ; MODE	
010A	0F		349	DEFB	X1+STOP2+EVEN+PARITY	
010B	05		350	DEFB	SIOWR5 ; TX PARAMS.	
010C	AA		351	DEFB	DTR+TX7+TXEN+RTS	
010D	03		352	DEFB	SIOWR3 ; RX PARAMS.	
010E	40		353	DEFB	RX7	
			354	SIOEA:	EQU	*
			355			
			356	SIOTB:		

LOC	OBJ CODE M	STMT	SOURCE	DMASIO STATEMENT		PAGE 8 ASM 5.9
010F	00	357		DEFB SIOWRO		; CH. RESET
0110	18	358		DEFB CHRES		
0111	01	359		DEFB SIOWR1		; STATUS AFFECTS VECTOR
0112	04	360		DEFB SIO SAV		
0113	02	361		DEFB SIOWR2		; VECTOR
0114	10	362		DEFB SIOVEC	AND. 255	
		363	SIOEB.	EQU	\$	
		364	*EJ			
		365				
		366	;	DATA AREA		
		367				
2000		368		ORG	RAM	
2000		369	SIOFLG:	DEFS	1	; SIO STATUS FLAG BYTE
2001		370	SRC:	DEFS	BLKSIZ	; DMA SOURCE ADDR
2041		371		DEFS	64	; STACK AREA
		372	STAK:	EQU	\$	
		373				
		374		END		

Using the Z80® SIO In Asynchronous Communications

Application Note

July 1980



SECTION 1

Introduction.

The Z80 Serial Input/Output (SIO) controller is designed for use in a wide variety of serial-to-parallel input and parallel-to-serial output applications. In this application note, only asynchronous applications are considered. The emphasis is almost completely on software

implementation, with only modest reference to hardware considerations.

While reference is made only to the Z80 SIO, the entire text also applies to the Z80 DART, which is functionally identical to the Z80 SIO in asynchronous applications.

Protocol

Communication, either on an external data link or to a local peripheral, occurs in one of two basic formats: synchronous or asynchronous. In synchronous communication, a message is sent as a continuous string of characters where the string is preceded and terminated by control characters; the preceding control characters are used by the receiving device to synchronize its clock with the transmitter's clock. In asynchronous communication, which is described in this application note, there is no attempt at synchronizing the clocks on the transmitting and receiving devices. Instead, each fixed-length character (rather than character string) is preceded and terminated by "framing bits" that identify the beginning and end of the character. The time between bits within a character is approximately constant, since the clocks or "baud rates" in the transmitter and receiver are selected to be the same, but the time between

characters can vary.

Thus, in asynchronous communication, each character to be transmitted is preceded by a "start" framing bit and followed by one or more "stop" framing bits. A start bit is a logical 0 and a stop bit is a logical 1. The receiver will look for a start bit, assemble the character up to the number of bits the SIO has been programmed for, and then expect to find a stop bit. The time between the start and stop bits is approximately constant, but the time between characters can vary. When one character ends, the receiving device will wait idly for the start of the next character while the transmitter continues to send stop or "marking" bits (both the stop bits and the marking bits are logical 1). Figure 1 illustrates this. A very common application of asynchronous communication is with keyboard devices, where the time between the operator's keystrokes can vary considerably.

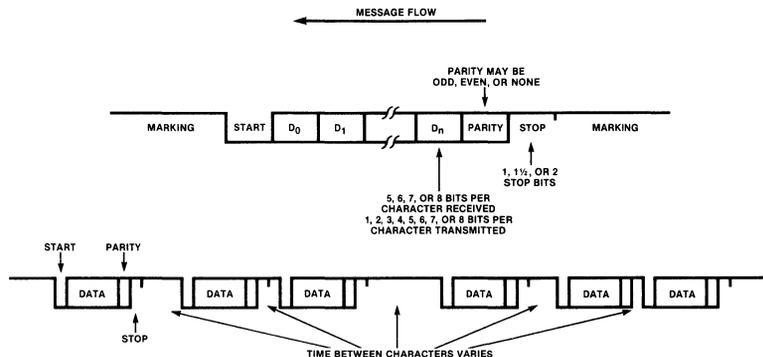


Figure 1. Asynchronous Data Format

Protocol
(Continued)

If the transmitter's clock is slightly faster than the receiver's clock, the transmitter can be programmed to send additional stop bits, which will allow the receiver to catch up. If the receiver runs slightly faster than the transmitter, then the receiver will see somewhat larger gaps between characters than the transmitter does, but the characters will normally

still be received properly. This tolerance of minor frequency deviations is an important advantage of using asynchronous I/O. Note however that errors, called "framing errors," can still occur if the transmitter and receiver differ substantially in speed, since data bits may then be erroneously treated as start or stop bits.

Modes

The SIO may be used in one of three modes: Polled, Interrupt, or Block Transfer, depending on the capabilities of the CPU. In Polled mode the CPU reads a status register in the SIO periodically to determine if a data character has been received or is ready for transmission. When the SIO is ready, the CPU handles the transfer within its main program.

In Interrupt mode, which is far more common, the SIO informs the CPU via an interrupt signal that a single-character transfer is required. To accomplish this, the CPU must be able to check for the presence of interrupt signals (or "interrupt requests") at the end of most instruction cycles. When the CPU detects an interrupt it branches to an interrupt service routine which handles the single-character transfer. The beginning memory address of this interrupt service routine can be derived, in part, from an "interrupt vector" (8-bit byte) supplied by the SIO during the interrupt acknowledge cycle.

In Block Transfer mode, the SIO is used in

conjunction with a DMA (direct memory access) controller or with the Z80 or Z8000 CPU block transfer instructions for very fast transfers. The SIO interrupts the CPU or DMA only when the first character of a message becomes available, and thereafter the SIO uses only its Wait/Ready output pin to signal its readiness for subsequent character transfers. Due to the faster transfer speeds achievable, Block Transfer mode is most commonly used in synchronous communication and only rarely in asynchronous formats. It is therefore not treated with specific examples in this application note.

Since Polled mode requires CPU overhead regardless of whether or not an I/O device desires attention, Interrupt mode is usually the preferred alternative when it is supported by the CPU. Note that the choice of Polled or Interrupt mode is independent of the choice of synchronous or asynchronous I/O. This latter choice is usually determined by the type of device to which the system is communicating.

SIO Configurations

The SIO comes in four different 40-pin configurations: SIO/0, SIO/1, SIO/2, and SIO/9. The first three of these support two independent full-duplex channels, each with separate control and status registers used by the CPU to write control bytes and read status bytes. The SIO/9 differs from the first three versions in that it supports only one full-duplex channel. The product specifications for these

versions explain this in full.

There are 41 different signals needed for complete two-channel implementation in the SIO/0, SIO/1, and SIO/2, but only 40 pins are available. Therefore, the versions differ by either omitting one signal or bonding two signals together. The dual-channel asynchronous-only Z80 DART has the same pin configuration as the SIO/0.

SIO-CPU Hardware Interfacing

The serial-to-parallel and parallel-to-serial conversions required for serial I/O are performed automatically by the SIO. The device is connected to a CPU by an 8-bit bidirectional data path, plus interrupt and I/O control signals.

The SIO was designed to interface easily to a Z80 CPU, as shown in Figure 2. Other microprocessors require a small amount of external logic to generate the necessary interface signals.

The SIO provides a sophisticated vectored-interrupt facility to signal events that require CPU intervention. The interrupt structure is based on the Z80 peripheral daisy chain. Non-Z80 microprocessors that are unable to utilize external vectored interrupts require some

additional external logic to utilize efficiently this interrupt facility. Some non-Z80 system designs do not utilize the vectored interrupt structure of the SIO at all. Instead, these require the CPU to poll the SIO's status through the data bus or to use non-vectored SIO interrupts.

Microprocessors such as the 8080 and 6800 need some signal translation logic to generate SIO read/write and clock timing. CPU signals which synchronize a peripheral device read or write operation are gated to form the proper I/O signals for the SIO. The SIO is selected by some processor-dependent function of the address bus in a memory or I/O addressing space.

Reference Material

In the next section we begin with a discussion of features common to all forms of asynchronous I/O. This is followed by discussions of polled asynchronous I/O and interrupt asynchronous I/O. Next is a series of frequently asked questions about the SIO when used in asynchronous applications. Finally, an example of a simple interrupt-driven asynchronous application is given and discussed in detail. For a complete understanding of the

material covered, the following publications are needed:

- *Z80 SIO Product Specification or Z80 DART Product Specification*
- *Z80 SIO Technical Manual*
- *Z80 Family Program Interrupt Structure*
- *Z80 CPU Technical Manual*
- *Z80 Assembly Language Programming Manual*

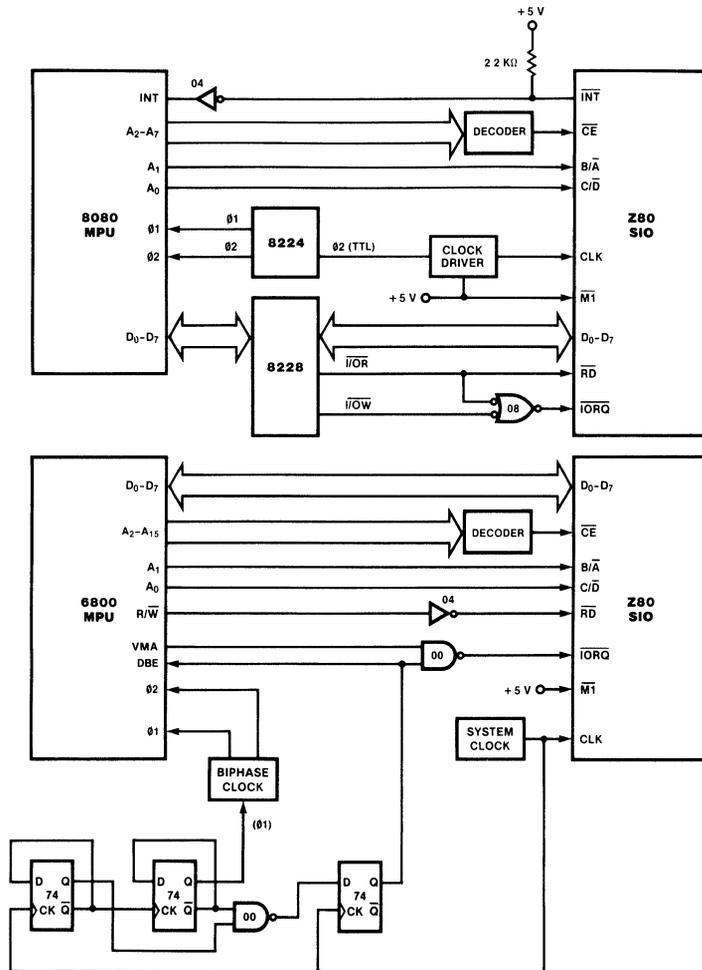


Figure 2. SIO Hardware Interfacing

**SECTION
2****Operational Considerations.**

All of the SIO options to be discussed here are software controllable and are set by the CPU. Thus, use of the SIO begins with an initialization phase where the various options are set by writing control bytes. These options are established separately for each of the two

channels supported by the SIO if both channels are used. Before giving an overview of how initialization is done, we will describe some of the basic characteristics of SIO operations that are common to both the Polled and Interrupt-driven modes.

**Addressing
the SIO**

The CPU must have a means to identify any specific I/O device, including any attached SIO. In a Z80 CPU environment, this is done by using the lower 8 bits of the address bus (A_0 - A_7). Typically, the A_1 bit is wired to the SIO's B/\bar{A} input pin for selecting access to Channel A or Channel B, and the A_0 bit is wired to the SIO's C/\bar{D} input pin for selecting the use of the data bus as an avenue for transferring control/status information (C) or actual data messages (D). The remaining bits of the address bus, A_2 - A_7 , contain a port address that uniquely identifies the SIO

device. These latter six lines are usually wired to an external decoding chip which activates that SIO's Chip Enable (CE) input pin when its address appears on A_2 - A_7 of the address bus.

The bar notation drawn above the names of certain signal lines, such as B/\bar{A} and C/\bar{D} , refer to signals which are interpreted as active when their logic sense—and voltage level—is Low. For example, the B/\bar{A} pin specifies Channel B of the SIO when it carries a logic 1 (high voltage) and it specifies Channel A when it carries a logic 0 (low voltage).

**Asynch-
ronous
Format
Operations**

Bits per Character. The SIO can receive or transmit 5, 6, 7, or 8 bits per character. This can be different for transmission and reception, and different for each channel. ASCII characters, for example, are usually transmitted as 7 bits. The SIO can in fact transmit fewer than 5 bits per character when set to the 5-bit mode; this is discussed further in the section entitled "Questions and Answers."

Parity. A parity bit is an additional bit added to a character for error checking. The parity bit is set to 0 or 1 in order to make the total number of 1s in the character (including parity bit) even or odd, depending on whether even or odd parity is selected. The SIO can be set either to add an optional parity bit to the "bits per character" described above, or not to add such a bit. When a parity bit is included, either even or odd parity can be chosen. This

selection can be made independently for each channel.

Start and Stop Bits. There are two types of framing bits for each character: start and stop. When transmitting asynchronously, the SIO automatically inserts one start bit (logic 0) at the beginning of each character transmitted. The SIO can be programmed to set the number of stop bits inserted at the end of each character to either 1, $1\frac{1}{2}$, or 2. The receiver always checks for 1 stop bit. Stop bits refer to the length of time that the stop value, a logic 1, will be transmitted; thus $1\frac{1}{2}$ stop bits means that a 1 will be transmitted for the length of clock time that $1\frac{1}{2}$ bits would normally take up. A logic 1 level that continues after the specified number of stop bits is called a "marking" condition or "mark bits."

**CPU-SIO
Character
Transfers**

The SIO always passes 8-bit bytes to the CPU for each character received, no matter how many "bits per character" are specified in the SIO initialization phase. If the number of "bits per character" is less than eight, parity and/or stop bits will be included in the byte sent to the CPU. The received character starts with the least-significant bit (D_0) and continues to the most-significant bit; it is immediately

followed by the parity bit (if parity is enabled) and by the stop bit, which will be logic 1 unless there is a framing error. The remainder of the byte, if space is still available, is filled with logic 1s (marking). If the "bits per character" is eight, then the byte sent to the CPU will contain only the data bits. In all cases, the start bit is stripped off by the SIO and is not transmitted to the CPU.

**Clock
Divider**

The SIO has five input pins for clock signals. One of these inputs (CLK) is used only for internal timing and does not affect transmission or reception rates. The other four clock inputs ($RxC\bar{A}$, $TxC\bar{A}$, $RxC\bar{B}$, and $TxC\bar{B}$) are used for timing the reception and transmission rates in Channels A and B. Only these last four are involved in "clock dividing." A clock divider within the SIO can be

programmed to cause reception/transmission clocking at the actual input clock rate or at $1/16$, $1/32$, or $1/64$ of the input clock rate. The receiver and transmitter clock divisions within a given channel must be the same, although their input clock rates can be different. The x1 clock rate can be used only if the transitions of the Receive clock are synchronized to occur during valid data bit times.

Auto Enables

The SIO has an Auto Enables feature that allows automatic SIO response and telephone answering. When Auto Enables is set for a particular channel, a transition to logical 0 (Low input level) on the respective Data Carrier

Detect ($\overline{\text{DCD}}$) input will enable reception, and a transition to logical 0 on the respective Clear To Send (CTS) input will enable transmission. This is described below under the heading "Modem Control."

Special Receive Conditions

There are three error conditions that can occur when the SIO is receiving data. Each of these will cause a status bit to be set, and if operating in Interrupt mode, the SIO can optionally be programmed to interrupt the CPU on such an error. The error conditions are called "special receive conditions" and they include:

■ **Framing error.** If a stop bit is not detected in its correct location after the parity bit (if used) or after the most-significant data bit (if parity is not used), a framing error will result. The start bit preceding the character's data bits is not considered in determining a framing error, although character assembly will not begin until a start bit is detected.

■ **Parity error.** If parity bits are attached by the external I/O device and checked by the SIO while receiving characters, a parity error will occur whenever the number of logic 1 data bits in the character (including the parity bit) does not correspond to the odd/even setting of the parity-checking function.

■ **Receiver overrun error.** SIO buffers can hold up to three characters. If a character is received when the buffers are full (i.e., characters have not been read by the CPU), an SIO receiver overrun error will result. In this case, the most recently received character overwrites the next most recently received character.

Modem Control

Five signal lines on the SIO are provided for optional modem control, although these lines can also be used for other general-purpose control functions. They are:

RTS (Request To Send). An output from the SIO to tell its modem that the SIO is ready to transmit data.

DTR (Data Terminal Ready). An output from the SIO to tell its modem that the SIO is ready to receive data.

CTS (Clear To Send). An input to the SIO from its modem that enables SIO transmission if the Auto Enables function is used.

DCD (Data Carrier Detect). An input to the SIO from its modem that enables SIO reception if the Auto Enables function is used.

SYNC (Synchronization). A spare input to the SIO in asynchronous applications. This input may be used for the Ring Indicator function, if necessary, or for general-purpose inputs.

In most applications of asynchronous I/O that use modems, the RTS and DTR control lines and the Auto Enables function are activated during the initialization sequence, and they are left active until no further I/O is expected. This causes the SIO to tell its modem continuously that the SIO is ready to transmit and receive data, and it allows the modem to enable automatically the SIO's transmission and reception of data. Figure 3 illustrates this.

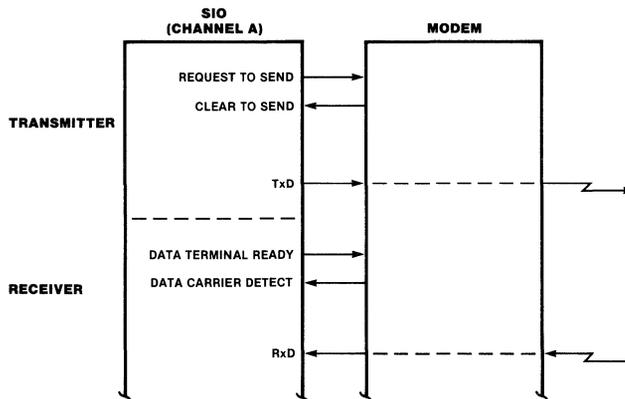


Figure 3. Modem Control (Single Channel)

External/ Status Interrupts

A change in the status of certain external inputs to the SIO will cause status bits in the SIO to be set. In the Polled Mode, these status bits can be read by the CPU. In the Interrupt mode, the SIO can also be programmed to interrupt the CPU when the change occurs. There are three such "external/status" conditions that can cause these events:

- **DCD.** Reflects the value of the $\overline{\text{DCD}}$ input.
- **CTS.** Reflects the value of the $\overline{\text{CTS}}$ input.
- **Break.** A series of logic 0 or "spacing" bits.

Note that the DCD and CTS status bits are the inverse of the SIO lines, i.e., the DCD bit will be 1 when the $\overline{\text{DCD}}$ line is Low.

Any transition in any direction (i.e., to logic 0 or to logic 1) on any of these inputs to the SIO will cause the related status bit to be latched and (optionally) cause an interrupt. The SIO status bits are latched after a transition on any one of them. The status must be reset (using an SIO command) before new transitions can be reflected in the status bits.

Initialization

The SIO contains eight write registers for Channel B (WR0-WR7) and seven write registers for Channel A (all except write register WR2). These are described fully in the *Z80 SIO Technical Manual* and are summarized in Appendix B. The registers are programmed separately for each channel to configure the functional personality of the channel. WR2 exists only in the Channel B register set and contains the interrupt vector for both channels. Bits in each register are named D₇ (most significant) through D₀. With the exception of WR0, programming the write registers requires two bytes: the first byte is to WR0 and contains pointer bits for selection of one of the other registers; the second byte is written to the register selected. WR0 is a special case in that all of the basic commands can be written to it with a single byte.

There are also three read registers, named RR0 through RR2, from which status results of operations can be read by the CPU (see Appendix B). Both channels have a set of

read registers, but register RR2 exists only in Channel B.

Let us now look at the typical sequence of write registers that are loaded to initialize the SIO for either Polled or Interrupt-driven asynchronous I/O. Figure 4 illustrates the sequence. Except for step E, this loading is done for each channel when both are used. Steps E and F are described further in the section on "Interrupt-Driven Environments."

Registers WR6 and WR7 are not used in asynchronous I/O. They apply only to synchronous communication.

The related publications on the SIO should be referred to at this point. They will be necessary in following the discussion of functions. In particular, the following material should be reviewed:

Z80 SIO Technical Manual, pages 9-12
("Asynchronous Operation")

Z80 SIO Technical Manual, pages 29-37
("Z80 SIO Programming")

- A. Load WR0.** This is done to reset the SIO
- B. Load WR4.** This specifies the clock divider, number of stop bits, and parity selection. Since register WR4 establishes the general form of I/O for which the SIO is to be used, it is best to set WR4 values first
- C. Load WR3.** This specifies the number of receive bits per character, Auto Enable selection, and turns on the receiver enabling bit
- D. Load WR5.** This specifies the number of transmit bits per character, turns off the bit that transmits the Break signal, turns on the bits indicating Data Terminal Ready and Request To Send, and turns on the transmitter enabling bit.
- E. Load WR2.** (Interrupt mode only and Channel B only.) This specifies the interrupt vector
- F. Load WR1.** (Interrupt mode only) This specifies various interrupt-handling options that will be explained later.

NOTES

Steps A through F are performed in sequence

*Channel B only

†Interrupt mode only. Polling mode begins I/O after step D

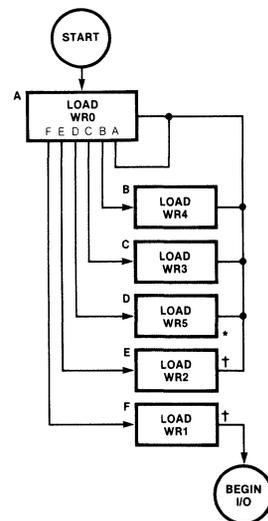


Figure 4. Typical Initialization Sequence (One Channel)

SECTION 3

Polled Environments.

In a typical Polled environment, the SIO is initialized and then periodically checked for completion of an I/O operation. Of course, if the checking is not frequent enough, received characters may be lost or the transmitter may be operated at a slower data rate than that of

which it is capable. Initialization for Polled I/O follows the general outline described in the last section. We now give an overview of routines necessary for the CPU to check whether a character has been received by the SIO or whether the SIO is ready to transmit a character.

Character Reception

To check whether a character has been received, and to obtain a received character if one is available, the sequence illustrated in Figure 5 should be followed after the SIO is initialized. We assume that reception was enabled during initialization; if it was not, the Rx Enable bit in register WR3 must be turned on before reception can occur. This must be done for each channel to be checked.

Bit D_0 of register RR0 is set to 1 by the SIO if there is at least one character available to be received. The SIO contains a three-character input buffer for each channel, so more than one character may be available to be received. Removing the last available character from the read buffer for a particular channel turns off bit D_0 .

If bit D_0 of register RR0 is 0, then no character is available to be received. In this case it is recommended that checks be made of bit D_7 to determine if a Break sequence (null character plus a framing error) has been received. If so, a Reset External/Status Interrupts command should be given; this will set the External/Status bits in register RR0 to the values of the signals currently being received. Thus, if the Break sequence has terminated, the next check of bit D_7 will so indicate. It may also be desirable to check bit 3 of register RR0 which reports the value of the Data Carrier Detect (DCD) bit.

In any case, if bit D_0 of register RR0 is 0, polled receive processing terminates with no character to receive. Depending on the facilities of the associated CPU, this step may be repeated until a character is available (or possibly a time-out occurs), or the CPU may return to other tasks and repeat this process later.

If bit D_0 of register RR0 is 1, then at least one character is available to be read. In this case, the value of register RR1 should first be read and stored to avoid losing any error information (the manner in which it is read is explained later). The character in the data register is then read. Note that the character must be read to clear the buffer even if there is an error found.

Finally, it is necessary to check the value stored from register RR1 to determine if the character received was valid. Up to three bits need to be checked: bit 6 is set to 1 for a framing error, bit 5 is set to 1 for a receiver overrun error (which occurs when the receive buffers are overwritten, i.e., no character has been removed and more than three characters have been received), and bit 4 is set to 1 for a parity error (if parity is enabled at initialization time). In case of a receiver overrun or parity error, an Error Reset command must be given to reset the bits.

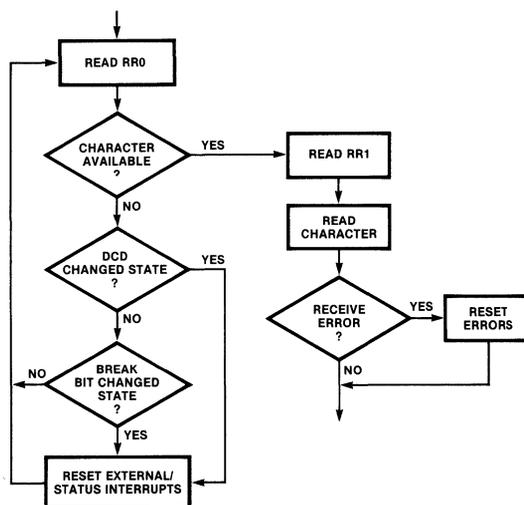


Figure 5. Polled Receive Routine

Character Transmission

To check that an initialized SIO is ready to transmit a character on a channel, and if so to transmit the character, the steps illustrated in Figure 6 should be followed. We assume that the Request To Send (RTS) bit in WR5, if required by the external receiving device, and the Transmit (Tx) Enable bit were set at initialization.

Depending on the external receiving device, the following bits in register RR0 should be checked: bit 3 (DCD), to determine if a data carrier has been detected; bit 5 (CTS), to determine if the device has signalled that it is clear to send; and bit 7 (Break), to determine if a Break sequence has been received. If any of these situations have occurred, the bits in register RR0 must be reset by sending the Reset External/Status Interrupts command, and the transmit sequence must be started again.

Next, bit 2 of register RR0 is checked. If this bit is 0, then the transmit buffer is not empty and a new character cannot yet be transmitted. Depending on the capabilities of the CPU, this is repeated until a character can be transmitted (or a timeout occurs), or the CPU may return to other tasks and start again later.

If bit 2 of register RR0 is 1, then the transmit buffer is empty and the CPU may pass the

character to be transmitted to the SIO, completing the transmit processing. On the Z80 CPU, this is done with an OUT instruction to the SIO data port.

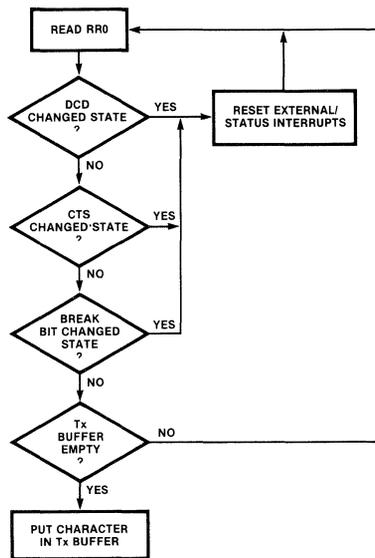


Figure 6. Polled Transmit

Assumptions for an Example

Now let us consider some examples in more detail. We assume we are given an external device to which we will input and output 8-bit characters, with odd parity, using the Auto Enables feature. We will support this device with I/O polling routines following the patterns illustrated in Figures 5 and 6. We assume that the CPU will provide space to receive characters from the SIO as fast as the characters are received by the SIO, and that the CPU will transfer characters as fast as the output can be accomplished by the SIO.

We specify this example by giving the control bytes (commands) written to the SIO and the status bytes that must be read from the SIO. Recall that to write a command to a register, except register WR0, the number of the register to be written is first sent to register WR0; the following byte will be sent to the named register. Similarly, to read a register other than RR0 (the default), the number of the register to be read is sent to register WR0; the following byte will return the register named.

Initialization

We begin with the initialization code for the SIO. This follows the outline illustrated in Figure 4. In the following sample code, each time register WR0 is changed to point to another register, the Reset External/Status Interrupts command is given simultaneously. Whenever a transition on any of the external lines occurs, the bits reporting such a transition are latched until the Reset External/Status Interrupts command is given. Up to two transitions can be remembered by the SIO. Therefore, it is desirable to do at least two different

Reset External/Status Interrupts commands as late as possible in the initialization so that the status bits reflect the most recent information. Since it doesn't hurt, we include these commands each time WR0 is changed to point to another register. This is an easy way to code the initialization to insure that the appropriate resets occur.

In the example below, the logic states on the C/D control line and the system data bus (D7-D0) are illustrated, together with comments.

Initialization
(Continued)

C/D	Bits sent to the SIO							
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	0	0	1	1	0	0	0
1	0	0	0	1	0	1	0	0
1	1	1	0	0	1	1	0	1
1	0	0	0	1	0	0	1	1
1	1	1	1	0	0	0	0	1
1	0	0	0	1	0	1	0	1
1	1	1	1	0	1	0	1	0

Effects and Comments

- Channel Reset command sent to register WR0 (D₅-D₃).
- Point WR0 to WR4 (D₂-D₀) and issue a Reset External/Status Interrupts command (D₅-D₃). Throughout the initialization, whenever we point WR0 to another register, we will also issue this command for the reasons noted above.
- Set WR4 to indicate the following parameters (from left to right):
 - A. Run at 1/64 the input clock rate (D₇-D₆).
 - B. Disable the sync bits and send out 2 stop bits per character (D₅-D₂).
 - C. Enable odd parity (D₁-D₀).
- Point WR0 to WR3.
- Set WR3 to indicate the following:
 - A. 8-bit characters to be received (D₇-D₆).
 - B. Auto Enables on (D₅).
 - C. Receive (Rx) Enable on (D₀).
- Point WR0 to WR5.
- Set WR5 to indicate the following:
 - A. Data Terminal Ready (DTR) on (D₇).
 - B. 8-bit characters to be transmitted (D₆-D₅).
 - C. Break not to be transmitted (D₄).
 - D. Transmit (Tx) Enable on (D₃).
 - E. Request To Send (RTS) on (D₁).

Reset and Error Sequences

In the receive and transmit routines that follow, we treat errors such as a transition on the Data Carrier Detect line by calling for a "reset sequence" to set the values in read register RR0 to reflect the current values found at the pins. This sequence consists of giving the Reset External/Status Interrupts command and beginning the driver over again. The command takes the form of a write to register WR0:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	0	1	0	0	0	0

Permits the status bits in RR0 to reflect current status.

This command does not turn off the latches for such things as parity errors stored in bits 4-6 of register RR1. When such an error occurs and the latches must be reset, we will

call for an "error sequence." This sequence consists of giving the Error Reset command and beginning the driver over again. The command also takes the form of a write to register WR0:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	1	1	0	0	0	0

Resets the latches in register RR1.

When specifying the result of reading register RR0 or RR1 or specifying data, we will indicate the values read as follows:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
D	D	D	D	D	D	D	D

Read a byte from the designated register..

Receive and Transmit Routines

Now we will first give an example of the receive routine. This parallels the preceding discussion of "Character Reception."

The framing error in this routine is reported on a character-by-character basis and it is not

necessary to execute an "error sequence" if it is the only error received. However, it is not harmful to do so.

Next, we give an example of transmission code that parallels the above discussion on "Character Transmission."

Receive and Transmit Routines
(Continued)

C/ \bar{D}	Bits sent and received							
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	D	D	D	D	D	D	D	D
1	0	0	0	0	0	0	0	1
1	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀

Effects and Comments (Receive Routine)

Read a byte from RR0 (the default read register); if D₀=0 then no character is ready to be received. In this case, if D₇ (Break) or D₃ (Data Carrier Detect) have changed state, then execute a "reset sequence." If D₀=0 and D₇ and D₃ have not changed state, then no character is ready to be received, either loop on this read or try again later.

Point WR0 to read from RR1, we will now check for errors in the character read. Note that Reset External/Status Interrupt Commands are not done normally to avoid losing a line-status change.

Read a byte from RR1; if either bit D₆=1 (framing error), D₅= (receive overrun error), or D₄=1 (parity error), the character is invalid and an "error sequence" should be executed after the following step.

Read in the data byte received. This must be done to clear the SIO buffer even if an error is detected.

C/ \bar{D}	Bits sent and received							
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	D	D	D	D	D	D	D	D
0	D	D	D	D	D	D	D	D

Effects and Comments (Transmit Routine)

Read a byte from RR0; if either bit D₃ (Data Carrier Detect), D₅ (Clear To Send) or D₇ (Break) have changed state, a "reset sequence" should be executed. If D₃, D₅ and D₇ have not changed state, then if D₂=0, the transmit buffer is not yet empty and a transmit cannot take place; either loop, reading RR0, or try again later.

Send the data byte to be transmitted.

SECTION 4

Interrupt-Driven Environments.

In a typical interrupt-driven environment, the SIO is initialized and the first transmission, if any, is begun. Thereafter, further I/O is interrupt driven. When action by the CPU is needed, an SIO interrupt causes the CPU to branch to an interrupt service routine after the CPU first saves state information.

In common usage, if I/O is interrupt driven, all interrupts are enabled and each different type of interrupt is used to cause a CPU branch to a different memory address. There is perhaps one frequent exception to this: parity errors are sometimes checked only at the end of a sequence of characters. The SIO facilitates this kind of operation since the parity error bit in read register RR1 is latched; once the bit is set it is not reset until an explicit

reset operation is done. Thus, if a parity error has occurred on any character since last reset, bit 4 in register RR1 will be set. It is then possible to set register WR1 so that parity errors do not cause an error interrupt when a character is received. The user then has the obligation to poll for the value of the parity bit upon completion of the sequence.

SIO initialization for Interrupt mode normally requires two steps not used in Polled mode: an interrupt vector (if used) must be stored in write register WR2 of Channel B and write register WR1 must be initialized to specify the form of interrupt handling. It is preferable to initialize the interrupt vector in WR2 first. In this way an interrupt that arrives after the enabling bits are set in WR1 will cause proper interrupt servicing.

Interrupt Vectors

The interrupt vector, register WR2 of Channel B, is an 8-bit memory address. When an interrupt occurs (and note that an interrupt can only occur after interrupts have been enabled by writing to register WR1) the interrupt vector is normally taken as one byte of an address used by the CPU to find the location of the interrupt service routine. It is also possible to cause the particular type of interrupt condition to modify the address vector in WR2 before branching, resulting in a branch

to a different memory location for each interrupt condition. This is a very useful construct; it permits short, special-purpose interrupt routines. The alternative, to have one general-purpose interrupt routine which must determine the situation before proceeding, can be quite inefficient. This is usually undesirable since the speed of interrupt-service routines is often a critical factor in determining system performance.

Interrupt Vectors

(Continued)

There are at most eight different types of interrupts that the SIO may cause, four for each of the two channels. If bit 1 in register WR1 of Channel B has been turned on so that an interrupt will modify the interrupt vector, the three bits (1-3) of the vector will be changed to reflect the particular type of interrupt. These interrupts follow a hardware-set priority as follows, starting with the highest priority:

Channel A Special Receive Condition sets bits 3-1 of WR1 to 111,

Channel A Character Received sets bits 3-1 to 110,

Channel A Transmit Buffer Empty sets bits 3-1 to 100,

Channel A External/Status Transition sets bits 3-1 to 101.

Channel B Special Receive Condition sets bits 3-1 to 011,

Channel B Character Received sets bits 3-1 to 010,

Channel B Transmit Buffer Empty sets bits 3-1 to 000,

Channel B External/Status Transition sets bits 3-1 to 001.

For example, suppose that the interrupt vector had the value 11110001 and the Status Affects Vector bit is enabled, along with all interrupt-enable bits. When an External/Status transition occurs in Channel A, the three zeros (bits 3-1) would be modified to 101, yielding an interrupt vector of 11111011. The value of the interrupt vector, as modified, may be obtained by reading register RR2 in Channel B.

Note that when a character is received, either the Special Receive Condition or Rx Character Available interrupt will occur, depending on whether or not an error occurred; the two will never occur simultaneously. Therefore, these two interrupts have equal priority. Note also that you can select not to be interrupted on some of the eight conditions; in this case, the presence of a particular condition for which interrupts are not desired can be determined by polling.

Suppose that interrupts have been enabled for all possible cases, and that the Status Affects Vector bit has also been enabled, allowing a different routine to handle each possible interrupt. As each interrupt causes a branch to a location only two bytes higher than the last interrupt, it is not possible to place a routine directly at the location where the vectored interrupt branches. In a Z80 CPU environment, these addresses refer to a table in memory which contains the actual starting location of the interrupt service routine. Also, since the state information saved by a CPU is rarely all of the information necessary to properly preserve a computation state, a typical interrupt service routine will begin by saving additional information and end by restoring that information. This is shown briefly in the examples of code in Appendix A.

It is possible to connect several SIOs using the interrupt mechanism and the IEI and IEO lines on the SIO to determine a priority for interrupt service. This mechanism is discussed on page 42 of the *Z80 SIO Technical Manual* and in the *Z80 Family Program Interrupt Structure Manual*. We do not go into it further in this application note.

Initialization

In general, the initialization procedure illustrated in Figure 4 can still be followed. All six steps (A through F) are required here. After completing the first four steps, which are the same as initialization for polled I/O, it is necessary to load an interrupt vector into WR2 of Channel B. Information is then written into register WR1 specifying which interrupts are to be enabled and whether a specific kind of interrupt should modify the interrupt vector.

Now let us give an example. As in the polled example, we assume that we are given a device to which we will input and output 8-bit characters, with odd parity, using the Auto Enables feature. We also assume the CPU will provide space to store characters as received.

We do not discuss the SIO commands and registers in detail. This is done in the *Z80 SIO Technical Manual*. A summary of the register bit assignments taken from the *Z80 SIO Serial Input/Output Product Specification* is included at the end of this note. Recall that to write a

register other than register WR0, the number of the register to be written is first sent to register WR0, and the following byte will be sent to the named register. Similarly, to read a register other than RR0 (the default), the number of the register to be read is first written to register WR0 and the next byte read will return the contents of the register named.

In our example below, each time register WR0 is changed to point to another register, the Reset External/Status Interrupts command is also given. Whenever a transition on any of the external/status lines occurs, the bits reporting the transition are latched until the Reset External/Status Interrupts command is given. Up to two transitions can be remembered by the internal logic of the SIO. Therefore, it is desirable to do at least two different Reset External/Status Interrupt commands as late as possible in the initialization so that the status bits reflect the most recent information. Since it doesn't hurt, we give these commands each

Initialization
(Continued)

time WR0 is changed to point to another register. This is an easy way to code the initialization to assure that the appropriate resets occur.

The columns below show the logic states on the C/D control line and the system data bus (D₇-D₀), together with comments.

C/D	Bits sent to the SIO								Effects and Comments
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	
1	0	0	0	1	1	0	0	0	Channel Reset command sent to register WR0 (D ₅ -D ₃).
1	0	0	0	1	0	1	0	0	Point WR0 to WR4 (D ₂ -D ₀) and issue a Reset External/Status Interrupts command (D ₅ -D ₃). Throughout the initialization, whenever we point WR0 to another register we will also issue a Reset External/Status Interrupts command for the reasons noted above.
1	1	1	0	0	1	1	0	1	Set WR4 to indicate the following parameters (from left to right): A. Run at 1/64 the clock rate (D ₇ -D ₆). B. Disable the sync bits and send out 2 stop bits per character (D ₅ -D ₂). C. Enable odd parity (D ₁ -D ₀).
1	0	0	0	1	0	0	1	1	Point WR0 to WR3.
1	1	1	1	0	0	0	0	1	Set WR3 to indicate the following: A. 8-bit characters to be received (D ₇ -D ₆). B. Auto Enables on (D ₅). C. Rx Enable on (D ₀).
1	0	0	0	1	0	1	0	1	Point WR0 to WR5.
1	1	1	1	0	1	0	1	0	Set WR5 to indicate the following: A. Data Terminal Ready (DTR) on (D ₇). B. 8-bit characters to be transmitted (D ₆ -D ₅). C. Break not to be transmitted (D ₄). D. Tx Enable on (D ₃). E. Request To Send (RTS) on (D ₁).
1	0	0	0	1	0	0	1	0	Point WR0 to WR2 (Channel B only).
1	1	1	1	0	0	0	0	0	Set the interrupt vector to point to address 11100000 (which is hex E0 and decimal 224). Once interrupts are enabled, they will cause a branch to this memory location, modified as described above if the Status Affects Vector bit is turned on (which it will be here). This vector is only set for Channel B, but it applies to both channels. It has no effect when set in Channel A.
1	0	0	0	1	0	0	0	1	Point WR0 to WR1.
1	0	0	0	1	0	1	1	1	Set WR1 to indicate the following: A. Cause interrupts on all characters received, treating a parity error as a Special Receive Condition interrupt (D ₄ -D ₃). B. Turn on the Status Affects Vector feature, causing interrupts to modify the status vector—meaningful only on Channel B, but will not hurt if set for Channel A (D ₂). C. Enable interrupts due to transmit buffer being empty (D ₁). D. Enable External/Status interrupts (D ₀).

Special Receive Condition Interrupts

A Special Receive Condition interrupt occurs (a) if a parity error has occurred, (b) if there is a receiver overrun error (data is being overwritten because the channel's three-byte receiver buffer is full and a new character is being received), or (c) if there is a framing error. The processing in this case is the following:

1. Issue an Error Reset command (to register WR0) to reset the latches in register RR1.
2. Read the character from the read buffer and discard it to empty the buffer.

It may be desirable to read and store the

value of register RR1 to gather statistics on performance or determine whether to accept the character. In some applications, a character may still be acceptable if received with a framing error.

In specifying the result of reading register RR0, RR1, or specifying data, we will indicate the values as follows:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
D	D	D	D	D	D	D	D

Read a byte from the designated register.

We now present an example of processing a Special Receive Condition interrupt.

C/D	Bits sent and received							
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	0	0	0	0	0	0	1
1	D	D	D	D	D	D	D	D
1	0	0	1	1	0	0	0	0
0	D	D	D	D	D	D	D	D

Effects and Comments

If we need to know what kind of error occurred, we point WR0 to read from RR1. Note that the Reset External/Status Interrupts command is not used. This avoids losing a valid interrupt.

Read a byte from RR1; one or more of bit D₆ (framing error), D₅ (receive overrun error), or D₄ (parity error) will be 1 to indicate the specific error.

Give an Error Reset command to reset all the error latches.

Read in the data byte received. This must be done to clear the receiver buffer, but the character will generally be disregarded.

Received (Rx) Character Interrupts

When an Rx Character Available interrupt occurs, the character need only be read from the read buffer and stored. If parity is enabled

with character lengths of 5, 6, or 7 bits, the received parity bit will be transferred with the character. Any unused bits will be 1s.

External/Status Interrupts

To respond to an External/Status Interrupt, all that is necessary is to send a Reset External/Status Interrupts command. However, if you wish to find the specific cause of the

interrupt, it is necessary to read register RR0. In this case, the complete processing takes the following form:

C/D	Bits sent and received							
	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	0	0	0	1	0	0	0	0

Effects and Comments

Read register RR0; bit D₇ (Break), D₅ (Clear To Send), or D₃ (Data Carrier Detect) will have had a transition to indicate the cause of the interrupt.

Give a Reset External/Status Interrupts command to set the latches in RR0 to their current values and stop External/Status Interrupts until another transition occurs.

Transmit (Tx) Buffer Empty Interrupts

The final kind of interrupt is a Tx Buffer Empty interrupt. If another character is ready to be transmitted on this channel, a Tx Buffer Empty interrupt indicates that it is time to do so. To respond to this interrupt, you need only send the next character. If no other character is ready to transmit, it may be desirable to mark the availability of the transmit mechanism for future use. In addition, you should send a Reset Tx Interrupt Pending command. This command prevents further transmitter inter-

rupts until the next character has been loaded into the transmitter buffer.

The Reset Tx Interrupt Pending command to WR0 takes the following form:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	0	1	0	1	0	0	0

Reset Tx Interrupt Pending command; no Tx Empty Interrupts will be given until after the next character has been placed in the transmit buffer.

Z80 Assembler Code

To take these examples further, let us use Z80 Assembler code to implement the routines for a single channel. We assume that the location stored in register WR2 points to the appropriate interrupt service routine. We also assume that the following constants have already been defined:

SIOctrl. The address of the SIO's Channel B control port (we assume Channel B in order to include code to initialize the interrupt vector).

SIOdata. The address of the SIO's Channel B data port.

X. An address pointing to locations in memory that will be used to store various values.

We will write data as binary constants; the "B" suffix indicates this. In most cases, binary constants will be referred to by the command names. We begin with the initialization routine:

INIT:	LD	C,SIOctrl	;place the address of the SIO in the C register for
	LD	A,00011000B	; use in subsequent output
	OUT	(C),A	;load Channel Reset command in A register
			;give Channel Reset command
	LD	A,00010100B	;write to register WR0 pointing it to register WR4
	OUT	(C),A	
	LD	A,11001101B	;output basic I/O parameters to WR4
	OUT	(C),A	
	LD	A,00010011B	;write to register WR0 pointing it to register WR3
	OUT	(C),A	
	LD	A,11100001B	;output receive parameters to WR3
	OUT	(C),A	
	LD	A,00010101B	;write to register WR0 pointing it to register WR5
	OUT	(C),A	
	LD	A,11101010B	;output transmit parameters to WR5
	OUT	(C),A	
	LD	A,00010010B	;write to register WR0 pointing it to register WR2
			; (Channel B only)
	OUT	(C),A	
	LD	A,11100000B	;output the interrupt vector to WR2; in this case it is
			; decimal location 224
	OUT	(C),A	
	LD	A,00010001B	;write to register WR0 pointing it to register WR1
	OUT	(C),A	
	LD	A,00010111B	;output interrupt parameters to WR1
	OUT	(C),A	
	RET		;return from initialization routine

Now let us look first at some sample codes for the Special Receive Condition interrupt routine, following the example above.

This is followed by a simple receive interrupt routine that will fetch the character received and store it in a temporary location.

SIOspecint:	PUSH	AF	;save registers which will be used in this routine
	LD	A,00000001B	;write to register WR0 pointing it to register RR1
	OUT	(SIOctrl),A	
	IN	A,(SIOctrl)	;fetch register RR1
	LD	(X),A	;store result for later error analysis
	LD	A,00110000B	;send an Error Reset command to reset device
			; latches
	OUT	(SIOctrl),A	
	IN	A,(SIOdata)	;fetch the character received—we will discard this
			; character since an error occurred during its
			; reception
	POP	AF	;restore saved registers
	EI		;enable interrupts
	RETI		;return from interrupt

**Z80
Assembler
Code**
(Continued)

```
SIOfecint:  PUSH  AF          ;save registers which will be used in this routine
            IN   A,(SIOdata)  ;fetch the character received
            LD   (X),A        ;store result for later use
            POP  AF          ;restore saved registers
            EI   ;enable interrupts
            RETI ;return from interrupt
```

Of course, this last routine is probably far too simple to be useful. It is more likely that an interrupt routine will fill up a buffer of characters. A more complex example of a receive interrupt routine is contained in the

chapter entitled "A Longer Example."

We now give a simple interrupt routine for an External/Status Interrupt, again assuming that the status contents of SIO register RRO are stored in temporary location X:

```
SIOextint:  PUSH  AF          ;save registers which will be used in this routine
            LD   A,00010000B  ;send a Reset External/Status Interrupts command
            OUT (SIOctrl),A
            IN   A,(SIOctrl)  ;fetch register RRO
            LD   (X),A        ;store result for later analysis
            POP  AF          ;restore saved registers
            EI   ;enable interrupts
            RETI ;return from interrupt
```

Finally, we give the processing for a transmit interrupt routine in the case where no more characters are to be transmitted.

It is likely that this code would just be a portion of a more general transmit interrupt

routine which would transmit a buffer-full of information at a time. A more complex example is included in the section entitled "A Longer Example."

```
SIOtrmint:  PUSH  AF          ;save registers which will be used in this routine
            LD   A,00101000B  ;send a Reset Tx Interrupt Pending command
            OUT (SIOctrl),A
            POP  AF          ;restore saved registers
            EI   ;Enable Interrupts
            RETI ;Return From Interrupt
```

SECTION**5****Questions and Answers.****Hardware Considerations**

Q: Can a sloppy system clock cause problems in SIO operation?

A: Yes; the specifications for the system clock are very tight and must be met closely to prevent SIO malfunction. The clock high voltage must be greater than $V_{CC} - 0.6V$ but less than $+5.5V$. The clock low voltage must be greater than $-0.3V$ but less than $+0.45V$. The transitions between these two levels must be made in less than 30 ns. This does not apply to the \overline{RxC} and \overline{TxC} inputs which are standard TTL levels.

Q: When is a received character available to be read?

A: Data will be available a maximum of 13 system clock cycles from the rising edge of the \overline{RxC} signal which samples the last bit of the data.

Q: What is the maximum time between character-insertion for transmission and next-character transmission?

A: This will vary depending on the speed of the line over which the character is being transmitted.

Q: Are the control lines to the SIO synchronous with the system clock so that noise may exist on the buses any time before setup requirements are satisfied?

A: Yes.

Q: In asynchronous use must receiver and transmitter clock rates be the same?

A: No, the SIO allows receive and transmit for each channel to use a different clock (thus up to four different clocks for receiving and transmitting data can be used on each SIO). However, the clock multiplier for each channel must be the same.

Q: Do Wait states have to be added when using the SIO with other processors other than the Z80 CPU?

A: No, provided that setup times specified for the SIO are met.

Q: If the Auto Enables bit in register WR3 is set, will a change in state on the \overline{DCD} (Data Carrier Detect) or \overline{CTS} (Clear To Send) lines still cause an interrupt?

A: Yes, provided that External/Status Interrupts are enabled (bit 0 in register WR1).

Q: Is the \overline{MI} line used by the SIO if no interrupts are enabled?

A: No, and in this case the \overline{MI} input should be tied high.

Q: Will the SIO continue to interrupt for a condition if the condition persists and the interrupt remains enabled?

A: Yes.

Q: What is the maximum data rate of the SIO?

A: It is 1/5 the rate of the system clock (CLK). For example, if the system clock operates at 4 MHz, the SIO's maximum transfer rate is 800K bits (100K bytes) per second.

Q: What pins are edge sensitive and should be strapped to avoid strange interrupts?

A: The external synchronization (\overline{SYNC}) pins and any other external status pins that are not used, including \overline{CTS} , and \overline{DCD} .

Q: What happens if the transmitter or receiver is disabled, while processing a character, by turning off its associated enable bit (bit 3 in register WR5 for transmit or bit 0 in register WR3 for receive)?

A: The transmitter will complete the character transmission in an orderly fashion. The receiver, however, will not finish. It will lose the character being received and no interrupt will occur.

Register Contents

Q: Does the Tx Buffer Empty (bit 2 in register RR0) get set when the last byte in the buffer is in the process of being shifted out?

A: No. The bit is set when the transmit buffer has already become empty. Similarly, the Tx Buffer Empty interrupt will not occur until the buffer is empty. The same is true for reception: the Rx Character Available bit (bit 0 in register RR0) is not set until the entire character is in the receive buffer, and the Rx Character Available interrupt will not occur until the entire character has been moved into the buffer.

Q: If an Rx Overrun error occurs (and bit 5 of register RR1 becomes latched on) because a new character has arrived, which character gets lost?

A: The most recently received character overwrites the next most recently received character.

Q: Does the Reset External/Status Interrupts command reset any of the status bits in register RR0?

A: No. However, when a transition occurs on any of the five External/Status bits in register RR0, all of the status bits are latched in their current position until a Reset External/Status Interrupts command is issued. Thus, the command does permit the appropriate bits of register RR0 to reflect the current signal values and should be done immediately after processing each transition on the channel.

Special Uses

- Q:** If the CPU does not have the return from interrupt sequence (RETI instruction on the Z80 CPU), how may the SIO be informed of the completion of interrupt handling?
- A:** This may be done by writing the Return From Interrupt command (binary, 00111000) to WR0 in Channel A of the SIO.
- Q:** If the CPU can be interrupted but cannot be used with vectored interrupts, how should processing be done?
- A:** Immediately after being interrupted, proceed in a manner similar to polling the SIO for both receive and transmit. Alternatively, the Status Affects Vector bit (bit 2 in register WR1) may be set and a 0 byte placed into the interrupt vector (register WR2 in Channel B). Then, the contents of the interrupt vector can be used to determine the cause of the interrupt and the channel on which the interrupt occurred. This can be queried by reading register RR1 of Channel B. Also, MI should be tied High and no equivalent to an interrupt acknowledge should be issued.
- Q:** How can the Wait/Ready ($\overline{W/RDY}$) signal be used by the CPU in asynchronous I/O?
- A:** The $\overline{W/RDY}$ signal is most commonly used in Block Transfer Mode with a DMA, and this use is described in the *Z80 DMA Technical Manual*. However, $\overline{W/RDY}$ may be directly connected to the Z80 CPU \overline{WAIT} line in order to use the block I/O instructions OTDR, OTIR, INDR, and INIR. In this case, the SIO can be used for block transfer reception. To do this, the SIO is configured to interrupt on the first character received only (by settings bits 4 and 3 of register WR1 to 01) and additional characters are sensed using the $\overline{W/RDY}$ line. The block I/O instructions decrement a byte counter to determine when I/O is complete.
- Q:** Can the \overline{SYNC} pin have any use in asynchronous I/O?
- A:** It may be used as a general-purpose input. For example, by connecting it to a modem ring indicator, the status of that ring indicator can be monitored by the CPU.

- Q:** How can the SIO be used to transmit characters containing fewer than 5 bits?
- A:** First, set bits 6 and 5 in register WR5 to indicate that five or fewer bits per character will be transmitted. The SIO then determines the number of bits to actually transmit from the data byte itself. The data byte should consist of zero or more 1s, three 0s, and the data to be transmitted. Thus, beginning the data byte with 11110001 will cause only the last bit to be transmitted:

Contents of data byte
(d = arbitrary value)

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	
1	1	1	1	0	0	0	d	1
1	1	1	0	0	0	d	d	2
1	1	0	0	0	d	d	d	3
1	0	0	0	d	d	d	d	4
0	0	0	d	d	d	d	d	5

*The rightmost number of bits indicated will be transmitted

- Q:** Can a Break sequence be sent for a fixed number of character periods?
- A:** Yes. Break is continuously transmitted as logic 0 by setting bit 4 of register WR5. You can then send characters to the transmitter as long as the Break level is desired to persist. A Break signal, rather than the characters sent, will actually be transmitted, but each bit of each character sent will be clocked as if it were transmitted. The All Sent bit, bit 0 of register RR1, is set to 1 when the last bit of a character is clocked for transmission, and this may be used to determine when to reset bit 4 of register WR5 and stop the Break signal.
- Q:** If a Break sequence is initiated by setting bit 4 of register WR5, will any character in the process of being transmitted be completed?
- A:** No. Break is effective immediately when bit 4 of WR5 is set. The "all sent" bit in register RR1 should be monitored to determine when it is safe to initiate a Break sequence.

SECTION

6

A Longer Example.

In this section, we give a longer example of asynchronous interrupt-driven full-duplex I/O using the SIO. The code for this example is contained in Appendix A, and the basic routines are flow charted in Figures 7-12.

The example includes code for initialization of the SIO, initialization of a receive buffer interrupt routine, and a transfer routine which causes a buffer of up to 80 characters of information to be transmitted on Channel A and a buffer of up to 80 characters of information to be received from Channel A. The transfer routine stops when either all data is received or an error occurs. Completion of an operation on a buffer for both receive and transmit is indicated by a carriage return character. Additional routines (not included in this example) would be needed to call the initialization code and initiate the transfer routine. Therefore, we do not present a complete example; that would only be possible when all details of a particular communication environment and operating system were known.

The code begins by defining the value of the SIO control and data channels, followed by location definitions for the interrupt vector. There is then a series of constant definitions of the various fields in each register of the SIO. This is followed by a table-driven SIO initialization routine called "SIO_init," shown in Figure 7, which uses the table beginning at the location "SIOtable." The SIO_init routine initializes the SIO with exactly the same

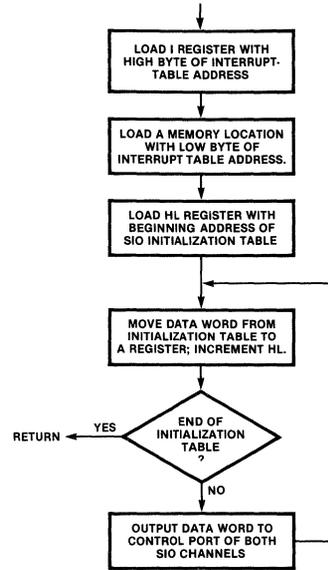


Figure 7. Interrupt-Driven Initialization Routine

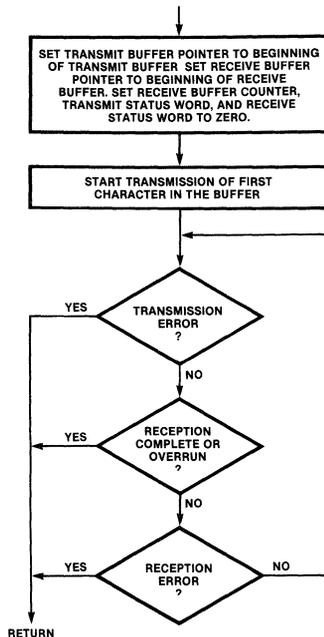


Figure 8. Interrupt-Driven Transmit Routine

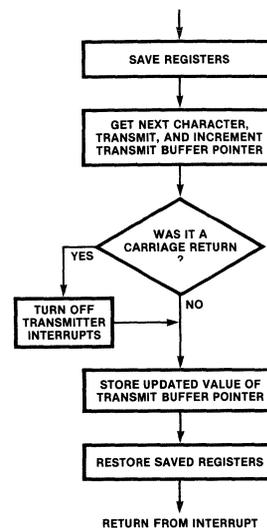


Figure 9. Transmitter Buffer Empty Interrupt Routine

A Longer Example
(Continued)

parameters as the interrupt-driven example in the previous section. The table-driven version is presented simply as an alternative means of coding this material.

A short routine for filling the receive buffer with "FF" (hex) characters and buffer definitions follows the SIO_Init routine. This in turn is followed by the transfer routine, Figure 8, which begins transmitting on Channel A; transmission and reception is thereafter directed by the interrupt routines. After the transfer routine begins output, it checks for various error conditions and loops until there is either completion or an error.

Then the four interrupt routines follow: TxBEmpty, Figure 9, is called on a transmit buffer interrupt; it begins transmission of the next character in the buffer. A carriage return stops transmission. RecvChar, Figure 10, is called on a normal receive interrupt; it places the received character in the buffer if the buffer is not full and updates receive counters. The routines SpRecvChar, Figure 11, and ExtStatus, Figure 12, are error interrupts; they update information to indicate the nature of the error.

The code of this example can be used in a situation where data is being sent to a device which echoes the data sent. In such a case, the transmit and receive buffers could be compared upon completion for line or transmission errors.

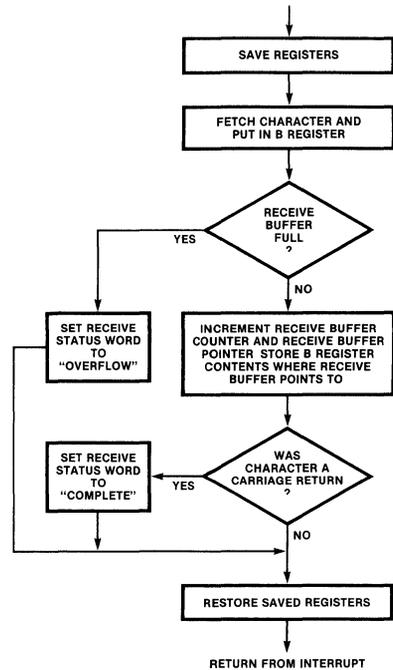


Figure 10. Receive Character Interrupt Routine

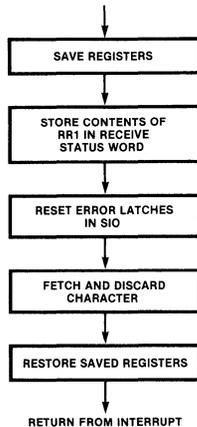


Figure 11. Special Receive Condition Interrupt Routine

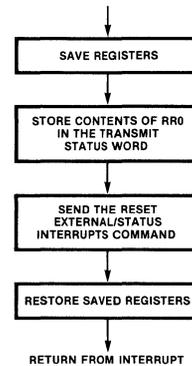


Figure 12. External/Status Interrupt Routine

Appendix A

Interrupt-Driven Code Example

SIO Port Identifiers and System Address Bus Addresses

SIO:	EQU	40H
SIOData:	EQU	SIO + 1
SIOACtrl:	EQU	SIO + 2
SIOBData:	EQU	SIO + 3
SIOBCtrl:	EQU	SIO + 4

Table of Interrupt Vectors

The table (Int_Tab) starts at the lowest priority vector, which should be dddd000d

ORG	0D0H	,starts at address with low , byte = 11010000
Int_Tab:	DEFW	TxBEmpty ,interrupt types for Channel B
	DEFW	ExtStat
	DEFW	RxChar
	DEFW	SpRxCond
	DEFW	TxBEmpty ,interrupt types for Channel A
	DEFW	ExtStat
	DEFW	RxChar
	DEFW	SpRxCond

Command Identifiers and Values

Includes all control bytes for asynchronous and synchronous I/O.

WR0 Commands

R0:	EQU	00H	,SIO register pointers
R1:	EQU	01H	
R2:	EQU	02H	
R3:	EQU	03H	
R4:	EQU	04H	
R5:	EQU	05H	
R6:	EQU	06H	
R7:	EQU	07H	
NC:	EQU	00H	,Null Code
SA:	EQU	08H	,Send Abort (SDLC)
RESI:	EQU	10H	,Reset Ext/Stat Int
CHRST:	EQU	18H	,Channel Reset
EIONRC:	EQU	20H	,Enable Int On Next Rx Char
RTIP:	EQU	28H	,Reset Tx Int Pending
ER:	EQU	30H	,Error Reset
RFI:	EQU	38H	,Return From Int
RRCC:	EQU	40H	,Reset Rx CRC Checker
RTCG:	EQU	80H	,Reset Tx CRC Generator
RTUEL:	EQU	0C0H	,Reset Tx Under/EOM Latch

WR1 Commands

WAIT:	EQU	00H	,Wait function
DRCVRI:	EQU	00H	,Disable Receive interrupts
EXTIE:	EQU	01H	,External interrupt enable
XMTRIE:	EQU	02H	,Transmit interrupt enable
SAVECT:	EQU	04H	,Status affects vector
FIRSTC:	EQU	08H	,Rx interrupt on first character
PAVECT:	EQU	10H	,Rx interrupt on all characters , (parity affects vector)
PDAVCT:	EQU	18H	,Rx interrupt on all characters , (parity doesn't affect vector)
WRONRT:	EQU	20H	,Wait/Ready on receive
RDY:	EQU	40H	,Ready function
WRDYEN:	EQU	80H	,Wait/Ready enable

WR2 Commands

IV:	EQU	00H
-----	-----	-----

WR3 Commands

B5:	EQU	00H	,Receive 5 bits/character
RENABL:	EQU	01H	,Receiver enable
ENRCVR:	EQU	01H	,Receiver enable
SCLINH:	EQU	02H	,Sync character load inhibit
ADSRCH:	EQU	04H	,Address search mode
RRCRCN:	EQU	08H	,Receive CRC enable
HUNT:	EQU	10H	,Enter hunt mode
AUTOEN:	EQU	20H	,Auto enables
B7:	EQU	40H	,Receive 7 bits/character
B6:	EQU	80H	,Receive 6 bits/character
B8:	EQU	0C0H	,Receive 8 bits/character

WR4 Commands

SYNC:	EQU	00H	,Sync modes enable
NOPTY:	EQU	00H	,Disable parity
ODD:	EQU	00H	,Odd parity
MONO:	EQU	00H	,8 bit sync character
C1:	EQU	00H	,X1 clock mode
PARITY:	EQU	01H	,Enable parity
EVEN:	EQU	02H	,Even parity
S1:	EQU	04H	,1 stop bit/character
S1HALF:	EQU	08H	,1 and a half stop bits/character
S2:	EQU	0CH	,2 stop bits/character
BISYNC:	EQU	10H	,16 bit sync character
SDLC:	EQU	20H	,SDLC mode
ESYNC:	EQU	30H	,External sync mode
C16:	EQU	40H	,X16 clock mode
C32:	EQU	80H	,X32 clock mode
C64:	EQU	0C0H	,X64 clock mode

WR5 Commands

T5:	EQU	00H	,Transmit 5 bits/character
XRCRCN:	EQU	01H	,Transmit CRC enable
RTS:	EQU	02H	,Request to send
SELCRC:	EQU	04H	,Select CRC-16 polynomial
XENABL:	EQU	08H	,Transmitter enable
BREAK:	EQU	10H	,Send break
T7:	EQU	20H	,Transmit 7 bits/character
T6:	EQU	40H	,Transmit 6 bits/character
T8:	EQU	60H	,Transmit 8 bits/character
DTR:	EQU	80H	,Data terminal ready

Initialization

SIO_Init:	LD	HL, Int_Tab
	LD	A,H
	LD	I,A
	LD	A,L
	LD	(I_Loc),A
	LD	HL, SIOtable
Init_Loop:	LD	A,(HL) ,loop for initialization
	INC	HL
	CP	0
	RET	Z
	OUT	(SIOACtrl),A
	OUT	(SIOBCtrl),A
	JR	Init_Loop
SIOtable:	DEFB	CR ,table for initialization
	DEFB	R4 + RESI
	DEFB	C64 + ODD + PARITY + S2
	DEFB	R3 + RESI
	DEFB	B8 + AUTOEN + ENRCVR
	DEFB	R5 + RESI
	DEFB	DTR + RTS + T8 + XENABL
	DEFB	R2 + RESI
I_Loc:	DEFS	1 ,location of int table
	DEFB	R1 + RESI ,address
	DEFB	EXTIE + XMTRIE + SAVECT + PAVECT
	DEFB	0

Receiver Buffer Initialization

```

Buf__Init. LD   A,BufLength ,fill receiver buffer
           LD   B,A       , with FF characters
           LD   HL,RBuffer ; to detect errors
           LD   A,OFFH
Buf__l    LD   (HL),A     ,a loop for Buf__Init
           INC   HL
           DJNZ Buf__l
           RET
BufLength: EQU   80      ;buffer length
XBuffer:  DEFS  BufLength ,Tx buffer starting location
RBuffer:  DEFS  BufLength ,Rx buffer starting location
XBufPtr   DEFS  2       ;Tx pointer
RBufPtr   DEFS  2       ;Rx pointer
RBufCtr.  DEFS  1       ;Rx counter

```

Transmit Routine (see Figure 8)

Initiates transmission of a buffer-full of data and terminates when an error is detected or a complete buffer has been received

```

RxStat    DEFS  1       ,Receive Status Word
TxStat    DEFS  1       ,Transmit Status Word
Complete  EQU   1
CR        EQU   0DH
Break     EQU   80H
EOM       EQU   80H
Overflow  EQU   OFFH
Transfer  LD   HL,XBuffer ,setup to begin Tx
           INC   HL
           LD   (XBufPtr),HL
           LD   HL,RBuffer
           LD   (RBufPtr),HL
           XOR   A       ,A = 0
           LD   (RBufCtr),A
           LD   (TxStat),A
           LD   (RxStat),A
           LD   A,SIOADat ;start Tx task
           LD   C,A
           LD   HL,(XBuffer) ,first character
           LD   A,(HL)
           OUT  (C),A
Tloop.    LD   A,(TxStat) ,await Tx completion or error
           CP   0
           RET  NZ
           LD   A,(RxStat)
           CP   Overflow
           RET  Z
           CP   Complete
           RET  Z
           JR   NZ,Tloop
           RET

```

Transmitter Buffer Empty Routine (see Figure 9)

```

TxBEmpty  PUSH  AF
           PUSH  BC
           PUSH  HL
           LD   HL,(XBufPtr)
           LD   A,SIOADat
           LD   C,A
           LD   A,(HL)
           OUTI
           CP   CR
           JR   NZ,TxBExit ,last character?
           LD   A,RTIP    ,Reset Tx Int Pending
           INC   C
           OUT  (C),A    ;to control port
TxBExit   LD   (XBufPtr),HL ;save pointer
           POP  HL
           POP  BC
           POP  AF
           EI
           RETI

```

Receive Character Routine (see Figure 10)

```

RxChar.   PUSH  AF
           PUSH  BC
           LD   A,SIOADat
           LD   C,A
           IN   A,(C)    ,get character
           LD   B,A
           LD   A,(RBufCtr)
           CP   BufLength
           JR   Z,Over
           INC  A       ;bump counter
           LD   (RBufCtr),A
           LD   A,B
           LD   HL,(RBufPtr) ,bump pointer
           LD   (HL),A
           INC  HL
           LD   (RBufPtr),HL
           CP   CR
           JR   NZ,RxExit
           LD   A,Complete
           LD   (RxStat),A
           JR   RxExit
Over       LD   A,Overflow ,indicate error
           LD   (RxStat),A
RxExit    POP  BC
           POP  AF
           EI
           RETI

```

Special Receive Condition Routine (see Figure 11)

```

SpRxCond: PUSH  AF
           PUSH  BC
           LD   A,SIOADat
           LD   C,A
           LD   A,R1    ,get RR1
           INC  C
           OUT  (C),A
           IN   A,(C)
           LD   (RxStat),A ,save status
           LD   A,ER    ;Reset Errors
           DEC  C
           OUT  (C),A
           DEC  C
           IN   A,(C)    ;get character
           POP  BC
           POP  AF
           EI
           RETI

```

External/Status Routine (see Figure 12)

```

ExtStatus: PUSH  AF
           PUSH  BC
           LD   A,SIOACtrl
           LD   C,A
           IN   A,(C)    ;get RR0
           LD   (TxStat),A
           LD   A,RESI   ,Reset Ext Stat Int
           OUT  (C),A
           POP  BC
           POP  AF
           EI
           RETI

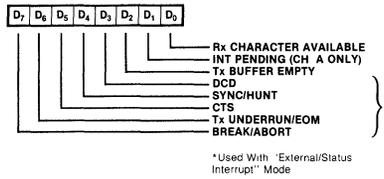
```

END

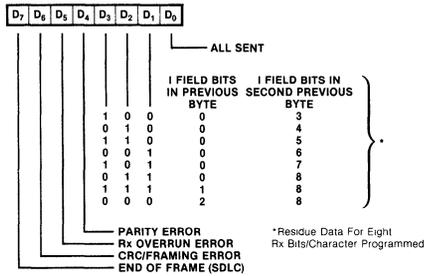
Appendix B

Read Register Bit Functions

READ REGISTER 0

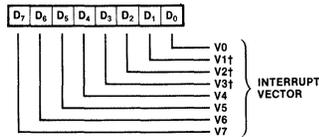


READ REGISTER 1†



†Used With Special Receive Condition Mode

READ REGISTER 2

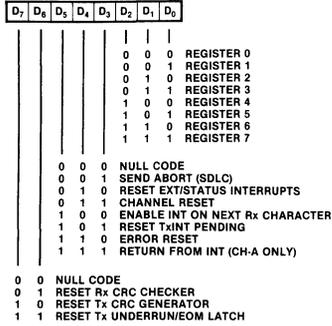


†Variable if "Status Affects Vector" is Programmed

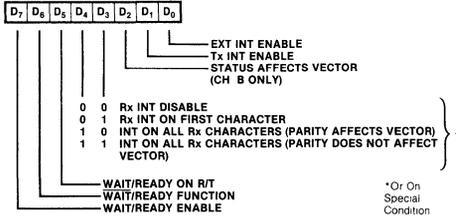
Appendix C

Write Register Bit Functions

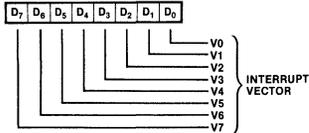
WRITE REGISTER 0



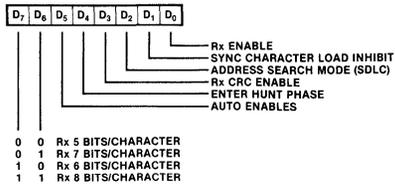
WRITE REGISTER 1



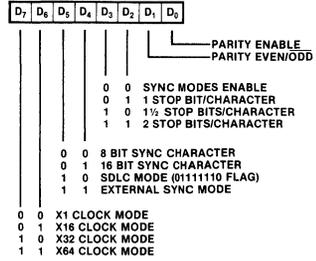
WRITE REGISTER 2 (CHANNEL B ONLY)



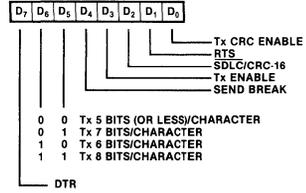
WRITE REGISTER 3



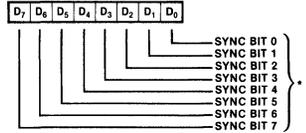
WRITE REGISTER 4



WRITE REGISTER 5

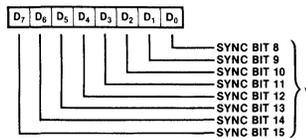


WRITE REGISTER 6



*Also SDLC Address Field

WRITE REGISTER 7



For SDLC it Must Be Programmed to 01111110 For Flag Recognition



Application Brief

March 1981

INTRODUCTION This application brief describes the use of the Z80 SIO with the increasingly popular Synchronous Data Link Control (SDLC) communications protocol. A general description of the SDLC protocol and implementation of the protocol using the SIO are discussed. Descriptions for transmit and receive operations are given for use with simple control frame sequences.

The reader should be familiar with hardware aspects of the SIO such as interfacing to the CPU and a modem. A more detailed description of the SDLC protocol is given in the IBM publication Synchronous Data Link Control General Information (document # GA27-3093-2). A description of the Z80 SIO can be found in the Zilog Data Book (document # 00-2034-A).

DESCRIPTION Data communication today requires a communication protocol that can transfer data quickly and reliably. One such protocol, Synchronous Data Link Control (SDLC), is the link control used by the IBM Systems Network Architecture (SNA) communication package. SDLC is actually a subset of the International Standards Organization (ISO) link control called High Level Data Link Control (HDLC), which is used for international data communication.

blished for a particular device, the other devices ignore the message until the next flag character is detected.

SDLC is a Bit-Oriented Protocol (BOP). It differs from Byte-Control Protocols (BCPs), such as bisync, in having a few bit patterns for control functions instead of several special character sequences. The attributes of the SDLC protocol are position dependent rather than character dependent, so control is determined by the location of the byte as well as by the bit pattern.

The address field contains one or more octets that are used to select a particular station on the data link. An address of all 1s is a global address code that selects all the devices on the link. When a primary station sends a frame, the address field is used to select a secondary station. When a secondary station sends a message to the primary station, the address field contains the secondary station address, i.e., the source of the message.

A character in SDLC is sent as an octet, a group of eight bits. Several octets combine to form a message frame in such a way that each octet belongs to a particular field. Each message frame consists of an opening flag, address, control, information, Frame Check Sequence (FCS), and closing flag fields. The flag field contains a unique binary pattern, 01111110, which indicates the beginning and end of a message frame. This pattern simplifies the hardware interface in receiving devices so that multiple devices connected to a common link do not conflict with one another. The receiving devices respond only after a valid flag character has been detected. Once communication is esta-

The control field follows the address field and contains information about the type of frame being sent. The control field consists of one octet and is always present.

The information field consists of zero or more 8-bit octets and contains any actual data transferred. However, because of the limitations of the error-checking algorithm used in the frame-check sequence, maximum recommended block size is approximately 4096 octets.

The Frame Check Sequence (FCS) follows the information field or the control field, depending on the type of message frame sent. The FCS is a 16-bit Cyclic Redundancy Code (CRC) of the bits in the address, control, and information fields. The FCS is based on the CRC-CCITT code, which uses the polynomial $(x^{16} + x^{12} + x^5 + 1)$. The Z80 SIO contains the circuitry necessary to generate and check the FCS field.

Zero Insertion/deletion is a feature of SDLC that allows any data pattern to be sent. Zero insertion occurs when five consecutive 1s in the data pattern are transmitted. After the fifth 1, a 0 is inserted before the next bit is sent. The data is not affected in any way except that there is an extra 0 in the data stream. The receiver counts the 1s and deletes the 0 following the five consecutive 1s, thus restoring the original data pattern. Zero insertion and deletion is necessary because of the hardware constraint of searching for a flag character or abort sequence. Six 1s preceded and followed by a 0 indicate a flag character. Seven to 14 1s signify an abort, while an idle line (inactive) is indicated by 15 or more 1s. Under these three conditions, zero insertion/deletion is inhibited. Figure 2 illustrates the various line conditions.

SDLC protocol differs from other synchronous protocols with respect to frame timing. In bisync, for example, a host computer might interrupt transmission temporarily by sending sync characters instead of data. This suspended condition could continue as long as the receiver does not time out. With SDLC, however, it is illegal to send flags in the middle of a frame to idle the line. Such an occurrence causes an error condition and disrupts orderly operation. Therefore, the transmitting device must send a complete frame without interruption. If a message cannot be completed, the primary station sends an abort and resumes message transmission later. These conditions are discussed later in the Programming section of this brief.

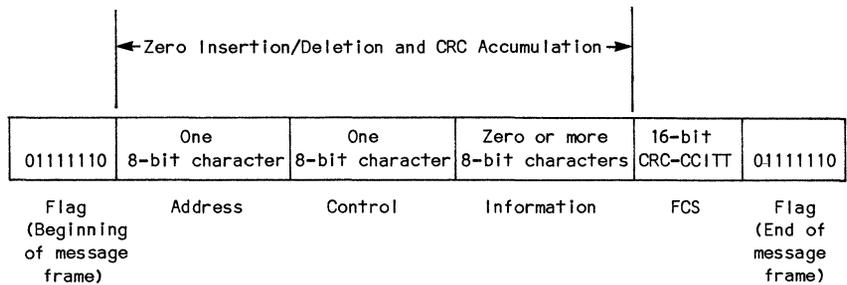


Figure 1. A Typical SDLC Message Frame Format

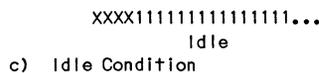
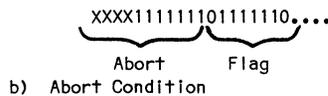
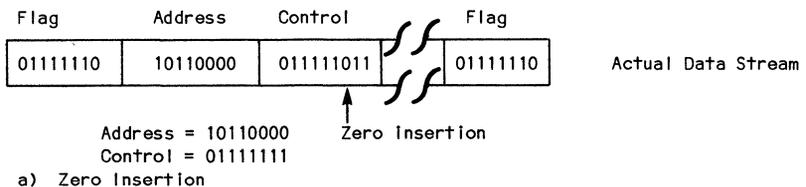


Figure 2. Bit Patterns for Various Line Conditions

**PROGRAMMING
THE SIO**

Implementation of the SDLC protocol with the Z80 SIO is simplified by the design of the SIO. This section discusses four areas of SIO programming: initialization, transmit operation, receive operation, and exception condition processing.

Initialization defines the basic mode of operation for the SIO. Table 1 shows the sequence of steps used to initialize the SIO, along with the necessary parameters. Since vectored interrupts are used, the SIO is programmed with the status affects vector (SAV) bit (WR1, bit 2) set.

Other function bits that can be included are the external interrupt enable bit (WR1, bit 0), which results in an interrupt for each DCD or CTS change, T_x underrun or abort change; address search bit (WR3, bit 2), which when set, prevents the SIO from responding to data received unless the address byte matches the contents of WR6 or the global (FFH) address; auto enable bit (WR3, bit 5), which causes the inactive CTS level to disable the transmitter and the inactive DCD level to disable the receiver; and DTR (WR5, bit 7) and RTS (WR5, bit 1), which can be used to control a modem or other such device.

Once the SIO is initialized and the transmitter is enabled, it sends flag characters continuously until a message begins transmission. These flag characters consist of the full 8-bit pattern. Although the SIO can receive flag characters with shared 0s (011111011111011110...), it can only transmit flag characters without shared 0s (011111001111100111110...).

Table 1. SIO Initialization Sequence

Register	Data	Function
0	00011000	Channel reset
2	(Vector)	Interrupt vector lower eight bits (channel B only)
4	00100000	SDLC mode
1	00011111	Interrupt control
6	(Address)	R_x address field
7	01111110	Flag field
5	11101011	T_x character length, enable, CRC enable RTS and DTR
3	11001001	R_x character length, enable, and CRC enable

**TRANSMIT
OPERATION**

After the SIO has been initialized and enabled, it can begin sending SDLC frames by software activation of the transmitter. Activating the transmitter includes resetting the transmitter inactive semaphore (a program indicator), resetting the T_x CRC accumula-

tion, sending a character to the SIO, and resetting the T_x underrun/EOM latch in the SIO. Figure 3 shows the sequence for transmitting a typical control message frame using interrupts.

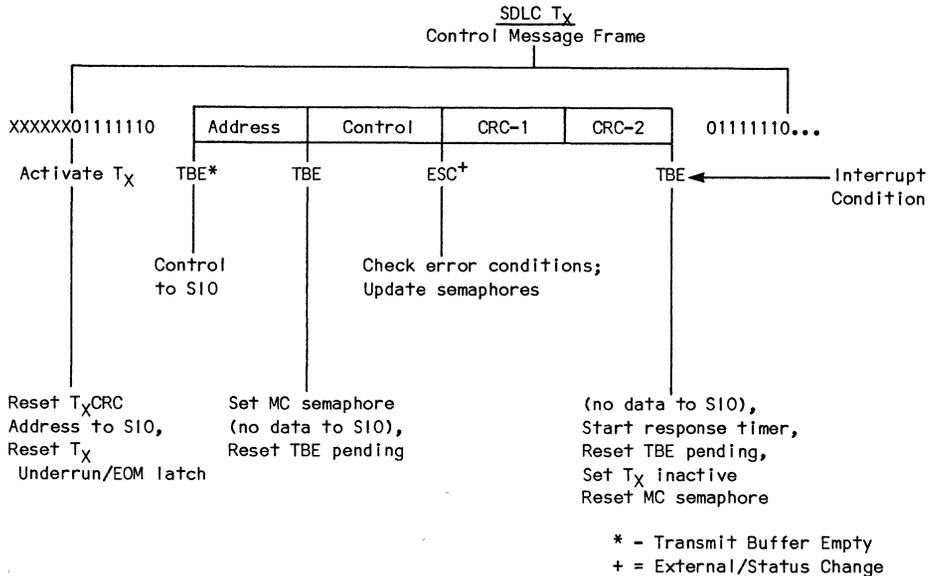


Figure 3. A Typical Transmit Control Frame Sequence

When the SIO is loaded with the first data character (address byte), it stores the character in the T_X buffer until the current flag character has completed shifting. After the address byte is transferred into the shift register, a Transmit Buffer Empty (TBE) interrupt occurs. The program then loads the control character into the SIO and continues processing. The next TBE interrupt is ignored by the program (and no further data is sent to the SIO), but a Reset T_X Interrupt Pending command is issued to the SIO to clear the TBE interrupt condition. Also, the program Message completed (MC) semaphore is set so that appropriate action can be taken when the next TBE interrupt occurs.

When the last data character (the control byte) has been shifted out of the SIO, the T_X underrun/EOM latch is set because the SIO buffer was not loaded with a character on the previous TBE interrupt. As a result, an External/ Status Change (ESC) interrupt occurs and the SIO begins transmitting the FCS bytes automatically. In the ESC inter-

rupt service routine, the program checks for other condition changes including CTS, DCD, and abort, and passes the status on to the program at the next-higher level.

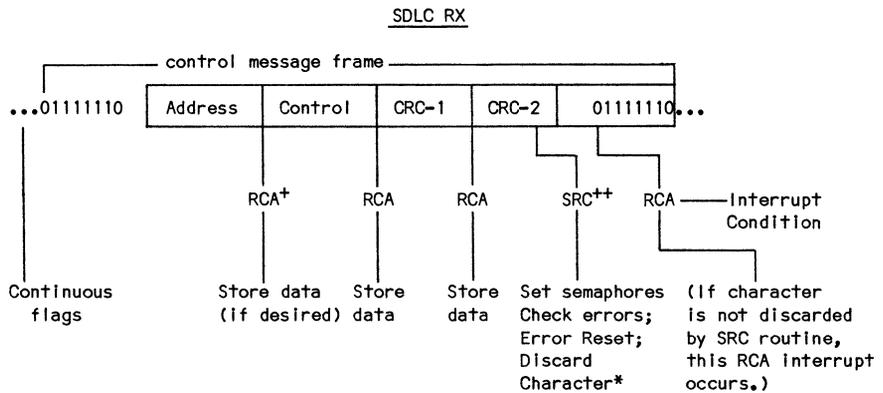
After the FCS bytes have been shifted out, the SIO generates a TBE interrupt to indicate that a flag character is being transmitted. The TBE interrupt service routine interprets the MC semaphore and determines that the frame has completed transmission. The program then clears the MC semaphore, sets the Transmitter Inactive semaphore, starts a timer for a response from the receiving device, and clears the TBE interrupt condition. At this point, transmission of an SDLC message frame is complete and another message frame may be sent.

If the transmitter is to be turned off, the program must allow at least a two-character time delay before disabling the transmitter. This can be accomplished by connecting the SIO T_X clock line to the input of a counter and having the counter interrupt the CPU when the bit count expires.

RECEIVE OPERATION

The SDLC receive sequence is slightly less complex than the transmit sequence. To begin, the SIO enters Hunt mode when any of three conditions occurs: receive enable, abort detect, or a software command. In Hunt mode the SIO searches for flag characters, and when it detects a flag, the SIO generates an ESC interrupt. This interrupt can be used to signal line activation or the end of an abort condition, depending upon the previous receive condition. For example, when the SIO has been initialized, the receive circuitry

is enabled and immediately begins searching for flag characters (Hunt mode operation). When the first flag is detected, the SIO exits from Hunt mode, which results in an ESC interrupt, and the SIO begins searching for the address field. If the SIO is programmed for Address Search mode and an address is received that does not match the programmed address byte in the SIO, the SIO does nothing until the next flag is found, after which the SIO again searches for an address match.



NOTES

* The SRC routine normally reads the data character to clear the SIO buffer. This should be done after the program issues an Error Reset command.

⁺RCA = Receive Character Available

⁺⁺SRC = Special Receive Condition (higher priority than RCA)

Figure 4. A Typical Receive Control Frame Sequence

If the address field matches the address byte programmed into the SIO, the SIO generates a Receive Character Available (RCA) interrupt when the address byte is ready to be transferred from the SIO to the CPU. If the SIO is programmed to interrupt on all receive characters, it generates an RCA interrupt for each character received thereafter. It should be noted that the SIO generates the RCA interrupt when a character reaches the top of the receive FIFO rather than when a character is transferred from the shift register to the FIFO. This means that if the FIFO is full of data, each character generates a separate RCA interrupt. This results in a more consistent software routine that does not need to check the receive FIFO, provided there is enough time between character transfers to allow the routine to complete the processing for each character.

After the last FCS byte of a frame is received and processed, the SIO generates a Special Receive Condition (SRC) interrupt, which is of higher priority than the RCA interrupt. In the SRC service routine, RRI is read to determine the cause of the interrupt and the appropriate program semaphores are updated. Normal completion results in no FCS or overrun errors and the End-of-Frame

bit is set. Upon completion of the SRC interrupt service routine, the program issues an Error Reset command to the SIO and reads the data port to discard the received data. If the data is not read and discarded, an RCA interrupt occurs. Now, a complete message frame and the first FCS byte are in the receive buffer.

Figure 4 shows the sequence for a typical control frame received by the SIO. If the address field byte is to be discarded, a program semaphore should initially be set to signal this to the RCA routine. After the address field has been received, the semaphore is cleared and reception continues normally. Note that upon completion of a frame, an RCA interrupt is generated for the first FCS byte and an SRC interrupt is generated for the last CRC byte.

Table 2 lists the contents of the interrupt service routines used with the SIO. The wake routine is not an interrupt service routine but is a routine called by the program on the next higher level to begin frame transmission. Once the wake routine is called, the program on the next higher level monitors the T_X active semaphore to determine when the current frame completes transmission and the next frame transmission can begin.

Wake:

```
Clear  $T_X$  inactive semaphore
Reset  $T_X$  CRC
Data to SIO
  (Address field byte)
Reset  $T_X$  Underrun/EOM latch
```

Transmit Buffer Empty (TBE):

```
If (MC cleared)
  If (buffer not empty)
    Data to SIO
  Else,
    Set MC semaphore
    Reset TBE condition
Else,
  Clear MC
  Set  $T_X$  inactive
  Reset TBE condition
  Start Response timer
```

External/Status Change (ESC):

```
Clear DCD, CTS, abort semaphores
If (abort)
  Set abort semaphore
Else if (DCD change)
  Set DCD semaphore
Else if (CTS change)
  Set CTS semaphore
```

Receive Character Available (RCA):

```
If (EOF)
  Read and discard data
Else,
  Store data
```

Special Receive Condition (SRC):

```
Read SIO RRI
If (EOF)
  Set EOF semaphore
Else if (CRC error)
  Set  $R_X$  CRC error semaphore
Else if ( $R_X$  overrun)
  Set  $R_X$  overrun semaphore
Issue Error Reset
Read data & discard
```

Table 2. SIO SDLC Interrupt Service Routines

**EXCEPTION
CONDITION
OPERATION**

Most of the exception conditions encountered in the SDLC protocol have been discussed in the previous sections. They include abort detect and DCD or CTS change. This section further describes some of the more unusual conditions.

DCD and CTS Change. The program handles DCD and CTS change by updating its semaphores each time an ESC interrupt occurs. In this manner, the program on the next higher level monitors the semaphores and determines a course of action based on what these semaphores indicate.

Abort and Idle Line Detect. Abort and idle line detect are a bit more complicated, since they result in similar interrupt operations. An abort occurs during a valid message frame. If the abort time is greater than 14 bits, an idle line is detected. This detection can be

done by activating a timer when the ESC interrupt that signals a marking line occurs. If another ESC interrupt occurs before the timer times out, the line is in an abort condition. If the timer times out before another ESC interrupt occurs, then the line is idle and the program can pursue an appropriate course of action. A possible mechanism for implementing the timer function is to use a programmable counter that is tied to the receive clock line to count bits. The counter is programmed for eight clock transitions and is started as soon as the SIO interrupts the CPU with an abort condition. Only eight clock transitions need to be counted because by the time the SIO generates the ESC interrupt, at least seven 1s have already passed. Figure 6 shows the abort/idle line timing and the interrupts resulting from the line changes.

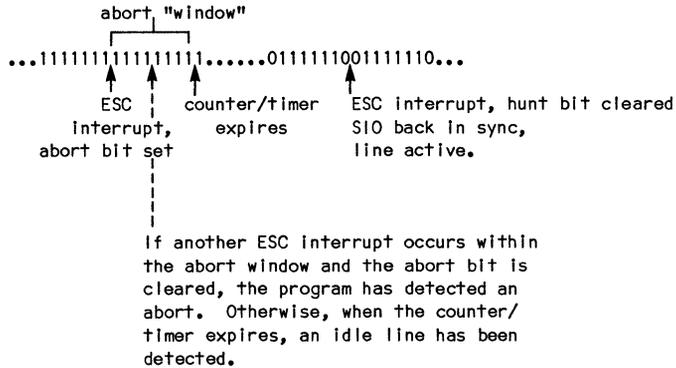


Figure 6. Abort/Idle Line Conditions

CONCLUSION

This brief describes implementation of the SDLC protocol using the SIO in an interrupt-driven environment. Descriptions for transmit and receive operations are given for use with simple control frame sequences. For frames that transfer data, the sequences are similar except for transmit, where a data character is sent to the SIO for a TBE interrupt. For receive, multiple RCA inter-

rupts occur for each data byte received.

The Z80 SIO enhances system performance by minimizing CPU intervention during data transfers using the SDLC protocol. Performance can be improved further by using the Z80 DMA with the SIO, resulting in an efficient system configuration that reduces CPU interaction to a minimum.

APPENDIX

Following is the listing of a simple SIO test program that uses the SDLC protocol. This program uses vectored interrupts to send a short SDLC control frame consisting of Address 9EH, Control 19H, and Data 81H. The response timer times the response of the receiving station after a message has been

sent. If the response timer expires, the program on the next higher level normally retransmits the message frame (if the retransmit count has not yet expired). This program transmits continuously until the processor is reset or interrupted by an external source.

LOC	OBJ	CODE	M	STMT	SOURCE	TEST SDLC STATEMENT	ASM 5 9
1	,					SIO SDLC TEST PROGRAM	
2							
3	,	[0]			01-21-81/MDP		INITIAL CREATION
4							

```

TEST. SDLC
LOC  OBJ CODE M STMT SOURCE STATEMENT                                ASM 5 9

5 ,      THIS PROGRAM SENDS ADDRESS 9EH, CONTROL 19H,
6 ,      AND DATA 81H CONTINUOUSLY USING THE Z80 VECTORED
7 ,      INTERRUPT MODE. THE SID IS INITIALIZED TO USE
8 ,      SDLC WITH THE BAUD RATE CLOCK SUPPLIED BY
9 ,      HARDWARE INTERNAL TO THE SYSTEM.
10
11 ,      EQUATES
12
13 ADDRESS: EQU      9EH      ; ADDRESS FIELD
14 CTRL:   EQU      19H      ; CONTROL FIELD
15 DATA:  EQU      81H      ; INFORMATION FIELD
16 MSQLEN: EQU      1        ; MESSAGE LENGTH
17 RAM:    EQU      2000H     ; RAM ORIGIN
18 RAMSIZ: EQU      1000H    ; RAM SIZE
19 SIODA:  EQU      0        ; SIO PORT A DATA
20 SIOCA:  EQU      SIODA+1   ; SIO PORT A CTRL
21 SIOCB:  EQU      SIODA+2   ; SIO PORT B DATA
22 SIOCB:  EQU      SIODB+1   ; SIO PORT B CTRL
23 CIOC:   EQU      8        ; CIO PORT C
24 CIOB:   EQU      CIOC+1    ; CIO PORT B
25 CIOA:   EQU      CIOC+2    ; CIO PORT A
26 CIOCTL: EQU      CIOC+3    ; CIO CTRL PORT
27 BAUD:   EQU      9600     ; ASYNC BAUD RATE
28 RATE:   EQU      BAUD/100
29 CIOCNT: EQU      9216/RATE
30 LITE:   EQU      0EOH     ; LIGHT PORT
31 RSPCNT: EQU      100      ; RESPONSE TIMER VALUE
32
33 ,      SID PARAMETERS
34
35 SIOWRO: EQU      0
36 CHRES:  EQU      18H      ; CH. RESET CMD
37 ESCRES: EQU      10H      ; ESC RESET CMD
38 TBERES: EQU      28H      ; TBE RESET CMD
39 RETIA:  EQU      38H      ; RETI CH. A
40 ENINRX: EQU      20H      ; ENAB. INT. NEXT RX
41 SRCRES: EQU      30H      ; SRC RESET CMD
42 RCRCRE: EQU      40H      ; RX CRC RESET CMD
43 TCRCRE: EQU      80H      ; TX CRC RESET CMD
44 EDMRES: EQU      0COH     ; EDM RESET CMD
45
46 SIOWR1: EQU      1
47 WREN:   EQU      80H      ; WAIT/RDY ENABLE
48 RDY:   EQU      40H      ; READY FUNCT.
49 WRONR: EQU      20H      ; WAIT/RDY ON RX
50 RXIFC: EQU      8        ; RX INT FIRST CHAR
51 RXIAP: EQU      10H      ; RX INT ALL + PARITY
52 RXIA:  EQU      18H      ; RX INT. ALL
53 SIOSAV: EQU      4        ; STATUS AFFECTS VECT
54 ; (CH. B ONLY)
55 TXI:   EQU      2        ; TX INT. ENABLE
56 EXTI:  EQU      1        ; EXT INT. ENABLE
57
58 SIOWR2: EQU      2        ; (CH B ONLY)
59
60 SIOWR3: EQU      3
61 RX8:   EQU      0COH     ; RX 8 BITS
62 RX6:   EQU      80H      ; RX 6 BITS
63 RX7:   EQU      40H      ; RX 7 BITS
64 RX5:   EQU      0        ; RX 5 BITS
65 AUTOEN: EQU      20H     ; AUTO ENABLES
66 HUNT:   EQU      10H     ; HUNT MODE
67 RXCRC: EQU      8        ; RX CRC ENABLE
68 ADSRCH: EQU      4        ; ADDR SEARCH
69 SYNINH: EQU      2        ; SYNC LOAD INHIBIT
70 RXEN:  EQU      1        ; RX ENABLE
71
72 SIOWR4: EQU      4
73 X64:   EQU      0COH     ; 64X CLOCK
74 X32:   EQU      80H      ; 32X CLOCK
75 X16:   EQU      40H      ; 16X CLOCK
76 X1:    EQU      0        ; 1X CLOCK

```

LUC	OBJ CODE	M	STMT	SOURCE	TEST. SDLC STATEMENT	ASM 5 9
			77		EXTSYN: EQU 30H	;EXT SYNC ENABLE
			78		SDLC: EQU 20H	;SDLC MODE
			79		SYN16: EQU 10H	;16 BIT SYNC
			80		SYN8: EQU 0	;8 BIT SYNC
			81		STOP2: EQU 0CH	;2 STOP BITS
			82		STOP15: EQU 8	;1.5 STOP BITS
			83		STOP1: EQU 4	;1 STOP BIT
			84		SYNCEN: EQU 0	;SYNC ENABLE
			85		EVEN: EQU 2	;EVEN PARITY
			86		PARITY: EQU 1	;PARITY ENABLE
			87			
			88	SIQWR5:	EQU 5	
			89	DTR:	EQU 80H	;ACTIVATE DTR
			90	TX8:	EQU 60H	;TX 8 BITS
			91	TX6:	EQU 40H	;TX 6 BITS
			92	TX7:	EQU 20H	;TX 7 BITS
			93	TX5:	EQU 0	;TX 5 BITS
			94	BREAK:	EQU 10H	;TX BREAK
			95	TXEN:	EQU 8	;TX ENABLE
			96	CRC16:	EQU 4	;CRC-16 MODE
			97	RTS:	EQU 2	;ACTIVATE RTS
			98	TXCRC:	EQU 1	;TX CRC ENABLE
			99			
			100	SIQWR6:	EQU 6	;LOW SYNC OR ADDR
			101			
			102	SIQWR7:	EQU 7	;HIGH SYNC OR FLAG
			103			
			104	;	SIQFLG = FLAGS FOR SID STATUS	
			105			
			106	;	BIT --- SET CONDITION	
			107			
			108	;	0 TX ACTIVE	
			109	;	1 MESSAGE COMPLETE	
			110	;	2 CTS ACTIVE	
			111	;	3 DCD ACTIVE	
			112	;	4 ABORT DETECT	
			113	;	5 RX OVERRUN ERROR	
			114	;	6 RX CRC ERROR	
			115	;	7 RX END OF FRAME	
			116	*E		
			117			
			118	;	*** MAIN PROGRAM ***	
			119			
0000			120	ORG	0	
0000	C32000		121	JP	BEGIN	;GO MAIN PROGRAM
			122			
			123	;	INTERRUPT VECTORS	
			124	;	(MUST START ON EVEN BOUNDARY)	
			125			
0010			126	ORG	\$. AND. OFFFOH. OR. 10H	
			127	INTVEC:		
			128	SIQVEC:		
0010	9C00		129	DEFW	CHBTBE	
0012	D100		130	DEFW	CHBESC	
0014	0101		131	DEFW	CHBRCA	
0016	0F01		132	DEFW	CHBSRC	
0018	3B01		133	DEFW	CHATBE	
001A	4301		134	DEFW	CHAESC	
001C	4B01		135	DEFW	CHARCA	
001E	5101		136	DEFW	CHASRC	
			137			
			138	BEGIN:		
0020	314020		139	LD	SP, STAK	; INIT SP
0023	ED5E		140	IM	2	; VECTOR INTERRUPT MODE
0025	3E00		141	LD	A, INTVEC/256	; UPPER VECTOR BYTE
0027	ED47		142	LD	I, A	
0029	214520		143	LD	HL, BUFFER	
002C	369E		144	LD	(HL), ADDRESS	; STORE ADDRESS
002E	23		145	INC	HL	
002F	3619		146	LD	(HL), CTRL	; STORE CTRL BYTE
0031	23		147	INC	HL	
0032	36B1		148	LD	(HL), DATA	; STORE DATA BYTE
0034	CD4C00		149	CALL	INIT	; INIT DEVICES

LOC	OBJ CODE	M	STMT	SOURCE	TEST. SDLC STATEMENT	ASM 5.9
0037	218720		150		LD HL, RBUF	; SETUP READ BUFFER
003A	228520		151		LD (RBPTR), HL	
			152	LOOP:		
003D	213D00		153		LD HL, LOOP	; SETUP STACK FOR RETURN
0040	E5		154		PUSH HL	
0041	CD7D00		155		CALL WAKE	; WAKE TX
			156	LOOP1:		
0044	3A4020		157		LD A, (SIOFLG)	; CHECK TX ACTIVE FLAG
0047	CB47		158		BIT 0, A	
0049	20F9		159		JR NZ, LOOP1	; LOOP IF TX ACTIVE
004B	C9		160		RET	
			161			
			162	INIT:		
			163	SIOINI:		
004C	217001		164		LD HL, SIOTA	; INIT CH A
004F	0E01		165		LD C, SIOCA	
0051	060A		166		LD B, SIOEA-SIOTA	
0053	EDB3		167		OTIR	
0055	217A01		168		LD HL, SIOTB	; INIT CH B
0058	0E03		169		LD C, SIOCB	
005A	0610		170		LD B, SIOEB-SIOTB	
005C	EDB3		171		OTIR	
005E	3E00		172		LD A, 0	; CLEAR FLAG BYTE
0060	324020		173		LD (SIOFLG), A	
			174	CIOINI:		
0063	DB0B		175		IN A, (CIOCTL)	; INSURE STATE 0
0065	AF		176		XOR A	; POINT TO REG 0
0066	D30B		177		OUT (CIOCTL), A	
0068	DB0B		178		IN A, (CIOCTL)	; CLEAR RESET OR STATE 0
006A	AF		179		XOR A	
006B	D30B		180		OUT (CIOCTL), A	; POINT TO REG 0
006D	3C		181		INC A	; WRITE RESET
006E	D30B		182		OUT (CIOCTL), A	
0070	AF		183		XOR A	; CLEAR RESET COND
0071	D30B		184		OUT (CIOCTL), A	
0073	218A01		185		LD HL, CLST	; INIT CIO
0076	060E		186		LD B, CEND-CLST	
0078	0E0B		187		LD C, CIOCTL	
007A	EDB3		188		OTIR	
007C	C9		189		RET	
			190			
			191	WAKE:		
007D	3A4020		192		LD A, (SIOFLG)	; SET ACTIVE FLAG
0080	CBC7		193		SET 0, A	
0082	324020		194		LD (SIOFLG), A	
0085	214520		195		LD HL, BUFFER	; SET BUFFER PTR
0088	224320		196		LD (BUFPTR), HL	
008B	3E03		197		LD A, 2+MSGLEN	; SET BYTE COUNT
008D	324120		198		LD (BYTES), A	
0090	3E80		199		LD A, TCRCRE	; CLEAR TX CRC
0092	D303		200		OUT (SIOCB), A	
0094	CD9C00		201		CALL CHBTBE	; START TRANSMIT
0097	3E00		202		LD A, EOMRES	; RESET EOM LATCH
0099	D303		203		OUT (SIOCB), A	
009B	C9		204		RET	
			205	*E		
			206			
			207	,	INTERRUPT SERVICE ROUTINES	
			208			
			209	CHBTBE.		
009C	CD5901		210		CALL SAVE	; CH B TX BUFFER EMPTY
009F	214020		211		LD HL, SIOFLG	; POINT TO FLAG BYTE
00A2	CB4E		212		BIT 1, (HL)	; CHECK MC FLAG
00A4	201D		213		JR NZ, CHBTB2	; BRANCH IF MESSAGE COMPLE
	TE					
00A6	3A4120		214		LD A, (BYTES)	; CHECK BYTE COUNT
00A9	B7		215		OR A	
00AA	280F		216		JR Z, CHBTB1	; BRANCH IF DATA DONE
00AC	3D		217		DEC A	
00AD	324120		218		LD (BYTES), A	
00B0	2A4320		219		LD HL, (BUFPTR)	
00B3	7E		220		LD A, (HL)	
00B4	D302		221		OUT (SIODB), A	

				TEST SDLC			ASM 5 9
LOC	OBJ CODE	M	STMT	SOURCE	STATEMENT		
00B6	23		222		INC HL		
00B7	224320		223		LD (BUFPTR), HL		
00BA	C9		224		RET		
			225	CHBTB1:			
00BB	CBCE		226		SET 1, (HL)	; SET MC FLAG	
00BD	3E0C		227		LD A, EOMRES		
00BF	D303		228		OUT (SIOCB), A		
00C1	1809		229		JR CHBTB3		
			230	CHBTB2:			
00C3	CB8E		231		RES 1, (HL)	; CLEAR MC FLAG	
00C5	CB86		232		RES 0, (HL)	; SET TX INACTIVE	
00C7	3E64		233		LD A, RSPCNT	; START RESPONSE TIMER	
00C9	324220		234		LD (RSP TMR), A		
			235	CHBTB3:			
00CC	3E2B		236		LD A, TBERES	; RESET TBE INT. PEND	
00CE	D303		237		OUT (SIOCB), A		
00D0	C9		238		RET		
			239				
			240	CHBESC:			
00D1	CD5901		241		CALL SAVE	; CH. B EXTERNAL/STATUS CHG	
00D4	214020		242		LD HL, SIOFLG	; GET FLAG BYTE	
00D7	CB96		243		RES 2, (HL)		
00D9	CB9E		244		RES 3, (HL)		
00DB	CBA6		245		RES 4, (HL)		
00DD	DB03		246		IN A, (SIOCB)	; READ RRO	
00DF	47		247		LD B, A	; STORE IN %B	
00E0	CB58		248		BIT 3, B	; CHECK DCD BIT	
00E2	C4FB00		249		CALL NZ, SETDCD		
00E5	CB68		250		BIT 5, B	; CHECK CTS BIT	
00E7	C4FE00		251		CALL NZ, SETCTS		
00EA	CB78		252		BIT 7, B	; CHECK ABORT BIT	
00EC	C4FB00		253		CALL NZ, SETABT		
00EF	CB4E		254		BIT 1, (HL)	; CHECK MC FLAG	
00F1	2B00		255		JR Z, CHBES1	; BRANCH IF CLEAR	
			256	CHBES1:			
00F3	3E10		257		LD A, ESCRES	; RESET ESC	
00F5	D303		258		OUT (SIOCB), A		
00F7	C9		259		RET		
			260	SETABT:			
00FB	CBE6		261		SET 4, (HL)		
00FA	C9		262		RET		
			263	SETDCD:			
00FB	CBDE		264		SET 3, (HL)		
00FD	C9		265		RET		
			266	SETCTS:			
00FE	CBD6		267		SET 2, (HL)		
0100	C9		268		RET		
			269				
			270	CHBRCA:			
0101	CD5901		271		CALL SAVE	; CH. B RX CHAR AVAIL	
0104	DB02		272		IN A, (SIODB)		
0106	2A8520		273		LD HL, (RBPTR)	; GET READ BUFF PTR	
0109	77		274		LD (HL), A		
010A	23		275		INC HL		
010B	228520		276		LD (RBPTR), HL		
010E	C9		277		RET		
			278				
			279	CHBSRC:			
010F	CD5901		280		CALL SAVE	; CH. B SPECIAL RX COND	
0112	3E01		281		LD A, 1		
0114	D303		282		OUT (SIOCB), A	; READ RR1	
0116	DB03		283		IN A, (SIOCB)		
0118	47		284		LD B, A	; SAVE IN %B	
0119	214020		285		LD HL, SIOFLG		
011C	CB86		286		RES 6, (HL)	; CLEAR CRC ERROR FLAG	
011E	CB78		287		BIT 7, B	; CHECK EOF BIT	
0120	C43801		288		CALL NZ, SETEFF	; BRANCH IF NOT EOF	
0123	CB70		289		BIT 6, B	; CHECK CRC ERROR	
0125	C43501		290		CALL NZ, SETCRC		
0128	CB68		291		BIT 5, B	; CHECK OVRUN BIT	
012A	C43201		292		CALL NZ, SETOVR		
			293	CHBSR1:			
012D	3E30		294		LD A, SRCRES	; ERROR RESET CMD	

LOC	OBJ CODE	M	STMT	SOURCE STATEMENT	TEST. SDLC	ASM 5 9
012F	D303		295	OUT	(SIOCB), A	
0131	C9		296	RET		
			297	SETDVR:		
0132	CBEE		298	SET	5, (HL)	
0134	C9		299	RET		
			300	SETCRC:		
0135	CBF6		301	SET	6, (HL)	
0137	C9		302	RET		
			303	SETEFF:		
0138	CBFE		304	SET	7, (HL)	
013A	C9		305	RET		
			306			
			307	CHATBE:		
013B	CD5901		308	CALL	SAVE ; CH. A TX BUFFER EMPTY	
013E	3E28		309	LD	A, TBERES	
0140	D301		310	OUT	(SIOCA), A	
0142	C9		311	RET		
			312			
			313	CHAESC:		
0143	CD5901		314	CALL	SAVE ; CH. A EXTERNAL/STATUS CHG	
0146	3E10		315	LD	A, ESCRES	
0148	D301		316	OUT	(SIOCA), A	
014A	C9		317	RET		
			318			
			319	CHARCA:		
014B	CD5901		320	CALL	SAVE ; CH A RX CHAR AVAIL	
014E	DB00		321	IN	A, (SIOCA)	
0150	C9		322	RET		
			323			
			324	CHASRC		
0151	CD5901		325	CALL	SAVE ; CH. B SPECIAL RX COND	
0154	3E30		326	LD	A, SRCRES	
0156	D301		327	OUT	(SIOCA), A	
0158	C9		328	RET		
			329			
			330		SAVE REGISTER ROUTINE	
			331			
			332	SAVE.		
0159	E3		333	EX	(SP), HL ; SP = HL	
015A	D5		334	PUSH	DE ; DF	
015B	C5		335	PUSH	BC ; BC	
015C	F5		336	PUSH	AF ; AF	
015D	DDE5		337	PUSH	IX ; IX	
015F	FDE5		338	PUSH	IY ; IY	
0161	CD6F01		339	CALL	GO ; PC	
0164	FDE1		340	POP	IY	
0166	DDE1		341	POP	IX	
0168	F1		342	POP	AF	
0169	C1		343	POP	BC	
016A	D1		344	POP	DE	
016B	E1		345	POP	HL	
016C	FB		346	EI		
016D	ED4D		347	RETI		
			348			
			349	GO		
016F	E9		350	JP	(HL)	
			351	*E		
			352			
			353		CONSTANTS	
			354			
			355	SIOCA:		
0170	00		356	DEFB	SIOWR0 ; CHAN RESET	
0171	18		357	DEFB	CHRES	
0172	01		358	DEFB	SIOWR1 ; CHAN CHARACS	
0173	D2		359	DEFB	WREN+RDY+RXIAP+TXI	
0174	04		360	DEFB	SIOWR4 ; MODE	
0175	4F		361	DEFB	X16+STOP2+EVEN+PARITY	
0176	05		362	DEFB	SIOWR5 ; TX PARAMS.	
0177	AA		363	DEFB	DTR+TX7+TXEN+RTS	
0178	03		364	DEFB	SIOWR3 ; RX PARAMS.	
0179	41		365	DEFB	RX7+RXEN	
			366	SIOEA:	EQU \$	
			367			

LOC	OBJ CODE	M	STMT	SOURCE	STATEMENT	ASM 5 9
			368			
			369	SIOTB:		
017A	00		369	DEFB	S10WR0	; CHAN RESET
017B	18		370	DEFB	CHRES	
017C	02		371	DEFB	S10WR2	; VECTOR REG
017D	10		372	DEFB	S10VEC. AND. 255	
017E	04		373	DEFB	S10WR4	; MODE
017F	20		374	DEFB	X1+SDLC+SYNCEN	
0180	01		375	DEFB	S10WR1	; CHAN CHARACS.
0181	1F		376	DEFB	RXIA+S10SAV+TXI+EXTI	
0182	06		377	DEFB	S10WR6	; ADDRESS
0183	9E		378	DEFB	ADDRESS	
0184	07		379	DEFB	S10WR7	; FLAG
0185	7E		380	DEFB	01111110B	
0186	05		381	DEFB	S10WR5	; TX PARAMS.
0187	EB		382	DEFB	DTR+TXB+TXEN+RTS+TXCRC	
0188	03		383	DEFB	S10WR3	; RX PARAMS.
0189	C1		384	DEFB	RXB+RXEN	
			385	SI0EB:	EQU	\$
			386			
			387	CLST:		
018A	28		388	DEFB	28H	; PORT B MODE
018B	00		389	DEFB	00000000B	
018C	2B		390	DEFB	2BH	; DATA DIRECTION
018D	EE		391	DEFB	11101110B	
018E	1C		392	DEFB	1CH	; CT1 MODE
018F	C2		393	DEFB	11000010B	
0190	16		394	DEFB	16H	; CT1 TC MSB
0191	00		395	DEFB	0	
0192	17		396	DEFB	17H	; LSB
0193	60		397	DEFB	C10CNT	
0194	01		398	DEFB	1	; MASTER CONFIG. REG
0195	F0		399	DEFB	11110000B	
0196	0A		400	DEFB	10	; CT1 TRIGGER
0197	06		401	DEFB	00000110B	
			402	CEND:	EQU	\$
			403	*E		
			404			
			405	;;	DATA AREA	
			406			
2000			407	ORG	RAM	
2000			408	DEFS	64	; STACK AREA
			409	STAK:	EQU	\$
2040			410	SI0FLG:	DEFS	1 ; S10 FLAG BYTE
2041			411	BYTES:	DEFS	1 ; BUFFER BYTE COUNT
2042			412	RSPTMR:	DEFS	1 ; RESPONSE TIMER
2043			413	BUFPTR:	DEFS	2 ; BUFFER POINTER
2045			414	BUFFER:	DEFS	64 ; BUFFER
2085			415	RBPTR:	DEFS	2 ; READ BUFF PTR
			416	RBUF:	EQU	\$
			417			
			418	END		

Binary Synchronous Communication Using the Z80 SIO



Application Note

October 1980

A popular communication protocol used to exchange information between data processing devices has been in use for some time. This protocol, developed by IBM, is called binary synchronous protocol, or bisync. The Z80 SIO provides a flexible and powerful tool for the implementation of the bisync protocol. However, there are some design considerations that require special attention. This paper will discuss these design considerations and offer an approach to using bisync with the Z80 SIO. Specific examples are presented and readers who are unfamiliar with the bisync protocol should refer to the ANSI standard (1) or the IBM publication (2) listed at the end of this paper.

Bisync is a character-oriented protocol with information transmitted in blocks between two (or more) data communication devices. The medium through which this information is conveyed is called the data link. The particular data link discussed in this paper is a point-to-point link using the ASCII transmission code. Other codes, such as EBCDIC, are not covered, but the format for bisync is basically the same. The data link consists of a master station (usually a computer) and a slave station (usually a terminal) with the associated communication gear in between—modems, phone lines, etc. The master station controls message flow by polling and selecting the slave station. Polling involves sending a general request message to the slave station(s) to determine whether or not any of the slaves have data to send (traffic). If a slave station does have traffic, it responds to the poll and the master can then select that particular slave for information exchange. Slaves can only respond to a master device and cannot initiate communication on the data link.

Information is exchanged by means of a well-defined block structure. Message blocks consist of a header, body, and trailer

(Figure 1). The header is made of two or more SYN characters (hence the name bisync), a start of header (SOH) character, and addressing and control information for a particular slave station.

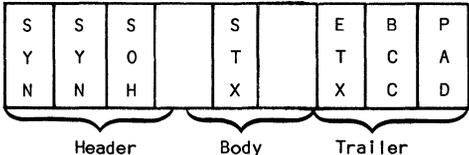


Figure 1. Basic Message Block Format for Bisync Protocol

The body begins with a start of text (STX) character and encompasses the entire text information. The body generally contains ASCII text data, although 8-bit binary data can be transmitted using transparent text mode.

The trailer contains the end of text (ETX) character and the block check character (BCC). The BCC is used for detecting errors through "cyclic redundancy checking" (CRC) or "longitudinal redundancy checking" (LRC).

Error detection is essential when transferring information between data processing equipment. Since ASCII specifies only seven bits for its code, the eighth bit is used for vertical redundancy checking (VRC), more commonly known as character parity. In synchronous communications, character parity is generally odd, whereas in asynchronous communications it is even. Figure 2 shows typical ASCII characters with parity. The SIO can be programmed for 7-bit characters with odd parity enabled to minimize software overhead.

0	1	1	0	0	1	0	1	0	1	1	0	1	0	1
L					M	P	L					M	P	
S					S	A	S					S	A	
B					B	R	B					B	R	
					I	I						I	I	
					T	T						T	T	
					Y	Y						Y	Y	

Figure 2. Odd VRC.
Number of 1s should be odd.

Because VRC applies only to the individual character, the entire message block has an LRC that makes up the BCC. The LRC is a simple bit position checksum where the number of 1s for each position (0 through 6) is even for a block of data. Since the BCC is a character, LRC is subject to the same character parity rules as the rest of the data block. The LRC includes all characters, except SYN, starting with the first character after SOH or STX and up to and including ETX in the trailer (Figure 3). Since the SIO cannot calculate the LRC, the task is left up to the user. LRC can be generated on a microprocessor with little effort by taking the message block and XORing the data with an initial value of zero to provide even LRC.

S	S	S		S	E	B
Y	Y	O		T	T	C
N	N	H		X	X	C

Included in BCC

Figure 3. Characters Included in BCC

Another type of BBC is generated by a cyclic redundancy check (CRC), which results in a more powerful method of block checking. CRC-12 is used for 6-bit transmission code and CRC-16 is used for 8-bit transmission code. CRC is used in lieu of character parity and LRC, as with transparent text mode operation.

The remainder of this paper illustrates how to use the SIO in three special cases of the bisync protocol: transparent text mode, abort/interrupt procedures, and error recovery procedures.

Transparent text mode is useful in bisync when information exchanged between master and slave is not ASCII data. For example, a binary data file (object program) might be sent from master to slave. ASCII transmission code is only seven bits long making it difficult to send 8-bit binary data. One alternative is to convert the binary data to ASCII hex format at the master, transmit it to the slave and reconvert it back into binary at the slave. However, two disadvan-

tages result from this. First, the master and slave require a means of conversion, by either software or hardware, adding cost to the data link. Since the slave (terminal) is burdened most by this, such an approach is usually not feasible. The other disadvantage is that the exchange of information is slower since two (or more) ASCII characters are sent for every eight bits of binary data. The bisync protocol has provisions for sending 8-bit binary data by using transparent text mode transmission. In this mode, character parity is disabled, allowing the full eight bits to be used for data. However, to allow control within the constraints of the protocol, there are certain limitations on the binary data pattern. The primary difference is that during transparent mode some communication control characters are preceded by a DLE character, actually making the control characters a two-character sequence. To distinguish a data byte from a control DLE, the protocol specifies insertion of another DLE. The receiver then throws away the first DLE, keeping the second as data. Table 1 shows the communication control characters that are valid during transparent mode.

Another character change occurs when the SYN character is used for line fill. Normally, the SYN character is ignored, but during transparent mode the SYN is preceded by a DLE, and both are consequently ignored by the receiver. In the event that the CPU does not have a character ready to send, the SIO automatically inserts SYN characters into the data stream. With the SIO programmed for 16-bit sync characters, two syncs are sent from the SIO (write registers WR6 and WR7) when its transmit buffer is empty. In transparent mode, the user must change WR6 and WR7 to DLE, SYN in order for the SIO to provide the proper line fill characters. In accordance with the ANSI standard, line fill characters are not included in the SIO CRC calculation during transmit. During reception in transparent mode, the software must disable CRC accumulation when the DLE SYN character sequence is detected.

While in transparent mode, the user must be concerned with the error detection codes. If parity is enabled in the SIO normally, it must be disabled during transparent mode. This change in SIO operation affects both transmit and receive and should therefore be considered if using full duplex.

Table 1. Control Codes Used In Transparent Mode

DLE	STX	Start of transparent text
DLE	ETB	End of transparent text block
DLE	ETX	End of transparent text
DLE	SYN	Idle sync
DLE	ENQ	Enquiry
DLE	DLE	DLE data
DLE	SOH	Start of transparent header

Since the SIO allows CRC enable/disable on the fly, the software can easily control CRC accumulation in both receive and transmit. During transmit, the CRC must be enabled/disabled before the character is transferred into the serial shift register. During receive, the CRC accumulation is delayed eight bits. After the character is transferred from the serial shift register into the buffer, the user has to read that character, decide whether or not to continue CRC accumulation, and disable/enable CRC before the next character is transferred to the buffer. This is not generally a problem, since character transfers occur about every 833 microseconds at 9600 baud. Table 2 shows the characters included and omitted in the CRC during transparent mode.

Table 2. Characters Included/Omitted in CRC During Transparent Mode

<u>Omitted from CRC</u>		<u>Included in CRC</u>
DLE	SYN	DLE of DLE DLE
DLE	SOH	ETX of DLE ETX
DLE	STX*	ETB of DLE ETB
		STX of DLE STX**

*If not preceded by transparent header within same block

**If preceded by DLE SOH within same block

When CRC accumulation is to be resumed, the software should enable CRC before the desired character is transferred to the receive buffer. For example, suppose a DLE pair is received during transparent text mode. The SIO generates an interrupt when the first DLE is transferred to the receive buffer. The driver program reads the DLE and immediately disables CRC. When the next interrupt occurs, the driver reads the second DLE and immediately enables CRC to include the second DLE into the CRC accumulation.

The second category of interest includes abort and interrupt procedures. There are two types of aborts: block abort and sending station abort. There are three types of interrupts: termination interrupt, reverse interrupt and temporary interrupt.

The block abort is used by the sending station when, in the process of transmitting a data block, the sending station detects an error condition in the data and decides to terminate the block so that the receiving station will discard it. In nontransparent mode, block abort is accomplished by ending the block with an ENQ character, instead of ETX or ETB. The sending station then waits for a reply from the receiver, which should be a NAK. The transparent mode procedure is identical except that a DLE ENQ character

sequence is used. Since a block abort puts the data link back in nontransparent mode, NAK is the valid response the receiver should send in both transparent and nontransparent modes.

The sending station abort is similar to the block abort, except that the sending station does not necessarily do a block abort but simply ends the current message block, waits for a response or timeout, and then sends an EOT to regain control of the data link. The sending station abort is useful when transmission to a particular receiver is necessary due to a higher priority message, buffer overflow condition, error detection, etc. Once the sending station abort sequence is made, the master can perform any data link control function.

From the receiver side, a termination interrupt causes the sending station to stop transmission. Such a procedure is useful when the receiver cannot accept any more data or incurs an error condition, such as paper jam, card jam, hardware error, etc. To accomplish a termination interrupt, the receiving station sends an EOT instead of the normal response. The EOT resets all stations on the link and allows the master to issue any control sequence.

The reverse interrupt (RINT) is used when the receiving station needs to transmit during reception of several message blocks. The RINT occurs when a receiver detects a valid CRC or LRC and, instead of returning an ACK, sends a DLE "<" character sequence to signal an affirmative acknowledgement and to stop transmission of data. Some exceptions and a more detailed description of RINT can be found in the ANSI standard.

The temporary interrupt procedure, WACK (Wait Before Sending Positive Acknowledge), is used by the receiving station to indicate positive acknowledgement and an inability to receive more data. Such a response may be necessary when the receiving station cannot accept data continuously, such as during a printing operation. The WACK consists of a DLE ";" character sequence and is sent in place of an ACK or ACKn. The sending station then sends ENQs (Enquiry) until the receiving station stops sending WACKs. The sending station can resume transmitting data when the receiving station sends an ACK or ACKn.

Recovery procedures provide a means of preventing data link instability. The recovery mechanism consists mainly of timers, grouped into four basic areas, and a NAK counter. The NAK counter is used to prevent repeated NAKs from inhibiting further communications. The sending unit counts how many NAKs it receives for a particular data block so that after a predetermined number of retries, it can recover and pursue another course of

action. The particular count value and course of action taken when the count expires are left up to the user.

Four timers (timer A or response timer, timer B or receiver timer, timer C or gross timer, and timer D or no activity timer) prevent the data link from getting "hung" or going idle for extended periods of time. Generally, the shortest interval is used with timer A, and the longest interval is used with timer D. For maximum system efficiency, however, the receiver timer (timer B) should timeout before the response timer (timer A). The particular implementation of these timers varies from system to system, and some flexibility of exact timer values is left up to the user.

Since it is assumed that interrupts will be used with the SIO, an interrupt driven receiver timer count is kept in memory and is reinitialized each time a character is received (receive interrupt). The same applies for the response timer, except that when a timeout occurs, the transmit driver has several options to follow.

If the SIO is set to transmit CRC on transmit underrun, then the driver could simply set its flags and not fill the buffer. This allows a normal exit, since the SIO will then send its CRC bytes. If the SIO is set to not transmit CRC on transmit underrun, then it sends sync characters (SYN SYN or DLE SYN, whichever was last written to WR6 and WR7) until the transmit buffer is filled or transmit data is set to marking.

In any event, enough time must be allowed after CRC is sent so that the receiver can

properly decode CRC. Because of the character delay in the SIO during CRC accumulation, about 20 clock cycles are necessary after the last CRC byte is sent to ensure adequate decoding time. (See the SIO Technical Manual for further details.) The SIO could be programmed to send pad characters either by disabling parity and sending 8-bit FFs (hex) or by filling WR6 and WR7 with FF hex. If enabled, the SIO automatically sends whatever is in its sync registers upon transmit underrun. Multiple message blocks do not have to be separated by pad characters as long as CRC is valid for the previous message block. However, to insure adequate time for the receiver to process CRC, it is recommended that at least two pad characters follow the last character of a block.

Using the SIO for the bisync protocol is fairly straightforward. Care should be exercised when using the SIO in transparent text mode, but the implementation is greatly simplified by the SIO's flexibility, as compared to other serial communications ICs. The CRC capabilities of the SIO provide a powerful means of maintaining maximum data integrity with minimum software overhead. Coupled with the DMA and the interrupt capabilities of the Z80 processor, the user will find the SIO an excellent choice in serving data communication needs.

- (1) American National Standards Institute.
ANSI X3.28 - 1976.
- (2) "General Information - Binary Synchronous Communications." Pub. number GA27-3004-2.

Serial Communication with the Z80A DART



Application Brief

January 1981

INTRODUCTION Serial data communication is among the most widely used forms of exchanging information with and between computers. The rapid expansion of this form of communication has created the need for low-cost, efficient, and flexible peripheral devices that provide the user with a wide variety of options. The Z80 DART is designed to fill this need by providing two independently programmable,

asynchronous communication channels for a Z80-based system.

This application brief describes the use of the Z80 DART in a Z80-based system. Further information on the Z80 CPU and Z80 DART is available in the Zilog Data Book (document number 00-2034-A), Z8400 Z80 CPU Product Specification (document number 00-2001-A), and the Z8470 Z80 DART Product Specification (document number 00-2044-A).

HARDWARE The hardware for this application consists of a Z8400 Z80 CPU, Z8470 Z80A DART, Z8536 CIO, 4K ROM, and 4K RAM. Figure 1 shows a block diagram of the system. The CIO supplies the bit rate clock for the DART and allows the baud rate for each channel to be determined by the software.

The DART-to-CPU interface consists of eight bidirectional data lines, seven control lines, and three daisy chain interrupt control lines. The data lines are used to transfer data between the DART and the CPU. The direction of data flow on the data lines is determined through the use of the \overline{CE} , \overline{RD} ,

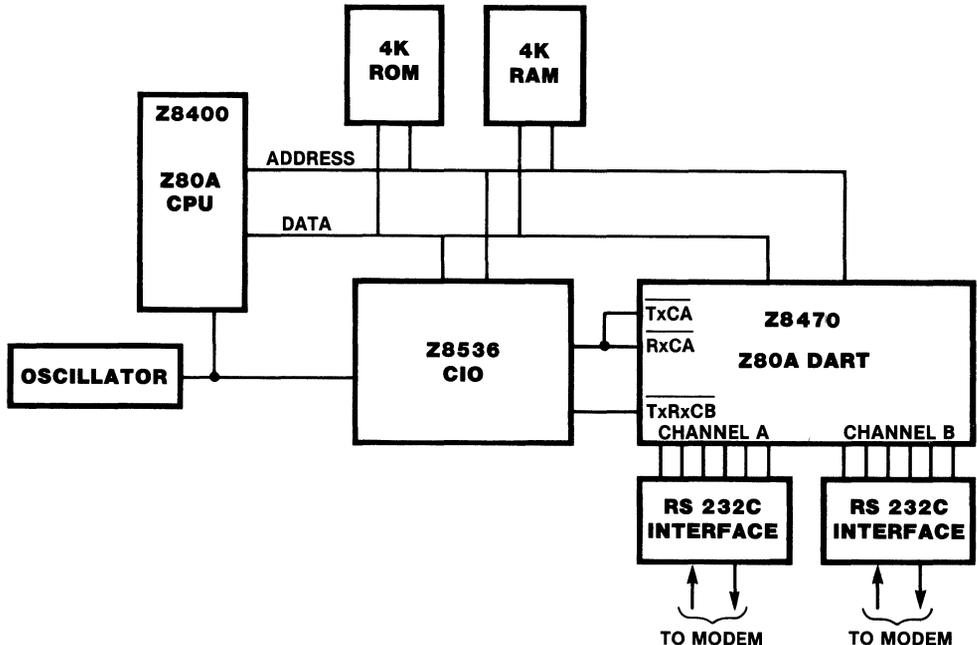


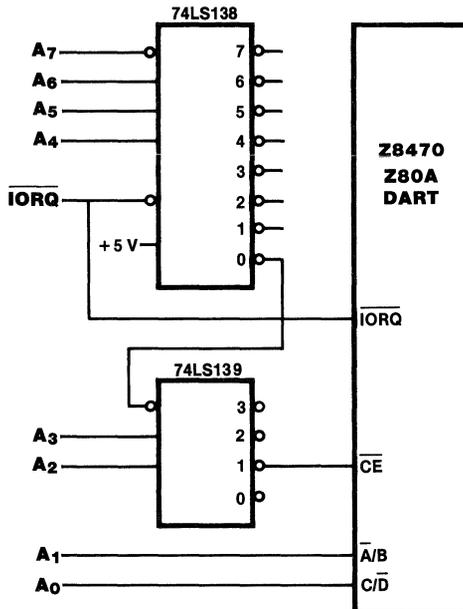
Figure 1. Z80 System Block Diagram

and $\overline{\text{IORQ}}$ control lines. When $\overline{\text{CE}}$ and $\overline{\text{IORQ}}$ are active, a data transfer occurs between the CPU and DART. If $\overline{\text{RD}}$ is active at the same time, data is sent from the DART to the CPU. If $\overline{\text{RD}}$ is not active, data is sent from the CPU to the DART. $\overline{\text{MI}}$ signals an interrupt acknowledge cycle from the CPU in conjunction with $\overline{\text{IORQ}}$. The $\overline{\text{RESET}}$ line performs a device reset on the DART, allowing it to be placed in a known state. The remaining two control lines determine which of the four ports are being accessed. Table 1 shows the relationship of these two lines to the ports.

Table 1. DART Port Addressing

Port	C/D	B/A
Channel A Data	0	0
Channel B Data	0	1
Channel A Control	1	0
Channel B Control	1	1

C/D and B/A are usually tied to the lowest two CPU address lines used for I/O device selection. Figure 2 shows the device-select decode logic used in this application.



NOTE Only the lower eight bits of the address bus are used for I/O select

Figure 2. DART Device Select Logic

External connections to the Z80 DART include serial data and control lines and modem control lines. The serial data lines are Transmit Data (TxD) and Receive Data (RxD) for each channel. Separate transmit and

receive clock inputs are available on channel A (TxCA and RxCA), while a combined transmit/receive clock input is provided for channel B (TxRxCB). To allow separate baud rates for both channels, TxCA and RxCA are tied together and connected to one counter/timer output, and TxRxCB is connected to another counter/timer output. This provides the user with a simple, software-programmable baud rate generator.

The modem control lines provide the user with a means of controlling some external device such as a modem. This is particularly useful for remote applications in which the CPU must determine a course of action based on the status of the modem control lines. For example, Ring Indicator (RI) can be used to signal the CPU that an incoming call needs to be answered, or Data Terminal Ready (DTR) can be used in conjunction with Data Carrier Detect (DCD) to signal the modem that data communications can take place. DTR remains active as long as the DART is communicating over the serial data link. The CPU can "hang up" or disconnect the telephone connection by deactivating DTR. Finally, Request To Send (RTS) and Clear To Send (CTS) are useful in a multidrop configuration; that is, when three or more modems are connected to the same telephone line RTS is used to switch the carrier for a particular modem on or off under software control. CTS is monitored so that after RTS is activated the CPU knows when to start sending data.

The IE1, IE0, and $\overline{\text{INT}}$ lines form the Z80 daisy-chain interrupt controls that enable proper interrupt sequencing. $\overline{\text{INT}}$ is an open-drain, active Low output that is connected to the Z80 CPU $\overline{\text{INT}}$ input, along with a pullup resistor. IE1 is usually connected to the preceding device in the daisy chain or is tied High if there is no preceding device. IE0 is connected to the following device in the daisy chain or is left open. This application example uses interrupts with the Status Affects Vector (SAV) programming option. Interrupts are prioritized internally in the DART according to the various conditions. There are four separate interrupt groups for each channel. Table 2 shows the relative priorities of these interrupts.

Table 2. DART Interrupt Priority

Priority	Function
Highest	Ch. A Special Rx Condition
	Ch. A Rx Char. Available
	Ch. A Tx Buffer Empty
	Ch. A External/Status Change
	Ch. B Special Rx Condition
	Ch. B Rx Char. Available
	Ch. B Tx Buffer Empty
	Ch. B External/Status Change
Lowest	

PROGRAMMING

Programming the Z80 DART consists of two parts: initialization and program operation. Initialization includes defining the operating characteristics of the DART. This is done by writing a series of bytes to the control port of each channel. A detailed description of the programming for the DART can be found in the DART Product Specification (document number 00-2044-A). A listing containing an initialization routine for the DART can be found in the appendix of this brief.

Once initialized, the DART interrupts the CPU for certain conditions that occur. These conditions include Transmit Buffer Empty, Receive Character Available, Special Receive Condition, and External/Status Change for each channel.

The DART generates a Transmit Buffer Empty (TBE) interrupt when a character is transferred from the internal buffer to the shift register. The interrupt service routine determines whether to send another character to the DART or to issue a Reset Tx Interrupt Pending command. If a character is loaded into the DART, the interrupt condition is automatically removed. If a character is not loaded, the software issues a Reset Tx Interrupt Pending command to remove the interrupt condition and also sets an internal program status flag that signals the transmit channel as inactive. When transmission starts from an inactive condition (such as after initialization), the main program must activate the transmitter by sending a character to the DART. In this application, a call to the transmit interrupt service routine activates the transmitter after the buffer and pointers have been initialized.

The Receive Character Available (RCA) interrupt occurs after the DART transfers a character from the serial shift register to the receiver FIFO. The DART can store up to three characters in the FIFO, giving the CPU

some flexibility in receive interrupt timing. Read Register 0 (RR0, bit 0) can be checked to see if any more characters are in the FIFO before exiting the interrupt service routine. If the DART is programmed so that parity does not affect the interrupt vector, parity errors must be checked in the receive service routine. This is done by writing a register pointer to the DART for Read Register 1 (RR1) and then reading the contents. The bit test instructions of the Z80 CPU are particularly useful in determining which bits are set or cleared. Processing for these errors is the same as processing for the Special Receive Condition.

The DART generates a Special Receive Condition (SRC) interrupt if it detects a parity error, overrun, or framing error during reception. When this occurs the programmer should reset the error condition by issuing an Error Reset command to the DART. After the Error Reset command is issued, the programmer should read and discard the data if necessary. If the data is not discarded, then an RCA interrupt occurs immediately after exiting the SRC service routine.

An External/Status Change (ESC) interrupt occurs when the DART detects a change in the external signals (RI, CTS, DCD) or when a receive break condition is initiated or terminated. This is useful in monitoring the interface to the modem where a software flag is set when the break condition is detected and reset when the break condition is cleared. With CTS, DCD, and RI, the same procedure is followed as with a break condition. However, if the auto enable bit is set in the DART, the DART does not transmit data until CTS becomes active, nor does it receive data until DCD becomes active.

The appendix contains the listing of a test program for the DART. While it is by no means complete, it does highlight the interrupt features of the Z80 DART.

CONCLUSION

As do other Z80 peripheral products, the Z80 DART interfaces well with the Z80 CPU. The software required to utilize the features of the DART is conducive to efficient programming. Interrupts provide a key method of maintaining efficient system operation, keeping CPU processing overhead to a minimum.

Other methods of utilizing the DART include a "polled" (noninterrupt) system. Because the

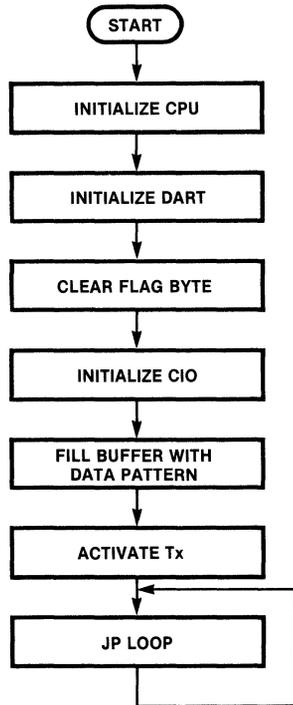
Z80 CPU has three interrupt modes, the DART can be used with the CPU without vectored interrupts. However, such simplicity is usually at the expense of program size and speed.

Nevertheless, the user will find the Z80 DART a viable alternative to more expensive devices when considering the asynchronous communication requirements for any Z80 system.

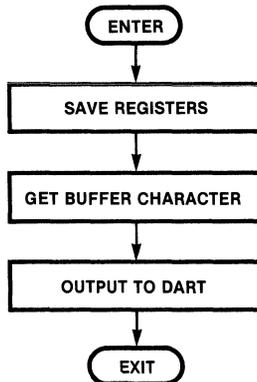
APPENDIX

Following is the listing of a DART test program. Note that all interrupt service routines are dummy routines, except DATBE, which

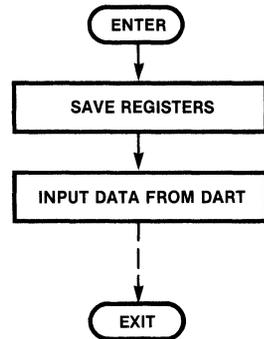
transfers characters from the buffer to Port A transmitter.



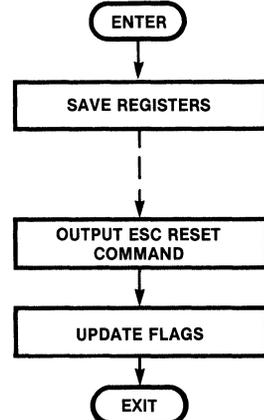
a) Main Program



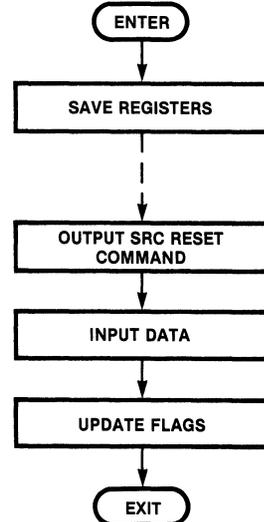
b) DATBE-DART Channel A Transmit Buffer Empty Interrupt Service Routine



c) DARCA-Channel A Receive Character Interrupt Routine



d) DAESC-Channel A External/Status Change Interrupt Routine



e) DASRC-Special Receive Condition Interrupt Routine

NOTE. DARCA, DAESC, AND DASRC are dummy routines.

Figure 3. Flow Diagram for DART Test Program

```

1 , DART TEST PROGRAM
2
3 , EQUATES
4
5 RAM EQU 2000H ,RAM ORIGIN
6 RAMSIZ: EQU 1000H ,RAM SIZE
7 CIA EQU 8 , (CI) PORT A
8 CIOB EQU CIA+1 , CIO PORT B
9 CIOC EQU CIA+2 , CIO PORT C
10 CIOCTL EQU CIA+3 , CIO CTRL PORT
11 BAUD EQU 9600 ,ASYNC BAUD RATE
12 RATE EQU BAUD/100
13 CIOCNT EQU 576/RATE
14 DRTDA: EQU 4 , DART PORT A DATA
15 DRTCA EQU DRTDA+1 , DART PORT A CTRL
16 DRTDB EQU DRTDA+2 , DART PORT B DATA
17 DRTCB EQU DRTDA+3 , DART PORT B CTRL
18
19 , DART PARAMETERS
20
21 DRTWR0 EQU 0
22 CHRES EQU 18H , CH. RESET CMD
23 ESCRES EQU 10H , ESC RESET CMD
24 TBERES EQU 28H , TBE RESET CMD
25 SRCRES EQU 30H , SRC RESET CMD
26 RETIA EQU 38H , RETI CH A
27 ENINRX EQU 20H , ENAB INT. NEXT RX
28
29 DRTWR1 EQU 1
30 WREN EQU 80H , WAIT/RDY ENABLE
31 RDY EQU 40H , READY FUNCT.
32 WRONR: EQU 20H , WAIT/RDY ON RX
33 RXIFC EQU 8 , RX INT FIRST CHAR
34 RXIAP: EQU 10H , RX INT ALL + PARITY
35 RXIA EQU 18H , RX INT. ALL
36 DRTSAV EQU 4 , STATUS AFFECTS VECT
37 , (CH B ONLY)
38 TXI EQU 2 , TX INT. ENABLE
39 EXTI EQU 1 , EX1. INT. ENABLE
40
41 DRTWR2 EQU 2 , (CH. B ONLY)
42
43 DRTWR3 EQU 3
44 RX8 EQU 0COH , RX 8 BITS
45 RX6 EQU 80H , RX 6 BITS
46 RX7 EQU 40H , RX 7 BITS
47 RX5 EQU 0 , RX 5 BITS
48 AUTOEN EQU 20H , AUTO ENABLES
49 RXEN EQU 1 , RX ENABLE
50
51 DRTWR4 EQU 4
52 X64 EQU 0COH , 64X CLOCK
53 X32 EQU 80H , 32X CLOCK
54 X16 EQU 40H , 16X CLOCK
55 X1 EQU 0 , 1X CLOCK
56 STOP2 EQU 0CH , 2 STOP BITS
57 STOP15 EQU 8 , 1.5 STOP BITS
58 STOP1 EQU 4 , 1 STOP BIT
59 EVEN EQU 2 , EVEN PARITY
60 PARITY EQU 1 , PARITY ENABLE
61
62 DRTWR5 EQU 5
63 DTR EQU 80H , ACTIVATE DTR
64 TX8 EQU 60H , TX 8 BITS
65 TX6 EQU 40H , TX 6 BITS
66 TX7 EQU 20H , TX 7 BITS
67 TX5 EQU 0 , TX 5 BITS
68 BREAK EQU 10H , TX BREAK
69 TXEN EQU 8 , TX ENABLE
70 RTS EQU 2 , ACTIVATE RTS
71 *E
72

```

LOC	OBJ CODE	M	STMT	SOURCE	TEST. DART STATEMENT	ASM 5.9
			73		*** MAIN PROGRAM ***	
			74			
0000			75	ORG	0	
0000	C32000		76	JP	BEGIN	; GO MAIN PROGRAM
			77			
			78		INTERRUPT VECTORS	
			79			
0010			80	ORG	\$. AND. OFFFOH. OR. 10H	
			81	INTVEC:		
			82	DRTVEC:		
0010	7E00		83	DEFW	DBTBE	
0012	9000		84	DEFW	DBESC	
0014	8A00		85	DEFW	DBRCA	
0016	A400		86	DEFW	DBSRC	
0018	B800		87	DEFW	DATBE	
001A	D100		88	DEFW	DAESC	
001C	CB00		89	DEFW	DARCA	
001E	E500		90	DEFW	DASRC	
			91			
			92	BEGIN:		
0020	318320		93	LD	SP, STAK	; INIT SP.
0023	ED5E		94	IM	2	; VECTOR INTERRUPT MODE
0025	3E00		95	LD	A, INTVEC/256	; UPPER VECTOR BYTE
0027	ED47		96	LD	I, A	
0029	CD4800		97	CALL	INIT	; INIT DEVICES
002C	210020		98	LD	HL, BUFFER	
002F	063E		99	LD	B, 62	
			100	LOOP:		
0031	78		101	LD	A, B	
0032	F640		102	OR	40H	
0034	77		103	LD	(HL), A	
0035	23		104	INC	HL	
0036	10F9		105	DJNZ	LOOP	
0038	360D		106	LD	(HL), 13	; CR
003A	23		107	INC	HL	
003B	360A		108	LD	(HL), 10	; LF
003D	210020		109	LD	HL, BUFFER	
0040	224120		110	LD	(BUFFTR), HL	
0043	CDB800		111	CALL	DATBE	; WAKE TX
			112			
0046	18FE		113	JR	\$; LOOP FOREVER
			114			
			115	INIT		
			116	DRTINI:		
0048	211001		117	LD	HL, DRTTA	; INIT CH. A
004B	0E05		118	LD	C, DRTCA	
004D	060A		119	LD	B, DRTEA-DRTTA	
004F	EDB3		120	OTIR		
0051	211A01		121	LD	HL, DRTTB	; INIT CH B
0054	0E07		122	LD	C, DRTCB	
0056	060C		123	LD	B, DRTEB-DRTTB	
0058	EDB3		124	OTIR		
005A	AF		125	XOR	A	; CLEAR FLAG BYTE
005B	324020		126	LD	(DRTFLG), A	
			127	CIOINI		
005E	DB0B		128	IN	A, (CIOCTL)	; INSURE STATE 0
0060	AF		129	XOR	A	; POINT TO REG 0
0061	D30B		130	OUT	(CIOCTL), A	
0063	DB0B		131	IN	A, (CIOCTL)	
0065	AF		132	XOR	A	
0066	D30B		133	OUT	(CIOCTL), A	
0068	3C		134	INC	A	; WRITE RESET
0069	D30B		135	OUT	(CIOCTL), A	
006B	AF		136	XOR	A	; ELSE, CLEAR RESET COND
006C	D30B		137	OUT	(CIOCTL), A	
006E	3EFE		138	LD	A, OFEH	; (FUDGE FOR CIO QUIRK)
0070	D30B		139	OUT	(CIOCTL), A	
0072	D30B		140	OUT	(CIOCTL), A	
0074	212601		141	LD	HL, CLST	; INIT CIO
0077	0620		142	LD	B, CEND-CLST	
0079	0E0B		143	LD	C, CIOCTL	

LOC	OBJ CODE	M	STMT	SOURCE	TEST. DART STATEMENT	ASM 5.9
007B	EDB3		144		OTIR	
007D	C9		145		RET	
			146	*E		
			147			
			148	;	SUBROUTINES	
			149			
			150	,	SETUP FOR ASYNC AS	
			151	,	9600 BAUD	
			152	,	2 STOP BITS	
			153	,	EVEN PARITY	
			154	,	7 BIT CHARACTERS	
			155			
			156	,	DRTFLG - X X 1 1 X X 1 1	
			157	,	/ ! /	
			158	,	ERROR ASLEEP ERROR ASLEEP	
			159	;	CHANNEL B CHANNEL A	
			160			
			161	DBTBE:		
007E	CDF900		162	CALL	SAVE	; CH. B TX BUFFER EMPTY
0081	3E00		163	LD	A, DRTWRO	; POINT TO REG. 0
0083	D307		164	OUT	(DRTCB), A	
0085	3E28		165	LD	A, TBERES	; RESET TBE
0087	D307		166	OUT	(DRTCB), A	
0089	C9		167	RET		
			168			
			169	DBRCA:		
008A	CDF900		170	CALL	SAVE	; CH. B RX CHAR AVAIL.
008D	DB06		171	IN	A, (DRTDB)	; READ DATA
008F	C9		172	RET		
			173			
			174	DBESC:		
0090	CDF900		175	CALL	SAVE	; CH. B EXTERNAL/STATUS
0093	3E00		176	LD	A, DRTWRO	; POINT TO REG. 0
0095	D307		177	OUT	(DRTCB), A	
0097	3E10		178	LD	A, ESCRES	; RESET ESC
0099	D307		179	OUT	(DRTCB), A	
009B	3A4020		180	LD	A, (DRTFLG)	; UPDATE FLAG
009E	CBE7		181	SET	4, A	
00A0	324020		182	LD	(DRTFLG), A	
00A3	C9		183	RET		
			184			
			185	DBSRC:		
00A4	CDF900		186	CALL	SAVE	; CH. B SPECIAL RX COND.
00A7	3E00		187	LD	A, DRTWRO	
00A9	D307		188	OUT	(DRTCB), A	
00AB	3E30		189	LD	A, SRCRES	; RESET SRC
00AD	D307		190	OUT	(DRTCB), A	
00AF	3A4020		191	LD	A, (DRTFLG)	; UPDATE FLAG
00B2	CBEF		192	SET	5, A	
00B4	324020		193	LD	(DRTFLG), A	
00B7	C9		194	RET		
			195			
			196	DATBE:		
00B8	CDF900		197	CALL	SAVE	; CH A TX BUFFER EMPTY
00BB	2A4120		198	LD	HL, (BUFPTR)	; GET BUFFER PTR
00BE	46		199	LD	B, (HL)	; GET CHAR
00BF	7D		200	LD	A, L	; UPDATE PTR.
00C0	3C		201	INC	A	
00C1	E63F		202	AND	3FH	; 64 BYTE WRAPAROUND
00C3	6F		203	LD	L, A	
00C4	224120		204	LD	(BUFPTR), HL	
00C7	78		205	LD	A, B	; OUTPUT CHAR.
00C8	D304		206	OUT	(DRTDA), A	
00CA	C9		207	RET		
			208			
			209	DARCA:		
00CB	CDF900		210	CALL	SAVE	; CH. A RX CHAR AVAIL.
00CE	DB04		211	IN	A, (DRTDA)	
00D0	C9		212	RET		
			213			
			214	DAESC:		
00D1	CDF900		215	CALL	SAVE	; CH. A EXTERNAL/STATUS

LOC	OBJ CODE	M	STMT	SOURCE	TEST. DART STATEMENT	ASM 5. 9
00D4	3E00		216		LD A, DRTWRO	
00D6	D305		217		OUT (DRTCA), A	
00D8	3E10		218		LD A, ESCRES	
00DA	D305		219		OUT (DRTCA), A	
00DC	3A4020		220		LD A, (DRTFLG)	
00DF	CBC7		221		SET 0, A	
00E1	324020		222		LD (DRTFLG), A	
00E4	C9		223		RET	
			224			
			225	DASRC		
00E5	CD900		226		CALL SAVE ; CH. B SPECIAL RX COND.	
00E8	3E00		227		LD A, DRTWRO	
00EA	D305		228		OUT (DRTCA), A	
00EC	3E30		229		LD A, SRCRES	
00EE	D305		230		OUT (DRTCA), A	
00F0	3A4020		231		LD A, (DRTFLG)	
00F3	CBCF		232		SET 1, A	
00F5	324020		233		LD (DRTFLG), A	
00F8	C9		234		RET	
			235			
			236		MATHEWS SAVE REGISTER ROUTINE	
			237			
			238	SAVE.		
00F9	E3		239		EX (SP), HL ; SP = HL	
00FA	D5		240		PUSH DE ; DE	
00FB	C5		241		PUSH BC ; BC	
00FC	F5		242		PUSH AF ; AF	
00FD	DDE5		243		PUSH IX ; IX	
00FF	FDE5		244		PUSH IY ; IY	
0101	CD0F01		245		CALL GO ; PC	
0104	FDE1		246		POP IY	
0106	DDE1		247		POP IX	
0108	F1		248		POP AF	
0109	C1		249		POP BC	
010A	D1		250		POP DE	
010B	E1		251		POP HL	
010C	FB		252		EI	
010D	ED4D		253		RETI	
			254			
			255	GD		
010F	E9		256		JP (HL)	
			257	*E		
			258			
			259		CONSTANTS	
			260			
			261	DRTTA		
0110	00		262		DEFB DRTWRO ; CHAN. RESET	
0111	18		263		DEFB CHRES	
0112	01		264		DEFB DRTWR1 ; CHAN. CHARACS.	
0113	13		265		DEFB RXIAP+TXI+EXTI	
0114	04		266		DEFB DRTWR4 ; MODE	
0115	4F		267		DEFB X16+STOP2+EVEN+PARITY	
0116	05		268		DEFB DRTWR5 ; TX PARAMS.	
0117	AA		269		DEFB DTR+TX7+TXEN+RTS	
0118	03		270		DEFB DRTWR3 ; RX PARAMS.	
0119	41		271		DEFB RX7+RXEN	
			272	DRTEA.	EQU \$	
			273			
			274	DRTTB		
011A	00		275		DEFB DRTWRO ; CHAN. RESET	
011B	18		276		DEFB CHRES	
011C	01		277		DEFB DRTWR1 ; CHAN. CHARACS.	
011D	17		278		DEFB RXIAP+DRTSVAV+TXI+EXTI	
011E	02		279		DEFB DRTWR2 ; VECTOR REG	
011F	10		280		DEFB DRTVEC. AND. 255	
0120	04		281		DEFB DRTWR4 ; MODE	
0121	4F		282		DEFB X16+STOP2+EVEN+PARITY	
0122	05		283		DEFB DRTWR5 ; TX PARAMS.	
0123	AA		284		DEFB DTR+TX7+TXEN+RTS	
0124	03		285		DEFB DRTWR3 ; RX PARAMS.	
0125	41		286		DEFB RX7+RXEN	

TEST DART
 LOC OBJ CODE M STMT SOURCE STATEMENT

ASM 5.9

```

      287 DRTEB EQU $
      288
      289 CLST
0126 28      290      DEFB 28H ; PORT B MODE
0127 00      291      DEFB 0000000B
0128 2B      292      DEFB 2BH ; DATA DIRECTION
0129 EE      293      DEFB 11101110B
012A 06      294      DEFB 6 ; " " PORT C
012B 0E      295      DEFB 00001110B
012C 1C      296      DEFB 1CH ; CT1 MODE
012D C2      297      DEFB 11000010B
012E 1D      298      DEFB 1DH ; CT2 MODE
012F C2      299      DEFB 11000010B
0130 1E      300      DEFB 1EH ; CT3 MODE
0131 C2      301      DEFB 11000010B
0132 16      302      DEFB 16H ; CT1 TC MSB
0133 00      303      DEFB 0
0134 17      304      DEFB 17H ; LSB
0135 06      305      DEFB C1DCNT
0136 18      306      DEFB 18H ; CT2 TC MSB
0137 00      307      DEFB 0
0138 19      308      DEFB 19H ; LSB
0139 06      309      DEFB C1DCNT
013A 1A      310      DEFB 1AH ; CT3 TC MSB
013B 00      311      DEFB 0
013C 1B      312      DEFB 1BH ; LSB
013D 06      313      DEFB C1DCNT
013E 01      314      DEFB 1 ; MASTER CONFIG. REG.
013F F0      315      DEFB 11110000B
0140 0A      316      DEFB 10 ; CT1 TRIGGER
0141 06      317      DEFB 00000110B
0142 0B      318      DEFB 11 ; CT2 TRIGGER
0143 06      319      DEFB 00000110B
0144 0C      320      DEFB 12 ; CT3 TRIGGER
0145 06      321      DEFB 00000110B
      322 CEND: EQU $
      323 *E
      324
      325 ; DATA AREA
      326
2000      327      ORG RAM
2000      328      BUFFER: DEFS 64
2040      329      DRTFLG: DEFS 1
2041      330      BUFPTR: DEFS 2
2043      331      DEFS 64 ; STACK AREA
      332      STAK: EQU $
      333
      334      END
  
```


Interfacing 8500 Peripherals To The Z80



Application Brief

December 1980

INTRODUCTION

There are several differences between the 8500 devices and the Z80 family peripheral devices, including interrupt handling, reset to the device, and daisy-chain control.

This application brief describes the hardware interface requirements and interrupt struc-

ture of the 8500 series peripherals in Z80 systems. The 8500 peripherals are general-interface versions of the Z-BUS counterparts and are designed to interface to nonmultiplexed buses (such as in a Z80 system), instead of multiplexed buses (such as in the Z8000).

CPU HARDWARE INTERFACING

The hardware interface consists of three basic groups of signals: the data bus, control and selection lines, and the interrupt control lines. Following is a table of the general interface signals used by the CPU. Additional information can be found in the peripherals' separate data sheets.

DATA BUS

D_0-D_7 Data bus, bidirectional, 3-state. This bus is used to transfer data between the CPU and the peripheral device.

CONTROL SIGNALS

A_0-A_n Address select lines (optional). These lines are normally used to select the port and/or control registers.

\overline{CE} Chip Enable. \overline{CE} should be gated with \overline{TORQ} or \overline{MREQ} to prevent spurious chip selects during other machine cycles.

\overline{RD}^* Read. \overline{RD} activates chip-read circuitry and gates data from chip onto data bus (to be read by the CPU).

\overline{WR}^* Write. \overline{WR} is used to strobe data from bus into chip.

INTERRUPT CONTROL

\overline{INTACK} Interrupt acknowledge signal from CPU. This replaces the \overline{MI} and \overline{TORQ} generated by the Z80 CPU for interrupt acknowledge. It is used in conjunction with \overline{RD} to gate the interrupt vector onto the data bus.

\overline{INT}, IEI Interrupt Request, Interrupt Enable Input and Interrupt Enable Output. These lines are functionally equivalent to those in the Z80 peripheral products. \overline{INT} is open-drain, active Low output.

*Chip reset is accomplished by activating \overline{RD} and \overline{WR} simultaneously.

INTERRUPT OPERATION

Understanding the 8500 interrupt operation requires basic operational knowledge of the Interrupt Pending (IP) and Interrupt Under Service (IUS) bits in relation to the daisy chain. IP is set in the SIO by an interrupt condition, such as the transmit buffer going empty, and is used with IUS to control the \overline{INT} signal. IP is not set while the CPU is executing an interrupt acknowledge cycle. Thus,

$$IP = INT * \overline{VREAD}$$

The IP latch is cleared either by a software

command to the device or by an implicit action generated by the interrupt service routine. The implicit action may be triggered by the CPU reading or writing a register in the device. For example, on a serial receive device like the SIO, IP may be reset when the CPU reads the character from the receive buffer that caused the interrupt. This removes the interrupt condition, allowing other interrupts to occur.

The Interrupt Under Service (IUS) latch is used to designate the interrupt that is

currently being serviced. IUS is set when the device receives an interrupt acknowledge from the CPU while IEI is High and IP is set. If IEI is Low, the device is prevented from setting the IUS latch and thus cannot issue a vector. In this way, the daisy chain can establish relative priority among peripheral devices. IUS is cleared on the 8500 devices by an explicit software command.

The daisy chain used in the Z80 peripherals is referred to as an IP and IUS daisy chain, because the IP and IUS bits control the IEO pin and the lower portion of the chain. If IP is set, IEO can be Low even if another peripheral has an interrupt under service. When the CPU executes a RETI instruction (ED-4D opcode), the peripheral monitors the bus and resets IUS. When the CPU reads the "ED" part of RETI, peripherals with IP set and IEI High bring IEO High momentarily. This enables the device in the chain with IUS set to clear its IUS latch when the "4D" byte is read by the CPU. (IUS for a device is not cleared unless IEI is High and the "ED-4D" instruction is decoded. This allows more than one device to have IUS set so that nested interrupts can be implemented.)

On the 8500 series devices, IP is used to control the daisy chain only during the interrupt acknowledge cycle. Under normal

conditions, only IUS is required to control the state of the IEO pin. Therefore, the daisy chain used in 8500 devices is referred to as an IUS daisy chain. Since IP is not a part of the daisy chain, there is no "ED" decoding pulling IEO High when IP is set. To allow more control over the daisy chain, the 8500 devices have a "Disable Lower Chain" (DLC) software command that unconditionally brings IEO Low. This can be used to deactivate parts of the daisy chain selectively, regardless of interrupt status. Figure 1 shows the functions of IP and IUS and the truth tables for each.

A unique feature of the 8500 devices is the INTACK pin. This pin acknowledges a CPU interrupt service cycle to the peripheral, allowing the peripheral to gate its vector onto the data bus. On the Z80 peripherals, interrupt acknowledge cycles from the CPU consist of a special M1 cycle where \overline{TORQ} is activated instead of \overline{MREQ} . This limits the control of devices in systems using a processor other than the Z80. As a result, a simpler implementation has been devised, which uses additional logic to accommodate a wider variety of processors. Figure 2 shows a circuit that generates INTACK for the 8500 devices in addition to wait states. Figure 3 shows the timing for INTACK and wait generation.

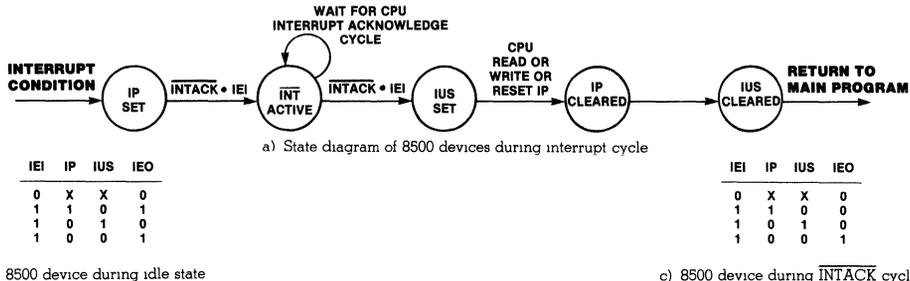


Figure 1. 8500 Device Interrupt-Processing Sequence

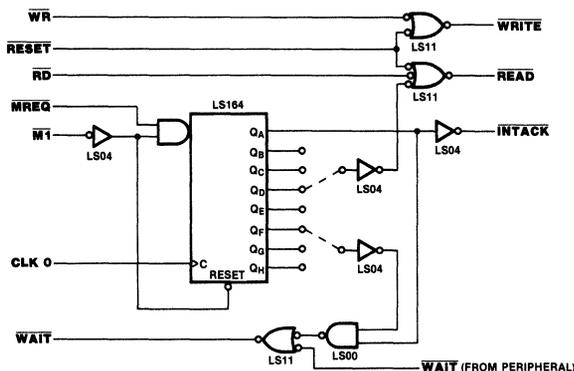
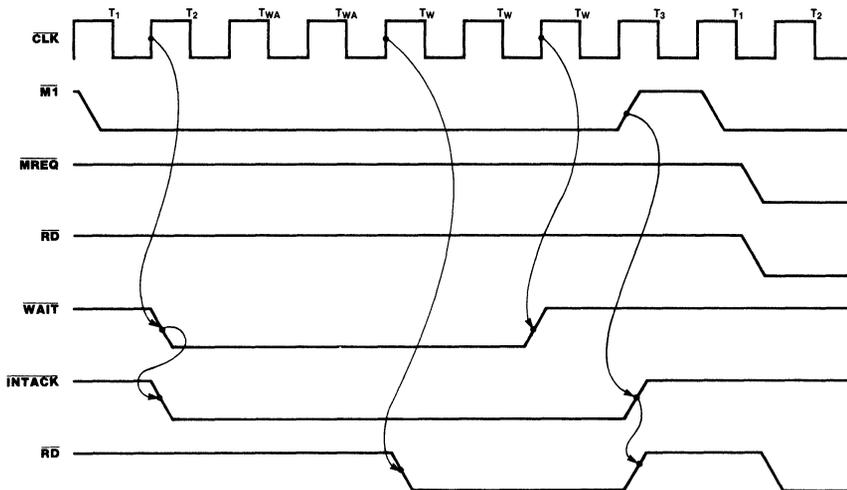


Figure 2. INTACK and WAIT Generation for 8500 Peripherals



NOTE: WAIT is assumed to be High.

Figure 3. Timing for 8500 Peripherals During Interrupt Acknowledge Without Z80 Peripheral Logic

On long daisy chains, wait states may be necessary to allow the IEI and IEO lines time to stabilize, thus avoiding conflict between devices and preventing IUS or IP from changing erroneously. Because of the IP and IUS configurations, the daisy chain used in Z80 peripherals needs to stabilize during the interrupt acknowledge and RETI operations.

However, on the 8500 devices, the daisy chain is IUS and wait states are generated for the INTACK cycle only, not for the return cycle. (There is no "ED-4D" decode.) As a result, hardware interfacing is greatly simplified and timing is less complicated than on the Z80 peripherals.

SOFTWARE CONSIDERATIONS

There are several options available for servicing interrupts on the 8500 devices. Since the vector register (or IP register) can be read at any time, the software can emulate the Z80 CPU interrupt response easily. The interrupt vector reflects the interrupt status condition, even if the

peripheral is programmed to return a vector that does not reflect the status change (SAV or VIS not set). This allows a simple software routine to emulate the Z80 vector response operation, as shown in the code of Figure 4.

Loc.	Obj	Code	M	Start	Source	AP.8500.1 Statement
				12		*E
				13		
				14		; This routine emulates the Z80 vector interrupt
				15		; operation by reading the device interrupt vector,
				16		; forming an address from a vector table, and exe-
				17		; cuting an indirect jump to the interrupt service
				18		; routine.
				19		
				20	INDX:	LD A,C1VREG ;CURRENT INT. VECTOR REG
0002	3E00			21	OUT	(CTRL),A ;WRITE REG. PTR.
0004	D3E0			22	IN	A,(CTRL) ;READ VECTOR REG.
0006	3C			23	INC	A ;VALID VECTOR?
0007	C8			24	RET	Z ;NO INTERRUPT - RETURN
0008	E60E			25	AND	00001110B ;MASK OTHER BITS
000A	5F			26	LD	E,A ;FORM INDEX VALUE
000B	1600			27	LD	D,0
000D	211600	R		28	LD	HL,VECTAB ;ADD VECTOR TABLE ADDR
0010	19			29	ADD	HL,DE
0011	7E			30	LD	A,(HL) ;GET LOW BYTE
0012	23			31	INC	HL
0013	66			32	LD	H,(HL) ;GET HIGH BYTE
0014	6F			33	LD	L,A ;PUT ROUTINE ADDR IN #HL
0015	E9			34	JP	(HL) ;GO TO ROUTINE !
				35		
				36	VECTAB:	
0016	0010			37	DEFW	INT1
0018	0011			38	DEFW	INT2
001A	0012			39	DEFW	INT3
001C	0013			40	DEFW	INT4
001E	0014			41	DEFW	INT5
0020	0015			42	DEFW	INT6
0022	0016			43	DEFW	INT7
0024	0017			44	DEFW	INT8

Figure 4. Z80 Vector Interrupt Response Emulation by Software

Because the 8500 devices have considerable program flexibility, a Master Interrupt Enable (MIE or IE) bit in the control register determines the device response to the CPU. If MIE is not set, interrupts are

not generated to the CPU and the device ignores any interrupt response from the CPU. This is used as a global enable and simplifies the programming of interrupts so that they can be easily changed on the fly.

A SIMPLE Z80 SYSTEM

The 8500 devices interface easily to the Z80 CPU, providing a system of considerable flexibility. Figure 5 illustrates a simple system using the Z80 CPU and a Z8536 CIO in a noninterrupt environment. Since $\overline{\text{INTACK}}$ is not used, it is tied High and no additional logic is needed. Because the CIO can be used in a polled interrupt system, the $\overline{\text{INT}}$ pin is connected to the CPU. The Z80 should not be programmed for Interrupt Mode 2, because the vector from the CIO is never sent to the CPU. Instead, the CPU can be set for Interrupt Mode 1, and a global interrupt routine that reads the vector register from the CIO can determine which routine to go to when an interrupt occurs, as previously illustrated in Figure 4.

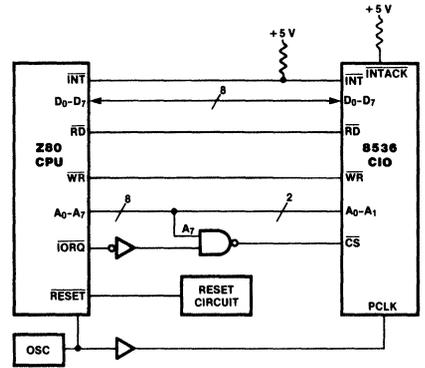
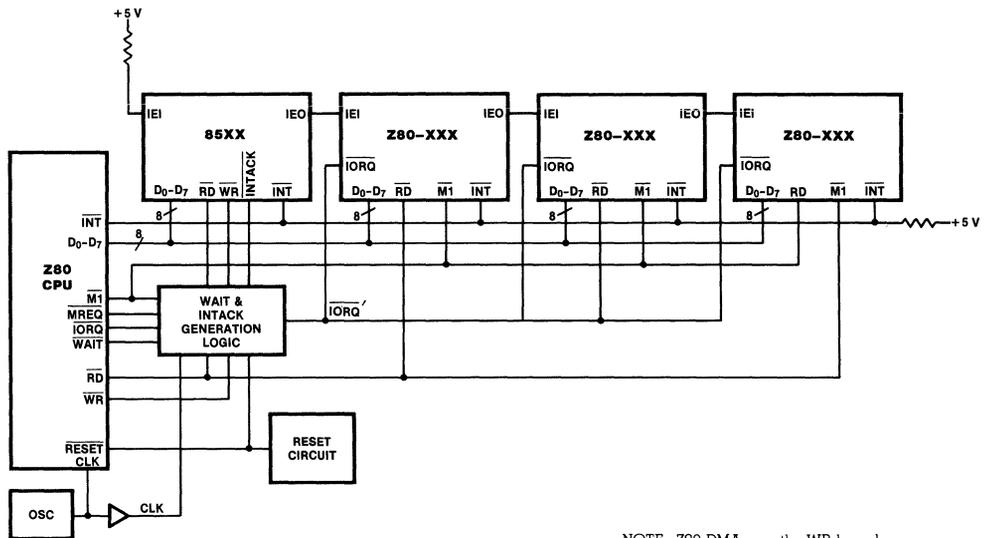


Figure 5. Non-Interrupt CPU Interface

Z80 PERIPHERALS WITH 8500 PERIPHERALS

A Z80 system using a combination of Z80 family peripherals and 8500-type peripherals is easily constructed, as shown in Figure 6. There is no placement restriction on the 8500 devices within the daisy chain, but it is recommended that they be near the beginning

of the chain in order to minimize propagation delays during the "ED-4D" decoding. The 8500 devices do not decode the "ED" during an opcode fetch cycle, so IEO will not change state during this time.



NOTE: Z80 DMA uses the WR line also.

Figure 6. A Z80 System Using 8500 Devices and Z80 Peripherals

Figure 7 is a diagram of the logic represented by the WAIT and INTACK logic box in Figure 6. The WAIT signal is OR-wired to the output of each peripheral device (if used). The RD and WR signals only go to the 8500

device. The Z80 peripherals are wired to the Z80 as usual. The timing for the INTACK and WAIT generation logic is illustrated in Figure 8.

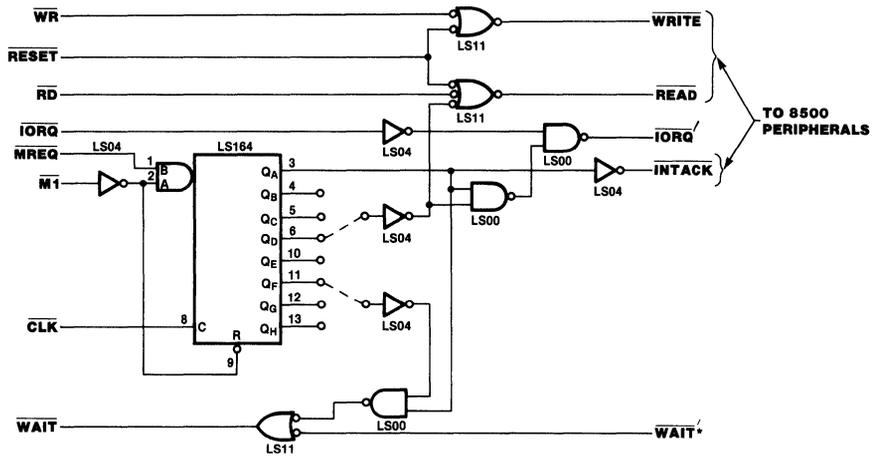


Figure 7. $\overline{\text{WAIT}}$ and $\overline{\text{INTACK}}$ Generation Logic

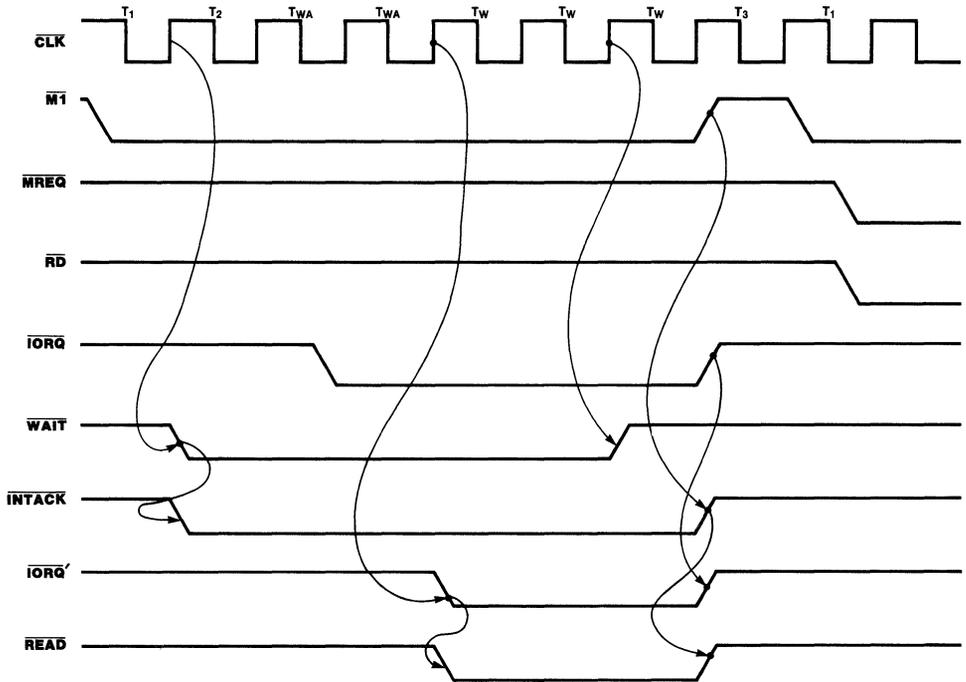


Figure 8. Timing for 8500 and Z80 Peripherals During Interrupt Acknowledge

Serial Clock Generation Using the Z8536 CIO



Application Brief

February 1981

INTRODUCTION When an external clock is not provided in a Z80-based system, it is often necessary to generate a bit-rate clock for serial devices. The most efficient way to accomplish this is to use a programmable counter that can change the bit-rate clock under CPU control. In this example, the Z8536 Counter/Timer I/O device (CIO) was chosen to generate the bit-rate clocks for a Z80-based statistical mul-

tiplexor project that used a Z80 SIO and a Z80 DART.

This application brief describes the use of the Z8536 CIO device in a Z80-based system for generating the bit-rate clocks for asynchronous communications. The Z8536 CIO contains the circuitry necessary to generate the clock pulses required by asynchronous communication devices.

HARDWARE The Z8536 CIO is housed in a 40-pin package and contains both system bus interface and I/O port connections. The three 16-bit counters can be programmed to output a pulse, square wave, or one-shot waveform on the timer's corresponding output pin. Three bits of the output ports (two from Port B and one from Port C) are used as the counter/timer outputs and provide the bit-rate pulses used in this application.

CIO is placed in a reset state and remains there until cleared by the program. Reset can also be initiated by issuing a command to Register 0 with bit 0 set or by a hardware condition (\overline{RD} and \overline{WR} simultaneously active). The reset state is described in detail in the programming section. Once the reset state is cleared, the CIO is placed in state 0, in which the control registers can be accessed by writing a Register Pointer to the CIO control port. This places the CIO in state 1, after which the next CPU access (read or write register data) causes the CIO to revert to state 0. The last register addressed may be accessed simply by reading the CIO control port. It should be noted that the Register Pointer can be written only while in state 0. Also, data can be written to a control register only after a Register Pointer has been written. Figure 1 shows the state diagram for the CIO.

Interfacing the CIO to the Z80 CPU requires eight bidirectional data lines and five control lines. The data lines are used to transfer register address and data to or from the CIO via the \overline{RD} , \overline{WR} , \overline{CE} , and address control lines. Two address lines (A_0 and A_1) select the port the CPU is accessing. Table 1 shows the port selected by the address bits.

Table 1. Port Addressing for the CIO

Address Line	A_1	A_0
Port C	0	0
Port B	0	1
Port A	1	0
CTRL	1	1

The control port (CTRL) is used for control register selection and parameter transfer. To select a particular register, a Register Pointer is written to the CTRL port and the data is written into or read from the register.

The CIO contains a state machine that controls the CPU interface. Upon power-up, the

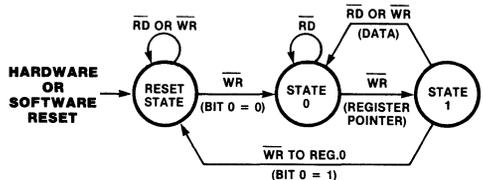


Figure 1. State Diagram for Z8536 CIO

The \overline{RD} and \overline{WR} control lines determine the data path direction into or out of the CIO. When activated simultaneously, they also perform the device's reset function. Figure 2 illustrates how the reset function can be implemented using external circuitry.

Since interrupts are not used in this application, INTACK is tied High to prevent spurious interrupt operation of the CIO due to noise.

Each counter/timer uses one or more bits on one of the parallel ports to provide for counter input and counter/timer output. Table 2 shows which output port bits correspond to particular counter/timer inputs and outputs.

The outputs of the counter/timers (PB4, PB0, and PC0) are fed to the rest of the circuitry to supply the serial clock pulses.

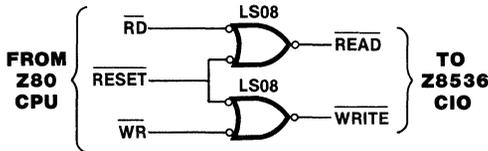


Figure 2. RESET Interface to the Z8536

PROGRAMMING

Once the hardware has been defined, the functional operation and configuration of the Z8536 are determined entirely by the software programming. Several considerations concerning initialization must be made when using the CIO. When the device receives a reset from either hardware or a software command, the reset state must be removed before any data can be written to the CIO. To clear the reset state, the user writes to register 0 with bit 0 cleared. Once the internal reset latch is cleared, the programmer can initialize the CIO and begin normal operations. The program listed in the appendix shows a reset sequence that brings the CIO to state 0 even if the previous state is undefined.

The configuration of the CIO defines the general operating characteristics of the device with respect to its internal functions. The Port Mode Specification register sets to output those bits in Port B that are used for the counter/timer outputs. In this example, Bit mode is used on Ports B and C to output the counter/timer pulses.

The Counter/Timer mode, time constant values, and trigger commands are the last parameters to be set. Finally, the Master Configuration Control register is set to enable Port B, all the counter/timers, and Port C (Port C is enabled along with the counter/timers). The Counter/Timer mode is programmed for continuous cycle square wave with external output enabled. The square-wave cycle time is two times the programmed time constant, which must be taken into account when programming time constant values. The downcounters in the CIO are 16-bit counters that are decremented by one for each internal clock cycle. The internal clock cycle is the PCLK cycle divided by two, so the time constant value is determined by the following formula:

Table 2. Counter/Timer External Interface Bits

Function	C/T1	C/T2	C/T3
C/T Output	PB4*	PB0	PC0
Counter Input	PB5	PB1	PC1
Trigger Input	PB6	PB2	PC2
Gate Input	PB7	PB3	PC3

*PB4 = Port B, bit 4

The last hardware consideration involves the clock input, PCLK. Since the Z8536 does not need to be synchronized with the CPU clock, PCLK can come from any source so long as it meets the timing and interface requirements. In fact, PCLK can come from a source external to the system if desired. Once inside the device, PCLK is divided by two before it is sent to the counter/timer circuits. There is no other prescaling done and the resulting clock is fed to the 16-bit counters.

$$\text{Time Constant} = \text{PCLK} / (4 * \text{Output Frequency})$$

PCLK is divided by four in the formula because it is divided by two inside the CIO before being fed into the downcounter and by two again because a square wave cycle is two times the time constant value. Substituting the baud rate and a multiplier of 16 for the output frequency, the formula reduces to a simple time constant formula.

$$\text{TC} = \text{PCLK} / (4 * 16 * \text{Baud Rate})$$

With a 3.6864 MHz PCLK input and a desired 9600 baud rate, the formula simplifies to:

$$\begin{aligned} \text{TC} &= 3,686,400 / (4 * 16 * 9600) \\ &= 57600 / 9600 \\ &= 6 \end{aligned}$$

Other 16X baud rates may be generated by using the above formula in a general form.

$$\text{TC} = 57600 / \text{Baud Rate}$$

The user must exercise caution when choosing values for the PCLK and baud rates since they must result in nearly integral time constant values. For example, a 2.4576 MHz clock input with 9600 baud and a 16X clock output give a time constant value of 4. Greater flexibility is available for selecting time constant values because the SIO does not require a square wave input when programmed for 16X, 32X, or 64X clock inputs. Pulses may be used with the SIO provided the user adheres to the SIO timing requirements.

The last operation performed on the CIO is a trigger command to "kick it off." This also includes setting the gate command bit in the Counter/Timer Command and Status registers, which allows the clock pulses to toggle the

downcounter. The trigger command bit loads an initial value into the downcounter and begins operation of the counter/timer circuitry. Once triggered, the counter/timer runs continuously, performing automatic reloads to

the downcounter after it reaches zero (terminal) count. At this time, the CIO is finished being programmed and the user has three clean square waveforms at the output pins.

CONCLUSION

The designer should find the Z8536 CIO a versatile and cost-effective component to satisfy his or her system needs. Coupled with other Zilog components, the Z8536 architecture enhances the performance of any Z80 system by providing the essential timing, I/O functions, and interrupt control functions necessary for efficient system operation.

The Z8536 CIO was chosen after considering device count, performance, and ease of use. Alternatives to the CIO include discrete (TTL) hardware counters and gates, external clock sources, or the Z80 CTC. These methods are generally too parts-intensive, and power consumption is therefore higher. For applications where two 8-bit ports and three counter/timers are needed, the CIO proves to be the ideal component.

APPENDIX

Following is a listing of a test program written for the Z80 CPU. This program simply initializes the CIO and then loops until

stopped, with the CIO continuously providing pulses. All three counter/timers are used to generate square waves corresponding to a 16X 9600 baud clock.

				TEST.CIO			
LOC	OBJ CODE	M STMT	SOURCE	STATEMENT	ASM 5.9		
			1	;	CIO TEST PROGRAM		
			2	;	[1] 01-07-81/MDP	INITIAL	CREATION
			3				
			4	;	THIS PROGRAM INITIALIZES THE THREE COUNTER		
			5	;	TIMERS IN THE Z8536 CIO TO GENERATE SQUARE		
			6	;	WAVES, THEN LOOPS FOREVER.		
			7				
			8	;	PROGRAM EQUATES		
			9				
			10	CIOC:	EQU	B	; CIO PORT C
			11	CIOB:	EQU	CIOC+1	; CIO PORT B
			12	CIOA:	EQU	CIOC+2	; CIO PORT A
			13	CIOCTL:	EQU	CIOC+3	; CIO CTRL PORT
			14	BAUD:	EQU	9600	; ASYNC BAUD RATE
			15	RATE:	EQU	BAUD/100	
			16	CIOCNT:	EQU	576/RATE	
			17	RAM	EQU	2000H	; RAM START ADDR
			18	RAMSIZ	EQU	1000H	; RAM SIZE
			19	*E			
			20				
			21	;	*** MAIN PROGRAM ***		
			22				
	0000		23		ORG	0	
			24	BEGIN:			
	0000	314020	25	LD	SP, STAK		; INIT SP.
	0003	CDO800	26	CALL	INIT		; INIT DEVICES
			27				
	0006	18FE	28	JR	\$; LOOP FOREVER
			29				
			30	INIT:			
			31	CIOINI:			
	0008	DB0B	32	IN	A, (CIOCTL)		; INSURE STATE 0
	000A	3E00	33	LD	A, 0		; REG 0 OR RESET
	000C	D30B	34	OUT	(CIOCTL), A		; WRITE PTR OR CLEAR RESET
	000E	DB0B	35	IN	A, (CIOCTL)		; STATE 0
	0010	3E00	36	LD	A, 0		; REG 0
	0012	D30B	37	OUT	(CIOCTL), A		; WRITE PTR
	0014	3E01	38	LD	A, 1		; WRITE RESET
	0016	D30B	39	OUT	(CIOCTL), A		
	0018	3E00	40	LD	A, 0		; CLEAR RESET
	001A	D30B	41	OUT	(CIOCTL), A		
	001C	212600	42	LD	HL, CLST		; INIT CIO
	001F	0620	43	LD	B, CEND-CLST		
	0021	0E0B	44	LD	C, CIOCTL		
	0023	EDB3	45	OTIR			
	0025	C9	46	RET			
			47	*E			
			48				

```

    49 ;;          CONSTANTS
    50
    51 CLST:
0026 28          52      DEFB      28H          ; PORT B MODE
0027 00          53      DEFB      0000000B
0028 28          54      DEFB      28H          ; PORT B DIRECTION
0029 EE          55      DEFB      11101110B
002A 06          56      DEFB      06H          ; PORT C DIRECTION
002B FE          57      DEFB      11111110B
002C 1C          58      DEFB      1CH          ; CT1 MODE
002D C2          59      DEFB      11000010B
002E 1D          60      DEFB      1DH          ; CT2 MODE
002F C2          61      DEFB      11000010B
0030 1E          62      DEFB      1EH          ; CT3 MODE
0031 C2          63      DEFB      11000010B
0032 16          64      DEFB      16H          ; CT1 TC MSB
0033 00          65      DEFB      0           ;
0034 17          66      DEFB      17H          ;          LSB
0035 06          67      DEFB      C1DCNT
0036 18          68      DEFB      18H          ; CT2 TC MSB
0037 00          69      DEFB      0           ;
0038 19          70      DEFB      19H          ;          LSB
0039 06          71      DEFB      C1DCNT
003A 1A          72      DEFB      1AH          ; CT3 TC MSB
003B 00          73      DEFB      0           ;
003C 1B          74      DEFB      1BH          ;          LSB
003D 06          75      DEFB      C1DCNT
003E 01          76      DEFB      1           ; MASTER CONFIG. REG.
003F FO          77      DEFB      11110000B
0040 0A          78      DEFB      0AH          ; CT1 TRIGGER
0041 06          79      DEFB      00000110B
0042 0B          80      DEFB      0BH          ; CT2 TRIGGER
0043 06          81      DEFB      00000110B
0044 0C          82      DEFB      0CH          ; CT3 TRIGGER
0045 06          83      DEFB      00000110B
    84 CEND:      EQU      $
    85
    86 ;;          DATA AREA
    87
2000          88      ORG      RAM
2000          89      DEFS    64          ; STACK AREA
    90 STAK:      EQU      $
    91
    92          END
  
```

Timing in an Interrupt-Based System with the Z80[®] CTC



Application Note

March 1981

INTRODUCTION In many computer systems, an accurate time base is needed so that critically timed events do not go awry. Use of a counter or timer to monitor time-dependent activities is essential in such systems. In an interrupt-driven system, the Z80 CTC can provide regular program time intervals. Single-event

counts or single-event time delays can also be implemented under program control. This application note describes both continuous time-interval operations and single-interval count operations using the Z80 CTC in a Z80 system.

HARDWARE CONFIGURATION In the example used here, the hardware consists of a Z80 CPU with 4K bytes of RAM, 4K bytes of ROM, a Z80A SIO, and a Z80A CTC. There are two external inputs to the CTC: one is derived from the ac power line to provide

60Hz pulses; the other is connected to a transmit clock line on the SIO. One of the counter/timer outputs is connected to the SIO transmit and receive clock input, as shown in Figure 1.

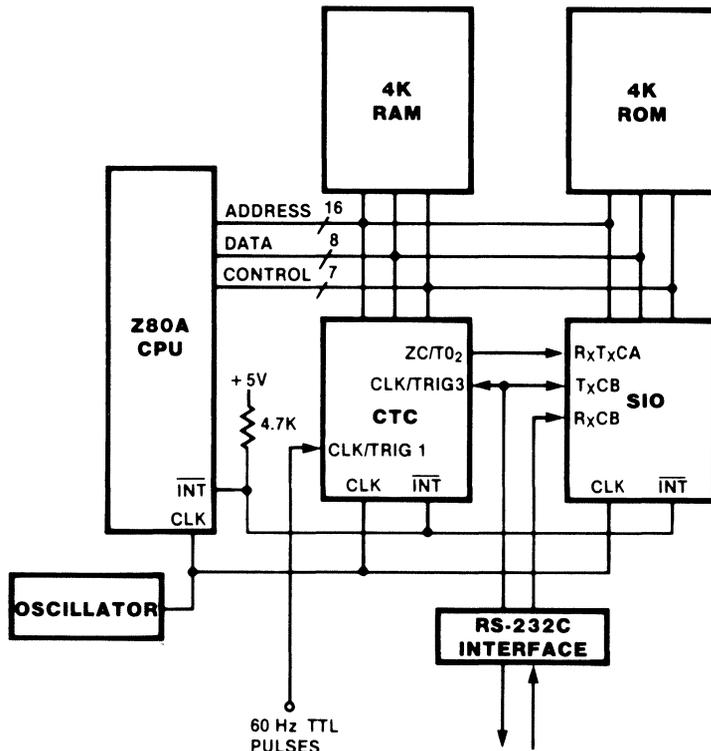


Figure 1. Z80A System Block Diagram

The Z80 CTC is designed for easy interface to the Z80 CPU. An 8-bit bidirectional data bus is used to transfer information between the CTC and CPU. The control lines, RD, TORQ, M1, and CE, determine what data is being transferred and when. M1 and TORQ are used during the interrupt acknowledge cycle to allow the CTC to present its 8-bit interrupt vector to the CPU. TORQ is also used in conjunction with CE to enable transfers between the CTC and the CPU. RD is used to control the direction of data flow between the CTC and the CPU. The channel select lines (CS₀ and CS₁) are connected to the lowest two bits of the address bus and are used to access one of the four counter/timer channels. Table 1 shows the relationships between the CS pins and the counter/timer channels.

Table 1. Channel Select Values

CS ₁	CS ₀	C/T Channel
0	0	Channel 0
0	1	Channel 1
1	0	Channel 2
1	1	Channel 3

The CTC system clock input requirements are similar to those of the Z80 CPU. For both, the system clock input Low level should be no greater than 0.45 V, the High level should be no less than V_{CC}-0.6 V, and the clock rise and fall times should be less than 30 ns. A clock-driver device that meets these requirements, such as the HH-3006-A¹, works well

with the CTC. Several devices can be connected to the driver, but the user should be careful not to overload the driver. The capacitance of the clock input to the CTC (20 pF) should be noted as this may affect the system clock rise and fall times.

Interrupt control logic within the CTC is used to initiate interrupts and to control the interrupt acknowledge cycle generated by the CPU. An interrupt is generated by the CTC when one of the counter/timer down counters reaches terminal count (0) and IEI is High. IEI and IE0 allow the CTC to operate within the Z80 interrupt daisy chain and to connect to the next higher-priority and next lower-priority devices in the chain, respectively. If there is no higher-priority device, IEI is tied to +5 V.

The CTC internally prioritizes each counter/timer with respect to interrupt generation. This maximizes performance by resolving contention between channels should two or more interrupt conditions occur simultaneously. Table 2 shows the relative priority levels of each counter/timer within the CTC.

Table 2. CTC Channel Interrupt Priority

Priority	Channel
Highest	0
	1
Lowest	2
	3

CTC MODES There are two basic modes under which the CTC can operate: Timer mode and Counter mode. Each mode has certain programmable character-

istics that enable the CTC to be used in a wide variety of applications.

TIMER MODE A typical use of the CTC in Timer mode is to provide regular, fixed-interval interrupts to the CPU used as a time-base reference to allocate the processor resources efficiently. For example, a multitasking system might have the processor execute a task for a given length of time and then interrupt execution of the program at one-second intervals to scan the task queue for higher-priority tasks. This system time interval can be provided by the CTC in Timer mode. In Timer mode, the CTC downcounter is decremented by the output of the prescaler, which is toggled by the system clock input. The prescaler has a programmable value of 16 or 256, depending on the condition of bit 5 in the channel control word (CCW). Thus, with a 4 MHz system clock fed into the CTC, a timer resolution of 4μs (prescaler count of 16) or 64μs (count of 256) is possible.

Another use of CTC Timer mode operation is to implement a nonretriggerable one-shot using external circuitry. The digital approach to the one-shot provides a programmable time delay under CPU control and provides greater noise immunity than the more common analog delay circuits provide. Figure 3 shows a circuit that uses part of a 74LS02 package in addition to one CTC channel.

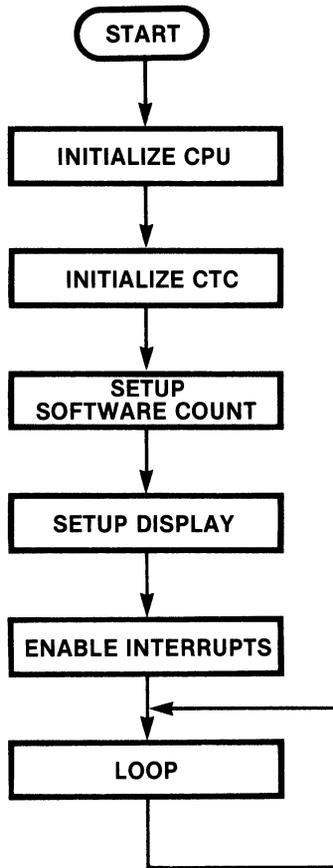
In the example shown, the interrupt interval is set to 8.33 ms, which is provided by the CTC with a 3.6864 MHz input clock, 256 prescaler value, and a time constant value of 120. The CTC interrupt service routine uses a software count of 120 to maintain a one-second system time interval. Each time the service routine is executed, the software count is decremented by 1. When the count reaches 0, a flag is set and the program pursues an appropriate course of action. Figure 2 shows the initialization and interrupt service routine coding for a CTC channel using the Timer mode.

The trigger waveform should be positive-going and should meet the CTC setup time for the CLK/TRIG input. Also, the trigger High level time should be less than the CTC delay time in order to prevent the two 74LS02s from latching in the triggered state. An additional gate can be added to initialize the 74LS02 flip-flop to a defined state when the system is reset or else the software can pulse the timer output to set the flip-flop, as is done in this case. A third use of the Timer mode is to provide a bit rate clock for a serial transceiver device, such as the Z80 SIO. The SIO can accept a 1x, 16x, 32x, or 64x bit rate clock input from an external source, and with a 16x, 32x, or 64x multiplier, the SIO can accept a pulse waveform input for the bit rate clocks, as long as the pulses meet the rise, fall, and hold time requirements of the SIO. The CTC meets these requirements and can be connected directly to the SIO to provide the necessary bit rate clocks. Figure 4 shows the code needed to generate a bit rate clock for the SIO.

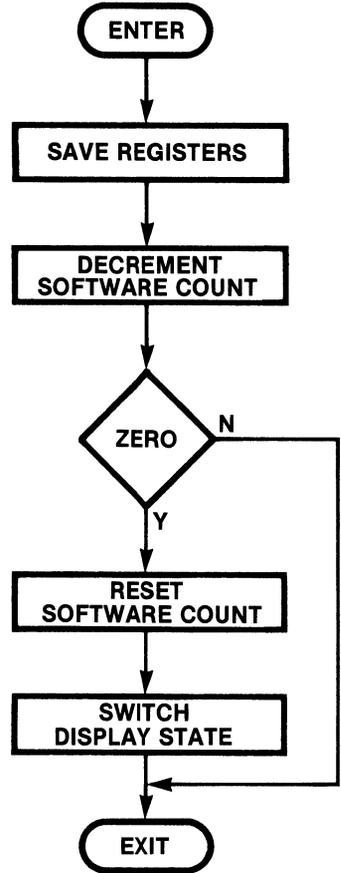
¹A clock driver by Hybrid House, 1615 Remuda La., San Jose, CA 95112.

With a 1x bit rate clock programmed into the SIO, a square-wave input must be supplied. This can be done by adding a flip-flop between the CTC and the SIO. The time constant

value should be set to half the baud rate value, since the CTC output is divided in half by the flip-flop.



a) Main Program



b) Interrupt Service Routine

Figure 2. Software for CTC Timer Mode Operation

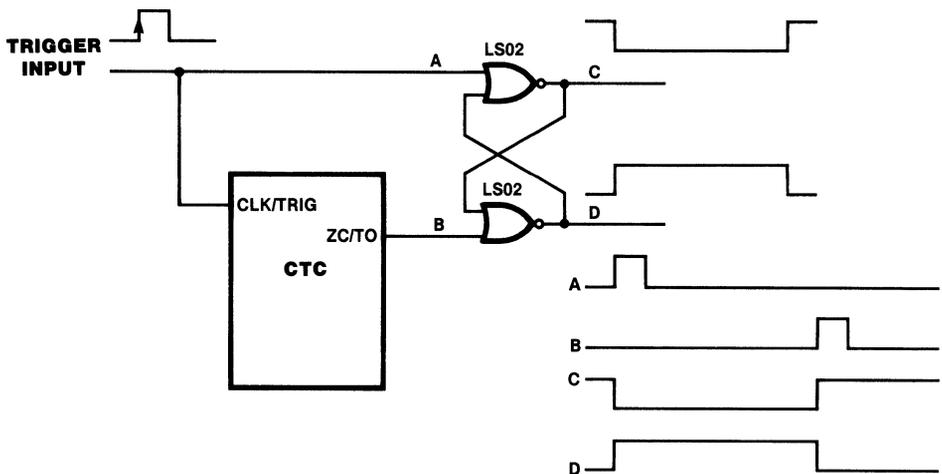


Figure 3. Monostable Multivibrator Using the Z80 CTC

```

TEST CTCO
LOC  OBJ CODE M STMT SOURCE STATEMENT
      1 .          CTC TEST PROGRAM
      2
      3 ,          THIS PROGRAM USES THE CTC IN CONTINUOUS
      4 ,          TIMER MODE THE CTC COUNTS SYSTEM CLOCK
      5 ,          PULSES AND INTERRUPTS EVERY 120 PULSES,
      6 ,          THEN DECREASES A COUNT, THEN SWITCHES
      7 ,          THE LED STATE WHEN THE COUNT REACHES ZERO
      8
      9 .          PROGRAM EQUATES
     10
     11 CTC0 EQU 12 , CTC 0 PORT
     12 CTC1 EQU CTC0+1 , CTC 1 PORT
     13 CTC2 EQU CTC0+2 , CTC 2 PORT
     14 CTC3 EQU CTC0+3 , CTC 3 PORT
     15 LITE EQU 0EOH ; LIGHT PORT
     16 RAM EQU 2000H , RAM START ADDR
     17 RAMSIZ EQU 1000H
     18 TIME EQU 120 , COUNT VALUE
     19
     20
     21 .          CTC EQUATES
     22
     23 CCW EQU 1
     24 INTEN EQU 80H
     25 CTRMODE EQU 40H
     26 P256 EQU 20H
     27 RISEDC EQU 10H
     28 PSTRT EQU 8
     29 TCLOAD EQU 4
     30 RESET EQU 2
     31 *E
     32
     33 .          *** MAIN PROGRAM ***
     34
     35 ORG 0
     36 JP BEGIN
     37
     38 ORG $ AND OFFFOH OR 10H
     39 INTVEC
     40 DEFW ICTC0
     41 DEFW ICTC1
     42 DEFW ICTC2
     43 DEFW ICTC3
     44
     45 BEGIN
     46 LD SP,STAK , INIT SP
     47 IM 2 , VECTOR INTERRUPT MODE
     48 LD A,INTVEC/256 , UPPER VECTOR BYTE
     49 LD I,A
     50 CALL INIT , INIT DEVICES
     51 EI , ALLOW INTERRUPTS
     52
     53 JR $ , LOOP FOREVER
     54
     55 INIT
     56 LD A,INTEN+P256+TCLOAD+RESET+CCW
     57 OUT (CTC0),A , SET CTC MODE
     58 LD A,TIME
     59 OUT (CTC0),A ; SET TIME CONSTANT
     60 LD A,INTVEC AND 11111000B
     61 OUT (CTC0),A ; SET VECTOR VALUE
     62 XOR A
     63 LD (DISP),A , CLEAR DISPLAY BYTE
     64 LD A,TIME ; INIT TIMER VALUE
     65 LD (COUNT),A
     66 RET
     67 *E
     68
     69 ,          INTERRUPT SERVICE ROUTINE
     70

```

LOC	OBJ CODE	M	STMT	SOURCE	STATEMENT	TEST. CTCO
			71	ICTC1		
			72	ICTC2		
			73	ICTC3		
003D	FB		74	EI		, DUMMY ROUTINES
003E	ED4D		75	RETI		
			76			
			77	ICTCO		
0040	CD5A00		78	CALL	SAVE	, SAVE REGISTERS
0043	3A4020		79	LD	A, (COUNT)	, CHANGE TIMER COUNT
0046	3D		80	DEC	A	
0047	324020		81	LD	(COUNT), A	
004A	CO		82	RET	NZ	, EXIT IF NOT DONE
004B	3E78		83	LD	A, TIME	, ELSE, RESET TIMER VALUE
004D	324020		84	LD	(COUNT), A	
0050	3A4120		85	LD	A, (DISP)	, BLINK LITES
0053	2F		86	CPL		
0054	324120		87	LD	(DISP), A	
0057	D3E0		88	OUT	(LITE), A	
0059	C9		89	RET		
			90			
			91		SAVE REGISTER ROUTINE:	
			92			
			93	SAVE		
005A	E3		94	EX	(SP), HL	
005B	D5		95	PUSH	DE	
005C	C5		96	PUSH	BC	
005D	F5		97	PUSH	AF	
005E	CD6800		98	CALL	GO	
0061	F1		99	POP	AF	
0062	C1		100	POP	BC	
0063	D1		101	POP	DE	
0064	E1		102	POP	HL	
0065	FB		103	EI		
0066	ED4D		104	RETI		
			105			
			106	GO		
0068	E9		107	JP	(HL)	
			108	*E		
			109			
			110		DATA AREA	
			111			
2000			112	ORG	RAM	
2000			113	DEFS	64	, STACK AREA
			114	STAK	EQU	\$
2040			115	COUNT	DEFS	1
2041			116	DISP.	DEFS	1
			117			
			118	END		

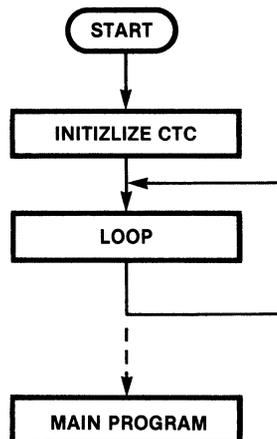


Figure 4. Software for CTC Bit Rate Generator

```

TEST.CTC2
LOC  OBJ CODE M STMT SOURCE STATEMENT
      1 ,          CTC TEST PROGRAM
      2
      3 ,          THIS PROGRAM USES THE CTC IN CONTINUOUS
      4 ,          TIMER MODE THE CTC SUPPLIES A BIT RATE
      5 ,          CLOCK TO THE SIO FROM THE SYSTEM CLOCK
      6 ;          THE SYSTEM CLOCK IS 3.6864 MHZ, WHICH IS
      7 ,          DIVIDED BY 16 BY THE PRESCALER, AND DIVIDED
      8 ,          BY A TIME CONSTANT VALUE OF 3 TO
      9 ,          PROVIDE A 16X, 4800 BAUD CLOCK
     10 ,          TO THE SIO. OTHER BAUD RATES CAN BE OBTAINED
     11 ,          BY PROGRAMMING DIFFERENT TIME CONSTANT
     12 ,          VALUES INTO THE CTC.
     13
     14 ;          PROGRAM EQUATES
     15
     16 CTC0 EQU      12          ; CTC 0 PORT
     17 CTC1 EQU     CTC0+1      ; CTC 1 PORT
     18 CTC2 EQU     CTC0+2      ; CTC 2 PORT
     19 CTC3 EQU     CTC0+3      ; CTC 3 PORT
     20 TIME EQU      3          ; TIME CONSTANT VALUE
     21
     22
     23 ,          CTC EQUATES
     24
     25 CCW EQU       1
     26 INTEN EQU     80H
     27 CTRMODE EQU  40H
     28 P256 EQU     20H
     29 RISEDG EQU   10H
     30 PSTRT EQU    8
     31 TCLOAD EQU   4
     32 RESET EQU    2
     33 *E
     34
     35 ,,          *** MAIN PROGRAM ***
     36
0000   37          ORG      0
     38 BEGIN:
0000   39          LD      A, TCLOAD+RESET+CCW
0002   40          OUT    (CTC2),A      ,SET CTC MODE
0004   41          LD      A, TIME
0006   42          OUT    (CTC2),A      ,SET TIME CONSTANT
     43
     44 ,          MAIN PROGRAM GOES HERE
     45 *E
     46
0008   47          JR      $          ; LOOP FOREVER
     48
     49          END

```

COUNTER MODE A typical computer system often uses a time-of-day clock. In the United States, the 60 Hz power line provides an accurate time base for synchronous motor clocks. A computer system can take advantage of the 60 Hz accuracy by incorporating a circuit that feeds 60 Hz square waves into a CTC channel. With a time constant value of 60, the CTC generates an interrupt once every second, which can be used to update a time-of-day clock. The CTC is set to Counter mode and with a time constant value of 60, as shown in Figure 5.

The interrupt service routine does nothing more than update the time-of-day clock. A more sophisticated operating system kernel would use the CTC to check the task queue status. In synchronous data communications, it is often necessary to ensure that a flag or sync character separates two adjacent message packets. Since some serial controller devices have no way to determine the status of sync characters sent, the user must use

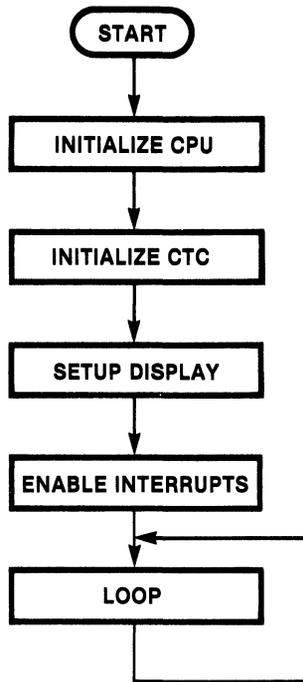
time delays to separate messages with the appropriate number of sync characters. Typically, software or timer delays are used to provide the time necessary to allow the characters to shift out of the serial device. The disadvantage of using this method is that variable baud rates shift characters at variable times so a worst-case time must be allowed if the baud rate is not known. If the bit rate clock is supplied by the modem, as is normally the case, this problem becomes even more acute.

A solution to this problem is to use a counter to count the number of bits shifted out of the serial device. With the CTC tied to the transmit clock line of the serial device, the CTC can be programmed to delay a certain number of bits before the CPU sends another message. This solves all of the problems mentioned and simplifies the message-handling software. Figure 6 shows the program needed to achieve the counting function. Note

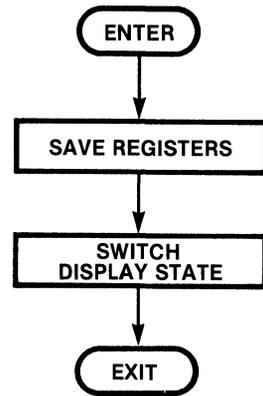
that the interrupt service routine disables the CTC, because the CTC is used only once with each message. Otherwise, the CTC would generate an interrupt each time the counter

reached terminal count.

Figure 1 shows the hardware implementation of the character delay counter using the CTC.



a) Main Program



b) Interrupt Service Routine

Figure 5. Software for CTC Counter Mode

```

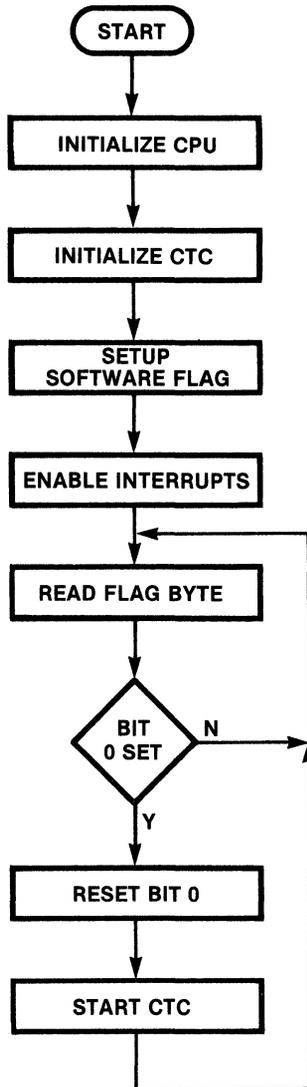
TEST CTC1
LOC  OBJ CODE M STMT SOURCE STATEMENT
      1 ,      CTC TEST PROGRAM
      2
      3 ,      THIS PROGRAM COUNTS EXTERNAL PULSES AND
      4 ,      CHANGES THE LED STATE EVERY 60 COUNTS
      5
      6 ,      PROGRAM EQUATES
      7
      8 CTC0: EQU    12          ; CTC 0 PORT
      9 CTC1: EQU   CTC0+1      ; CTC 1 PORT
     10 CTC2: EQU   CTC0+2      ; CTC 2 PORT
     11 CTC3: EQU   CTC0+3      ; CTC 3 PORT
     12 LITE: EQU   0E0H        ; LIGHT PORT
     13 RAM: EQU   2000H        ; RAM START ADDR
     14 RAMSIZ EQU   1000H
     15 COUNT EQU   60          ; COUNTER TIME CONSTANT
     16
     17
     18 ,      CTC EQUATES
     19
     20 CCW: EQU    1
     21 INTEN EQU   80H
     22 CTRMODE EQU   EQU    40H
     23 P256: EQU   20H
     24 RISEDG EQU   10H
     25 PSTRT EQU   8
     26 TLOAD: EQU   4
     27 RESET EQU   2
  
```

```

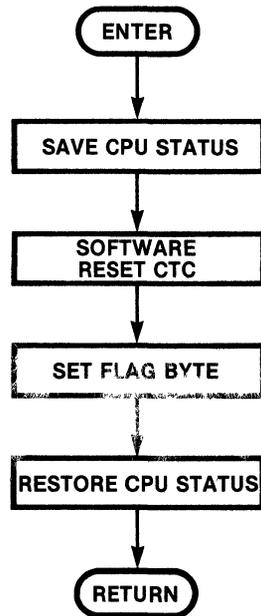
TEST. CTC1
LOC  OBJ CODE M STMT SOURCE STATEMENT
      28 *E
      29
      30 ; ; *** MAIN PROGRAM ***
      31
0000      32          ORG      0
0000      33          JP        BEGIN
      34
0010      35          ORG      $. AND. OFFFOH. DR. 10H
      36          INTVEC:
0010      37          DEFW     ICTCO
0012      38          DEFW     ICTC1
0014      39          DEFW     ICTC2
0016      40          DEFW     ICTC3
      41
      42          BEGIN:
0018      43          LD        SP, STAK          ; INIT SP
001B      44          IM        2              ; VECTOR INTERRUPT MODE
001D      45          LD        A, INTVEC/256    ; UPPER VECTOR BYTE
001F      46          LD        I, A
0021      47          CALL     INIT              ; INIT DEVICES
0024      48          EI              ; ALLOW INTERRUPTS
      49
0025      50          JR        $              ; LOOP FOREVER
      51
      52          INIT:
0027      53          LD        A, INTEN+CTRMODE+TCLDAD+RESET+CCW
0029      54          OUT      (CTC1), A          ; SET CTC MODE
002B      55          LD        A, COUNT
002D      56          OUT      (CTC1), A          ; SET TIME CONSTANT
002F      57          LD        A, INTVEC. AND 1111000B
0031      58          OUT      (CTCO), A          ; SET VECTOR VALUE
0033      59          XOR      A
0034      60          LD        (DISP), A          ; CLEAR DISPLAY BYTE
0037      61          RET
      62 *E
      63
      64 ; ; INTERRUPT SERVICE ROUTINE
      65
      66          ICTCO.
      67          ICTC2:
      68          ICTC3:
0038      69          EI              ; DUMMY ROUTINES
0039      70          RETI
      71
      72          ICTC1:
003B      73          CALL     SAVE              ; SAVE REGISTERS
003E      74          LD        A, (DISP)          ; BLINK LITES
0041      75          CPL
0042      76          LD        (DISP), A
0045      77          OUT      (LITE), A
0047      78          RET
      79
      80 ; ; SAVE REGISTER ROUTINE
      81
      82          SAVE:
0048      83          EX        (SP), HL
0049      84          PUSH    DE
004A      85          PUSH    BC
004B      86          PUSH    AF
004C      87          CALL    GO
004F      88          POP     AF
0050      89          POP     BC
0051      90          POP     DE
0052      91          POP     HL
0053      92          EI
0054      93          RETI
      94
      95          GO.
0056      96          JP        (HL)
      97 *E
      98

```

		TEST CTC1				
LDC	OBJ	CODE	M	STMT	SOURCE	STATEMENT
			99	..	DATA AREA	
			100			
2000			101		ORG	RAM
2000			102		DEFS	64 ; STACK AREA
			103	STAK	EQU	\$
2040			104	DISP:	DEFS	1 ; LITE DISPLAY BYTE
			105			
			106		END	



a) Main Program



b) Interrupt Service Routine

Figure 6. Software for CTC Single-Cycle Use

```

TEST. CTC3
LOC  OBJ CODE M STMT SOURCE STATEMENT
      1 ;      CTC TEST PROGRAM
      2
      3 ;      THIS PROGRAM INITIALIZES CTC INTERRUPT VECTOR,
      4 ;      THEN STARTS CTC 3, THEN WAITS FOR CTC 3 TO
      5 ;      TERMINATE. AFTER TERMINATING, THE CTC INTERRUPT
      6 ;      THE CPU AND ENTERS A SERVICE ROUTINE THAT SETS
      7 ;      A PROGRAM FLAG TO INDICATE ZERO COUNT, AND
      8 ;      RESETS CTC 3.
      9
     10 ;      EQUATES
     11
     12 RAM:  EQU    2000H      ; RAM START ADDRESS
     13 RAMSIZ: EQU    1000H   ; RAM SIZE
     14 CTC0:  EQU    12      ; CTC 0 PORT
     15 CTC1:  EQU    CTC0+1   ; CTC 1 PORT
     16 CTC2:  EQU    CTC0+2   ; CTC 2 PORT
     17 CTC3:  EQU    CTC0+3   ; CTC 3 PORT
     18 COUNT: EQU    20      ; COUNT 20 PULSES
     19
     20 ;      CTC PARAMETERS
     21
     22 CCW:    EQU    1        ; CTRL BYTE
     23 INTEN:  EQU    80H      ; INTERR. ENABLE
     24 CTRMODE: EQU    40H    ; COUNTER MODE
     25 P256:   EQU    20H     ; PRESCALE BY 256
     26 RISEDG: EQU    10H     ; START ON RISING EDGE
     27 PSTRT:  EQU    8       ; PULSE STARTS TIMING
     28 TCLOAD: EQU    4      ; TIME CONST. FOLLOWS
     29 RESET:  EQU    2       ; SOFTWARE RESET
     30 *E
     31
     32      ORG    0
     33      JP    BEGIN        ; GO MAIN PROGRAM
     34
     35      ORG    *. AND. OFFFOH. OR. 10H
     36 INTVEC:
     37 CTCVEC:
     38      DEFW  ICTC0
     39      DEFW  ICTC1
     40      DEFW  ICTC2
     41      DEFW  ICTC3
     42
     43 ;;      MAIN PROGRAM
     44
     45 BEGIN:
     46      LD    SP, STAK      ; INIT SP
     47      LD    A, INTVEC/256 ; INIT VECTOR REG.
     48      LD    I, A
     49      IM    2            ; VECTORED INTERRUPT MC
     50      LD    A, CTCVEC. AND. 11111000B
     51      OUT  (CTC0), A     ; SETUP CTC VECTOR
     52      LD    A, 1        ; SET FLAG BYTE
     53      LD    (FLAG), A
     54      EI
     55
     56 LOOP:
     57      LD    A, (FLAG)    ; READ FLAG BYTE
     58      BIT  0, A
     59      JR    Z, LOOP      ; BRANCH IF NOT SET
     60      RES  0, A         ; CLEAR FLAG BYTE
     61      LD    (FLAG), A
     62      LD    A, INTEN+CTRMODE+RISEDG+TCLOAD+1
     63      OUT  (CTC3), A     ; LOAD CTC 3
     64      LD    A, COUNT
     65      OUT  (CTC3), A
     66      JR    LOOP
     67 *E
     68
     69 ;      INTERRUPT SERVICE ROUTINES FOR CTC
     70
     71 ICTC0:
     72 ICTC1:

```

TEST. CTC3					
LOC	OBJ CODE	M	STMT	SOURCE	STATEMENT
			73	ICTC2:	
0041	FB		74	EI	
0042	ED4D		75	RETI	; DUMMY INTERRUPT ROUTI
			76		
			77	ICTC3:	
0044	08		78	EX	AF, AF'
0045	3E03		79	LD	A, 00000011B ; RESET CTC 3
0047	D30F		80	OUT	(CTC3), A
0049	3A0020		81	LD	A, (FLAG) ; SET PROGRAM FLAG
004C	CBC7		82	SET	0, A
004E	320020		83	LD	(FLAG), A
0051	08		84	EX	AF, AF'
0052	FB		85	EI	
0053	ED4D		86	RETI	
			87	*E	
			88		
			89	;;	DATA AREA
			90		
2000			91	ORG	RAM
2000			92	FLAG: DEFS	1 ; PROGRAM FLAG BYTE
2001			93	DEFS	128
			94	STAK: EQU	\$
			95		
			96	END	

CONCLUSION

The versatility of the Z80 CTC makes it useful in a myriad of applications. System efficiency and throughput can be improved through prudent use of the CTC with the Z80 CPU. Coupled with the powerful, vectored

Interrupt capabilities of the Z80 CPU, the CTC can be used to supply counter/timer functions to the CPU. This reduces software overhead on the CPU and significantly increases system throughput.



Interfacing 16-Pin Dynamic RAMS to the Z80A Microprocessor

TABLE OF CONTENTS

	<u>PAGE</u>
INTRODUCTION.	2-121
16 PIN DYNAMIC RAM ADDRESSING	2-122
MEMORY REFRESH.	2-124
ACCESS TIME	2-125
Z80A/Z80 - CPU TIMING CONSIDERATIONS.	2-127
MEMORY CYCLE SELECTION.	2-131
DYNAMIC RAM MEMORY ORGANIZATION	2-132
SLOW MEMORY INTERFACE	2-138
DESIGN EXAMPLE.	2-140
CONCLUSIONS	2-146

INTERFACING 16-PIN DYNAMIC RAMS
TO THE Z80A MICROPROCESSOR

This application note will present the major design considerations and a design example for interfacing the 16-pin dynamic RAM devices, both 4K and 16K, to the Z80 and Z80A microprocessors. These devices will be emphasized because they are fast becoming the favorite memory component for data storage in microprocessor based systems. The 16K RAM (Zilog 6116), in particular, with design improvements over the 4K devices, will substantially reduce memory cost by quadrupling memory density in a package that is pin compatible with the 4K RAM.

This application note assumes a basic understanding of the Z80A CPU and dynamic RAM elements. The reader is referred to selected specification sheets on the various 4K and 16K dynamic RAMS and to the following Zilog literature:

Z80A CPU Technical Manual, and

Z6116 16K Dynamic RAM Product Specification

INTRODUCTION

16-pin dynamic RAMs are increasingly being used as the memory component for data storage in microprocessor-based systems. Their main features are low cost per bit and high bit density. These features, coupled with a low stand-by power mode, TTL-compatible inputs and outputs, and simple upgrade from 4K to 16K systems, have made these devices an attractive alternate to 18- or 22-pin dynamic RAMs.

Now, however, the system designer has to be concerned with the interface requirements of 16-pin dynamic RAMs. The characteristics of this memory element requires that refreshing of the memory be performed at periodic intervals in order to retain the stored data. This, coupled with the requirement for multiplexing address lines, has been the main drawback to their use. A typical interface generally required 12 to 20 standard TTL devices and included timing generators, decode logic, multiplexer circuitry, refresh logic, and buffers.

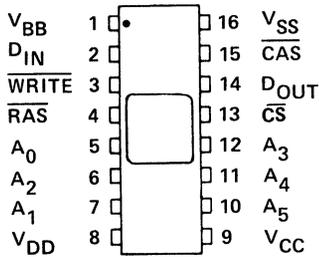
The Zilog Z80A microprocessor has been designed to simplify this interface with built-in refresh logic. This allows totally transparent RAM refresh without the need for a refresh counter or its associated multiplexer. During each memory opcode fetch cycle, a dedicated line from the CPU ($\overline{\text{RFSH}}$) is used to indicate that a refresh read of all dynamic memories should be performed. With $\overline{\text{RFSH}}$ in the true state (LOW), the lower seven bits of the address bus identify one ROW address to be refreshed. Before the next opcode fetch, this address will have been incremented to point to the next ROW address. Since it is only necessary to refresh the 'ROWS', a total of 64 refresh cycles will refresh an entire 4K RAM, or 128 refresh cycles for a 16K RAM. Z80A-CPU refreshing is automatically performed during a portion of the instruction fetch cycle which is used for internal processing. Thus, the effect of refreshing the RAM is totally transparent to program execution, preventing the necessity of stealing cycles or stopping the CPU as would otherwise be required.

16 PIN DYNAMIC RAM ADDRESSING

Each cell of a dynamic RAM array is arranged in a matrix. Selection of a unique bit location within this matrix in a 4K RAM element will require 12 address lines while the 16K device requires 14. For the 16-pin RAM device to accommodate these lines, it will be required to divide them into two groups; Row addresses and Column addresses (six each for the 4K RAM and seven each for the 16K RAM). Each group is applied to the RAM on the same input lines (Figure 1) through an external multiplexer and latched into the chip by applying two clock strobes in succession. The first clock, the Row Address Strobe ($\overline{\text{RAS}}$), latches the Row address bits into the RAM (A0-A5 for the 4K, A0-A6 for the 16K). The second clock, the Column Address Strobe ($\overline{\text{CAS}}$), latches the Column address bits, (A6-A11 for the 4K, A7-A13 for the 16K) into the RAM.

Each cell, therefore, is uniquely addressed by row and column. When $\overline{\text{RAS}}$ goes active, all of the cells in the selected row respond (there are 64 rows in the 4K RAM matrix and 128 rows in the 16K RAM matrix) and are gated to sense amplifiers where the logic level of each cell is discriminated, latched, and rewritten. $\overline{\text{CAS}}$ activates a column in the matrix (there are 64 columns in the 4K RAM matrix and 128 columns in the 16K RAM matrix) which uniquely identifies the cell in the row output and yields the required bit to the output buffer.

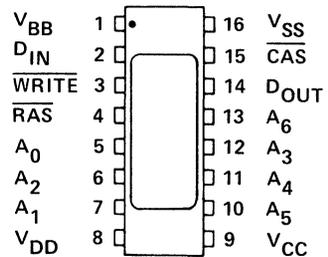
During refresh, the interface logic will enable the Row Address lines from the multiplexer. The CPU, with a true condition on the Refresh line ($\overline{\text{RFSH}}$), will then present the address (A0-A7) of the Row to be refreshed, and activate the memory request line ($\overline{\text{MREQ}}$) to initiate a memory cycle.



4K RAM

PIN NAMES

A ₀ -A ₆	ADDRESS INPUTS
CAS	COLUMN ADDRESS STROBE
D _{IN}	DATA IN
D _{OUT}	DATA OUT
RAS	ROW ADDRESS STROBE
WRITE	READ/WRITE INPUT
V _{BB}	POWER (-5V)
V _{CC}	POWER (+5V)
V _{DD}	POWER (+12V)
V _{SS}	GROUND



16K RAM

FIGURE 1. The pin assignments for 4K and 16K RAMs show identical functions for each, except Pin 13, which is used as a chip select in 4K RAMs and as the 7th multiplexed address line in the 16K RAM.

MEMORY REFRESH

When any row in a 16-pin dynamic RAM is actively cycled, all locations within that row are refreshed. To refresh the entire RAM, it is only necessary to perform a $\overline{\text{RAS}}$ only memory cycle ($\overline{\text{CAS}}$ is not required for a refresh sequence) at each of the 64 row addresses for the 4K device and 128 row addresses for the 16K device, every 2 milliseconds or less.

The Z80 CPU refreshes the memory more frequently than is necessary to meet the 2ms row refresh requirement. Under worst case conditions, no more than 19T states will separate opcode fetch cycles (the EX (SP),HL instruction is representative of the longest time between opcode fetches). Assuming this worst case period between opcode fetches and, therefore, refresh cycles, the following times for total refresh for both 4K and 16K RAMS at 2.5 MHZ and 4 MHZ are shown below:

REFRESH TIME			
MEMORY SIZE	Z80-CPU 2.5 MHZ	Z80A-CPU 4.0 MHZ	NO. OF REQUIRED REFRESH CYCLES/2 mS
4K	487 us (max)	304 us (max)	64
16K	974 us (max)	608 us (max)	128

TABLE 1. WORST CASE MEMORY REFRESH CYCLES ASSUMING NO WAIT STATES

From the above table, it can be seen that the worst case refresh time for 16K RAMS consumes approximately 1/2 of the available 2ms time interval while the 4K RAM consumes only about 1/4 of the allotted time. This provides for optional use of the refresh cycle for other CPU transparent bus activity, such as DMA and CRT refresh.

ACCESS TIME

Most dynamic RAMS have access times in the range of 150ns to 300ns. This access begins with the leading edge of the row address strobe ($\overline{\text{RAS}}$). The column address strobe ($\overline{\text{CAS}}$) completes this access cycle. The time between the fall of $\overline{\text{RAS}}$ and the fall of $\overline{\text{CAS}}$ is identified as the $\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ delay time (t_{RCD}), and can be related to the previous access times as follows:

$$t_{\text{RACmax}} = t_{\text{RCDmax}} + t_{\text{CACmax}}$$

WHERE t_{RACmax} = Access time from $\overline{\text{RAS}}$

t_{RCDmax} = max $\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ delay time

t_{CACmax} = Access time from $\overline{\text{CAS}}$

As long as t_{RCD} is less than max value (but greater than t_{RCDmin}), the worst case access is from $\overline{\text{RAS}}$ (see Figure 1). If $\overline{\text{CAS}}$ is applied at a point in time beyond the t_{RCDmax} limit, the access time from $\overline{\text{RAS}}$ will be lengthened by the amount that t_{RCD} exceeds the t_{RCDmax} limit and the access time from $\overline{\text{CAS}}$ (t_{CAC}) will be the critical parameter. Note, however, that reducing t_{RCD} to something less than t_{RCDmax} will have no effect at reducing t_{RACmax} .

The significance of the min/max value on t_{RCD} is that $\overline{\text{CAS}}$ can be brought low any time within this window and not affect access time. This is a great improvement from early 4K designs that required $\overline{\text{CAS}}$ to be brought low at a set minimum time from $\overline{\text{RAS}}$ low in order to avoid increasing access time. This made no allowance for the time required to switch the MUX from ROW to COLUMN addresses, requiring that the worst case multiplexing time delay be added to the specified access time.

This window, for the application of the external $\overline{\text{CAS}}$, is the result of gating $\overline{\text{CAS}}$ internal to the chip. The internal $\overline{\text{CAS}}$ is inhibited until the occurrence of a delayed signal derived from $\overline{\text{RAS}}$. Therefore, $\overline{\text{CAS}}$ can be activated as soon as the requirement for the row address hold time (t_{RAH}) has been satisfied and the address inputs have been changed from row to column. Note that the column address set-up time (t_{ASC}) can be assumed to be zero for all dynamic RAMs (See Figure 2).

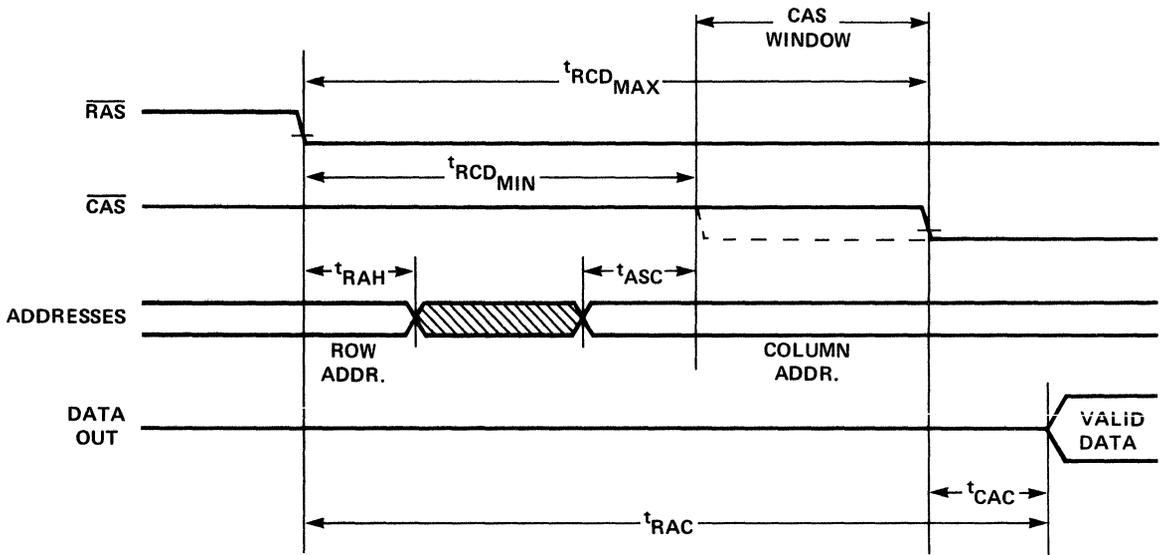


Figure 2 Dynamic ram access time parameters

Z80A/Z80 - CPU TIMING CONSIDERATIONS

The Z80A/Z80 CPU is designed to allow efficient and effective interface with dynamic RAM memories. Figures 3 through 8 identify the timing for CPU data, address signals, and control signals associated with memory interface for the Z80A and Z80. The opcode fetch, with its associated refresh cycle, will represent the worst case memory access, requiring data to be returned to the CPU in the first two T states. Memory read and write cycles have relaxed timing requirements as indicated in Figures 5 through 8. This will require memories with access times of 250ns or less for the Z80A and 400ns or less for the Z80. These numbers, however, do not take into consideration the propagation delays through any buffer logic added.

Notice that addresses are stable well before $\overline{\text{MREQ}}$ goes active, giving sufficient time for address decode logic to settle. The main concern, therefore, is propagation delay from $\overline{\text{MREQ}}$ to $\overline{\text{RAS}}$. This should be kept to a minimum since it will directly affect access time.

From Figure 7, it can be seen that write ($\overline{\text{WR}}$) goes active on the trailing edge of T2. The CPU, therefore, usually performs a read-modify-write cycle ($\overline{\text{CAS}}$ before $\overline{\text{WR}}$). To utilize the early write cycle ($\overline{\text{WR}}$ active before $\overline{\text{CAS}}$) and allow 16K systems to tie their inputs and outputs together, the read line ($\overline{\text{RD}}$) from the CPU can be inverted and used instead of $\overline{\text{WR}}$. This requires, however, that write data be valid before $\overline{\text{CAS}}$.

From Figure 4, it can be seen that the minimum high time for $\overline{\text{MREQ}}$ between opcode fetch and refresh cycles is 105ns for the Z80A. For systems that use $\overline{\text{MREQ}}$ to generate $\overline{\text{RAS}}$, this is not sufficient to satisfy $\overline{\text{RAS}}$ precharge time requirements of the slower RAMs. However, as will be shown in the design example, relatively simple logic can be used to extend $\overline{\text{RAS}}$ high time between these cycles.

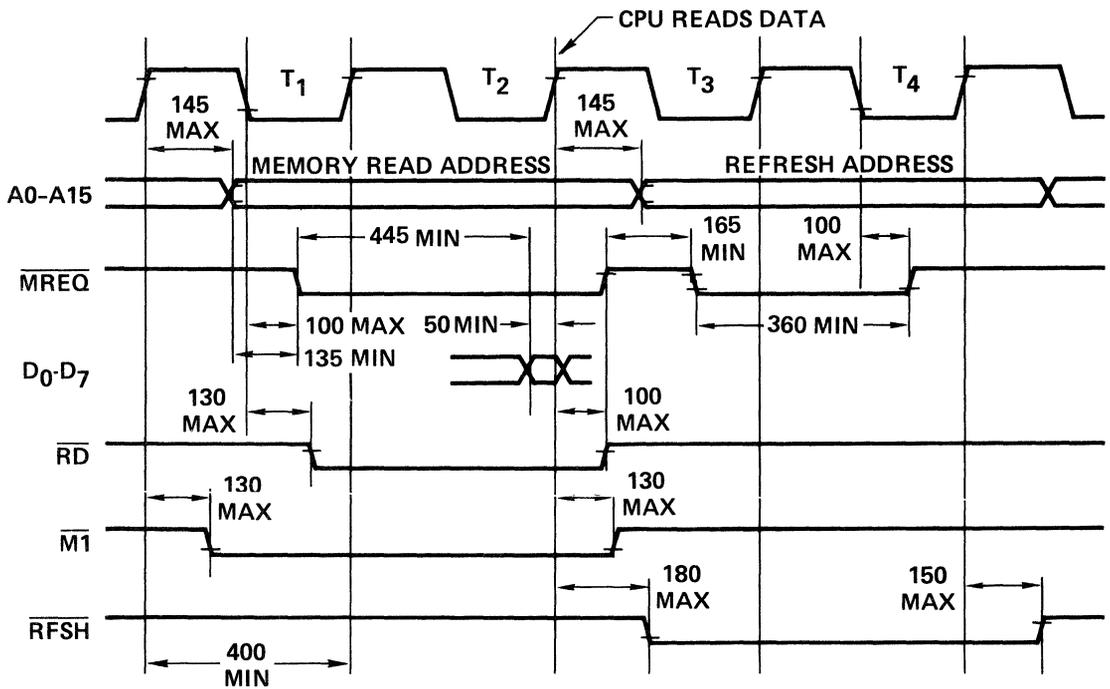
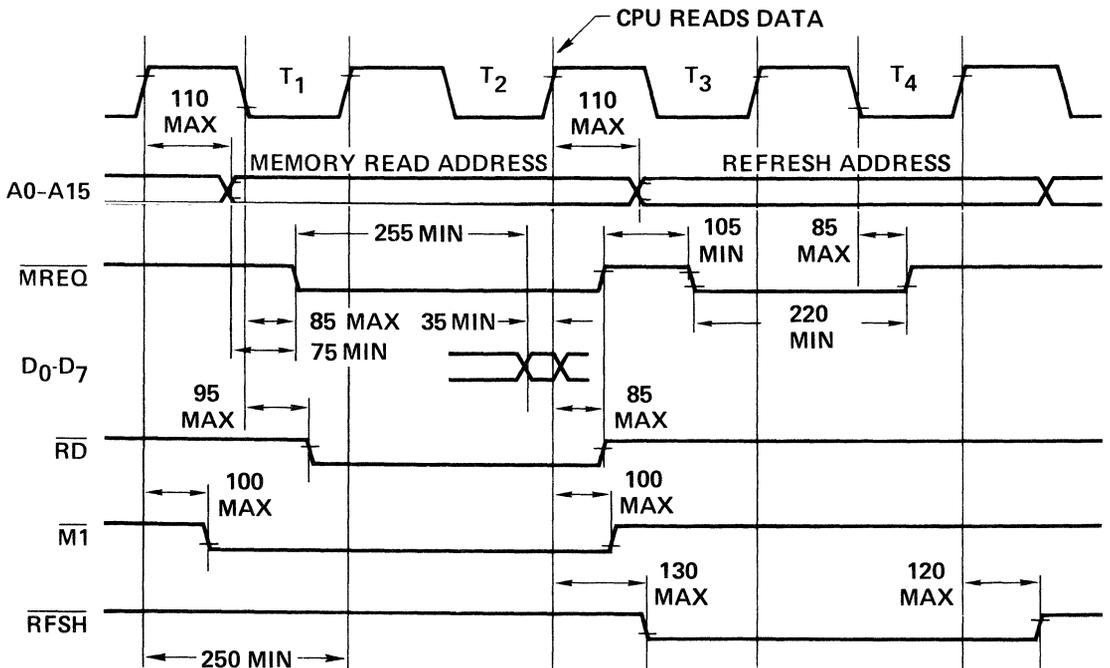


Figure 3 Z80-CPU op code fetch cycle timing at 2.5 MHz clock



NOTE: ALL TIMING IN ns ASSUME RISE/FALL TIME: 15ns

Figure 4 Z80A-CPU op code fetch cycle timing at 4 MHz clock

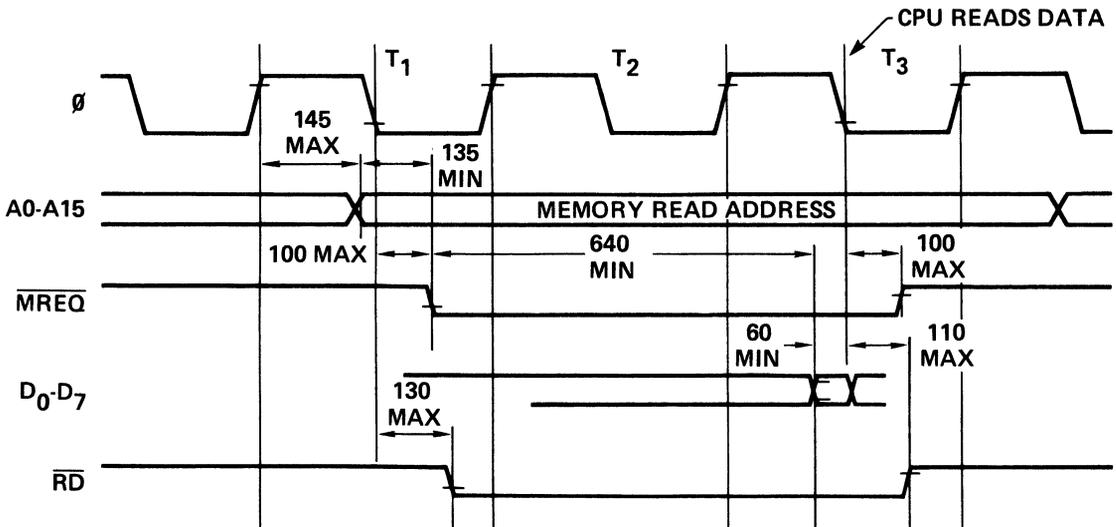
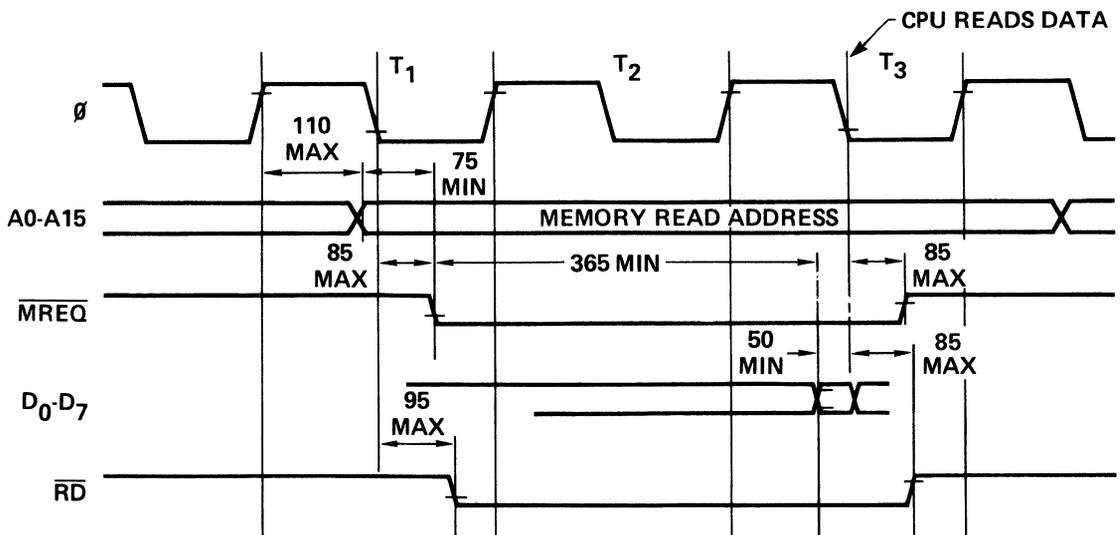


Figure 5 Z80-CPU read cycle timing at 2.5 MHz clock



NOTE: ALL TIMING IN ns
ASSUME RISE/FALL TIME: 15ns

Figure 6 Z80A-CPU read cycle timing at 4 MHz clock

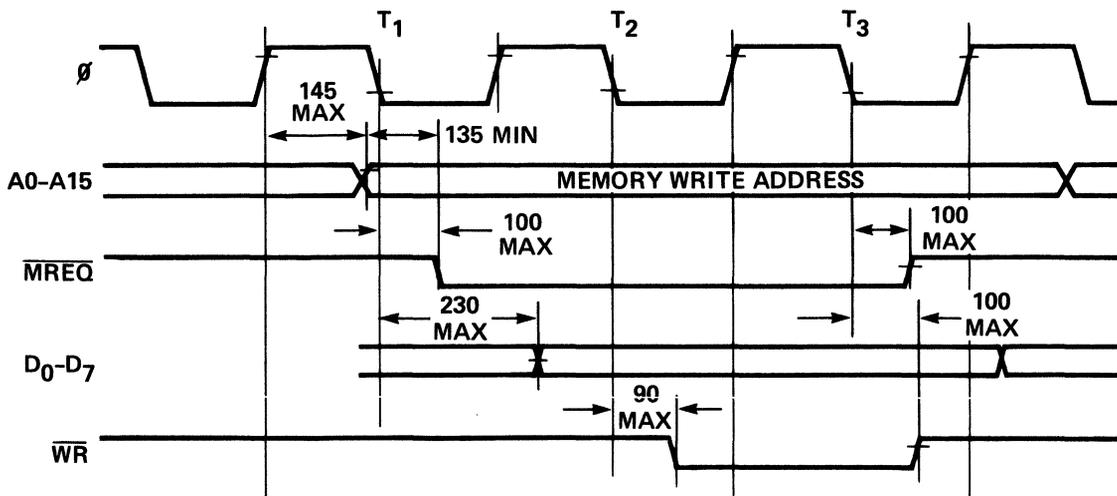
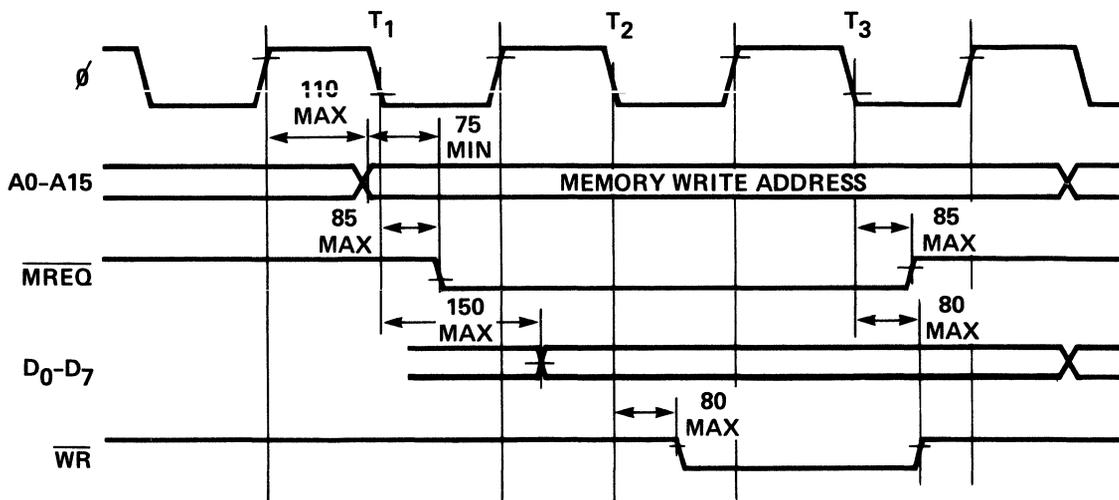


Figure 7 Z80-CPU write cycle timing at 2.5 MHz clock



NOTE: ALL TIMING IN ns ASSUME RISE/FALL TIME: 15ns

Figure 8 Z80A-CPU write cycle timing at 4 MHz clock

MEMORY CYCLE SELECTION

Selection of an operating mode is controlled by a combination of CAS and WRITE while RAS is active. The available modes in most 4K and 16K RAMS are a read cycle, a write cycle, a read-write cycle, and a read-modify-write cycle. For some of the newer 4K devices and the 16K device, another type of cycle known as page mode allows for faster access time by keeping the same row address and strobing successive column addresses onto the chip.

The read-modify-write cycle can be accomplished in less time than a read cycle followed by a write cycle because the addresses do not change in between. It is, therefore, possible to generate the write strobe as soon as the data modification is complete. In other words, data is read from a cell, modified, and then rewritten in its modified form into the same cell. In contrast, a read-write cycle does not require data to be valid at the output before the write operation is started.

In a write cycle, if the WRITE input is brought low before CAS (early write), the data is strobed in by CAS. In a delayed write cycle, the WRITE line goes low after CAS and data is strobed in with WRITE.

DYNAMIC RAM MEMORY ORGANIZATION

Careful attention must be given to dynamic RAM memory array layout. Page decoding, power line routing and filtering, noise suppression and generation, buffer drive requirements, and system upgrading are all important considerations during the design phase.

If a memory array consisting of 4K devices exceeds 4K bytes (one page), it will be necessary to configure multiple rows. Each row, or page, is selected by decoding address lines A12-A15. If the system is intended to be upgraded with 16K devices, the chip select line (\overline{CS}) should not be used for device selection. Instead, \overline{RAS} should be gated to the selected 4K bank with \overline{CAS} being applied to all devices. (Chips that receive \overline{CAS} but no \overline{RAS} will be unselected.) The \overline{CS} line should be distributed to all devices and tied to ground. It can then be used as the seventh address line when upgrading to 16K RAMS.

The \overline{CAS} line is used to control the output buffer in a configuration where the outputs are or-tied. If true data is still available from a previous cycle (assuming latched output 4K RAMS), then \overline{CAS} deselects these devices if they are not being accessed during the current cycle. Note also that if \overline{RAS} is inactive and \overline{CAS} active, the only function that is performed is to change any true outputs to the high impedance state. Figure 9 shows the logic for one data bit in an 8K-byte system utilizing two banks of 4K RAM devices.

The absence of an output latch on most 16K RAMS can allow for simplification in system design. Unlike the latched 4K devices which need an extra cycle to clear the latch, the 16K non-latched device maintains data valid only during the time the \overline{CAS} clock is active. Each memory cycle, therefore, can be maintained as an independent cycle, allowing the data input and output pin to be directly connected. This is assuming, however, that the write line goes true before \overline{CAS} (early write mode).

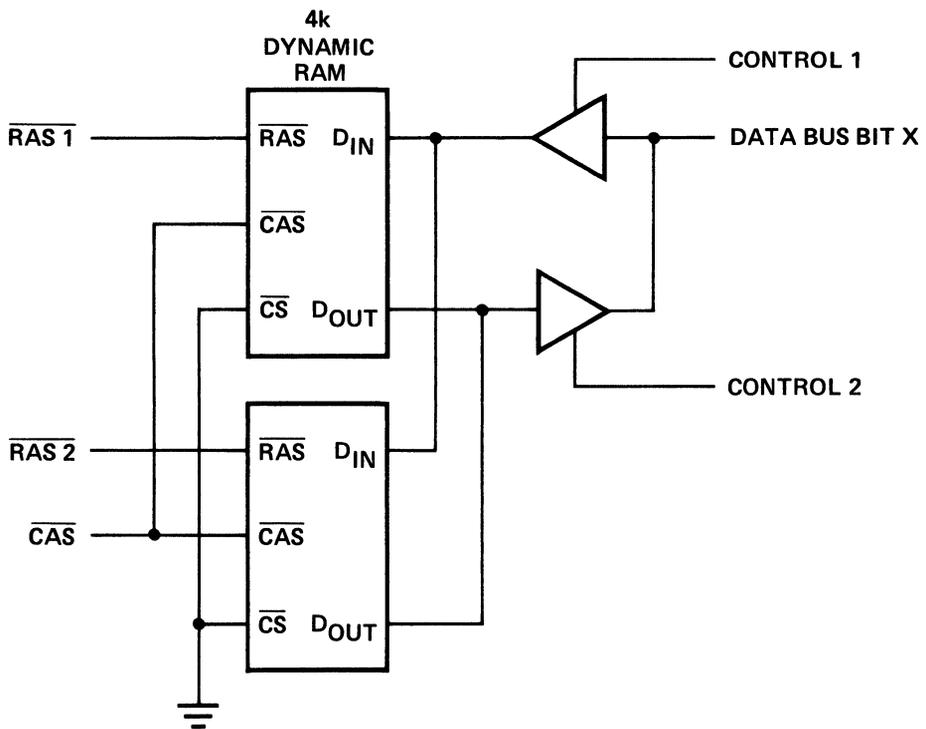


Figure 9 Partial memory configuration in 8k byte system

All inputs on most dynamic RAMS are TTL compatible (on some 4K devices \overline{RAS} , \overline{CAS} , and the \overline{WRITE} line require a 2.7 volt minimum logic 1 level which will require a pull-up resistor on the TTL driver). These TTL inputs, however, do not source current; but instead, present purely capacitive loads. This capacitance will vary between 5pf and 10pf on most 4K and 16K devices. With a large number of RAMS in a memory array, capacitive loading becomes a consideration. A 16K byte memory array made up of 4K devices will present from 150pf to 250pf of input capacitance to the input buffers. Most TTL outputs are not specified above 50pf. Therefore, a TTL driver must be used that can provide enough charging and discharging current to achieve the required voltage transition within the allotted time. A fairly accurate calculation can be made for determining the required drive current by using the standard relationship between the charging current i , the capacitance C , the voltage transition V , and the allotted time T :

$$i = C \frac{\Delta V}{\Delta T}$$

For example, if the worst case capacitance on an address line is 250pf and it is required to change this address line within a 60ns period from zero volts to 3 volts, the driving current is:

$$i = 250 \times 10^{-12} \frac{(3)}{60 \times 10^{-9}} = 12\text{mA}$$

The power consumption of dynamic RAMS, which generally varies from 350mw to 1 watt, depends on the state of the \overline{RAS} and \overline{CAS} clocks. The device draws minimum current when these clocks are inactive (standby mode). At each transition of the clocks, the device will draw current. This current corresponds to the precharging of these lines which represent large capacitance loads. During standby, the power consumption is usually less than 20mw. Also, because of this very low power dissipation when the clocks are turned off, the technique of decoding \overline{RAS} to selected chips results in a sizable decrease in power consumption (approximately 60% of all active power is due to \overline{RAS} and only 40% is due to \overline{CAS}). Because the memory is dynamic, the power dissipation is a function of the rate of memory access and, therefore, operating frequency.

The resulting current spike, which occurs when the $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ clocks go through their negative transitions, is coupled onto the power supply busses causing noise throughout the system. To compensate for this noise, high-frequency ceramic bypass capacitors should be placed within the memory array. A good practice is to supply a .1uf capacitor every other device between +12V and ground. Alternating between these capacitors, a Decoupling on the +5V line to prevent noise from affecting TTL logic should consist of a .01uf capacitor every 4 or 5 devices. For low frequency decoupling, a 10uf tantalum capacitor between +12 and ground should be supplied every 16 devices with a 10uf tantalum between -5V and ground every 32 devices.

The use of a multi-layer board with internal power and ground planes would be beneficial in a dynamic RAM system. However, proper routing of power lines on a two-sided card should provide satisfactory results. It has been found that bussing the +12 volt and ground lines both horizontally and vertically at every device will reduce noise and greatly improve RAM performance. The -5 and +5 volt lines need not be bussed in this fashion since they are less heavily loaded and are less likely to see current spikes.

Keeping the layout as small as possible and locating the address and data bus buffers as close to the array as possible will also reduce potential ringing and reflections.

Figure 10 represents a typical expandable RAM interface for a total memory capability of either 16K using 4K devices or 64K using 16K devices. The multiplexing of address lines is done by "wire-oring" 8T97 drivers and controlling the tri-state input for row to column switching. Since the minimum voltage on any RAM input is -1 volt, a small series resistor (about 30ohms) is inserted on each RAM address line to surpass any undershoot that might occur. When using 4K RAMs, the lower section of the 74S139 decoder selects the desired 16K quadrant by decoding address lines A14 and A15. The upper section of the decoder selects the desired 4K bank in this quadrant by decoding address lines A12 and A13. When using 16K RAMs, the lower section of the decoder is not used and the upper section decodes the desired 16K

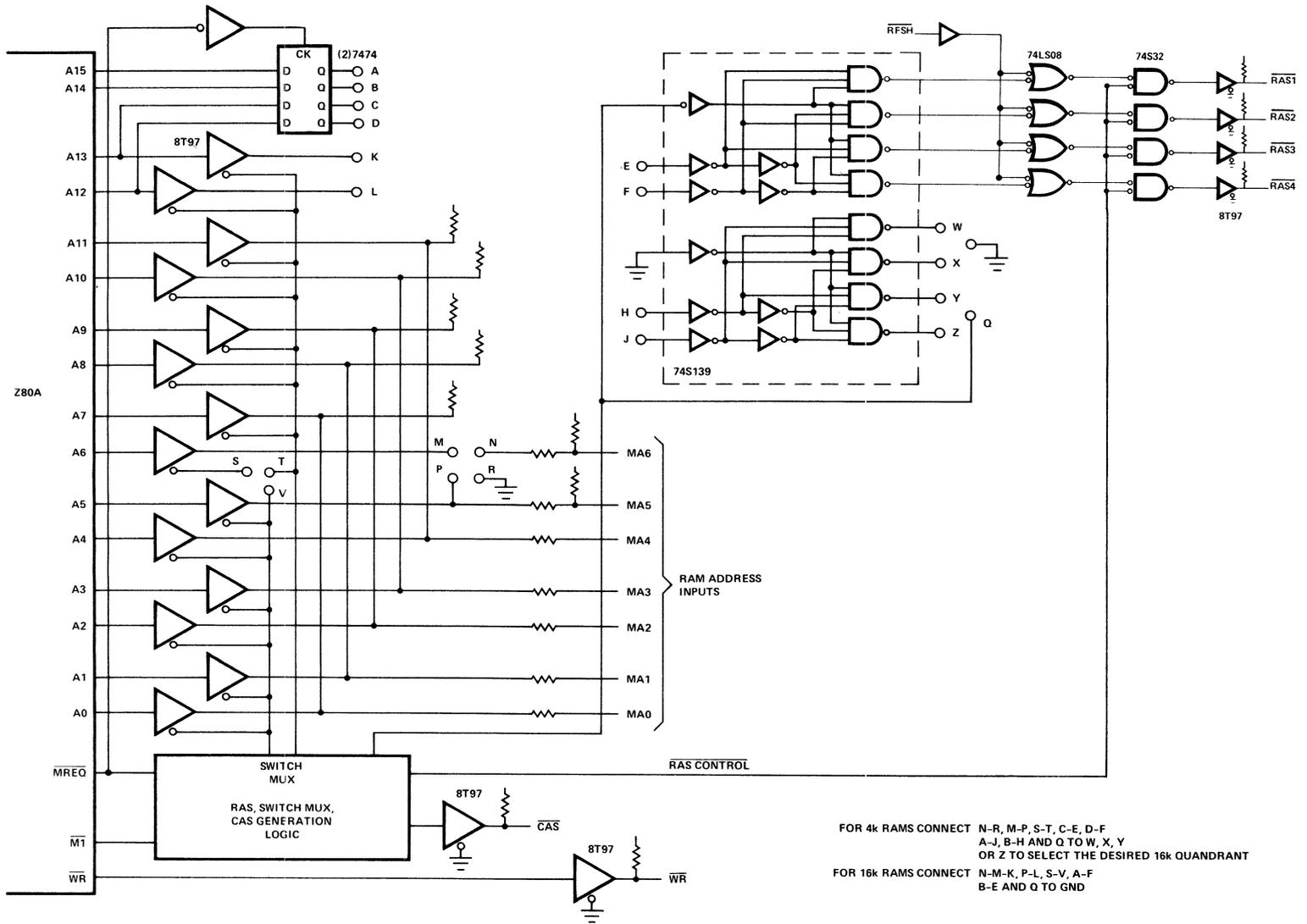


Figure 10 Z80A-16K/64K dynamic ram interface

quadrant with address lines A14 and A15. The latch on the upper address line is used to prevent potential spikes on the $\overline{\text{RAS}}$ lines as $\overline{\text{MREQ}}$ and the address lines change at the end of the cycle.

The use of 8T97 drivers, with an external pull-up resistor, will insure proper logic level and capacitance drive capability. When using 4K devices, memory address line 6 (MA6) is not needed and tied to ground (this is the chip select line on 4K RAMs). When using 16K RAMs, this line is the 7th address line (A6 for row and A13 for column).

SLOW MEMORY INTERFACE

When working with memory devices with long access times (2708 EPROMS with a 450ns max access time, for example), it will be necessary to add wait states to Z80A timing. Figure 11 shows how a JK flip-flop can be configured for adding one wait state (250ns with a 4MHz clock) to each memory cycle. When using dynamic memories that have access times between 250 and 350ns, it is only necessary to add wait states for Op Code fetch cycles, since this cycle is the critical one in terms of memory access requirements. In this case, the logic in Figure 11 can be controlled by $\overline{M1}$ instead of \overline{MREQ} to accommodate these memories.

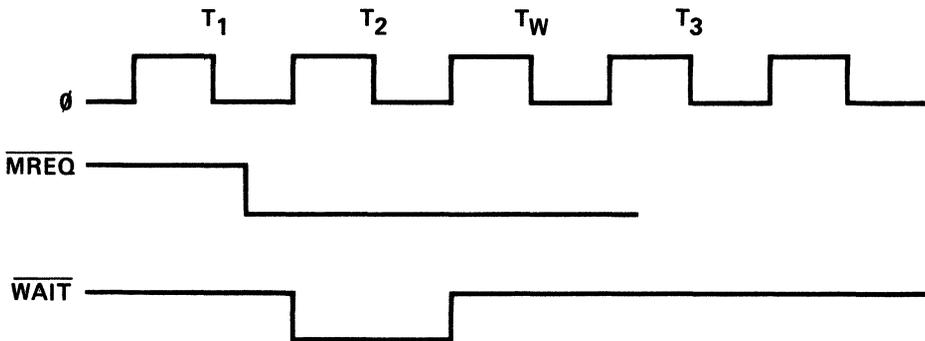
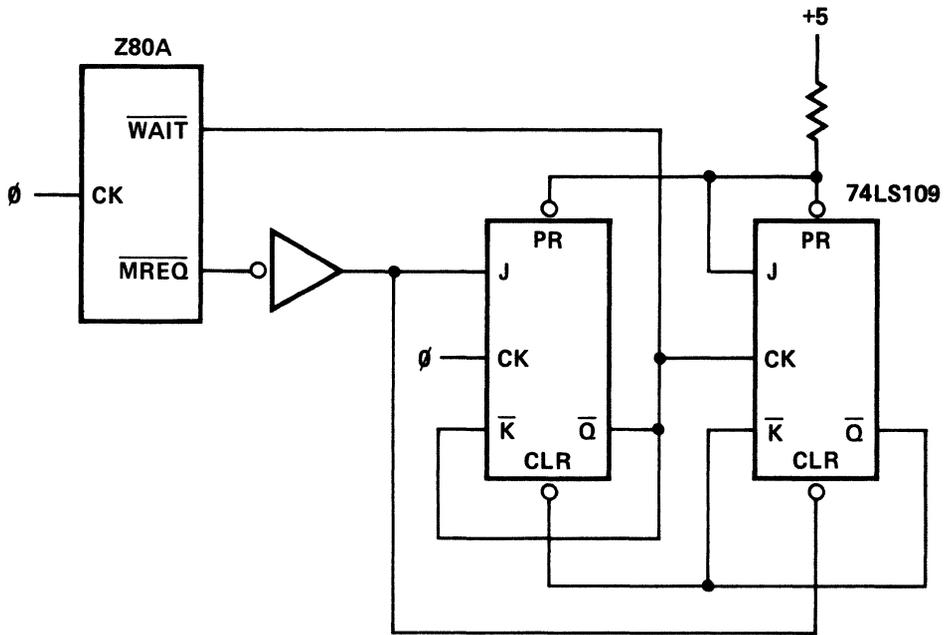


Figure 11 Adding one wait state to each memory cycle

DESIGN EXAMPLE

A typical design is presented to demonstrate a technique for dynamic RAM interface to the Z80A operating at 4MHz. Of the several approaches that could have been used for generating the timing signals needed for this interface, the tradeoffs for considering this approach consisted of the following:

1. Monostable-multivibrators could have been used to generate the time delays for the MUX switching and CAS signals, but one-shots are hard to adjust and are less reliable than other approaches.
2. The inherent delay in low power TTL gates could be used for this timing, but predictable timing intervals are hard to achieve at 4MHz.
3. A tapped delay line produces very accurate timing signals but is less attractive from a cost standpoint.

A synchronous technique has been chosen for this design because it generates accurate signals with a minimum of logic complexity and produces predictable results from system to system. The approach is to generate the CPU 4MHz clock from an 8MHz source. This 20 clock is then divided by two and used with the resulting 0 clock to generate the MUX switch and $\overline{\text{CAS}}$ signals after $\overline{\text{RAS}}$ has been generated from the fall of $\overline{\text{MREQ}}$. Figure 12 is a schematic diagram of this interface. Figure 13 indicates the timing relationship involved.

The ROW Address Strobe ($\overline{\text{RAS}}$) is generated at the fall of $\overline{\text{MREQ}}$. On the next rising edge of the 0 clock, 'A' flip-flop is clocked to generate the signal used to switch the multiplexer from ROW addresses to Column addresses. The following falling edge of the 20 clock is used to generate the $\overline{\text{CAS}}$ signal (B flip-flop). Flip-flop C is used to insure sufficient $\overline{\text{RAS}}$ precharge time, which must be taken into account since $\overline{\text{MREQ}}$ has a minimum high time of 100ns between Op Code fetch and refresh cycles and $\overline{\text{MREQ}}$, therefore, cannot be used to set $\overline{\text{RAS}}$ high. With $\overline{\text{CAS}}$ true, the trailing edge of $\overline{\text{RAS}}$ is clocked high with the 0 clock. This will extend the $\overline{\text{RAS}}$ high time to approximately 150ns.

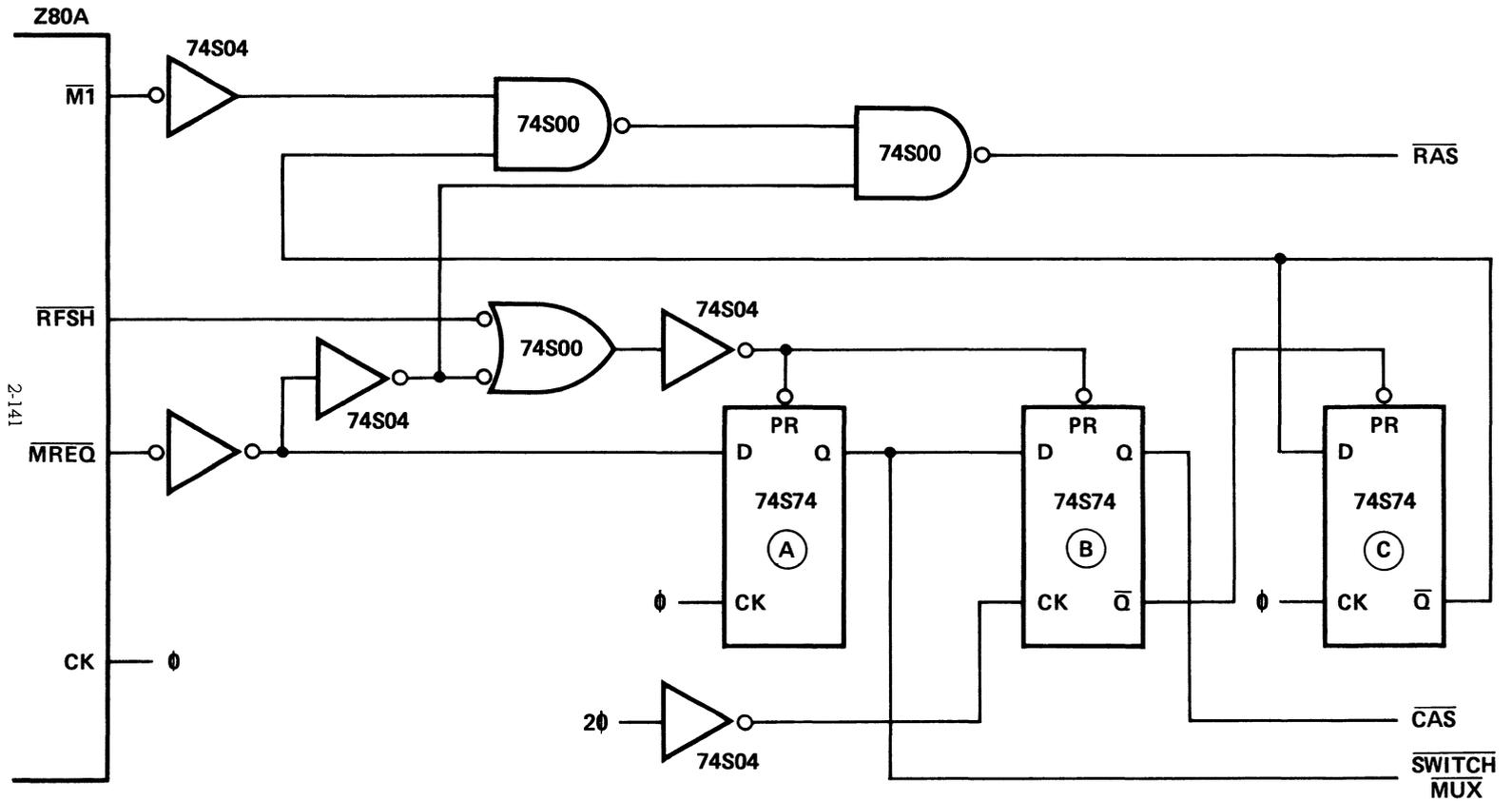


Figure 12 Z80A dynamic ram interface

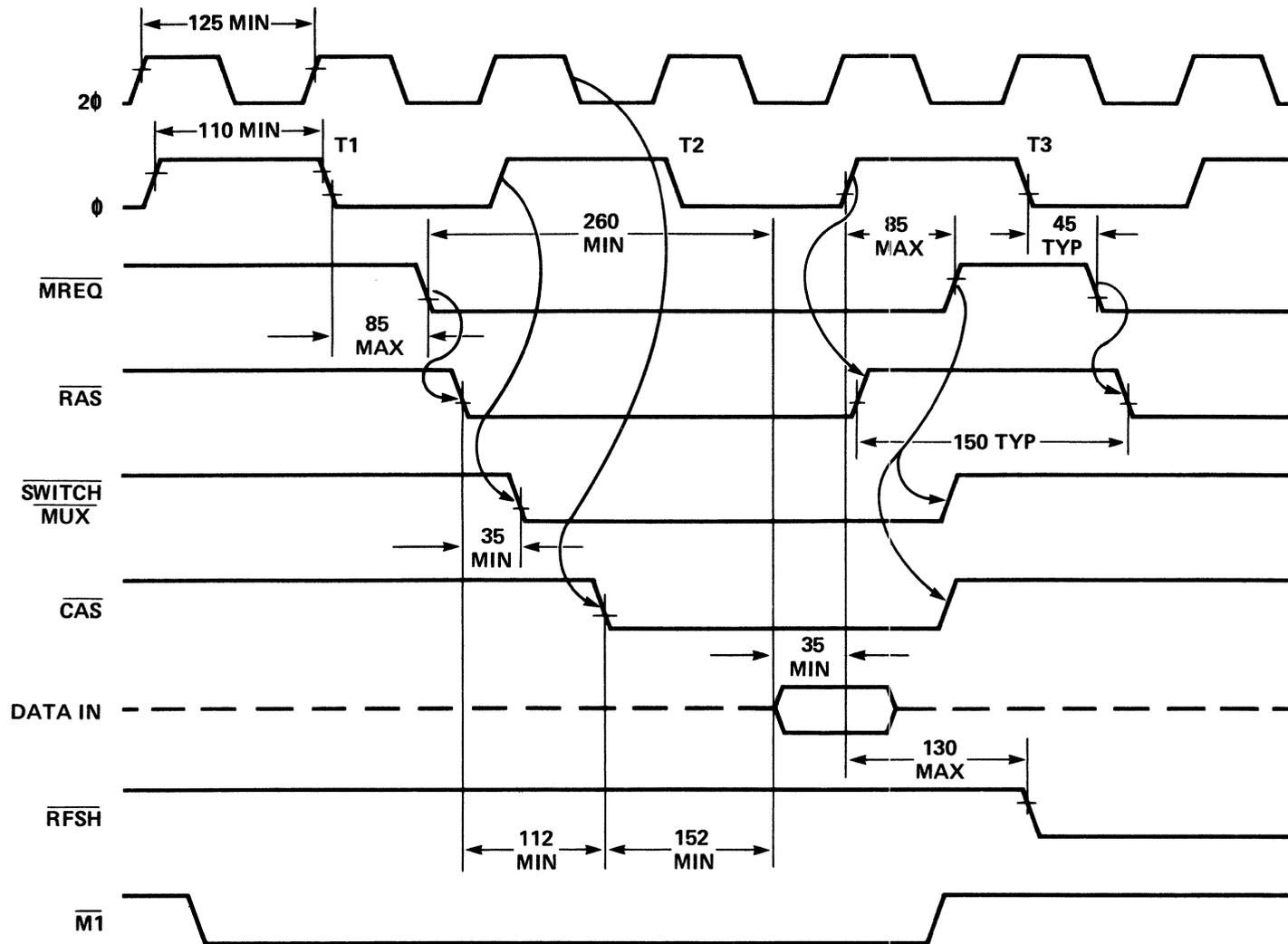


Figure 13 Z80A dynamic ram interface timing

This basic logic structure is configured into a microcomputer system and is seen in Figure 14. For simplicity, only the logic pertaining to the RAM interface is shown. Additional logic consisted of monitor software and a serial I/O interface to a CRT terminal.

Calculated timing parameters matched measured data quite accurately. Figures 15 through 20 indicate recordings taken during the Op Code fetch and refresh cycles at room temperature and at a Vcc of +5.0 volts.

From Figure 19, it can be seen that the interval between the leading edge of $\overline{\text{RAS}}$ and the leading edge of the switch MUX signals is approximately 50ns. The calculated interval is 35ns minimum and is consistent with the ROW address hold time tRAH (see Z6116 Product Specification) of all RAMs that are access time compatible with the Z80A.

At the leading edge of the switch MUX signal, the RAM addresses are switched from ROW to Column addresses. Assuming the column address set up time (tASC) to be zero (consistent with most dynamic RAMs), the interval for address switching is approximately 70ns as confirmed from calculated and measured data (see Figures 13 and 19). Scope triggering records both Row and Column addresses which appear to be superimposed on a typical RAM address line as seen in Figure 17.

Since the $\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ interval exceeds the tRCD max value of most access-compatible RAMs, the RAM access time is measured from the leading edge of $\overline{\text{CAS}}$. RAMs with $\overline{\text{CAS}}$ access times of 150ns or less should be compatible with this interface approach. If it is desired to keep the $\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ interval within or closer to the tRCD max limit, a 40 clock could be applied to the clock input of Flip-Flop B (Figure 12) instead of the 20 clock. This would reduce the $\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ interval to approximately 65ns.

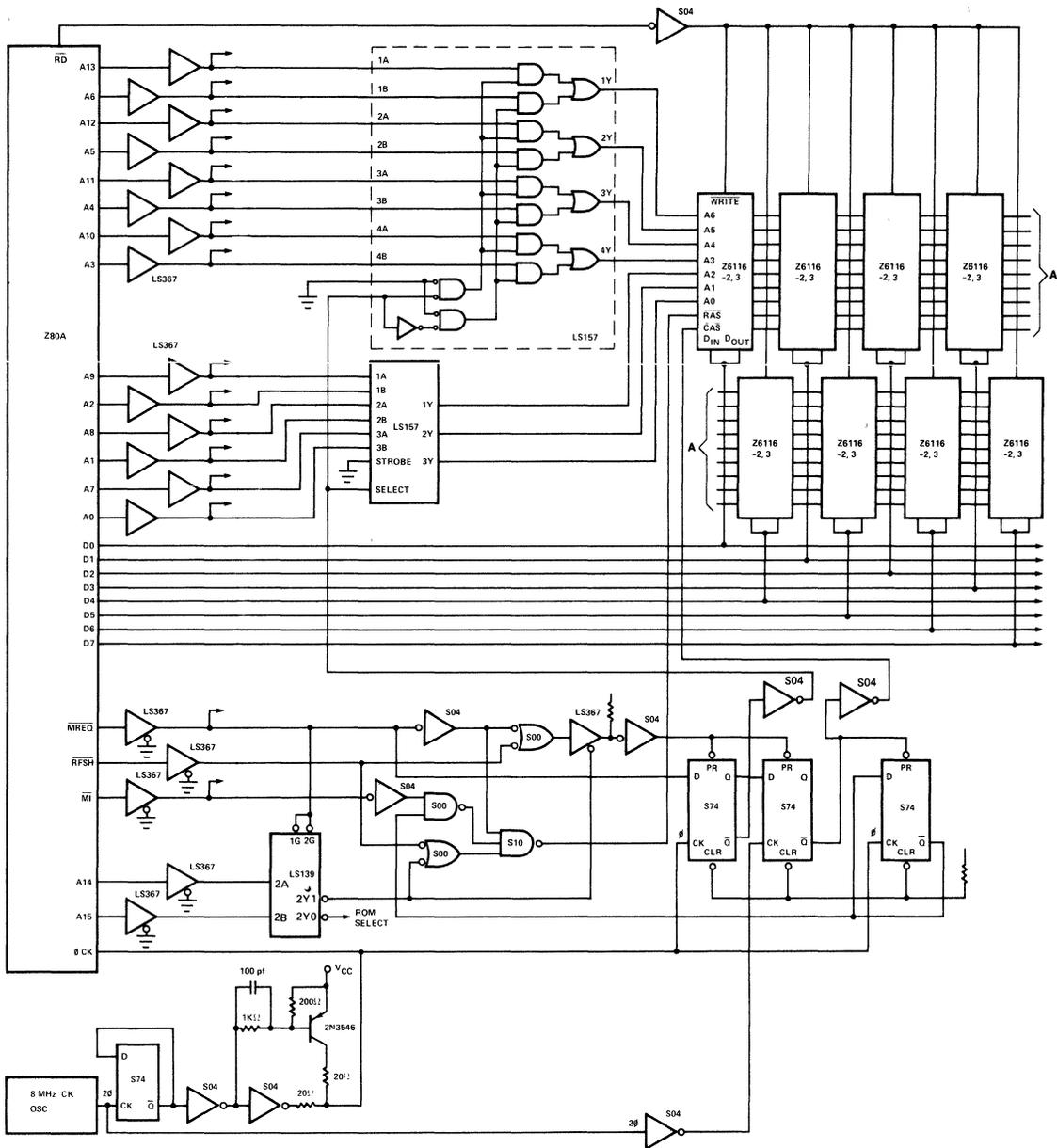


Figure 13 Z80A 16K ram interface

FIGURE 15

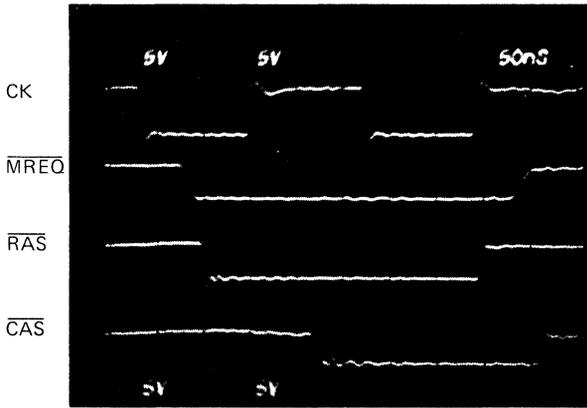


FIGURE 18

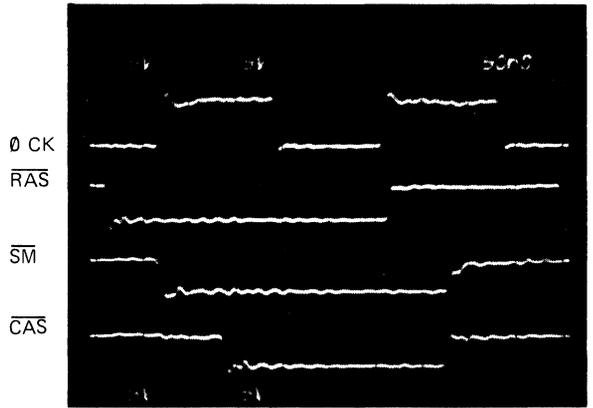


FIGURE 16

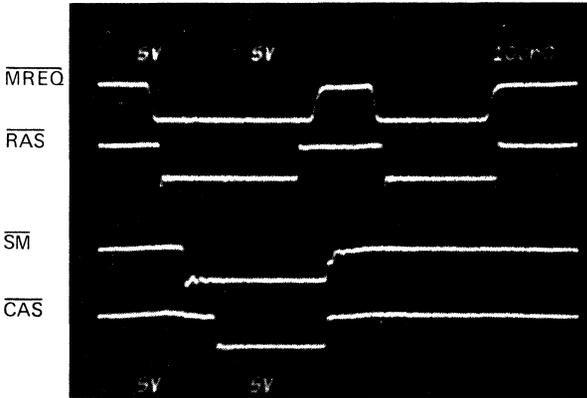


FIGURE 19

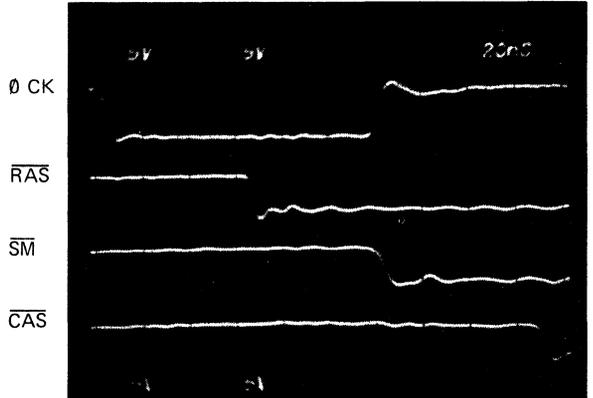


FIGURE 17

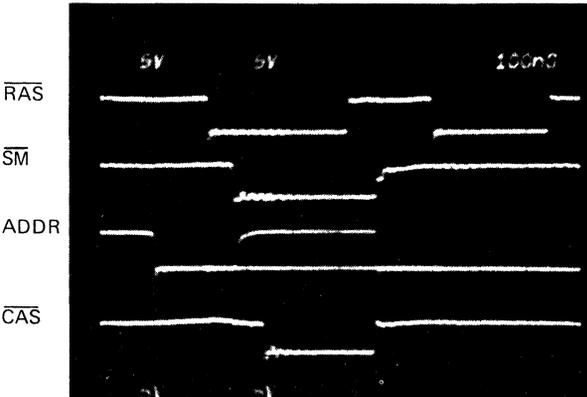
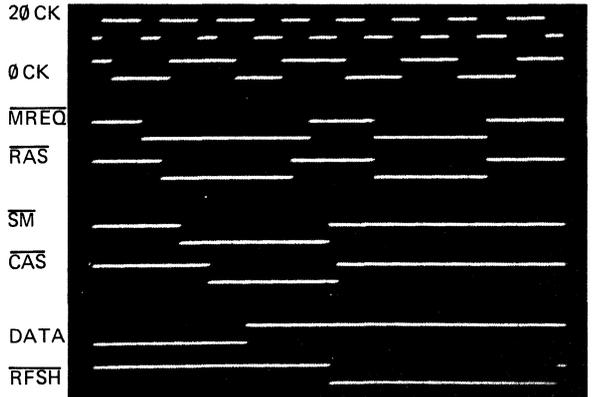


FIGURE 20



The recordings also show the relation between $\overline{\text{MREQ}}$ and $\overline{\text{RAS}}$ high time between Op Code fetch and refresh cycles. The calculated value for $\overline{\text{RAS}}$ high time was 150ns while the measured value was approximately 170ns. This allows adequate $\overline{\text{RAS}}$ precharge time for all access-compatible RAMs.

A composite of all the major control signals, including one data line, is seen in Figure 20. This recording, done with a logic analyzer, shows the relative relation of these signals during the Op Code fetch and refresh cycles. Notice, that during refresh, only $\overline{\text{RAS}}$ is active with the switch MUX and $\overline{\text{CAS}}$ signals disabled.

CONCLUSIONS

The high density, reduced standby power, and reduced cost per bit of 16-pin dynamic RAMs have made these devices suitable for an increasing number of applications. Their attractiveness is also enhanced by the ease of upgrading from 4K to 16K devices. With this increased usage, however, the interface logic between the memory array and the microprocessor becomes an important consideration. The Z80A has simplified this interface. Internal logic operates totally transparent to CPU operation, supplying refresh capability without the need for a refresh counter and its associated multiplexer. This interface can be configured with just five standard TTL gates to obtain synchronous generation of the $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and the multiplexer switching signal. Additional design attention must be given to the RAM layout which can have a dramatic effect on system performance. Dynamic RAMs tend to be noise generators as well as being noise sensitive. However, with proper attention to filtering, power line routing, and array organization, dynamic RAM memory can provide a cost effective solution for high-density storage.

Controlling Z80 microcomputer I/O? An interrupt-driven program could help

Although interrupts are not necessarily the fastest way to control I/O in a microcomputer system, they are often the most practical—especially when several asynchronous external events must be serviced in preference to ongoing calculations.

Interrupt processing itself is, of course, a function of hardware architecture, but it must be supported by special routines in the user's software. To do that efficiently requires detailed knowledge of how an interrupt functions.

Say a peripheral generates an interrupt condition when a character becomes available on the Z80-SIO (serial-I/O) receiver. When the connected peripheral's interrupt-enable input line is high and its internal interrupt circuitry is enabled, it activates the interrupt line of the Z80 CPU.

The processor samples the interrupt line on the last T state of the last machine cycle in every instruction. If the interrupt line (\overline{INT}) is active, the interrupt-enable flip-flop in the Z80 is set and the data-bus request line (\overline{BUSRQ}) is inactive, the CPU acknowledges the interrupt by entering a special M1 cycle called the interrupt-acknowledge cycle. An I/O request is then made during the last T state of this cycle to the device, which is now able to put its vector on the data bus. This vector, together with the I register, forms a 16-bit pointer in the interrupt service routine's starting-address table (ISR-SAT). The Z80 CPU then obtains a 2-byte address from the table and jumps to that address.

But before the interrupt can be processed properly, the user has to prepare the ground:

1. An interrupt "page" must be chosen and the I-register programmed accordingly.
2. The device interrupt vector must be programmed.
3. An entry (or several) must be made in the ISR-SAT.

When the interrupt has been acknowledged, the machine state can be described as follows:

- The user program has been interrupted.
- Control has been passed to the proper interrupt service routine (to which the table entry for the

Two other ways

Besides interrupts, μC -system I/O can be handled by software polling (handshaking) or by direct memory access (DMA).

Software polling is the simplest technique—the CPU is left idle until a peripheral is ready to transfer data. Even if the inefficient processor use is acceptable, transfer rates under 42.3 kbytes/s for the Z80 or 67.8 kbytes/s for the Z80A are only adequate for paper-tape or card readers, and inadequate for tape or disk drives. Unless a separate CPU is dedicated to I/O, the inefficiency usually makes polling unacceptable.

Worse yet, while the CPU waits for one peripheral to get ready, several others may go begging. So when several devices need CPU access according to predetermined priorities, interrupt-controlled I/O is usually preferred—especially when asynchronous external events have to be serviced in preference to some ongoing calculations.

But interrupt servicing also takes time. The Z80 needs up to 40 T cycles to detect and acknowledge the interrupt, in addition to I/O processing. Compared with polling, interrupt-driven I/O can be much slower. While polling can service a single-density floppy that needs data at 31.3 kbytes/s, interrupt control under a Z80 CPU would lead to the loss of some data.

When high speed is critical, DMA takes the prize. By synchronizing a peripheral to the central memory rather than the CPU, the software overhead of both polling and interrupts can be avoided. However, DMA excludes the possibility of any data preprocessing (field masking, character search), and tends to be expensive. Direct memory access is often combined with interrupts; for instance, serial communications can be started via interrupts, and then carried on at high speed under DMA.

Eric A. Benhamou, Software Group Manager, General Systems, Zilog, Inc., 10460 Bubb Rd., Cupertino, CA 95014.

interrupting device is pointing).

- All maskable interrupts are disabled, as they would be after a Disable Interrupts instruction.

- The program then executes the interrupt handler routine. Since the Z80 CPU does not preserve the state of the system automatically when interrupts occur, the interrupt handler must do this job, in one of three ways.

First, registers and flags may be saved on the stack with the following sequence of instructions and T cycles:

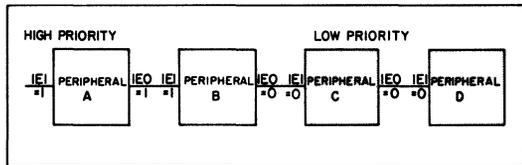
```
PUSH AF 11
PUSH HL 11
PUSH DE 11
PUSH BC 11
PUSH IX 15
PUSH IY 15
Total T cycles: 74
```

When this method is used, a similar sequence of POP instructions restores the stack at the end of the service routine:

```
POP IY 14
POP IX 14
POP BC 10
POP DE 10
POP HL 10
POP AF 10
Total T cycles: 68
```

Second, other RAM locations may be used for the same purpose:

```
LD (ASAVE),A      13
LD (HLSAVE),HL   16
LD (DESAVE),DE   20
LD (BCSAVE),BC   20
LD (IXSAVE),IX   20
LD (IYSAVE),IY   20
Total T cycles: 109
```



1. A daisy-chain mechanism resolves priority conflicts between Z80 peripherals. A device can only interrupt when its IEI line is high. While being serviced, the interrupting device (highlighted) keeps its IEO line low, preventing lower-priority devices (gray) from issuing an interrupt.

The flat register F, however, cannot be saved directly in the RAM method—a serious limitation. Furthermore, a sequence taking 109 T states could be prohibitively long in some applications.

Third, when interrupt-overhead time and program space must both be minimized, the Z80 CPU's alternate register set may be used to save registers and flags. Here, EXX exchanges the contents of the BC, DE and HL registers with the contents of the BC', DE' and HL' registers, respectively, while EX AF, AF' exchanges the contents of AF and AF'.

This operation requires only eight T states, 2 bytes of code and no additional stack space. However, this method is of limited use when multiple interrupts are needed since only one interrupt handler may use the alternate register set; this excludes the use of nested interrupts. The method also assumes that the alternate register set is not already used by the program that is being interrupted.

Since only those registers whose content is destroyed by the interrupt service routine need to be preserved, a combination of the three methods is frequently used.

More interruptions

If successive interrupts are to be allowed, the interrupt handler must explicitly enable them with an Enable Interrupts (EI) instruction. This can be done anywhere in the program after the first interrupt occurs, and before the next one is expected.

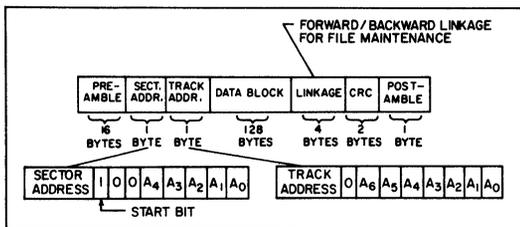
In most cases, however, exactly where in the main program an interrupt occurs is not known. Consequently, it is good practice to enable interrupts in the interrupt handler itself. If the interrupt handler executes an EI instruction immediately, higher-priority devices could interrupt as early as the instruction immediately after EI, producing nested interrupts. Program-timing calculations must take this into account.

When several peripheral devices operate simultaneously in interrupt mode on a Z80 CPU, interrupts from any device may have to be acknowledged within a very short time to avoid data loss or other I/O malfunctions. In such cases, interrupts should be reenabled as early as possible in the interrupt handler to minimize the interval during which real-time events will not be recognized. This time will not exceed the value T_{lim} , which is defined as follows:

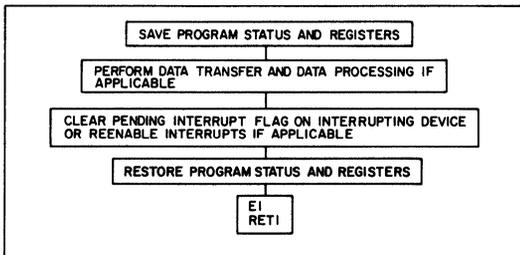
$$T_{lim} = \text{Interrupt-acknowledge time} \\ + \text{instruction time for EI} \\ + \text{instruction time for following instruction}$$

The T_{lim} can be minimized by forcing the instruction following EI to be a NOP or any other instruction of four T states.

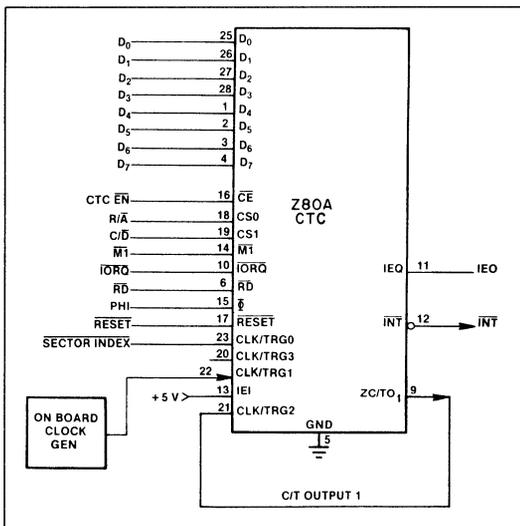
When the programmer allows nested interrupts to occur, he must remember that only one service routine may use the alternate register set for preserving the interrupted program state. If all the interrupt handlers save registers on the stack, he must verify that



2. All interrupt routines start by saving program status, which they restore at the end. They terminate with Enable Interrupt (EI) and Return from Interrupt (RETI) instructions—what happens in between depends largely on the specific application.



3. Sectors on a floppy disk contain not only data (highlighted) but also pre and postambles, as well as control information.



4. A Z80A CTC chip generates interrupts for a disk controller. Its channel 0 counts index or sector pulses (pin 23), while the cascaded channels 1 and 2 provide a wide timing range.

the stack area is large enough to prevent overflow.

If nested interrupts are not desirable, the EI instruction is usually inserted at the end of the interrupt service routine, immediately before the Return from Interrupt (RETI) instruction. Any pending interrupt will then be delayed until the end of the interrupt handler, that is, until RETI has been executed.

Watch for the daisy chain

The interrupt handler must terminate with RETI, which is decoded by the interrupting device and allows the device to leave the interrupt state. In particular, it reactivates the lower daisy chain by resetting lines IEI (Interrupt Enable In) and IEO (Interrupt Enable Out) so that interrupts from lower-priority devices can be recognized (Fig. 1).

Keep in mind that EI and RETI do different things. The EI instruction does a general interrupt enable on the Z80 CPU for maskable interrupts. This does not imply, however, that all Z80 peripheral devices are allowed to interrupt—only those that have an active IEI line. In an interrupt-service routine, therefore, only devices higher in the chain than the interrupting device may, in turn, interrupt after EI and before RETI.

The RETI instruction is the only way to reestablish the interrupt daisy chain, below the interrupting device, by software. By resetting the interrupt-under-service flip-flop inside the peripheral, RETI affects the state of the interrupting device, while EI affects the state of the Z80 CPU.

Fig. 2 shows a generalized interrupt routine that summarizes the discussed steps. But the flow chart should not be regarded as a rigid model—interrupt handlers can differ widely from one application to another.

Making tracks—carefully

A hard-sectored floppy-disk driver provides many opportunities for interrupt-service routines: Sector count, head-load-delay time-out and track stepping in particular require interrupt techniques to achieve acceptable performance levels.

One problem stems from slight head misalignments, which cause the physical location of sector data to vary slightly with respect to the sector hole. The impact on data readability may be severe because flexible diskettes are likely to be read on several drives with different alignment characteristics.

For example, if data are written “early” with respect to the sector hole, the read gate may be turned on too late to catch the start bit, and synchronization will occur on the first ONE bit in the data field. A sector-address error or a CRC error are typical results. If data are written “late” with respect to the sector hole, the last bytes of the previous sector may still be under the read head when the read gate is turned on.

The best point to turn on the read gate is at the

```

;PROGRAM CTC TO REQUIRED MODE OF OPERATION
.
.
LD A,13H      ; SET Z80 INTERRUPT REGISTER
LD I,A
LD HL,INT0    ; LOAD INTRPT ROUT ADDRESS
LD (CTCOIV),HL ; IN INTERRUPT TABLE
LD A, (CTCOIV.SHL.8)/256
OUT (CTCO), A ; PROGRAM CTC INTRPT VECTOR
LD A, (COUNT)
DEC A         ; PROGRAM CTC CHAN 0 TO INTRPT
              ; ON REQUESTED SECTOR - 1

LD BC,CTRMOD.SHL.8+CTCO
OUT (C), B   ; SEND OUT MODE-CTRL BYTE
OUT (C), A   ; SEND OUT COUNT
.
.
INT0:  PUSH AF      ; SAVE REGISTERS
        PUSH BC
        LD A,3      ; RESET CTC CHANNEL 0
        OUT (CTCO), A
        ;DELAY 4.5 MS OR .9 SECTOR TIME
        LD HL,INTL ; LOAD INTRPT ROUTINE ADDRESS
        LD (CTC2IV), HL ; IN INTERRUPT TABLE

        ; PROGRAM CTC1 IN COUNTER MODE, NO INTRPTS
        LD BC, CNTMOD.SHL.8+CTC1
        OUT (C),B
        LD B,TIME32 ; ZERO-COUNT PERIOD = 32US

        ; PROGRAM CTC2 IN COUNT MODE WITH INTRPTS
        LD BC,CTRMOD.SHL.8+CTC2
        OUT (C),B
        LD A,DELAY1 ; ZERO-COUNT REACHED
        OUT (C),A   ; AFTER 4.5 MS

        POP BC
        POP AF     ; RESTORE REGISTERS
        EI
        RETI

INT1:  PUSH AF      ; SAVE REGISTER
        PUSH BC
        PUSH DE
        PUSH HL
        LD A,3
        OUT (CTC2) ; RESET CTC CHANNEL 2

        LD HL,INT2 ; LOAD INTRPT ROUT ADDRESS
        LD (CTC2IV),HL ; IN INTERRUPT TABLE

        ; PREPARE TO SET UP CTC CHANNELS 1 AND 2
        LD C,CTC1
        LD DE,CNTMOD.SHL.8+TIME32
        LD HL,CTRMOD.SHL.8+DELAY2

        ; DETECT REQUESTED SECTOR HOLE

INT11: IN A, (PIOB) ; INPUT STATUS
        BIT SECTOR,A ; TEST SECTOR + INDEX LINE
        JR NZ,INT11 ; LOOP UNTIL SECTOR FOUND

        ; START DELAY TO MIDDLE OF PREAMBLE
        OUT (C),D
        OUT (C),E ; ENABLE CHANNEL 1

        LD C,CTC2 ; ENABLE CHANNEL 2
        OUT (C),H
        OUT (C),L

        POP HL ; RESTORE REGISTERS
        POP DE
        POP AF
        EI
        RETI

INT2:  PUSH AF      ; SAVE REGISTERS
        PUSH BC
        LD A,3
        OUT (CTC2),A
        .
        .
CNTMOD EQU 47H
CTRMOD EQU 0C7H
TIME32 EQU 16H
DELAY1 EQU 8EH

```

5. Several routines are needed to handle the sector interrupts via Counter-Timer-Circuit (CTS) programming: Routine INTO generates the time delays; Routine INT1 prepares the CTC; Routine INT11 handles the polling phase; and INT2 resets the CTC.

middle of the preamble (Fig. 3)—and this point must be found with good accuracy.

A Z80 CTC (counter-timer circuit) may be used to generate floppy-disk controller interrupts. In this application, three of its four programmable counters are needed.

Counter channel 1 is driven by a 1.4056- μ s periodic signal (derived from a standard 11.3828-MHz crystal oscillator, divided by 16) that is programmed to count 22 (=16H) such pulses. The counter, therefore, reaches a full count every 32 μ s, which corresponds to the length of 1 byte.

The second counter is driven in cascade by the first counter's output (pin 9) and programmed to generate an interrupt after a varying number of byte units on pin 12 (line \overline{INT}).

The sector pulse line from the floppy drive is connected to the counter input of CTC channel 0 (pin 23) to generate an interrupt on any desired sector. The sector pulse line is also connected to a Z80 PIO bit on this same controller so that the presence of a sector hole may be detected by a simple port read followed by a bit test.

Several routines are used in the sector-read operation of a typical hard-sectoring floppy-disk driver. The program (Fig. 5) assumes that the number of holes between the current sector and the requested sector has been computed in a previous routine, the result being held in a variable named COUNT.

The interrupt routines generate the proper delay to look for the sector hole—not of the sector to be read, but the one before. Then the program adds 0.9 times the length of a sector and, at that moment, switches from an interrupt to a polling technique. This approach assures that the CPU can execute other tasks as long as possible, without running the risk of missing a sector hole.

It would have been simpler to program the Z80 CTC to interrupt on the requested sector hole rather than on the previous one. In that case, however, the interrupt overhead delay, added to the time required to preserve registers and flags and to set up the Z80 CTC time delay, would be in the order of 186 T states (Fig. 6).

If the controller must operate both on a 2.5-MHz (Z80) and a 4-MHz (Z80A) processor, the described method becomes unacceptable because the variable overhead time between the moment the sector pulse occurs and the moment the preamble-delay countdown starts. Instead, the sector hole is detected by software polling, which only introduces a CPU clock-dependent delay of 86 to 101 T states (Fig. 7). The polling loop, however, is entered about 4.5 ms after the previous hole has been encountered, to return control to the main program for as long as possible.

Polling and interrupt techniques can be combined in this case because sector holes are regularly spaced on a disk. Knowing when one hole flies by is, in theory, enough to predict the start of any following sector; in practice, however, speed variation, electrical delays

<u>T-STATES</u>	<u>FUNCTION</u>
21	Max. Interrupt detection delay
19	(=longest Z-80 CPU instruction time) Interrupt acknowledge delay
11	PUSH AF ; SAVE FLAGS,REGISTERS
11	PUSH HL
11	PUSH DE
11	PUSH BC
20	PUSH IX
7	LD C,CTC1
10	LD DE,CNTMOD.SHL.8+TIME32
10	LD HL,CTRMOD.SHL.8+DELAY2
12	OUT (C),D
12	OUT (C),E ; ENABLE CHANNEL 1
7	LD C,CTC2 ; ENABLE CHANNEL 2
12	OUT (C),H
12	OUT (C),L
186	Total number of T states

6. Timing analysis shows that detecting sectors purely by interrupts requires 186 T states—too much for a CPU-clock-independent floppy controller.

<u>T STATES</u>	<u>FUNCTION</u>
11	INT11: IN A,(PIOB) ; SEE FIGURE 4
8	BIT SECTOR,A
12 T027	JR NZ,INT11 ; SEE NOTE
12	OUT (C),D
12	OUT (C),E ; ENABLE CHANNEL 1
7	LD C,CTC2 ; ENABLE CHANNEL 2
12	OUT (C),H
12	OUT (C),L
86 TO 101	Total number of T states

NOTE: If sector pulse occurs after execution of input instruction, loop is entered up to 15 T states later.

7. Detecting sectors by polling takes less time; the exact number of T states depends on the timing of the sector pulse with respect to program execution.

and mechanical delays introduce too many uncertainties. Still, the approximate knowledge of the requested hole's occurrence permits the calling program to utilize additional CPU time before the polling loop for sector-hole sensing takes over. This provides the preamble delay-time count with an accurate starting point.

The presence of one index hole on the disk, whose position with respect to the first encountered hole is not known, requires special attention. When the Z80 CTC interrupts, its internal counter is reloaded and remains enabled, and any pulse on the Z80 CTC input line will decrement the counter, even though the device may not have left the interrupt state. To avoid spurious interrupts from the index hole, the Z80 CTC is always reset in all Z80 CTC interrupt-service routines, even though this may not always be necessary.■

How useful?

Immediate design application
Within the next year
Not applicable

Circle No.

550
551
552

Z8000™ 16-Bit Microprocessor Family 3

Z8000
Z8001
Z8002
Z8003
Zilog
Zilog

Advanced Architectural Features of the Z8000 CPU



Tutorial Information

March 1981

Introduction The Zilog Z8000 CPU microprocessor is a major advance in microcomputer architecture. It offers many minicomputer and mainframe features for the first time in a microprocessor chip. This tutorial describes the Z8000 CPU with emphasis placed on those features that set it apart from its microprocessor predecessors. For a detailed description of all Z8000 CPU features, consult the Zilog publications listed in the bibliography at the end of this tutorial.

The features to be discussed are grouped into four areas: CPU organization, handling of interrupts and traps, use of memory, and new

instructions and data capabilities.

Before discussing these features in more detail, a word about nomenclature is in order. The term Z8000 refers to the concept and architecture of a family of parts. Zilog has adopted the typical conductor industry 4-digit designation for Z8000 Family parts, while also keeping the traditional 3-letter acronym that proved so popular for the Z-80 Family. Thus, the 48-pin version of the Z8000 CPU is called the Z8001 CPU; the 40-pin version is known as the Z8002 CPU.

CPU Organization The Z8000 CPU is organized around a general-purpose register file (Figure 1). The register file is a group of registers, any one of which can be used as an accumulator, index register, memory pointer, stack pointer, etc. The only exception is Register 0, as explained later.

Flexibility is the major advantage of a general-purpose register organization over an organization that dedicates particular registers to each function. Computation-oriented routines can use general registers as accumulators for intermediate results whereas data manipulation routines can use these registers for memory pointers.

Dedicated registers, however, have a disadvantage: when more registers of a given type are needed than are supplied by the machine, the performance degrades by the extra instructions to swap registers and memory locations. For example, a processor with two index registers suffers when three are needed because a temporary variable in memory (or in another register) must be used for the third

index. When the third index is needed, it must be swapped into an index register. In contrast, on a general-register machine three of the registers could be dedicated for index use. In addition, since the need for index registers may vary over the course of a program, a general-register architecture, such as the Z8000, can be adapted to the changing needs of the computation with respect to the number of accumulators, memory pointers and index registers. Thus flexibility results in increased performance and ease of use.

In addition, the registers of the Z8000 are organized to process 8-bit bytes, 16-bit words, 32-bit long words and 64-bit quadruple words. This readily accommodates applications that process data of variable sizes as well as different tasks that require different data sizes.

Although all registers can—in general—be used for any purpose, certain instructions such as Subroutine Call and String Translation make use of specific registers in the general register file, and this must be taken into account when these instructions are used.

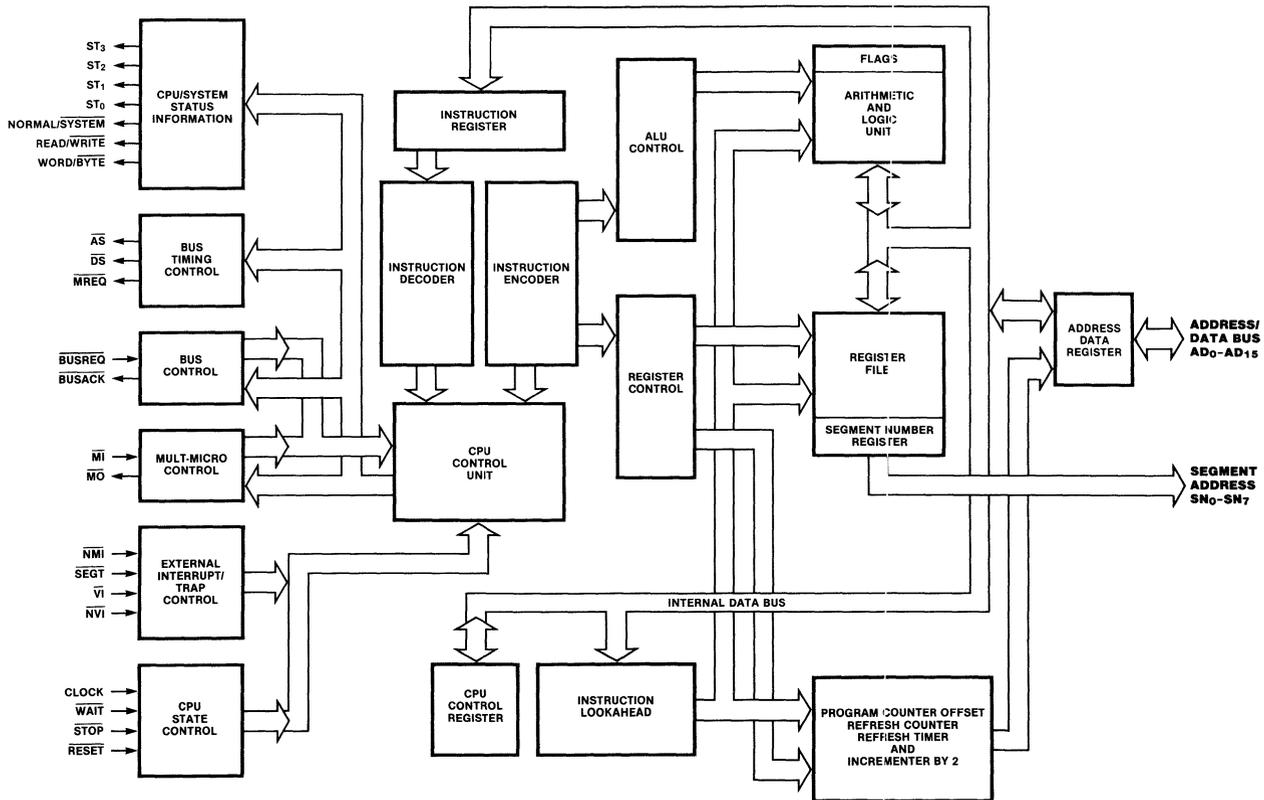


Figure 1. CPU Organization

CPU Organization
(Continued)

The Z8000 CPU also contains a number of special-purpose registers in addition to the general-purpose ones. These include the Program Counter, Program Status registers and

the Refresh Counter. These registers are accessible through software and provide some of the interesting features of Z8000 CPU architecture.

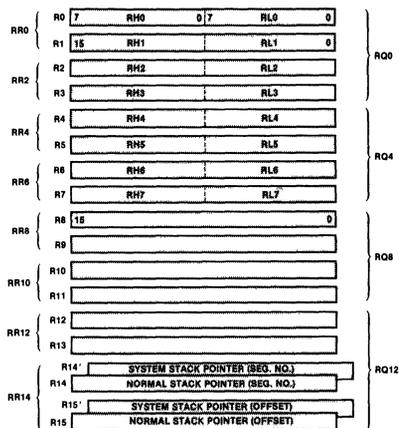


Figure 2. Z8001 General Purpose Registers

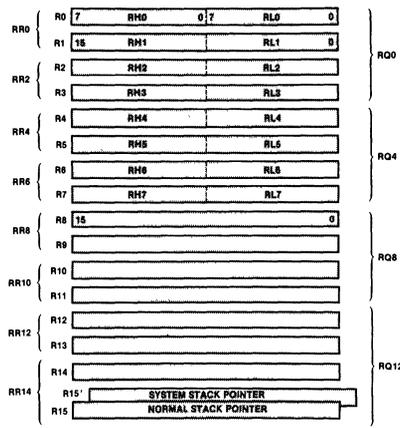


Figure 3. Z8002 General Purpose Registers

Register Organization

All general-purpose registers can be used as accumulators, and all but one as index registers or memory pointers. The one register that cannot be used as an index register is Register 0. Specifying Register 0 is used as an escape mechanism to change the address mode from IR to IM, from X to DA, or—with Load instructions—from BA to RA. This has been done so that the two addressing mode bits in the instruction can specify more than four addressing modes for the same opcode.

The Z8000 CPU register file can be addressed in several groupings: as sixteen byte registers (occupying the upper half of the file only), as sixteen word registers, as eight long-word registers, as four quadruple-word registers, or as a mixture of these. Instructions either explicitly or implicitly specify the type of register. Table 1 illustrates the correspondence between the 4-bit source and destination register fields in the instruction (Figure 4) and the location of the registers in the register file (Figures 2 and 3).

Register Designator	Byte	Word	Long Word	Quadruple Word
0 0 0 0	RH0	R0	RR0	RQ0
0 0 0 1	RH1	R1		
0 0 1 0	RH2	R2	RR2	
0 0 1 1	RH3	R3		
0 1 0 0	RH4	R4	RR4	RQ4
0 1 0 1	RH5	R5		
0 1 1 0	RH6	R6	RR6	
0 1 1 1	RH7	R7		
1 0 0 0	RL0	R8	RR8	RQ8
1 0 0 1	RL1	R9		
1 0 1 0	RL2	R10	RR10	
1 0 1 1	RL3	R11		
1 1 0 0	RL4	R12	RR12	RQ12
1 1 0 1	RL5	R13		
1 1 1 0	RL6	R14	RR14	
1 1 1 1	RL7	R15		

Table 1

Register Organization
(Continued)

Note that the byte register-addressing sequence (most significant bit distinguishes between the two bytes in a word register) is different from the memory addressing sequence (least significant bit distinguishes between the two bytes in a word). Long-word (32-bit) and quadruple-word (64-bit) registers are addressed by the binary number of their starting word registers (most significant word). For example, RR6 is addressed by a binary 6 and occupies word registers 6 and 7.

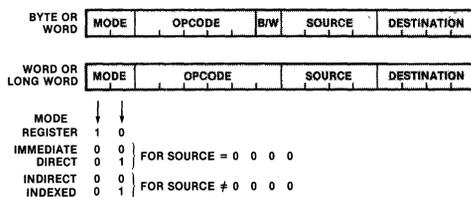


Figure 4. Instruction Format

System/Normal Mode of Operation

The Z8000 CPU can run in one of two modes: System or Normal. In System Mode, all of the instructions can be executed and all of the CPU registers can be accessed. This mode is intended for use by programs that perform operating system type functions. In Normal Mode, some instructions, such as I/O instructions, are not all allowed, and the control registers of the CPU are inaccessible. In general, this mode of operation is intended for use by application programs. This separation of CPU resources promotes the integrity of the system since programs operating in Normal Mode cannot access those aspects of the CPU which deal with time-dependent or system interface events.

Normal Mode programs that have errors can always reproduce those errors for debugging purposes by simply re-executing the programs with their original data. Programs using facilities available only in System Mode may have errors due to timing considerations (e.g.,

based on the frequency of disk requests and disk arm position) that are harder to debug because these errors are not easily reproduced. Thus a preferred method of program development would be to partition the task into that portion which can be performed without recourse to resources accessible only in System Mode (which will usually be the bulk of the task) and that portion requiring System Mode resources. The classic example of this partitioning comes from current minicomputer and mainframe systems: the operating system runs in System Mode and the individual users write their programs to run in Normal Mode.

To further support the System/Normal Mode dichotomy, there are two copies of the stack pointer—one for the System Mode and another for Normal. Although the stacks are separated, it is possible to access the normal stack registers while in the System Mode by using the LDCTL instruction.

Status Lines

The Z8000 CPU outputs status information over its four status lines (ST₀-ST₃) and the System/Normal line (S/N). This information can be used to extend the addressing range or to protect accesses to certain portions of memory. The types of status information and their codes are listed in Table 2.

Status conditions are mutually exclusive and can, therefore, be encoded without penalty. Most status definitions are self-explanatory. One code is reserved for future enhancements of the Z8000 Family.

Extension of the addressing range is accomplished in a Z8000 system by allocating physical memory to specific usage (program vs. data space, for example) and using external circuitry to monitor the status lines and select the appropriate memory space for each address. For example, the direct addressing range of the Z8002 CPU is limited to 64K bytes; however, a system can be configured

with 128K bytes if additional logic is used, say, to select the lower 64K bytes for program references and the upper 64K bytes for data references.

ST ₃ -ST ₀	Definition
0 0 0 0	Internal operation
0 0 0 1	Memory refresh
0 0 1 0	I/O reference
0 0 1 1	Special I/O reference
0 1 0 0	Segment trap acknowledge
0 1 0 1	Non-maskable interrupt acknowledge
0 1 1 0	Non-vectored interrupt acknowledge
0 1 1 1	Vectored interrupt acknowledge
1 0 0 0	Data memory request
1 0 0 1	Stack memory request
1 0 1 0	Data memory request (EPU)
1 0 1 1	Stack memory request (EPU)
1 1 0 0	Instruction space access
1 1 0 1	Instruction fetch, first word
1 1 1 0	Extension processor transfer
1 1 1 1	Reserved

Table 2

Status Lines
(Continued)

Protection of memory by access types is accomplished similarly. The memory is divided into blocks of locations and associated with each block is a set of legal status signals. For each access to the memory, the external circuit checks whether the CPU status is appropriate for the memory reference. The Z8010 Memory Management Unit is an example of an external memory-protection circuit, and it is discussed later in this tutorial.

The first word in an instruction fetch has its

Refresh

The idea of incorporating the Refresh Counter in the CPU was pioneered by the Z-80 CPU, which performs a refresh access in a normally unused time slot after each opcode fetch. The Z8000 is more straightforward (each refresh has its own memory-access time slot of three clock cycles), and is more versatile (the refresh rate is programmable and capable of being disabled altogether).

The Refresh Register contains a 9-bit Row Counter, a 6-bit Rate Counter and an Enable Bit (Figure 5). The row section is output on AD_0 - AD_8 during a refresh cycle. The Z8000 CPU uses word-organized memory, wherein A_0 is only employed to distinguish between the lower and upper bytes within a word during reading or writing bytes. A_0 therefore plays no role in refresh—it is always 0. The Row Counter is—at least conceptually—always incremented by two whenever the rate counter passes through zero. The Row Counter cycles through 256 addresses on lines AD_1 - AD_8 , which satisfies older and current 64- and 128-row addressing schemes, and can also be used with 256-row refresh schemes for 64K RAMs.

The Rate Counter determines the time between successive refreshes. It consists of a programmable 6-bit modulo-n prescaler

Instruction Prefetch (Pipelining)

Most instructions conclude with two or three clock cycles being devoted to internal CPU operations. For such instructions, the subsequent instruction-fetch machine cycle is overlapped with the concluding operations, thereby improving performance by two or three clock cycles per instruction.

Examples of instructions for which the subsequent instruction is fetched while they complete are Arithmetic and Shift instructions.

own dedicated status code, namely 1101. This allows the synchronization of external circuits to the CPU. During all subsequent fetch cycles within the same instruction (remember, the longest instruction requires a total of four word fetches), the status is changed from 1101 to 1100. Load Relative and Store Relative also have a status of 1100 with the data reference, so information can be moved from program space to data space.

($n = 1$ to 64), driven at one-fourth the CPU clock rate. The refresh period can be programmed from 1 to 64 μ s with a 4 MHz clock. A value of zero in the counter field indicates the maximum time between refreshes; a value of n indicates that refresh is to be performed every 4n clock cycles. Refresh can be disabled by programming the Refresh Enable Bit to be zero.

A memory refresh occurs as soon as possible after the indicated time has elapsed. Generally, this means after the T_3 clock cycle of an instruction if an instruction execution has commenced. When the CPU does not have control of the bus (during the bus-request/bus-acknowledge sequence, for example), it cannot issue refresh commands. Instead, it has internal circuitry to record "missed" refreshes; when the CPU regains control of the bus it immediately issues the "missed" refresh cycles. The Z8001 and Z8002 CPU can record up to two "missed" refresh cycles.

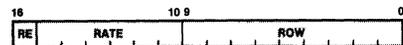


Figure 5. Refresh Counter

Some instructions for which the overlap is logically impossible are the Jump instructions (because the following instruction location has not been determined until the instruction completes). Some instructions for which overlap is physically impossible are the Memory Load instructions (because the memory is busy with the current instruction and cannot service the fetch of the succeeding instruction).

**Extended
Instruction
Facility**

The Z8000 architecture has a mechanism for extending the basic instruction set through the use of external devices. Special opcodes have been set aside to implement this feature. When the CPU encounters instructions with these opcodes in its instruction stream, it will perform any indicated address calculation and data transfer, but otherwise treat the "extended instruction" as being executed by the external device. Fields have been set aside in these extended instructions which can be interpreted by external devices (called Extended Processing Units—EPU's) as opcodes. Thus by using appropriate EPU's, the instruction set of the Z8000 can be extended to include specialized instructions.

In general, an EPU is dedicated to performing complex and time consuming tasks in order to unburden the CPU. Typical tasks suitable for specialized EPU's include floating-point arithmetic, data base search and maintenance operations, network interfaces, graphics support operations—a complete list would include most areas of computing. EPU's are generally designed to perform their tasks on data resident in their internal registers. Moving information into and out of the EPU's internal registers, as well as instructing the EPU as to what operations are to be performed, is the responsibility of the CPU.

For the Z8000 CPU, control of the EPU's takes the following form. The Z8000 CPU fetches instructions, calculates the addresses of operands residing in memory, and controls the movement of data to and from memory. An EPU monitors this activity on the CPU's AD lines. If the instructions fetched by the CPU are extended instructions, all EPU's and the CPU latch the instruction (there may be several different EPU's controlled by one CPU). If the instruction is to be executed by a particular EPU, both the CPU and the indicated EPU will be involved in executing the instruction.

If the extended instruction indicates a transfer of data between the EPU's internal registers and the main memory, the CPU will calculate the memory address and generate the appropriate timing signals (AS, DS, MREQ, etc.), but the data transfer itself is between the memory and the EPU (over the

AD lines). If a transfer of data between the CPU and EPU is indicated, the sender places the data on the AD lines and the receiver reads the AD lines during the next clock period.

If the extended instruction indicates an internal operation to be performed by the EPU, the EPU begins execution of that task and the CPU is free to continue on to the next instruction. Processing then proceeds simultaneously on both the CPU and the EPU until a second extended instruction is encountered that is destined for the same EPU (if more than one EPU is in the system, all can be operating simultaneously and independently). If an extended instruction specifies an EPU still executing a previous extended instruction, the EPU can suspend instruction fetching by the Z8000 CPU until it is ready to accept the next extended instruction: the mechanism for this is the STOP line, which suspends CPU activity during the instruction fetch cycle.

There are four types of extended instructions in the Z8000 CPU instruction repertoire: EPU internal operations; data transfers between memory and EPU; data transfers between EPU and CPU; and data transfer between EPU flag registers and CPU flag and control word. The last type is useful when the program must branch based on conditions determined by the EPU. Six opcodes are dedicated to extended instructions: 0E, 0F, 4E, 4F, 8E and 8F (in hexadecimal). The action taken by the CPU upon encountering these instructions is dependent upon an EPU control bit in the CPU's FCW. When this bit is set, it indicates that the system configuration includes EPU's; therefore, the instruction is executed. If this bit is clear, the CPU traps (extended instruction trap), so that a trap handler in software can emulate the desired operation.

In conclusion, the major features of this capability are, that multiple EPU's can be operating in parallel with the CPU, that the five main CPU addressing modes (Register, Immediate, Indirect Register, Direct Address, Indexed) are available in accessing data for the EPU; that each EPU can have more than 256 different instructions; and that data types manipulated by extended instructions can be up to 16 words long.

Program Status Information

The Program Status Information consists of the Flag And Control Word (FCW) and the Program Counter (PC). The Z8000 CPU uses one byte in FCW to store flags and another byte to store control bits.

Arithmetic Flags. Flags occupy the low byte in the FCW and are loaded, read, set and reset by the special instruction LDCTLB, RESFLG and SETFLG. The flags are:

- C** Carry
- Z** Zero
- S** Sign (1 = negative; two's complement notation is used for all arithmetic on data elements)
- P/V** Even Parity or Overflow (the same bit is shared)
- D** Decimal Adjust (differentiates between addition and subtraction)
- H** Half Carry (from the low-order nibble)

Interrupt and Trap Structure

The Z8000 provides a powerful interrupt and trap structure. Interrupts are external asynchronous events requiring CPU attention, and are generally triggered by peripherals needing service. Traps are synchronous events resulting from the execution of certain instructions. Both are processed in a similar manner by the CPU.

The CPU supports three types of interrupts

Control Bits. The control bits occupy the upper byte in the FCW. They are loaded and read by the LDCTL instruction, which is privileged in that it can be executed only in the System Mode. The control bits are:

- NVIE** Non-Vectored Interrupt Enable
- VIE** Vectored Interrupt Enable
- S/N** System or Normal Mode
- SEG** Segmented Mode Enable (Z8001 only)

The SEG bit is always 0 in the Z8002 even if the programmer attempts to set it. In the Z8001, a 1 in this bit indicates segmented operation. A 0 in the Z8001 SEG bit forces non-segmented operation and the CPU interprets all code as non-segmented. Thus, the Z8001 can execute modules of user code developed for the non-segmented Z8002.

(non-maskable, vectored and non-vectored), three internal traps (system call, unimplemented instruction, privileged instruction) and a segmentation trap. The vectored and non-vectored interrupts are maskable.

The descending order of priority for traps and interrupts is: internal traps, non-maskable interrupts, segmentation trap, vectored interrupts and non-vectored interrupts.

Effects of Interrupts on Program Status

The Flag and Control Word and the Program Counter are collectively called the *Program Status Information*—a useful grouping because both the FCW and PC are affected by interrupts and traps. When an interrupt or trap occurs, the CPU automatically switches to the System Mode and saves the Program Status plus an identifier word on the system stack. The identifier supplies the reason for the interrupt. (The Z8002 pushes three words on the stack; the Z8001 pushes four words.)

After the pre-interrupt or "old" Program Status has been stored, the "new" Program Status is automatically loaded into the FCW and PC. This new Program Status Information is obtained from a specified location in memory, called the Program Status Area.

The Z8000 CPU allows the location of the Program Status Area anywhere in the addressable memory space, although it must be aligned to a 256-byte boundary. Because the Status Line code is 1100 (program reference) when the new Program Status is loaded, the Program Status must be located in program memory space if the memory uses this attribute (for example, when using the Z8010 Memory Management Unit or when separate memory modules are used for program and for data).

The Program Status Area Pointer (PSAP) specifies the beginning of the Program Status Area. In the Z8002, the PSAP is stored in one word, the lower byte of which is zero. The Z8001, however, stores its PSAP in two words. The first contains the segment number and the second contains the offset, the lower byte of which is again zero. The PSAP is loaded and read by the LDCTL instruction.

In the Z8002, the first 14 words (28 bytes) of the Program Status Area contain the Program Status Information for the following interrupt conditions:

Location (In Bytes)	Condition
0-3	Not used (reserved for future use)
4-7	Unimplemented instruction has been fetched, causing a trap
8-11	Privileged instruction has been fetched in Normal Mode, causing a trap
12-15	System Call instruction
16-19	Not used
20-23	Non-maskable interrupt
24-27	Non-vectored interrupt

Effects of Interrupts on Program Status
(Continued)

Bytes 28-29 contain the FCW that is common to all vectored interrupts. Subsequent locations contain the vector jump table (new PC for vectored interrupts). These locations are addressed in the following way: the 8-bit vector that the interrupting device has put on the lower byte of the Address/Data bus (AD_0-AD_7) is doubled and added to $PSAP + 30$. Thus,

- Vector 0 addresses $PSAP + 30$,
- Vector 1 addresses $PSAP + 32$, and
- Vector 255 addresses $PSAP + 540$.

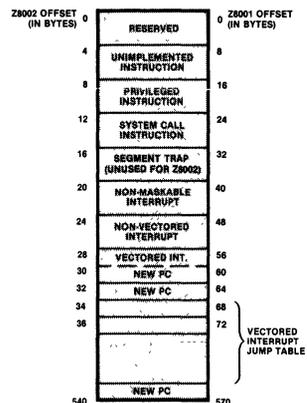
In the segmented Z8001, the first 28 words of the Program Status Area (56 bytes) contain the Program Status Information (reserved word, FCW, segment number, offset), for the following interrupt conditions:

Location (In bytes)	Condition
0-7	Not used (reserved for future use)
8-15	Unimplemented instruction has been fetched causing a trap
16-23	Privileged instruction has been fetched in Normal Mode causing a trap
24-31	System Call instruction
32-39	Segmentation trap (memory violation detected by the Z8010 Memory Management Unit)
40-47	Non-maskable interrupt
48-55	Non-vectored interrupt

Bytes 56-59 contain the reserved word and FCW common to all vectored interrupts. Subsequent locations contain the vector jump table (the new segment number and offset for all vectored interrupts). These locations are addressed in the following way: the 8-bit vector that the interrupting device has put on the lower byte of the Address/Data bus (AD_0-AD_7) is doubled and added to $PSAP + 60$. Thus,

- Vector 0 addresses $PSAP + 60$,
- Vector 2 addresses $PSAP + 64$, and
- Vector 254 addresses $PSAP + 568$.

Care must be exercised in allocating vector locations to interrupting devices; always use even vectors. Thus there are effectively only 128 entries in the vector jump table. (Figure 6 illustrates the Program Status Area.)



**Z8000 CPU
Memory
Features**

The way a processor addresses and manages its memory is an important aspect in both the evaluation of the processor and the design of a computer system that uses the processor. Z8000 architecture provides a consistent memory address notation in combining bytes into words and words into long words. All three data types are supported for operands in the Z8000 instruction set. I/O data can be either byte- or word-oriented.

The Z8001 CPU provides a segmented addressing space with 23-bit addressing. The Z8010 Memory Management Unit can increase the address range of this processor. To support a memory management system, the Z8001 processor generates Processor Status Information.

**Address
Notation**

In the Z8000 CPU, memory and I/O addresses are always byte addresses. Words or long words are addressed by the address of their most significant byte (Figure 7). Words always start on even addresses ($A_0 = 0$), so both bytes of a word can be accessed simultaneously. Long words also start on even addresses.

Within a word, the upper (or more significant) byte is addressed by the lower (and always even) address. Similarly, within a long word, the upper (more significant) word is addressed by the lower address. Note that this format differs from the PDP-11 but is identical to the IBM convention.

There is good reason for choosing this format. Because the Z8000 CPU can operate on 32-bit long words and also on byte and word strings, it is important to maintain a continuity of order when words are concatenated into long words and strings. Making ascending addresses proceed from the highest byte of the first word to the lowest byte of the last word maintains this continuity, and allows compar-

ing and sorting of byte and word strings. These signals are also generated by the Z8002 CPU and—as mentioned earlier—can be used to increase the address range of this processor beyond its nominal 64K byte limit. It is not necessary to use a Z8010 Memory Management Unit with a Z8001. The segment number (upper six bits of the address) can be used directly by the memory system as part of the absolute address.

These issues are discussed in more detail in the following sections, along with a description of the method used to encode certain segmented addresses into one word. A brief comment on the use of 16K Dynamic RAMs with the Z8001 concludes this group of sections that deal with Z8000 CPU memory features.

ing and sorting of byte and word strings.

Bit labeling within a byte does not follow this order. The least significant bit in a byte, word or long word is called Bit 0 and occurs in the byte with the highest memory address. This is consistent with the convention where bit n corresponds to position 2^n in the conventional binary notation. This ordering of bit numbers is also followed in the registers.

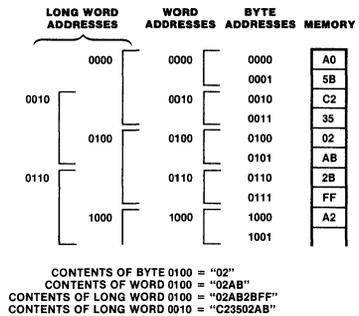


Figure 7. Memory Addressing

Memory and I/O Addressing

Like most 16-bit microprocessors, the Z8000 CPU uses a 16-bit parallel data bus between the CPU and memory or I/O. The CPU is capable of reading or writing a 16-bit word with every access. Words are always addressed with even addresses ($A_0 = 0$). All instructions are words or multiple words.

The Z8000 CPU can, however, also read and write 8-bit bytes, so memory and I/O addresses are always expressed in bytes. The Byte/Word (B/\bar{W}) output indicates whether a byte or word is addressed (High = byte). A_0 distinguishes between the upper and lower byte in memory or I/O. The most significant byte of the word is addressed when A_0 is Low (Figure 8).

For word operations in both the read and write modes, $B/\bar{W} = \text{Low}$, A_0 is simply

ignored and A_1 - A_{15} address the memory or I/O. For byte operations in the read mode, $B/\bar{W} = \text{High}$, A_0 is again ignored, and a whole word (both bytes) is read, but the CPU internally selects the appropriate byte. For byte operations in the write mode, the CPU outputs identical information on both the Low (AD_0 - AD_7) and the High (AD_8 - AD_{15}) bytes of the Address/Data bus. External TTL logic must be used to enable writing in one memory byte and disable writing in the other byte, as defined by A_0 . The replication of byte information for writes is for the current implementation and may change for subsequent Z8000 CPUs; therefore system designs should not depend upon this feature.

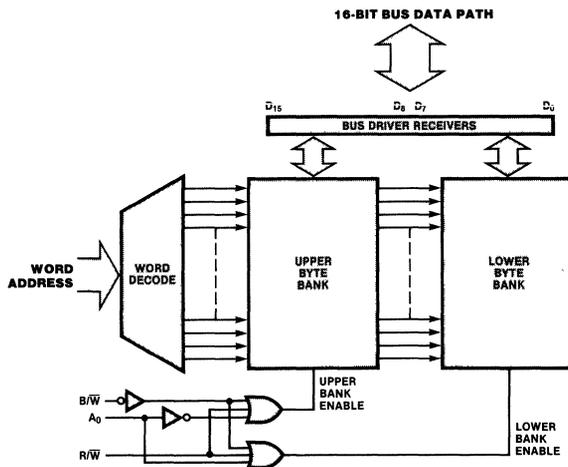


Figure 8. Byte/Word Selection

Segmentation

In organizing memory, segmentation is a powerful and useful technique because it forms a natural way of dividing an address space into different functional areas. A program typically partitions its available memory into disjointed areas for particular uses. Examples of this are storing the procedure instructions, holding its global variables, or serving as a buffer area for processing large, disk-resident data bases. The requirements for these different areas may differ, and the areas themselves may be needed only part of the time.

Segmentation reflects this use of memory by allowing a user to employ a different segment for each different area. A memory management system can then be employed to provide system support, such as swapping segments from disk to primary memory as requested (as in overlays), or in monitoring memory accesses and allowing only certain types of accesses to

a particular segment. Thus, dealing with segments is a convenient way of specifying portions of a large address space.

When segmentation is combined with an address translation mechanism to provide relocation capability, the advantages of segmentation are enhanced. Now segments can be of variable user-specifiable sizes and located anywhere in memory.

The Z8001 generates 23-bit logical addresses, consisting of a 7-bit segment number and a 16-bit offset. Thus each of its six memory address spaces consists of 128 segments, and each segment can be up to 64K bytes. Different routines of a program can reside in different segments, and different data sets can reside in different segments. The Z8010 Memory Management Unit translates these logical addresses into physical-memory locations.

Long Offset and Short Offset Addressing

When a segmented address is stored in memory or in a register, it occupies two 16-bit words as previously described for the PC and PSAP. This is a consequence of the large addressing range. When a segmented address is part of an instruction in the Direct Address and Indexed Address Modes, there are two representations: Long and Short Offset addressing.

In the general unrestricted case of Long Offset, the segmented address occupies two words, as described before. The most significant bit in the segment word is a 1 in this case.

The Short Offset Mode squeezes the segment number and offset into one word, saving pro-

gram size and execution time. Since 23 bits obviously don't fit into a 16-bit word, the 8 most significant bits of the offset are omitted and implied to be zero. The most significant bit of the address word is made 0 to indicate Short Offset Mode. Short Offset addresses are thus limited to the first 256 bytes at the beginning of each segment. This may appear to be a severe restriction, but it is very useful, especially in the Index Mode, where the index register can always supply the full 16-bit range of the offset. Short Offset saves one instruction word and speeds up execution by two clock cycles in Direct Address Mode and three clock cycles in Indexed Mode.

Using the Z8010 Memory Management Unit

The Z8001 CPU can be combined with another 48-pin LSI device—the Z8010 MMU—for sophisticated memory management. The MMU provides address translation from the logical addresses generated by the Z8001 CPU to the physical addresses used by the memory. An address translation table, containing starting addresses and size information for each of the 64 segments, is stored in the MMU. The translation table can be written and read by the CPU using Special I/O instructions. The MMU thus provides address relocation under software control, making software addresses (i.e., logical addresses) independent of the physical memory addresses.

But the MMU provides much more than address relocation; it also monitors and protects memory access. The MMU provides a Trap input to the CPU and—if necessary—an inhibit signal ($\overline{\text{SUP}}$) to the memory write logic when specific memory-access violations occur. The MMU provides the following types of memory protection:

- Accesses outside the segment's allotted memory can be prevented.
- Any segment can be declared invalid or non-accessible to the CPU.
- Segments can be declared Read Only.
- By designating a segment as System Only, access can be prohibited during the Normal Mode.
- Declaring a segment Execute Only means it can be accessed only during instruction access cycles. Data or stack use is prohibited.
- Any segment can be excluded from DMA access.
- Segments can have a Direction And Write Warning attribute, which generates a trap when a write access is made in the last 256 bytes of its size. This mechanism can be used to prevent stack overflow.

Multiple MMUs must be used when more than 64 segments are needed. Thus, to support the full complement of 128 segment numbers provided for each Z8001 CPU address space, two MMUs are required. The MMU has been designed for multiple-chip configurations, both to support 128-segment translation tables and to support multiple translation table systems.

Note that the memory management features do not interfere with the ability to directly address the entire memory space. Once programmed, the MMU (or MMUs) translates and monitors any memory address generated by the CPU.

The MMU contains status bits that describe the history of each segment. One bit for each segment indicates whether the segment has been accessed; another bit indicates whether the segment has been written. This is important for certain memory management schemes. For example, the MMU indicates which segments have been updated and, therefore, must be saved on disk before the memory can be used by another program.

When translating logical addresses to physical memory addresses, the MMU must do the following: access its internal 64×32 -bit RAM, using the segment number as the address, then add the 16 bits of RAM output to the most significant address byte (AD_8 – AD_{15}) and finally place the result on its Address outputs. The least significant byte (AD_0 – AD_7) bypasses the MMU.

The internal RAM access time is approximately 150 ns. Throughput delay is avoided by making the segment number available early: SN_0 – SN_7 are output one clock period earlier than the address information on AD_0 – AD_7 .

In summary, the Z8000 CPU supports sophisticated memory management through such architectural features as the Status Lines, the R/\overline{W} and S/\overline{N} lines, Segment Trap input line, and early output of segment numbers.

Using 16K Dynamic RAMs with the Z8001

Z8000 systems usually implement most of their memory with 16K × 1-bit dynamic RAMs that have time-multiplexed addresses (Zilog also manufactures this device—the Z6116). In Z8001-based systems with MMUs, CPU Address/Data lines AD₁-AD₇ supply row addresses, MMU address outputs A₈-A₁₄ supply column addresses, and MMU outputs A₁₅-A₂₃ are decoded to generate Chip Select signals that gate either $\overline{\text{RAS}}$ or $\overline{\text{CAS}}$ or both.

Gating $\overline{\text{RAS}}$ reduces power consumption because all non-selected memories remain in the standby mode. But this technique

requires that $\overline{\text{RAS}}$ must wait for the availability of the most significant address bits from the MMU. During refresh, the $\overline{\text{RAS}}$ decoder must be changed to activate all memories simultaneously.

Gating $\overline{\text{CAS}}$ does not achieve lower power consumption; however, this technique allows the use of slower memories because $\overline{\text{RAS}}$ can be activated as soon as the CPU address outputs are stable, without waiting for the MMU delay. Also, there is no need to change the $\overline{\text{CAS}}$ decoder during refresh.

Data Types and Instructions

The Z8000 architecture directly supports bits, digits, bytes, and 16- or 32-bit integers as primitive operands in its instruction set. In addition, the rich set of addressing modes supports higher-level data constructs such as arrays, lists and records. The Z8000 also intro-

duces a number of powerful instructions that extend the capabilities of microprocessors. The remaining sections of this paper describe Z8000 data types, addressing modes, and a selection of novel instructions.

Data Types

Operands are 1, 4, 8, 16, 32, or 64 bits, as specified by the instruction. In addition, strings of 8- or 16-bit data can be manipulated by single instructions. Of particular interest are the increased precisions of the arithmetic instructions. Add and Subtract instructions can

operate on 8-, 16-, or 32-bit operands; Multiply instructions can operate on 16- or 32-bit multiplicands; and Divide instructions can operate on 32- or 64-bit dividends. The Shift instructions can operate on 8-, 16-, and 32-bit registers.

Addressing Modes

The rich variety of addressing modes offered by Z8000 architecture includes: Register, Immediate, Indirect Register, Direct Address, Index, Relative Address, Base Address, and Base Index. Three are of particular interest with respect to high-level data structures: Indirect Register, Base Address, and Base Index. These modes can be used for lists, records, and arrays, respectively.

Indirect Register. In this addressing mode, the contents of the register are used as a memory address. This mode is needed whenever special address arithmetic must be performed to reference data. Essentially, the address is calculated in a register and then used to fetch the data. For example, this mode is useful when manipulating a linked list, where each entry contains a memory pointer to the memory location of the next entry. Essentially, the pointer is loaded into a register and used to access the next item on the list. When the list item is large or has a complex structure, the Base Address or Base Index Modes can be used to access various components of the item.

Base Address. In this addressing mode, the memory address contained in the register (the base) is modified by a displacement in the instruction (known at compile time). This mode

is useful, for example, in accessing fields within a record whose format is fixed at compile time.

Base Index. The memory address in this addressing mode is contained in a register (the base) and is modified by the contents of another register (the index). This mode can be useful in accessing the components of an array, because the index of the component is usually calculated during execution time—as a function of the index of a DO-Loop, for example.

Index vs. Base Address. In the Z8002 and in the Z8001 running non-segmented, these two addressing modes are functionally equivalent, because the base address and displacement are both 16-bit values.

When the Z8001 runs segmented, there is a difference: in the Index mode, the base address (including the segment number) is contained in the instruction, in either Short Offset or Long Offset notation. The 16-bit displacement stored in a register is then added to the offset in the base address to calculate the effective address. In the Base Address Mode, on the other hand, the 16-bit displacement is specified in the instruction and is added to the offset of the base address that is stored in a long-word register.

The Instruction Set

The Z8000 offers an abundant instruction set that represents a major advance over its predecessors. The Load and Exchange instructions have been expanded to support operating system functions and conversion of existing microprocessor programs. The usual Arithmetic instructions can now deal with higher-precision operands, and hardware Multiply and Divide instructions have been added. The Bit Manipulation instructions can access a calculated bit position within a byte or word, as well as specify the position statically in the instruction.

The Rotate and Shift instructions are considerably more flexible than those in previous microprocessors. The String instructions are useful in translating between different character codes. Special I/O instructions are included to manage peripheral devices, such as the Memory Management Unit, that do not respond to regular I/O commands. Multiple-processor configurations are supported by special instructions.

The following instructions exemplify the innovative nature of the Z8000 instruction set. A complete list of Z8000 instructions can be found in the reference materials listed at the end of this tutorial.

Load and Exchange Instructions.

Exchange Byte (EX) is practical for converting Z-80, 8080, 6800 and other microprocessor programs into Z8000 code, because the Z8000 uses the opposite assignment of odd/even addresses in 16-bit words.

Load Multiple (LDM) saves *n* registers and is useful for switching tasks.

Load Relative (LDR) loads fixed values from program space into data space.

Arithmetic Instructions.

Add With Carry and Subtract With Carry (ADC, SBC) are conventionally used in 8-bit microprocessors for multiprecision arithmetic operations. These instructions are rarely used with the Z8000 CPU because it has 16- and 32-bit arithmetic instructions.

Decrement By N and Increment By N (DEC, INC) are intended for address and pointer manipulation, but can also be used for Quick Add/Subtract Immediate with 4-bit nibbles. The flag setting is different from Add/Subtract instructions—as is conventional—in that the Carry and Decimal adjust flags are unaffected by the Increment and Decrement instructions to support multiple precision arithmetic.

Decimal Adjust (DAB) automatically generates the proper 2-digit BCD result after a byte Add or Subtract operation, and eliminates the need for special decimal arithmetic instructions.

Multiply (MULT) provides signed (two's complement) multiplication of two words, generating a long-word result; or of two long-words generating a quadruple word result. No byte multiply exists because it is rarely used and, after sign extension, can be performed by a word multiply.

Divide (DIV) provides signed (two's complement) division of a long word by another word, generating a word quotient and a remainder word; or of one quadruple-word by a long-word, generating a long-word quotient and long-word remainder.

Both Multiply and Divide use a conforming register assignment. That is, a multiply followed by a divide on the same registers is essentially a no-op. The register designation used in the operation description must be even for word operations and must be a multiple of four for long-word operations.

Logical Instructions.

Test Condition Code (TCC) performs the same test as a Jump instruction, but affects the least significant bit of a specified register instead of changing the PC.

Program Control Instructions.

Call Relative (CALR) is a shorter, faster version of Call, but with a limited range.

Decrement And Jump If Non-Zero (DJNZ) is a one-word basic looping instruction.

Jump Relative (JR) is a shorter, faster version of Jump, but with a limited range.

Bit Manipulation Instructions.

Test Bit, Reset Bit, Set Bit (BIT, RES, SET) are available in two forms: static and dynamic. For the static form, any bit (the position is defined in the immediate word of the instruction) located in any byte or word in any register or in memory can be set, reset or tested (inverted and routed into the Z flag).

For the dynamic form, any bit (the position is defined by the content of a register that is, in turn, specified in the instruction) located in any byte or word in any register, but not in memory, can be set, reset or tested.

Test And Set (TSET) is a read/modify/write instruction normally used to create operating system locks. The most significant bit of a byte or word in a register or in memory is routed into the S flag bit and the whole byte or word is then set to all 1s. During this instruction, the processor does not relinquish the bus.

Test Multi-Micro Bit and Multi-Micro Request/Set/Reset (MBIT, MREQ, MSET, MRES) are used to synchronize the access by multiple microprocessors to a shared resource,

The Instruction Set
(Continued)

such as a common memory, bus, or I/O device.

Note that the instruction MREQ (Multi-Microprocessor Request) has nothing whatsoever in common with the MREQ (Memory Request) output from the Z8000 CPU.

Rotate and Shift Instructions.

The Z8000 CPU has a complete set of shift instructions that shift any combination of bytes or words, right or left, arithmetically or logically, by any meaningful number of positions as specified either in the instruction (static) or in a register (dynamic).

The CPU also has a smaller repertoire of rotate instructions that rotates bytes or words, either right or left, through carry or not, and by one bit or by two bits.

The instructions Rotate Digit Left and Rotate Digit Right (RLDB, RRDB) rotate 4-bit BCD digits right or left, and are used in BCD arithmetic operations.

Block Transfer and String Manipulation Instructions.

Translate And Decrement/Increment (TRDB, TRIB) is used for code conversion, such as ASCII to EBCDIC. These instructions translate a byte string in memory by substituting one string by its table-lookup equivalent. TRDB and TRIB execute one operation and decrement the contents of the length register; thus they are useful as part of loop performing several actions on each character.

Translate, Decrement/Increment and Repeat (TRDRB, TRIRB) are the same as TRDB and

TRIB, except they repeat automatically until the contents of the length register become zero. They are therefore useful in straightforward translation applications.

Translate And Test, Decrement/Increment (TRTDB, TRTIB) tests a character according to the contents of the translation table.

Translate And Test, Decrement/Increment And Repeat (TRTDRB, TRTIRB) scans a string of characters. The first character is tested and, depending on the contents of the translation table, the process stops or skips to the next character. Stopped characters can be used for further processing.

I/O and Special I/O Instructions.

The Z8000 CPU has two complete sets of I/O instructions: Standard I/O and Special I/O. The only difference is the status information on the ST₀-ST₃ outputs. Standard I/O instructions are used to communicate with Z-Bus compatible peripherals. Special I/O instructions are typically used for communicating with the Memory Management Unit.

Both types of instructions transfer 8 or 16 bits and use a type of 16-bit addressing analogous to the Z8002 memory-addressing scheme: For word operations, A₀ is always zero; in byte-input operations, A₀ is used internally by the CPU to select the appropriate byte; in byte-output operations, the byte is duplicated in the high and low bytes of the address/data bus, and external logic uses A₀ to enable the appropriate output device.

Bibliography

Selected Publications on the Z8000 Family
Z8001/Z8002 CPU Product Specification
(00-2045)
Z8000 CPU Instruction Set (03-8020-01)

Z8000 PLZ/ASM Assembly Language
Programming Manual (03-3055-01)
Z8010 Z-MMU Product Specification (00-2046)

A Small Z8000 System



Application Note

Component Application Engineering

January 1980

Introduction This application note describes the hardware design implementation of a small computer using the Zilog Z8002 16-bit microprocessor, ROMs/EPROMs and dynamic RAMs plus parallel and serial I/O devices. The interface requirements of the Z8002 to memory and to Z80A peripherals are described and design alternatives are given whenever possible. This design is similar in structure and is software compatible with the Zilog Z8000 Development

Module (part number 05-6101-01).

The design uses a minimal number of TTL support devices and, whenever possible, gate functions have been combined into MSI circuits. The result is a design that uses MSI TTL circuits in a very efficient—but sometimes non-obvious—way that minimizes the package count. Because some of the design techniques may not be self-explanatory, an effort has been made to explain them.

General Structure

Figure 1 shows a block diagram of the design. The Z8002 16-bit microprocessor is the heart of the system. This high-performance CPU offers a regular architecture, a powerful instruction set, a sophisticated interrupt structure, and high throughput at a modest 4 MHz clock rate.

For a description of the Z8002, see the *Z8001/Z8002 CPU Product Specification* (03-8002-01). For a detailed description of the Z8000 instruction set, refer to the *Z8000 PLZ/ASM Assembly Language Manual* (03-3055-01).

Fixed program and data information is stored in an array of 2K x 8 ROMs or EPROMs; 16 16K x 1 dynamic RAMs provide 32K bytes of read/write storage. Input/output is handled by five I/O devices. Two Z80A PIOs provide 4 byte-wide bidirectional ports (32 lines) with handshake control. A Z80A SIO provides two fully independent full-duplex asynchronous or synchronous serial data communications channels. Four counter/timers in the Z80A CTC

relieve the processor from simple counting and timing tasks and generate the programmable baud-rates for the serial I/O channels. Eight switches can be interrogated and interpreted by the program.

The block diagram also indicates the various support functions. A crystal-controlled clock circuit generates a Z8002 and Z80A compatible clock signal plus two complementary TTL clocks. Address buffers drive the memory and I/O devices; address latches demultiplex the time-shared Address/Data bus.

The ROM array uses a One-of-Eight Address Decoder and the RAMs are driven by an address multiplexer and a $\overline{RAS}/\overline{CAS}$ generator. The timing for all these functions originates in the bus control and timing circuit. The I/O devices are selected by an I/O decoder and receive Z80A equivalent control signals generated by the Z8002 to Z80A Control Translator. The following sections contain detailed descriptions of these circuits.

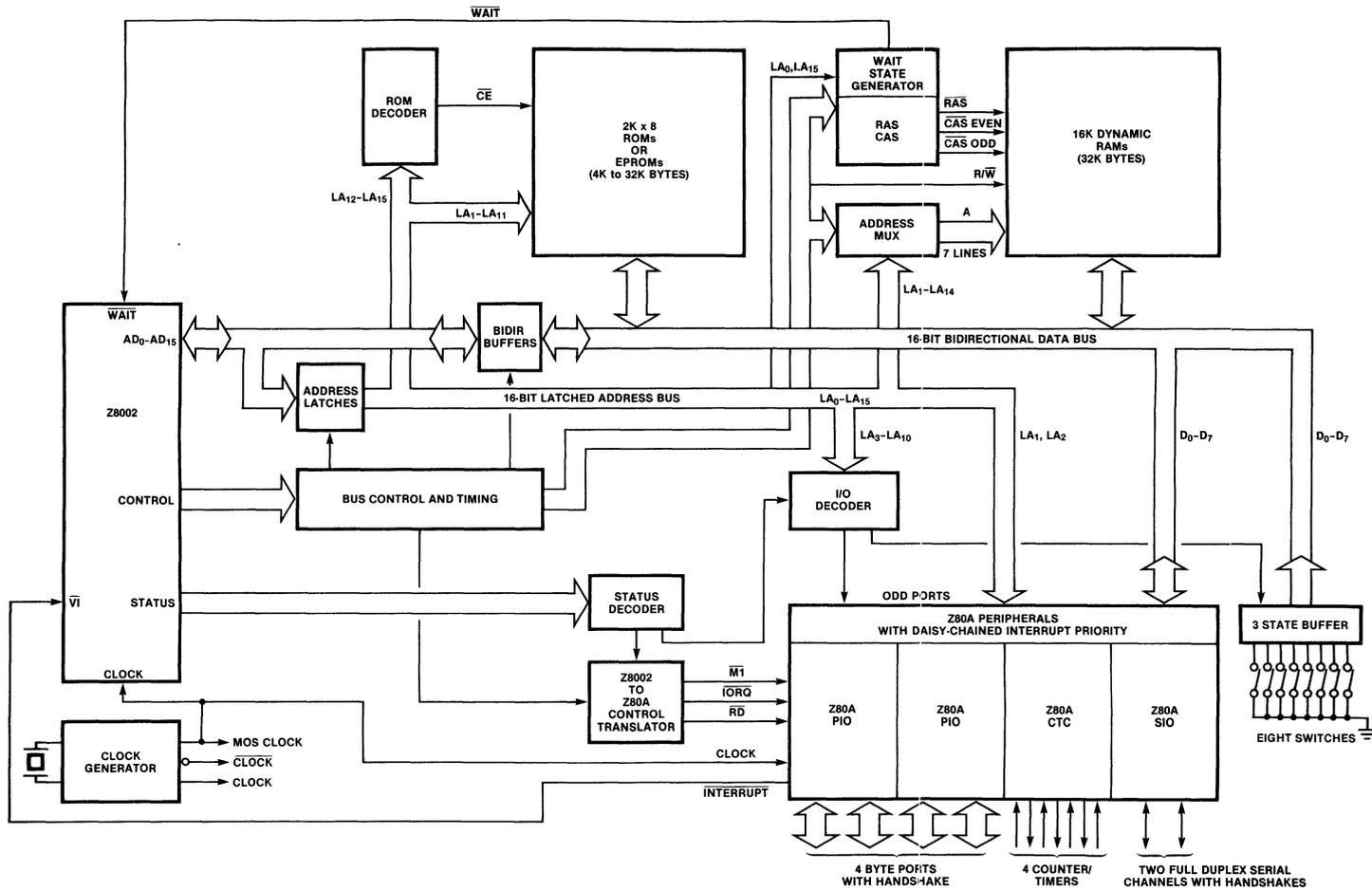


Figure 1. Block Diagram

CPU Output Buffering

The Z8002 outputs can sink 2 mA while maintaining TTL noise margins and can thus drive five LS-TTL inputs. All output delays are specified for a capacitive load of up to 50 pF. They increase by approximately 0.1 ns/pF of additional capacitive load.

Very small systems can be built without TTL buffering of the CPU outputs, but most systems require TTL buffering of the Address/Data lines and major control outputs, like \overline{AS} , \overline{DS} , \overline{MREQ} and R/W.

Bidirectional buffering of the A/D lines. The Address/Data lines require Bus Transceivers, such as the LS243 Quad Non-Inverting Bus Transceiver with separate Enable inputs for the two directions (one active High; the other active Low), or the LS245 Octal Non-Inverting Bus Transceiver with a Direction Control input and an active Low Enable input.

Figure 3 shows the logic that controls four LS243 Quad Transceivers; Figure 4 shows the even simpler logic that controls two LS245 Octal Transceivers.

The bus transceivers are controlled by three CPU control outputs as shown in the following truth table.

\overline{BUSACK}	R/W	\overline{DS}	
H	H	L	Enable Receiver (input Data into CPU)
H	H	H	Enable Transmitter (output Address or Data from CPU)
H	L	H	
H	L	L	
L	X	X	Disable Transceiver

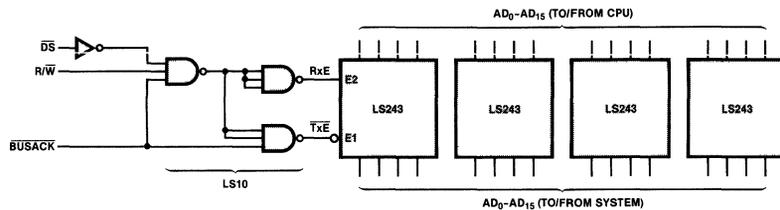


Figure 3. Bidirectional Address/Data Buffering Using Quad Transceivers

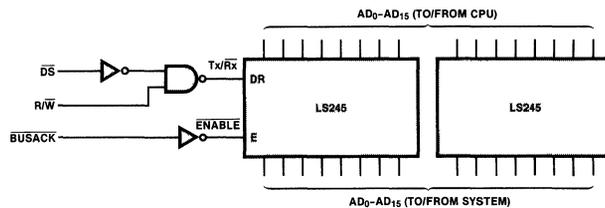


Figure 4. Bidirectional Address/Data Buffering Using Octal Transceivers

Unidirectional Buffering of CPU Control Outputs. The following CPU control outputs may require unidirectional buffering: \overline{AS} , \overline{DS} , \overline{MREQ} , R/W, N/S, B/W.

The buffered signals must be 3-stated when \overline{BUSACK} is Low. One LS365A or LS367A Hex 3-State Buffer can perform this function as shown in Figure 5. The LS244 Octal 3-State Buffer buffers eight signals, but uses a 20-pin package.

In a simple system, such as the one described here, \overline{BUSREQ} is not used, so \overline{BUSACK} is therefore always High. In a more complex system with direct memory access, a Low on \overline{BUSACK} indicates that the CPU has relinquished the bus. If the buffered bus is

shared, \overline{BUSACK} must be used to control the latches and transceivers, as shown in Figures 4 through 6.

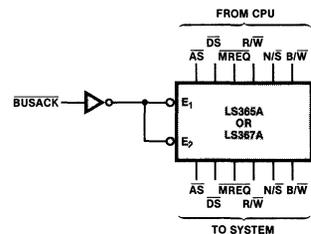


Figure 5. Control Signal Buffering

Address Latching (Demultiplexing the A/D lines)

The Z8002 uses a 16-bit time-shared Address/Data bus that must be demultiplexed, that is, latched for use with standard (not edge-activated) memories. \overline{AS} is the obvious control signal for address latching and two LS373 Octal Transparent Latches are the best choice for this function (Figure 6). Note that

addresses are not guaranteed valid when \overline{AS} goes Low. It is therefore not possible to use the falling edge of \overline{AS} to clock the addresses into edge-triggered registers. The rising edge of \overline{AS} may be used as a clock, but this delays address availability by almost 100 ns. Transparent latches are the better choice.

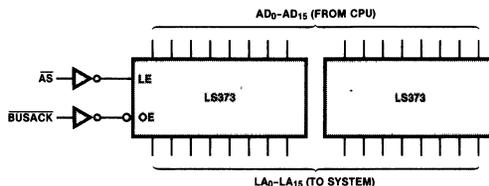


Figure 6. Address Latches

ROM Addressing

Most microprocessors use nonvolatile memory for part of their program memory. Since the program status information for the Z8002 is read after Reset from locations 0002 and 0004, it is natural to use the lower half of the addressing space for ROM or EPROM.

This application uses 2716-type 2K × 8 EPROMs addressed by the latched addresses LA₁-LA₁₁. Pairs of 2716s store the low and high byte of each word. A₀ is ignored since the Z8002 always reads a full word from memory. LA₁₅ must be used as a Chip Select input to separate the ROM and RAM areas. When more than 2K words of ROM or EPROM are used, an LS138 one-of-eight decoder selects between the ROM and EPROM pairs.

When driven with a 4 MHz clock, the Z8002 requires a read access time (address valid from the CPU to data required into the CPU) of 400 ns. After subtracting a 27 ns propagation delay through the LS373 address latches and an 18 ns propagation delay through the LS243 transceivers, the ROM or EPROM must have an access time (address in to data out) of better than 355 ns. Some ROMs and EPROMs have a longer access time and therefore require an additional wait state that relaxes the access time requirement by an additional 250 ns. Figure 8 shows a 2-input NAND gate that generates a Wait signal whenever LA₁₅ is Low and Q2 is High, thereby adding a wait state to every ROM/EPROM access.

RAM Address Multiplexing and RAS CAS Generation

Dynamic 16K × 1 RAMs such as the Z6116 provide read/write random-access storage. Sixteen of these devices populate the upper half of the addressable memory space (LA₁₅ = High). Dynamic 16K RAMs use address multiplexing to reduce the package pin count, thus requiring only seven address inputs plus strobe inputs RAS and CAS.

Address Multiplexing. Two LS157 Quad Two-Input Multiplexers route the 14 address outputs LA₁-LA₁₄ into the seven RAM address inputs. MREQ synchronized with the rising clock edge is a convenient signal to control this multiplexer (Figure 7).

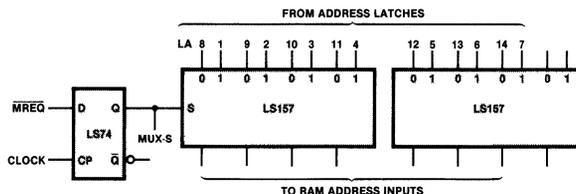


Figure 7. Address Multiplexer

RAM Address Multiplexing
(Continued)

RAS and CAS Generation. The address strobes $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ must be timed carefully with respect to the address information and the multiplexer control. Conceptually, $\overline{\text{MREQ}}$ might be used as $\overline{\text{RAS}}$ and $\overline{\text{DS}}$ as $\overline{\text{CAS}}$. This would, however, require a memory read access time from the falling edge of $\overline{\text{CAS}}$ of approximately 120 ns (parameter 33 in the *Z8001/Z8002 Product Specification Composite AC Timing Diagram*, minus the 30 to 40 ns used by the $\overline{\text{CAS}}$ drivers and bus transceivers). Only the fastest 16K dynamic RAMs (the Zilog Z6116-2, for example) meet this requirement. Consequently, it is more practical to use a small amount of clocked TTL logic to generate earlier $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ signals and thus relieve the access time requirements so that even slow 16K RAMs (Z6116-3 and -4) can be used. Figure 8 shows the circuit that generates $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$.

$\overline{\text{RAS}}$ is a 2-clock period (500 ns at 4 MHz) wide active-Low signal starting on the falling clock edge when $\overline{\text{AS}}$ is Low. The address information is valid and stable during the specified hold time (<50 ns) immediately after the falling edge of $\overline{\text{RAS}}$. $\overline{\text{RAS}}$ is generated by an LS109 edge-triggered dual JK flip-flop, clocked by $\overline{\text{CLOCK}}$ (that is, of a polarity opposite to the Z8002 clock). At the end of a machine cycle both Q1 and Q2 are High. The

falling edge of $\overline{\text{CLOCK}}$ during $\overline{\text{AS}}$ clocks Q1 Low. The next falling clock edge leaves Q1 unaffected, but clocks Q2 Low. The next falling edge clocks Q1 High and leaves Q2 unaffected. The next falling clock edge clocks Q2 High and leaves Q1 High unless $\overline{\text{AS}}$ is Low, in which case the cycle is repeated. Q1 is Low from the center of the first to the center of the third T state. Q2 is Low from the center of the second to the center of the fourth T state.

The left half of the LS139 Dual One-of-Four Decoder generates $\overline{\text{CAS}}$ by ANDING three signals: LA_{15} , MUX-S, and an auxiliary signal active during Read or DS.

During a read operation, $\overline{\text{CAS}}$ becomes active at the beginning of T_2 ; that is, on the rising edge of $\overline{\text{CLOCK}}$ after $\overline{\text{MREQ}}$ has gone Low. During a write operation, $\overline{\text{CAS}}$ is delayed until the beginning of DS, when output data is guaranteed valid. The flip-flop stretches the width of DS, thus stretching $\overline{\text{CAS}}$ (during write operations) from 160 to 200 ns, as required by slower memories.

The right half of the LS139 decoder controls the routing of $\overline{\text{CAS}}$ to the two memory byte banks. The Z8002 addresses memory as bytes, but usually accesses words, ignoring A_0 . It uses A_0 only when writing a byte, in which case it suppresses $\overline{\text{CAS}}$ to the byte bank that is not being written.

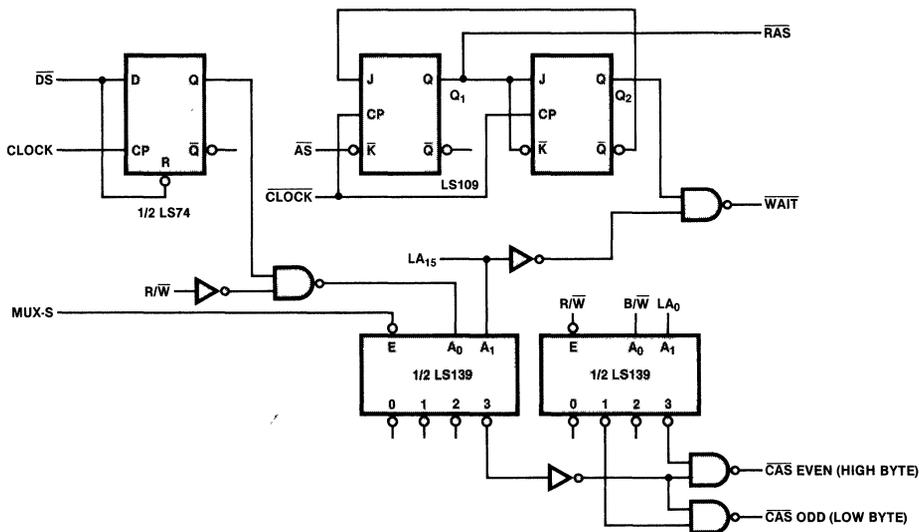


Figure 8. $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and $\overline{\text{WAIT}}$ STATE Generators

RAM Address Multiplexing
(Continued)

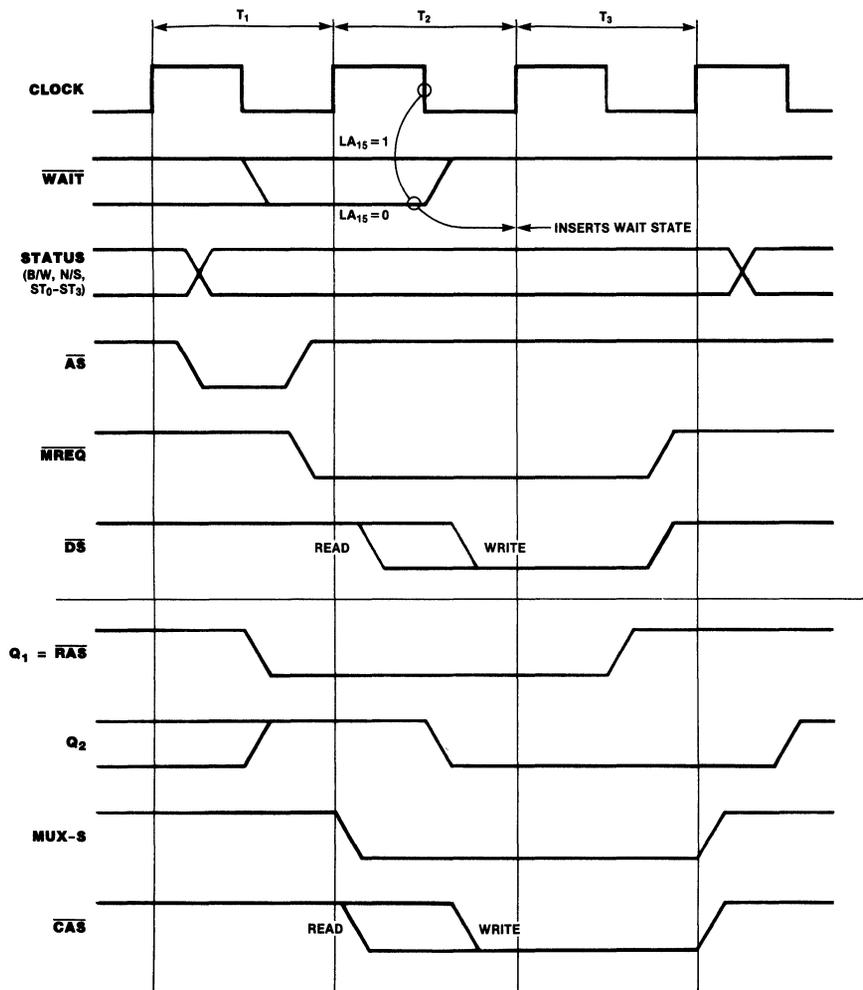


Figure 9. RAS and CAS Generation

RAM Address Multiplexing
(Continued)

Dynamic Memory Refresh. No external hardware is required for memory refresh. The Z8002 provides automatic memory refresh if properly initiated through a LDCTL instruction into the Refresh Control Register (Figure 10). Loading a 9E00 generates a refresh operation every 60 clock cycles (15 μ s with a 4 MHz clock). This satisfies the worst-case

refresh requirements of typical 16K dynamic RAMs.

Figure 11 shows the relationship between the upper byte of the refresh control register and the refresh period expressed in clock cycles. (Refer to the *Z8000 PLZ/ASM Assembly Language Programming Manual*, 03-3055-01 for further information.)

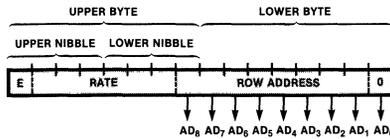


Figure 10. Refresh Control Register

		LOWER NIBBLE OF UPPER BYTE															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
UPPER NIBBLE OF UPPER BYTE	0																
	1																
	2																
	3																
	4					NO REFRESH											
	5																
	6																
	7																
	8	256	4	8	12	16	20	24	28								
	9	32	36	40	44	48	52	56	60								
	A	64	68	72	76	80	84	88	92								
	B	96	100	104	108	112	116	120	124								
	C	128	132	136	140	144	148	152	156								
	D	160	164	168	172	176	180	184	188								
	E	192	196	200	204	208	212	216	220								
	F	224	228	232	236	240	244	248	252								

Figure 11. Refresh Period in Clock Cycles

Status Decoding

The Z8002 provides encoded status information on four outputs (ST₀-ST₃), which distinguish between three different interrupt acknowledge cycles; memory refresh; I/O reference; internal operation; data memory, stack or program memory access; and the first

word of an instruction fetch. Two LS138 One-of-Eight Decoders can generate all the individual status signals. For a simple system, only the first ten status codes have to be decoded. A single LS42 One-of-Ten Decoder is sufficient for this purpose (Figure 12).

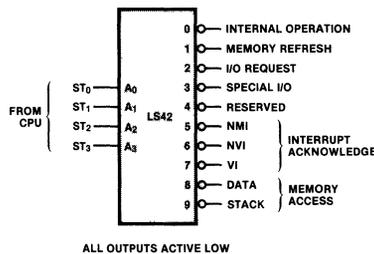


Figure 12. Status Decoder

Interfacing Peripheral Devices

(Continued)

The first four I/O devices addressed when \overline{LA}_5 is Low are four Z80A peripheral components. The fifth peripheral is a set of eight switches that can be read by the CPU, which addresses them as a peripheral device. The user can thus specify any one of 256 different conditions (for example, choosing between 16 different baud rates for each of the two serial I/O channels). The sixth \overline{CE} output addresses a phantom peripheral called RETI, which is activated at the end of an interrupt service operation.

The interrupt operation requires some extra logic and software to make the Z80A peripherals compatible with the Z8002. Z80A peripherals request a vectored interrupt by pulling the \overline{VI} input of the CPU Low. The CPU (Z80A or Z8002) samples this input at a specified time prior to the end of any instruction execution. The Z8002 then acknowledges the interrupt with a specific Status code (\overline{VIACK}). The Z80A, which has no dedicated Interrupt Acknowledge output, acknowledges interrupts by issuing a unique combination of control signals: \overline{IORQ} active during an $\overline{M1}$ cycle ($\overline{M1}$ normally indicates the opcode fetch cycle of an instruction execution). Z80A peripherals resolve potential conflicts between overlapping interrupt requests from different interrupting devices by means of a daisy-chain arrangement between the IEO outputs and the IEI inputs of the peripheral components. The highest-order peripheral has its IEI permanently tied High. For any peripheral that has no interrupt pending or under service, $IEO = IEI$. Any peripheral that has an interrupt pending or under service forces its IEO Low.

To insure stable conditions in the daisy chain, all interrupt status signals are prevented from changing while $\overline{M1}$ is Low. When

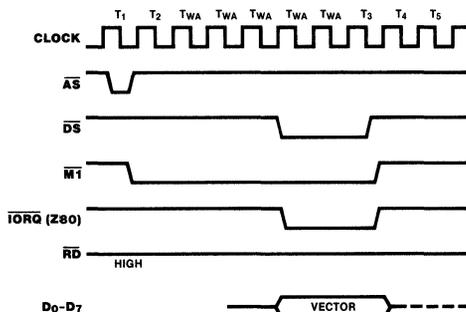


Figure 14. Interrupt Acknowledge Cycle

\overline{IORQ} is Low, the highest priority interrupt requestor (the one with IEI High) places its interrupt vector on the data bus and sets its internal interrupt-under-service latch. Figure 14 shows the Interrupt Acknowledge timing.

The circuit shown in Figure 13 generates the $\overline{M1}$, \overline{IORQ} , and \overline{RD} signals required by the Z80A peripherals during an interrupt acknowledge cycle.

Return From Interrupt. At the end of an interrupt service routine the interrupt-under-service latch in the Z80A peripheral that has been serviced must be reset. The Z80A CPU accomplishes this by executing a special 2-byte instruction with the opcode sequence ED-4D (RETI) appearing on the data bus. All peripherals monitor this sequence and manipulate the daisy chain to reset the appropriate internal interrupt-under-service latch. The normal daisy-chain operation can be used to detect a pending interrupt; however, it cannot distinguish between an interrupt under service and a pending unacknowledged interrupt of a higher priority. Whenever "ED" is decoded, the daisy chain is modified by forcing High the IEO of any interrupt that has not yet been acknowledged. Thus the daisy chain identifies the device presently under service as the only one with an IEI High and an IEO Low. If the new opcode byte is "4D," the interrupt-under-service latch is reset (Figure 15).

The Z8002 does not have the equivalent RETI instruction and must therefore simulate it with a combination of hardware and software. A software sequence at the end of every interrupt service routine writes two consecutive bytes

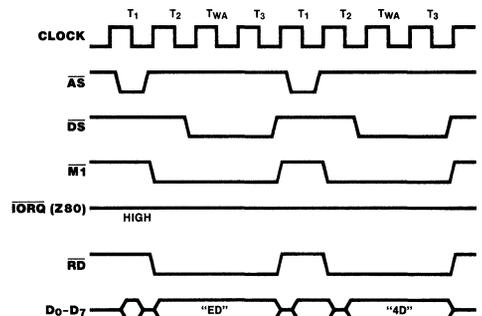


Figure 15. Return from Interrupt Cycle

**Interfacing
Peripheral
Devices**

(Continued)

(ED followed by 4D) into the phantom peripheral called RETI. The recommended software sequence is as follows:

DI		Disable Interrupts
LDB	RL1, #%ED	Load First Byte
OUTB	RETI, RL1	Output First Byte
LDB	RL1, #%4D	Load Second Byte
OUTB	RETI, RL1	Output Second Byte
EI		Enable Interrupts
RET		Return From Interrupt

To prevent the two byte simulated RETI instruction from being interrupted, interrupts must be disabled. If NMIs can occur at any time, then interrupts must remain disabled throughout the NMI service routine. This allows the Z80A peripheral devices to decode correctly a RETI instruction. During the two OUTB operations, each four clock cycles long, RETI is Low, \overline{VIACK} is High and the Z80A control signals \overline{MI} and \overline{RD} are Low.

Driving Z80A Peripherals. The Z80A PIO, CTC and SIO are directly connected to the appropriate lines, as follows. The bidirectional AD₀-AD₇ buffers are connected to the D₀-D₇ data inputs/outputs on the peripherals.

The address bits LA₁ and LA₂ are used as Port Select (A/B) and Control Data select (C/D) on the PIO and SIO, and as Channel Select (CS₀, CS₁) on the CTC.

The Interrupt outputs of all peripherals are interconnected (pulled up with a 4.7k Ω resistor to V_{CC} and connected to the \overline{VI} input of the Z8002). The IEI-IEO interrupt daisy chain of the Z80A peripheral devices must be connected appropriately to establish the desired hierarchy of interrupt priorities.

The Z80A PIO requires a \overline{MI} to enable the peripheral circuit's internal interrupts. This can easily be accomplished by writing a dummy byte (00H) to the RETI port after PIO interrupts have been enabled.

Reset

The Z8000 Reset input requires a minimum High level of 2.4 V. While TTL High levels are guaranteed to be at least 2.4 V, this does not leave margin for noise immunity. If an open collector buffer (such as a 7407) is available, an output pullup resistor to +5 V will provide

more than adequate margin for noise immunity. If an open collector gate is not readily available, a standard TTL gate may be used with an output pullup resistor. In this case, the value of the pullup resistor should not be less than 300 Ω .

Conclusion

This Application Note demonstrates that a small, but powerful computer can be built around the Z8002 16-bit microprocessor using very few standard TTL support packages. It

also shows how the readily-available Z80A peripheral circuits interface easily to the Z8002, taking advantage of the similarity in the Z80A and Z8000 interrupt structures.

An Introduction to the Z8010 MMU Memory Management Unit



Tutorial Information

March 1981

Introduction

The declining cost of memory, coupled with the increasing power of microprocessors, has accelerated the trend in microcomputer systems to the use of high-level languages, sophisticated operating systems, complex programs and large data bases. The Z8001 microprocessor supports these advances by offering multiple 8M byte address spaces as well as a rich and powerful instruction set. The Z8010 Memory Management Unit (MMU) supports the Z8001 processor in the efficient and flexible use of its large address space.

Support for managing a large memory can take many forms:

- Providing a logical structure to the memory space that is largely independent of the actual physical location of the data
- Protecting the user from inadvertent mistakes such as attempting to execute data

- Preventing one user from unauthorized access to memory resources or data
- Protecting the operating system from unexpected access by the users.

The Z8010 provides all these features plus additional features that permit a variety of system hardware configurations and system designs.

This paper examines the various uses of memory management in computer systems and how memory management techniques generally meet these requirements. The major features of the Z8010 MMU illustrate how memory management functions can be supported by hardware. A few examples demonstrate how this LSI circuit can be used to configure several different memory management systems.

Motivations for Memory Management

The primary memory of a computer is one of its major resources. As such, the management of this resource becomes a major concern as demands on it increase. These demands can arise from different sources, three of which are of interest in the present context. The first stems from multiple users (or multiple tasks within a dedicated application) contending for a limited amount of physical memory. The second comes from the desire to increase the integrity of the system by limiting access to various portions of the memory. The final source arises from issues surrounding the development of large, complex programs or systems. Each of these three sources involves a multifaceted group of related issues.

When multiple tasks constitute a given system (for example, multiple users of a system or multiple sub-tasks of a dedicated application), the possibility exists that not all tasks may be in primary memory at the same time. (A task is the action of executing a program on its data; a task may be as simple as a single

procedure or as complex as a set of related routines.) If the population of memory-resident tasks can vary over time, a useful feature of a system would be the ability for a task to reside anywhere in memory, and perhaps in several different locations during its lifetime. Such tasks are called *relocatable*, and a system in which all tasks are relocatable generally offers greater flexibility in responding to changing system environments than a system in which each task must reside in a fixed location.

A second issue that arises in multi-task environments is that of sharing. Separate tasks may execute the same program on different data, and may therefore share common code. For example, several users compiling FORTRAN programs may wish to share the compiler rather than each user having a separate copy in memory. Alternatively, several tasks may wish to execute different programs using the same data as input, and it may be possible for these tasks to access the same copy of the

**Motivations
for Memory
Management**
(Continued)

input. For example, a user may wish to print a PASCAL program while it is being compiled; the print process and the compiler process could access the same copy of the text file.

A third issue in multi-task systems is protecting one task from unwanted interactions with another. The classic example of unwanted interaction is one user's unauthorized reading of another user's data. Prohibiting all such interactions conflicts with the goal of sharing and so this issue is usually one of selectively prohibiting certain types of interactions. The issue of protecting memory resources from unauthorized access is usually included in the larger set of issues relating to system integrity.

System integrity takes many forms in addition to protecting a task's data from unwanted access. Another aspect is preventing user tasks from performing operating system functions and thereby interrupting the orderly dispatch of these tasks. For example, most large systems prevent a user task from directly initiating I/O operations because this can disrupt the correct functioning of the system.

Another aspect of separating users from system functions relates to separating system I/O transfers from user tasks, especially with respect to error conditions. For example, an error during a direct memory access, say to a nonexistent memory location, should not cause an error in the program that is currently executing.

A final example of increasing the system integrity is protecting a user task from itself. Obvious errors, such as trying to execute data or overflowing an area set aside for a stack, can be detected while a program is executing and handled appropriately, provided the system is given sufficient information.

The notion of protecting an executing task from performing certain types of actions known to be erroneous introduces a third general motivation for memory management, namely support for the design and correct implementation of large, complex programs and systems.

Protecting a task from itself obviously helps in debugging a large program, but there are other system features that can aid in developing complex systems. Modern methodology for developing large systems dictates partitioning a task into a number of small, simple, self-contained sub-tasks with well defined interfaces. Each sub-task generally interacts with only a few other sub-tasks and this communication is carefully controlled. This methodology promotes a systems design that can be readily modified, but it also tends to promote the creation of a large number of nearly independent sub-tasks and many data structures accessible to only one or a few of these sub-tasks.

Because modern systems are increasingly driven to support many interacting tasks, possibly written and compiled separately, they must also enforce some communication protocol without sacrificing efficient operation. Modern memory management systems can offer effective tools for implementing large systems designed using this methodology.

In summary, the major goals of memory management systems are to:

- Provide flexible and efficient allocation of memory resources during the execution of tasks
- Support multiple, independent tasks that can share access to common resources
- Provide protection from unauthorized or unintentional access to data or other memory resources
- Detect obviously incorrect use of memory by an executing task
- Separate users from system functions.

Most of today's memory management systems support these functions to some degree. The extent of this support is largely a question of resources to be devoted to these functions and the understood demands of the intended applications for these systems.

**The Fundamentals of
Memory
Management**

Memory management has two functions: the *allocation* and the *protection* of memory. Dynamic relocation of tasks during their execution is accomplished by an address translation mechanism. The restriction of memory access is accomplished by memory attribute checking. Both operations occur with each memory request during the execution of a program and both are transparent to the user.

Address translation simply means treating the memory addresses generated by the program as logical addresses to be interpreted or translated into actual physical memory locations before dispatching the memory access requests to the memory unit. Memory attribute checking means that each area of memory has associated with it information as to who can

access it and what types of access can be made by each task. Each memory reference is checked to insure that the task has the right to access that location in the given fashion (for example, to read the contents of the location or to write data to that location).

Instead of a linear address space, more elaborate memory management systems have a hierarchical structure in which the memory consists of a collection of memory areas, called segments. Access to this structured memory requires the specification of a segment and an offset within that segment. Thus, instead of specifying memory location 1050 in a linear address space, a task specifies memory location 5 in segment number 23, for example.

The Fundamentals of Memory Management
(Continued)

Generally, segments can be of variable size, within limits, and a user can specify the size of each segment to be used. Thus one user may have two segments of two thousand and ten thousand words for his FORTRAN program and data, respectively, while another user might have three segments of three thousand, six thousand and two thousand words for her PASCAL program, data, and run-time stack. If the first user called his data segment number 5, then the first word in his data set would be accessed by the logical address (5,0) indicating segment 5, offset 0. The memory management system translates this symbolic name into the correct physical memory address.

Figure 1 gives a conceptual realization of these two users' logical program spaces. The first user, User A, has his program segment called "Segment 6" and his data segment called "Segment 5." The second user, User B, has her program segment called "Segment 5," her data segment called "Segment 12" and her stack segment called "Segment 2." Notice that both users have named one of their segments "Segment 5," but they refer to different entities. This causes no problem since the system keeps the two memory areas separate. The situation is analogous to both users having an integer variable called "I" in their programs: The system realizes that these are two separate variables stored in different memory locations.

User A's data segment, "Segment 5," is ten thousand words. If he references word 10,050

of Segment 5 he gets an error message from the system indicating that he has exceeded the allocation limit for Segment 5. Note that he does not access word 50 of Segment 6. That is, segments are logically distinct and unordered. A reference to one segment cannot inadvertently result in access to another segment. Thus, in this example, User A is prevented from accidentally (or deliberately) accessing his program as though it were part of his data segment.

Figure 2 illustrates one way that these segments could be arranged in the physical memory. The dotted lines indicate the memory-mapping function from the logical address space of the user to the physical memory locations allocated to him. The figure also indicates the access attributes associated with each user's segments. For example, program segments are "execute only" and data segments are "read/write." Thus a user is prevented from executing a data segment or writing into a code segment.

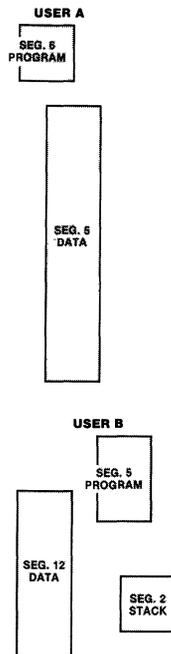


Figure 1. Two User's Logical Address Space

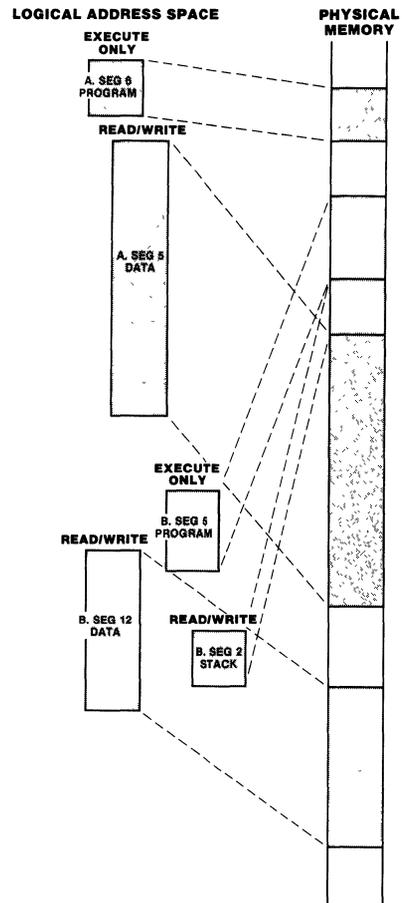


Figure 2. Mapping Logical Segments to Physical Memory

The Fundamentals of Memory Management
(Continued)

Figure 3 illustrates what happens when both users have access to the same data set in primary memory, say the results of a questionnaire that both intend to analyze. Each user has a logical name associated with that data set to specify the segment in which the data set is to reside. Note that the two users have chosen to put the data set in different segments of their personal address spaces. The system-mapping function translates these different segment names to the same physical memory locations. Thus User A's access to address (2, 17) references the same physical memory location as User B's access to address (7, 17). In the figure, note that two of B's segments have been moved in physical memory to create a space large enough to hold the questionnaire data.

Another topic in memory management that is supported by Z8001-Z8010 architecture but requires additional support hardware is demand swapping, or segmented virtual memory, which means that the logical memory

area may not actually reside in physical memory until a task actually tries to access it. At the time an access is made to a segment missing from physical memory, the instruction execution is held in abeyance until the logical memory can be brought into the physical memory and then the instruction is allowed to proceed with the memory access. The address translation is performed, access protection is checked and the instruction proceeds as if the logical memory area had been in the physical memory at the beginning of the instruction. The instructions in the Z8001 must run to completion before the CPU can perform any action, such as responding to a missing segment trap. But with the conjunction of hardware and software to simulate the above functions, a segmented virtual memory scheme can be implemented.

A final topic in memory management is paging, which is another method for partitioning a user address space and mapping it onto the physical memory. Paging is most effective when demand swapping can be supported. Essentially, paging divides the logical memory into fixed-size blocks, called pages. Like segments, the individual pages can be located anywhere in the physical memory and a translation mechanism maps logical addresses to physical memory locations. There are two differences between paging and segmenting a logical memory. First, pages are of fixed size whereas segments are of various sizes. Second, under paging, the logical memory is still linear, that is, a task accesses memory using a single number, rather than a pair as in segmentation. The major advantage of paging is in treating memory as blocks of fixed sizes, which simplifies allocating memory to users and deciding where to place the logical pages in physical memory. The major disadvantage of paging is in assigning different protection attributes to different areas in a user address space because a paged memory appears homogeneous to the user and the operating system. Paging can be combined with segmentation to produce a memory management system with the advantages of both paging and segmentation. The implementation of paging for the Z8001 requires additional support hardware and may be implemented independent of the Z8010.

Before proceeding to the mechanism of memory management, it is instructive to review how a segmented address translation mechanism with protection attributes achieves the five major goals of memory management outlined in the previous section. The first goal permits dynamic allocation of memory during the execution of tasks; that is, a task could be located anywhere in memory and even moved about when its execution is suspended. The address translation mechanism provides this flexibility because the task deals exclusively

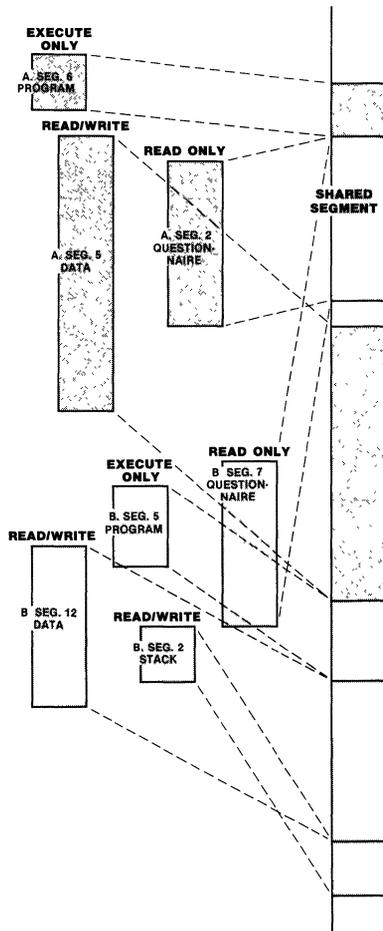


Figure 3. Two Users Sharing a Common Segment

The Fundamentals of Memory Management (Continued)

with logical addresses and hence is independent of the addresses of the physical memory locations it accesses. Moving the task to different physical memory locations requires that the address mapping function be changed to reflect the change in memory location, but the task's code need not be modified. Of course, this flexibility does incur the price of managing the various system tables required to implement memory management.

The second goal supports sharing of common memory areas by different tasks. This is accomplished by mapping different logical areas in different tasks to the same physical memory locations.

The third provides protection against certain types of memory accesses. This is accomplished by associating accessing attributes with each logical segment and checking the type of access to see if each access is permitted.

The fourth goal detects obvious execution errors related to memory accessing. This can be accomplished by checking each access to a segment to see whether the address falls within the allocated physical memory for that segment. It could also include affixing a read/write attribute to data to prevent a task from trying to execute a data segment, and affixing an execute-only attribute to code segments to prevent a task from trying to read or write data to this segment. Additionally, if a segment is used for a stack, the system could issue a warning to a task when the stack approaches the allocated limit of the segment. The task could then request more memory for the stack before the stack overflows and creates a fatal error.

The final goal listed for memory manage-

ment systems separates user functions from system functions. For processors that distinguish between System mode and User mode of operation, this goal can be accomplished by associating a system-only attribute with system segments so users cannot directly access system tables and tasks.

As a final point, it should be noted how segmentation can be used to support the development and execution of large, complex programs and systems. The concept of segmentation corresponds to the concept of partitioning a large system into procedures and data structures where each procedure and data structure can be associated with a separate segment. A task can then invoke a procedure or sub-task or access a data structure by referring to its logical segment name. Access to these objects can be individually restricted by using the protection-checking mechanism of the memory management system.

As a specific example of how segmentation could be used in the design of a large system, consider a multi-user interactive BASIC system with a large data base shared by all users. Such a system could be designed with segments 0 through 15 reserved for system use, segments 16 through 31 reserved for the BASIC interpreter and its internal tables, segments 32 through 63 allocated to user tasks and segments 64 through 127 reserved for portions of the data base when they are in primary memory being accessed by users. For this system, segments 0 through 31 would probably always be in memory; the other segments would be assigned as needed and the memory they require allocated dynamically.

The Mechanics of Memory Management

Essentially there are four issues in implementing a memory management system: how addresses are specified, how these addresses are translated, what attributes are checked for each access, and how the protection mechanism is implemented. Some of the major alternatives in each of these issues are briefly discussed here, primarily from the point of view of a segmented memory.

Two approaches have traditionally been taken for specifying addresses in a segmented memory. For simplicity, only addresses in instructions are discussed. The first way puts all the addressing information in the instruction itself. That is, each memory address in an instruction contains both the segment name and the offset within the segment. The alternative sets aside special registers that contain some of this information, for example the segment name or the address in physical memory where the segment resides.

The advantage of the latter approach lies in the fact that fewer bits are needed in an instruction to specify addresses. Thus programs may be shorter. Also, because there is

reduced traffic between the memory and the processor for fetching shorter instructions, a program may execute faster.

On the other hand, these special registers must be manipulated to access more segments than there are registers, and this manipulation adds to the number of instructions, the program size and the execution time. In practice, these can destroy the advantages described above. If the special registers contain physical memory locations, then these must be protected from user access to maintain the integrity of the system, and changing segments requires system calls which can be time consuming if too few registers are supplied. The Z8001 architecture specifies the complete logical address in the instruction.

Address translation is performed by adding the logical segment offset to the memory location where the segment begins. Thus, when an address of the form (a, b) is presented to the translation mechanism, the segment name "a" is used to determine where segment "a" resides in memory. Assume that it resides in locations 10000 to 25000. Then the actual

The Mechanics of Memory Management
(Continued)

memory location of (a, b) is memory location $10000 + b$. The major option in implementing this type of address translation is in determining the segment location in physical memory. When special registers have been set aside to contain the starting location of the segment instead of putting all address information in the instruction, the addressing mechanism is similar to using the segment register as an index register or a base register.

When logical addresses are either completely specified in the instruction or when the special register contains the symbolic segment name, a table must be used to translate the logical segment name into a physical memory location. The table may have an associative capability, that is, the segment name is presented to the table and the device returns the physical memory location where the segment begins. Alternatively, the table could have one entry for every possible segment name. The Z8010 implementation of the address translation table sets aside a specific table entry for each logical segment name.

A number of attributes can be associated with a segment and checked during each access. One of these is the allocated length of the segment, and each access is checked to see if it falls within the bounds of the segment. The Z8010 provides limit checking.

Another type of attribute deals with ownership or class of ownership: tasks are grouped into classes and only those in certain classes are permitted access. The simplest example is the system versus user classification, where tasks are either one or the other and this determines whether or not any type of access can be made to the segment. The Z8010 has this feature—users are prevented from accessing system segments.

Other types of attributes that can be associated with a segment involve modes of accessing, for example read only, read/write or execute only. For these attributes, the processor must indicate the type of access to be made, be it code fetch, read from memory, write to memory, etc. The Z8001 indicates when it is fetching code, reading or writing data, or performing stack operations, and thus the Z8010 can offer protection for these opera-

tions. The other issue with respect to attributes is whether they are permissive or prohibitive. That is, whether the attribute is in the form of "write to this segment is permitted" or of the form "write to this segment is prohibited." The Z8010 adopts the approach of specifying attributes that prohibit certain types of accessing.

The final issue in the mechanics of memory management systems is the implementation of the protection attributes. These may be associated either with the logical address space or with the physical memory itself. The IBM 360 series, for example, places the memory protection information with the physical memory itself. Thus the processor generates a memory address and the memory module checks to see if the access is permitted. The main difficulty with this approach is in the lack of flexibility, because protection is associated with fixed memory partitions. Also, sharing memory is cumbersome because each user is given a protection key to match the memory key; thus both users must have the same access key or a universal access key. Associating access attributes with the logical segment permits a versatile memory management scheme because different users can access the same segment and have different access attributes associated with their accessing. The Z8010 implements access attributes using the segment mapping information.

Other information associated with each segment does not pertain to the protection mechanism but can be of use to the memory management system. This information generally relates to the history of the segment; for example, whether a segment has been modified while resident in primary memory. If it has not been modified and the system requires the memory for another segment, the memory can be freed immediately; otherwise, the updated version of the segment must be stored in secondary memory and the primary memory is not available until the segment has been saved. Although not strictly necessary, such information can improve the performance of the memory management system. The Z8010 collects information on segment usage, and this information can be used to enhance performance of systems that use this device.

The Z8010 Memory Management Unit

The Z8001 CPU generates segmented addresses consisting of a 7-bit segment number and a 16-bit segment offset address. In addition, the CPU generates status signals indicating its current mode of operation (such as Instruction Fetch, Data Memory Reference, Stack Memory Reference, and Internal Operation), whether it is performing a Read or a Write Memory Reference and whether it is in Normal (User) or System Mode. The Z8010 Memory Management Unit uses this information to perform its memory management functions. This section describes the Z8010 MMU in

some detail, beginning with the translation procedure and continuing with a description of the internal registers of the chip. The section concludes with a description of the system commands that alter the contents of these registers.

The Z8010 MMU has three functional states. The first is the memory management state: when a logical address is presented to the unit, the MMU checks the access to insure its validity and translates the logical address to a physical memory location. The second state is a command state: when a special I/O instruc-

**The Z8010
Memory
Management
Unit**
(Continued)

tion is issued to the MMU, such as reading or writing one of its internal registers, the MMU responds to the command as appropriate. The third state is a quiescent state: when the CPU issues an I/O instruction or a refresh cycle, the MMU address lines remain 3-stated.

The inputs to the MMU are the Address/Data lines (A/D lines), Segment Number lines, Bus Status and Timing Lines, and special control lines for chip selection and DMA. The outputs from the MMU are Address lines, a Segment Trap line and a Suppress line (Figure 4). During address translation and access protection, logical addresses are presented to the MMU on the Segment Number and Address/Data lines; the MMU puts the translated physical memory location on its Address lines and, if appropriate, activates the Segment Trap and/or Suppress lines.

Segment Trap is a special type of synchronous interrupt for the Z8001 CPU; Suppress aborts the memory access. In the command state, the MMU receives commands on the A/D lines; data to be read from or written into the MMU is also placed on the A/D lines.

The MMU selects which of the three states it will be in according to the status information on the Bus Status lines during the initial clock cycle of an instruction or DMA cycle. The MMU performs address translation during a memory reference for either a regular instruction or a DMA request. Only I/O instructions (either regular or special), memory refresh and reserved bus status states cause the MMU to cease performing memory address translations and enter another state.

The MMU uses the segment number to access an internal table of segment descriptor registers, each register containing the starting memory location of the segment (called the base address), the segment's limit (used to determine the range of legal address offsets) and the types of accesses permitted to that segment.

Physical memory for segments is allocated in blocks of 256 bytes. The eight least significant bits of the base address are all zero and are not stored in the Segment Descriptor Register. Also, since the eight low-order bits of the segment base are always zero, the eight low-order bits of the segment offset need not participate

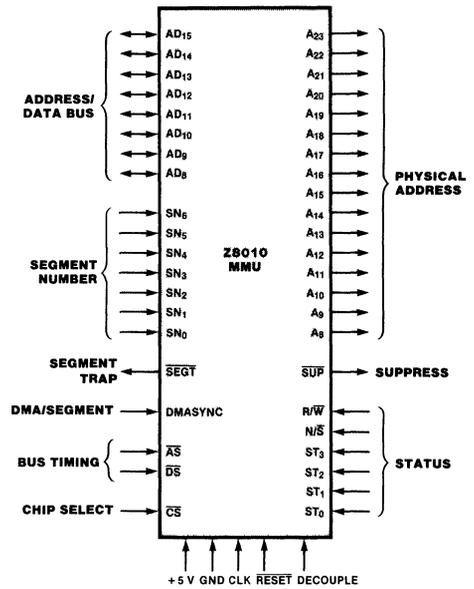


Figure 4. Z8010 MMU Pin Functions

in the addition of the base address to the offset. Rather, they can be juxtaposed to the result of adding the high-order byte of the offset to the most significant 16 bits of the base address.

This process is illustrated in Figure 5. Note that the low-order eight bits of the offset are not used by the MMU. Figure 6 goes through an example of mapping the logical address (5, 1528) to a physical memory location when segment 5 begins at location 231100.

Figure 6a illustrates the full addition to be performed during address translation. The segment number 5 selects Segment Descriptor Register 5 in the MMU. The base address field in this register contains 2311 which corresponds to a base address of 231100. The offset, 1528, is then added to 231100 to produce the physical memory location 232628. Figure 6b represents the same logical operation, but illustrates the actual operation of the MMU. Again segment number 5 is used to select the base address. However, only the high-order byte of the offset is added to the contents of the

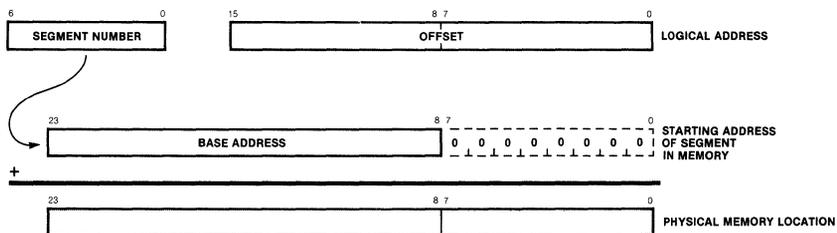


Figure 5. Generation of the Physical Memory Location from a Logical Address

**The Z8010
Memory
Management
Unit**
(Continued)

MMU base-address field: 15 is added to 2311 to produce the most significant 16 bits of the physical memory location. The low-order byte of the physical location is the same as the low-order byte of the offset.

The results of the two processes illustrated in figures 6a and 6b are the same, but in 6a a 24-bit addition is implied whereas in 6b only a 16-bit addition is needed. Also, the low-order eight bits of the offset are not needed by the MMU and this reduces the number of pins required by the MMU package.

The MMU checks memory references for two types of trap conditions. The first type is an access violation. This occurs when a memory reference is performed in a mode that is not allowed by the read-only, execute-only, CPU-inhibit or system-only attribute of a segment. A memory reference outside the allocated memory for the segment also constitutes an access violation.

The second type is a write warning. This occurs when a write is made to the last 256 bytes of a special type of segment (indicated by a special attribute flag called the Direction And Warning Flag). These segments are typically used for stacks and are therefore logically organized so that successive writes (or stack pushes) access lower-numbered memory locations. By generating a segment trap request when a write is performed into the lowest-numbered 256 bytes of the memory allocated for these segments, the MMU is signaling that a stack is in danger of overflowing. The operating system in servicing this trap can increase the memory allocated for the segment and avoid a fatal stack overflow condition.

The MMU generates two control signals that can be used by the system to perform memory management functions. Segment Trap Request is generated upon the first detected occur-

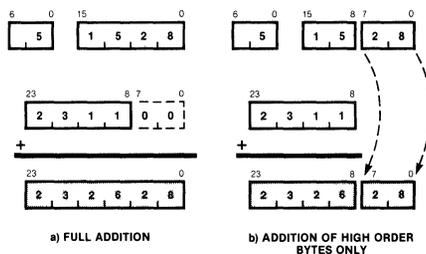


Figure 6. Two Methods of Address Translation

rence of a violation or write warning. Once asserted, this signal remains set until a trap acknowledge signal is received. Only when the Fatal Flag, a special MMU control flag, is set will a detected violation not cause a segment trap request. This flag is set only when a second violation is detected while a previous trap is being processed and thus indicates that the system software is in error.

The other control signal generated by the MMU is Suppress. Once a violation has been detected, this signal is asserted on that and every succeeding memory reference for the remainder of the instruction. In particular, I/O and Special I/O instructions are checked for memory access violations, and once a memory access violation is detected, subsequent memory accesses cause Suppress signals to be generated. I/O addresses, of course, bypass the MMU and are neither translated nor checked. Intervening DMA cycles and memory refresh cycles are exceptions to this rule. During such cycles Suppress is not asserted unless a violation is detected during that cycle. Only DMA can generate a violation; refresh can never cause a violation. Suppress can be used by the memory system to inhibit writes, thus protecting the memory from illegal alterations.

**MMU
Internal
Registers**

There are three groups of registers in the MMU: Segment Descriptor Registers, Control Registers and Status Registers. The Segment Descriptor Registers contain all the information relating to the address translation and access protection of a particular segment. The Con-

trol Registers contain information used to control the various functions of the MMU, including how to interpret various signals generated by the CPU. The Status Registers contain all the information the MMU generates when it detects an access violation.

**Segment
Descriptor
Registers**

Because there are 64 Segment Descriptor Registers in the MMU, two MMUs are required to handle all 128 segments that the Z8001 can manipulate directly. An MMU is programmed to handle either segments 0 through 63 or segments 64 through 127; the particular set of 64 segments in an MMU can be changed using special operating system commands. Each Segment Descriptor contains three fields, a 16-bit Base Field, an 8-bit Limit Field and an 8-bit Attribute Field (Figure 7). The segment number of a logical address determines which

segment descriptors are used in address translation.

The Base Field specifies the starting location in memory of the segment.

The Limit Field specifies the segment size in blocks of 256 bytes. The address offset is compared against the segment limit and a size violation occurs if the offset falls outside the segment boundaries. A write warning occurs if the destination is in the last block of a segment being used as a stack.

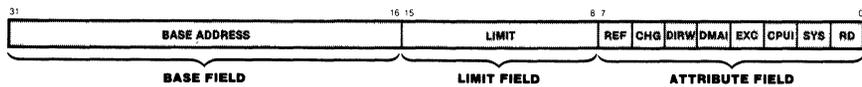


Figure 7. A Segment Descriptor

The *Attribute Field* contains eight flags. Five flags protect the segment against certain types of access, one indicates a special orientation of the segment, and two indicate the types of accesses that have been made to the segment. The following brief description explains how these flags are used.

The *Read-Only Flag (RD)* indicates that the only accesses to this segment are reads. Writes are prohibited when this flag is set. Thus this flag is a write-inhibit flag; in particular, code can be executed from a read-only segment. This flag is useful in protecting data from being written by unauthorized users. For example, if one user wants to give another access to a document that he has created, but does not want this user to be able to modify it, the system can set the Read-Only Flag when it copies the file into the user's address space. If the data is already in memory (in a read-only mode), then this same memory area can be made accessible to that user without another copy of the document being required.

The *System-Only Flag (SYS)* indicates that only accesses made in System Mode are to be permitted. When this flag is set, accesses in the Normal Mode are prohibited. This attribute is useful in protecting system tables and tasks from being accessed by users. For example, system I/O routines can be left in the memory with this flag set and a user is unable to call them directly. This feature is useful if a system is designed so that users are given certain segment names and other segment names are reserved for system use. This flag prevents users from accessing system segments, even though they can generate the logical addresses.

The *CPU-Inhibit Flag (CPUI)* indicates that the segment is not to be referenced by the CPU. When this flag is set, CPU access to this segment is prohibited, but DMA channels can access the segment. This flag is useful in preventing a program from accessing a segment whose data resides on secondary storage and has not been brought into primary memory. For example, a user may request the operating system to read a file from disk into segment number 19; if the operating system returns control to the user before the file has been read, this flag should be set in Segment Descriptor Register 19.

The *Execute-Only Flag (EXC)* indicates that the segment is to be referenced only during the instruction fetch cycle of the processor. When this flag is set, access to the segment during any other cycle of an instruction, for example during the memory request cycle, is

prohibited. This flag is useful in preventing a program from making a copy of a proprietary program. For example, if this flag is set for a segment containing code that a user can access, that code is protected from being read and hence from being copied.

The *DMA-Inhibit Flag (DMAI)* indicates that the segment is not to be referenced by a DMA Channel. When this flag is set, only the CPU has access to the segment. This flag is useful in preventing a DMA device from modifying a segment being used by an executing task. For example, segments with valid data should have this flag set to protect them from modification by a DMA device.

The *Direction And Warning Flag (DIRW)* indicates that memory accesses are to be monitored and certain accesses are to be signaled, although allowed to proceed. When this flag is set, any write to the lowest 256 bytes of the segment generates a write warning. This flag is useful for segments that are used as stacks since the Z8001 has special stack instructions to manipulate stacks that grow toward lower memory locations. Thus a write warning for a stack indicates that the stack may soon overflow its allotted memory space and that more physical memory should be obtained. For example, if a segment serves as a run-time stack for a block-structured programming language such as PASCAL, memory can be allocated to this segment only as a program requires during its execution. The alternative in a fixed allocation environment is to allocate as much memory for the stack as the system expects the program to need, whether or not it is actually used by the program.

The *Changed Flag (CHG)* indicates that a write has occurred to this segment. This flag is set automatically whenever a program or DMA device writes into the segment. This flag is useful in indicating which segments have been modified in the case where the segment must be written to a secondary storage device. Segments that have not been updated need not be copied back to disk if a copy already exists. For example, when a user task is suspended in a multiple-user environment and his task is to be swapped out of memory temporarily to make room for another task, only those segments that have been changed need to be updated on the disk.

The *Referenced Flag (REF)* indicates that a memory access has been made to a segment. This flag is set automatically whenever a program or DMA device accesses the segment. This flag is useful in indicating which segments are active in the case that a segment must be

Segment Descriptor Registers
(Continued)

selected to be swapped out of primary memory to make room for another task. For example, seldom-used operating-system tasks that usually reside in primary memory may be swapped

out to make room for users with large memory requirements. This flag is a way of ascertaining which segments contain seldom used tasks.

Control Registers

Three user-accessible 8-bit registers in the MMU control the functioning of the MMU (Figure 8). The Mode Register provides a sophisticated method for selectively enabling MMUs in a multiple-MMU configuration. The Segment Address Register (SAR) selects a particular segment descriptor to be accessed by a system routine when it is changing the organization of primary memory. The Descriptor Selection Counter Register selects the particular byte in the Segment Descriptor Register that is accessed.

Two flags in the Mode Register govern the functioning of the MMU. The Master Enable Flag (MSEN) indicates whether the device will perform address translation. When this flag is set, addresses translated by the MMU are placed on its Address lines; when this flag is clear, the Address lines are 3-stated. Thus, once this flag is reset, no memory request can pass through the MMU. In a single-MMU configuration, MSEN set to zero requires that the CPU must have access to a special memory, since it will not be able to fetch an instruction from the primary memory. This flag can be set during hardware reset (this is discussed later).

The second flag in the mode register that governs the functioning of the MMU is the Translate Flag (TRNS). This flag indicates whether the MMU is to translate the addresses presented to it. When the flag is set, the MMU translates logical addresses to physical memory locations and checks to see if a violation will occur on that access. When the flag is clear, addresses presented to the MMU are passed to the output Address lines without change, and no protection checking is done.

When multiple-MMUs are used in a memory-management system, some mechanism must be present to select those devices that are to be active during the memory translation process. More specifically, if two MMUs are employed so that all 128 segments can be used at random by an executing process, then some way must exist for each of the MMUs to know which 64 Segment Descriptors are located in its Segment Descriptor Registers. The Upper Range Select Flag (URS) indicates which set of 64 descriptors is stored in the MMU. When the flag is set, the MMU contains descriptors 64 through

127; when the flag is reset, the MMU contains descriptors 0 through 63.

When multiple-MMU devices keep separate tables for system descriptors and user descriptors, the Multiple Segment Table Flag (MST) and the Normal Mode Select Flag (NMS) in the Mode Register distinguish which MMUs contain system descriptors and which contain user descriptors. When the MST flag is set, multiple tables are present in the configuration, and each MMU is dedicated to one of the tables. In this case the MMU translates addresses only when the N/S signal matches the NMS flag. Thus, if there are two tables in the memory management system (one for the system and one for users), the NMS flag is set in those MMUs containing the users' segment descriptors, and is not set in the remaining MMUs. All MMUs in the system have the MST flag set to indicate more than one table in the system.

The final piece of control information in the Mode Register is a 3-bit Identification Field (ID) that indicates a logical name for the MMU. When a segment trap is acknowledged by the CPU, the MMU uses this field to select one of the A/D lines; each enabled MMU should select a different line. If an MMU requested a segment trap, it outputs a 1 on its assigned A/D line; otherwise it outputs a 0. Since the ID field is three bits, up to eight MMUs can be uniquely identified. One instruction might result in multiple violations in different MMUs, so that the segment trap software might have to deal with several MMUs to process the trap.

The other two control registers in the MMU are the Segment Address Register (SAR), which points to one of the 64 segment descriptors, and the Descriptor Selection Counter Register. Commands to read or write a segment descriptor use the SAR pointer to select which descriptor is to be accessed. This register has an auto-incrementing capability for accessing consecutive descriptors in succession without having to reload the SAR. Thus if descriptors 0 through 4 are to be modified, the SAR is initialized to 0 and then auto-incremented to point to descriptors, 1, 2, 3 and 4 in succession.

The Segment Descriptor Number is a 6-bit field that contains the address of the descriptor within the MMU. If the MMU holds segments 64 through 127 (that is, if the URS flag is set), the segment named 64 is accessed when the SAR number field is 0. This is a result of the 6-bit limit of the descriptor number field. The field indicates the 6 least-significant bits of the logical segment descriptor number.

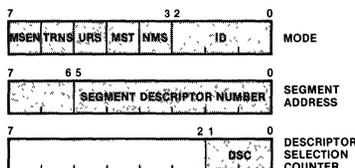


Figure 8. MMU Control Registers

Control Registers
(Continued)

Segment Descriptors consist of four bytes; the Descriptor Selection Counter indicates which byte is being accessed during a command (commands to the MMU can read or write only one byte at a time). A counter value of 0 indicates the high-order byte of the base address is being accessed, 1 indicates the low-order byte of the base address, 2 indicates the limit field, and 3 indicates the attribute field.

This counter is used by MMU commands that access multiple bytes within a descriptor. In general, the counter is handled automatically by the MMU commands. Only when a command could be interrupted—and intervening MMU commands issued—should this register be saved and later restored by the interrupting program.

Status Registers

Six 8-bit registers contain information useful in recovering from memory trap conditions (Figure 9). The Violation Type Register describes the conditions that generated the segment trap. The Violation Segment Number and Offset Registers contain the segment number and upper byte of the segment address offset for the logical address that caused the segment trap. The Instruction Segment Number and Offset Registers contain the segment number and upper byte of the segment address offset for the last instruction before the segment trap was issued. The Bus Cycle Status Register records the status of the bus at the time the trap condition was detected.

Only violations caused by CPU access have trap information stored in the status registers; DMA violations cause Suppress to be asserted, but the Status Registers are not altered. Thus if a DMA violation occurs between a CPU violation and entry to the trap service routine, the service routine still has the CPU trap information available to process the trap. It is the responsibility of the DMA device to save enough information in the event of a violation so that a software DMA violation service routine can process the violation correctly.

Eight flags in the Violation Type Register describe the cause of the segment trap. Four flags correspond to access protection modes in the segment descriptor attribute mode. A read-only violation sets the RDV flag, a system-only violation sets the SYSV flag, a CPU access to a CPU-Inhibit segment sets the CPUIV flag, an execute-only violation sets the EXCV flag.

Three flags correspond to addressing violation or warnings. The Segment Length Violation Flag (SLV) is set whenever the offset of the logical address falls outside the memory space allocated to the segment. The Primary Write Warning Flag (PWW) is set whenever a write occurs in the last 256 bytes of a segment whose Direction And Warning Flag is set (that is, for segments being used as stacks where the top of the stack is within 256 bytes of the allocated memory space of the segment). The Secondary Write Warning Flag (SWW) is similar to the PWW flag, only it is set when the CPU is in system mode, a stack push is being performed to a segment with a Direction And Warning Flag set, and some other addressing violation or warning has occurred (the EXCV, CPUIV, SLV, SYSV, RDV or PWW flags have been set). When the SWW flag is set it indicates

that the system stack is in danger of overflowing its allotted memory. Once the SWW flag is set, further write warnings are suppressed. This prevents the system from repeatedly being interrupted for the same warning while it is in the process of eliminating the cause of the warning.

The final violation-type register flag to be discussed is the Fatal Condition Flag (FATL). This flag is set when any other flag in the violation type register is set and either a violation is detected or a write-warning condition occurs in normal mode. This flag is not set during a stack push in system mode that results in a warning condition. This flag indicates that a memory access error has occurred in the trap processing routine. Once this flag has been set, no Trap Request signals are generated on subsequent violations. However, Suppress signals are generated on this and subsequent CPU violations until the FATL flag has been reset.

The Bus Cycle Status Register contains information pertaining to the status of the bus when a trap condition is detected. This includes CPU Status (ST₀-ST₃), plus flags indicating whether a read or a write was being performed and whether or not the N/S line was asserted.

The Violation Segment Number and Offset Registers record the first logical address to cause a trap. Only the high-order byte of the offset is saved, however, so that external support circuitry is needed to save the low-order eight bits of the logical address offset. If the trap occurred during the instruction fetch cycle, this information is the logical address of the instruction; otherwise it indicates the

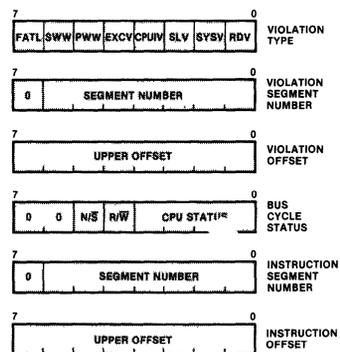


Figure 9. MMU Violation Information Registers

Status Registers (Continued)	<p>logical address of a data item which was to be accessed.</p> <p>The Instruction Segment Number and Offset Registers record the logical address of the last instruction fetch that occurred before the trap. Only the high-order byte of the offset is saved, however, so external support circuitry is needed to save the low-order eight bits of the offset.</p> <p>If an instruction fetch caused the trap, these</p>	<p>registers indicate the logical address of the previous instruction. Such information is useful if the preceding instruction was a branch instruction to an invalid address since—in this case—these registers indicate which branch instruction led to the erroneous situation. If a data reference caused the segment trap, then these registers indicate the logical address of the instruction that specified the illegal access.</p>
Stack Segments	<p>Segments are specified by a base address and a range of legal offsets to this base address. On each access to a segment, the offset is checked against this range to insure that the access falls within the allowed range. If an access outside the segment is attempted, a Trap Request and a Suppress signal are generated.</p> <p>Normally the legal range of offsets within a segment is from 0 to $256N + 255$ bytes, where $0 \leq N \leq 255$. (N is the value in the limit field of the segment descriptor.) However, a segment may be specified so that legal offsets range from $256N$ to $65,535$ bytes, where $0 \leq N \leq 255$. The latter type of segment is useful for stacks because the Z8001 stack-manipulation instructions cause stacks to grow toward lower memory locations. Thus, when a stack grows to</p>	<p>the limit of its allocated segment, additional memory can be allocated on the correct end of the segment. As an aid in maintaining stacks, the MMU detects when a write is performed to the lowest allocated 256 bytes of these segments and generates a Trap Request. No Suppress signal is generated so the write is allowed to proceed. This write warning can then be used to indicate that more memory should be allocated to the segment.</p> <p>The DIRW flag indicates that a segment is to be treated in this special way by the MMU. When the DIRW flag is set, the range of allowed offsets is from $256N$ to $65,535$ bytes and writes into the range $256N$ to $256N + 255$ generate Segment Trap but not Suppress, indicating a write warning.</p>
Segment Trap and Acknowledge	<p>The Z8010 MMU generates a Segment Trap whenever it detects an access violation or a write warning condition. In the case of an access violation, the MMU also activates Suppress. Suppress can be used to inhibit memory writes and to request that special data be returned on a read access. Segment Trap remains Low until a Trap Acknowledge signal is received. If a violation occurs, Suppress is asserted for that cycle and all subsequent CPU memory references until the end of the instruction. Intervening DMA cycles are not suppressed, however, unless they generate a violation. Violations detected during DMA cycles cause Suppress to be asserted during that cycle only; no segment trap requests are ever generated during DMA cycles. This is because the CPU would not be able to respond to these traps until the conclusion of the DMA cycle.</p> <p>Segment traps to the Z8001 CPU are handled similarly to other types of interrupts. To service a segment trap, the CPU enters a segment trap acknowledge cycle. The acknowledge cycle is always preceded by an instruction fetch cycle that is aborted. The MMU has been designed so that this dummy instruction fetch cycle is ignored. During the acknowledge cycle, all enabled MMUs use the Address/Data lines to indicate their status. An MMU that has generated a Segment Trap request outputs a 1</p>	<p>on the A/D line associated with the number in its ID field. An MMU that has not generated a segment trap request outputs a 0 on its associated A/D line. A/D lines for which no MMU is associated remain 3-stated. During a segment trap acknowledge cycle, an MMU uses A/D line $8 + 1$ if the content of its ID field is 1.</p> <p>Following the acknowledge cycle, the CPU automatically pushes the program status words and program counter onto the system stack, and loads a new program status word and program counter from the program status area. The Segment Trap line is reset during the segment trap acknowledge cycle, and no Suppress signal is generated during the stack push. If the store creates a write warning condition, a segment trap request is generated and is serviced at the end of the context swap; the SWW flag is also set. Servicing this second Segment Trap request also creates a write warning condition, but—because the SWW flag is set—no Segment Trap request is generated. If a violation rather than a write warning condition occurs during the context swap, the FATL flag is set rather than the SWW flag. In this case, subsequent violations cause the Suppress to be asserted but not Trap Request. Without the SWW and FATL flags, trap processing routines that generate memory violations would repeatedly be interrupted and called to pro-</p>

Segment Trap and Acknowledge (Continued) cess the violations they create. The CPU routine to process a trap request should first check the FATL flag to determine if a fatal system error has occurred. If not, the

Commands to the MMU When a memory management system must read or change information in the MMU to respond to a segment trap or to re-organize the physical memory, it can issue control commands to the MMU. These commands fall into two generic categories: reset commands and read/write commands. Reset commands are simply orders to the MMU to set or clear specified fields. For these commands, the Z8001 Special I/O output command can be used with the destination field set to be the MMU command code corresponding to the desired action.

Read and write commands are slightly more complicated because they consist of both commands and data. Such commands to the MMU are issued using the Z8001 Special I/O instructions. These instructions have a source and a destination field. For an input instruction, the source field contains an MMU command code and the destination field indicates where in primary memory the data is placed. For an output instruction, the destination field contains an MMU command and the source field indicates where the data to be written into the MMU resides in memory.

The high-order byte of the command contains the opcode for that command; the low-order byte of the command can be used to specify the particular MMU to be accessed. The MMU does not receive information on AD₀-AD₇, so external circuitry must decode information on these lines during the Special I/O commands and then select a particular MMU. The encoding of the low-order byte is dependent upon the system implementation. This paper always uses the convention that bit 1 specifies MMU number 1.

The reset commands to the MMU are: Reset Violation Type Register, Reset SWW Flag In Violation Type Register, and Reset Fatal Flag In Violation Type Register. Resetting the Violation Type Register is similar to a hardware reset in that it clears this register and returns the internal control of the MMU to an initial state (as if no violation had occurred since system initialization). Resetting the SWW flag or the FATL flag in the Violation Type Register clears these flags.

Two other commands are similar to reset commands in that they have no data associated with them. These are Set All CPU-Inhibit Flags in the segment attribute fields and Set All DMA-Inhibit Flags in the segment attribute fields, both of which cause all segment

SWW flag should be checked to determine if more memory is required for the system stack. Finally, the trap itself should be processed and the violation type register reset.

descriptors in the MMU to have the CPU1 or DMA1 flags set, respectively. These two set commands can be useful in initializing address translation tables or when swapping between tasks. For example, when swapping between tasks the Set All CPU1 Flags command automatically makes the previous task's segments inaccessible to the next task, unless the system explicitly initializes the segment attribute field in these segments.

As an example of using the Special Output instruction SOUT to control an MMU, consider resetting the fatal flag of MMU #1. The MMU command opcode for this is "%14" (% denotes hexadecimal). The assembler syntax for the SOUT instruction is "SOUT destination field, source field" so that the instruction to reset the fatal flag of MMU #1 is "SOUT %1402, R0." Specifying register 0 in this instruction is an arbitrary choice—the content of this register is placed on the A/D lines during the data phase of the SOUT instruction, but it is ignored by the MMU. The low-order byte of the command (the destination field of the instruction) encodes which MMU is to reset its fatal flag. The convention followed in this paper is that MMU 1 is specified by setting bit 1 in the low order byte of the command. (Bit 1 set is hex "%02.")

The rest of the MMU commands consist of both operation and data. The following internal registers can be read or written: the Mode Register, the Segment Address Register, the Descriptor Registers and the Descriptor Selection Counter Register. A Descriptor Register can be read or written as a whole, or selected subfields can be accessed. In addition, by using the auto-increment feature of the Segment Address Register, successive Descriptor Registers can be accessed, or a selected field within successive Descriptor Registers can be accessed. For example, one Special I/O command in block mode could read a number of segment attribute fields. This is useful in determining which segments have been modified.

As an example of using the Special Output instruction SOUT to write data into an MMU, consider writing the contents of Register 6 into the Mode Register of MMU #2. The opcode for this command is "%00" and so the command is "SOUT %0004, R6." Here the high-order byte of the destination field contains the opcode and the low-order byte has bit 2 set (hexadecimal 4 if 0100 in binary) indicating MMU #2.

Commands to the MMU
(Continued)

Certain MMU internal registers can only be read—there is no corresponding write instruction. This is because these registers contain information relating to a detected violation and thus it is not necessary to be able to write into these registers. These registers are the Violation Type Register, the Violation Segment Number Register, the Violation Offset Register,

the Instruction Segment Number Register, the Instruction Offset Register and the Violation Bus Status Register. Although the Violation Type Register cannot be written, it should be noted that it can be cleared and that two of its flags can be individually cleared: the SWW flag and the FATL flag.

Direct Memory Access

DMA operations may occur between Z8001 machine cycles and can be handled through the MMU. The MMU permits DMA in either the System or Normal Mode of operation. For each memory access, segment attributes are checked and—if a violation is detected—a Suppress signal is generated. Unlike a CPU violation, which automatically causes Suppress signals to be generated on subsequent memory accesses until the next instruction, DMA violations generate a Suppress only on a per-memory-access basis. The DMA device should note the Suppress signal and record sufficient information to enable the system to recover from the access violation. No Segment Trap Request is ever generated during DMA (hence warning conditions are not signaled). There are no trap requests because the CPU would not acknowledge the request until the end of the DMA cycle.

At the start of a DMA cycle, the DMASync line must go Low, indicating to the MMU the beginning of a DMA cycle. A Low DMASync inhibits the MMU from using an indeterminate segment number on lines SN₀-SN₆. When the DMA logical memory address is valid, DMASync must be High on one rising edge of Clock and the MMU then performs its address-translation and access-protection functions. Upon the release of the bus at the termination of the DMA cycle, DMASync must again be High. After two clock cycles of DMASync High, the MMU assumes that the CPU has control of the bus and that subsequent memory references are CPU accesses. The first instruction fetch occurs at least two clock cycles after the CPU regains bus control. During CPU cycles, DMASync should always be High.

Hardware and Software Reset

The MMU can be reset by either hardware or software mechanisms but note that they have different effects. A hardware reset occurs on the falling edge of the Reset input; a software reset is performed by an MMU command. A hardware reset clears the Mode Register, Violation Type Register and Descriptor Selection Counter. If the Chip Select line is Low while Reset is Low the Master Enable Flag in the Mode Register is set to 1. All other registers are undefined. After reset, the A/D and A lines are 3-stated. The $\overline{\text{SUP}}$ and $\overline{\text{SEG}}$

open-drain outputs are not driven. If the Master Enable Flag is not set during reset, the MMU does not respond to subsequent addresses on its A/D lines. To enable an MMU after a hardware reset, an MMU command must be used in conjunction with Chip Select.

A software reset occurs when the Reset Violation Type Register command is issued. This command clears the Violation Type Register and returns the MMU to its initial state as if no violations or warnings had occurred.

Multiple-MMU Configurations

Z8010 MMU architecture supports system configurations that use more than one MMU. Multiple MMU devices can be used either to manage 128 CPU segments rather than the 64 supported by one MMU, or to manage multiple translation tables.

The Z8001 CPU generates logical addresses that can specify up to 128 different segment names. Because the MMU contains only 64 Segment Descriptor Registers, two MMUs are needed to perform address translation for 128 logical segments. Systems designed with only one MMU device still have the power and flexibility offered by memory management, although tasks in such a system are restricted to manipu-

lating only 64 logical segment names. These names must either be 0 through 63 or 64 through 127. If the MMU in a single-MMU configuration is set to translate segment names in one range and the CPU generates a logical segment name in the other range, the MMU does not perform address translation and no physical memory location is output. In this case, no request is made to memory. Therefore, a single-MMU configuration should have additional external logic to detect erroneous segment names and generate a Segment Trap and Suppress signal.

The Upper Range Select flag (URS) is used in multiple MMU configurations to indicate which group of logical segment names

Multiple-MMU Configurations

(Continued)

are to be translated by an MMU. When this flag is set, the Segment Descriptor Registers in the MMU are used in translating logical addresses in the range 64 through 127. When the flag is clear, the range is 0 through 63. Thus the URS flag corresponds to the most significant bit (bit 6) in the logical segment names that the MMU translates. Because this flag is under program control, the range of logical segment names can be changed during execution in System Mode.

MMU architecture also supports multiple segment translation tables. This feature is useful when separate tables are maintained for different tasks. Each task has its own table and switching between tasks requires enabling the appropriate MMU devices. In contrast, systems with only one translation table must either restrict the logical segment names that an individual task can use, or change the Descriptor Register entries whenever tasks are swapped. Two flags in the Mode Register, together with the N/\bar{S} signal, are used in multiple table configurations.

The Multiple Segment Table (MST) flag indicates whether the configuration is being used to support multiple tables. When this flag is set, the MMU will compare the N/\bar{S} line against the Normal Mode Select Flag (NMS) before generating a physical memory location on its Address lines. When the line and the flag match (both asserted or both de-asserted), the MMU is enabled and an address translation is performed (assuming the URS flag matches the most significant bit in the logical segment

name). If the N/\bar{S} line fails to match the state of the NMS flag, no translated address is generated by the MMU. The MST flag and the NMS flag are under program control and can be changed in System Mode.

The simplest multiple translation table configuration has one table for Normal Mode access and one for System Mode access. In such a configuration, the Multiple Table Flag is set in all MMUs and the N/\bar{S} line of each MMU receives its input from the N/\bar{S} output of the Z8001 CPU. MMUs containing descriptors of system segments have the NMS flag clear, and those containing descriptors to be used in Normal Mode have the flag set. When the Z8001 is in System Mode, the N/\bar{S} line is Low and it matches the NMS flag in those MMUs whose Descriptor Registers contain system segment information. Therefore, these MMUs are used in address translation for system references.

When the Z8001 is in Normal Mode, the N/\bar{S} line is High and it matches the NMS flag in those MMUs whose Descriptor Registers contain user segment information. Consequently, these MMUs are used in address translation for user segments. In this configuration, system segments are separated from user segments. When the Z8001 changes from Normal to System Mode of operation, the appropriate translation table is automatically selected. A more elaborate example of a configuration with multiple translation tables is given in the next section.

Examples

This section describes two Z8001-Z8010 configurations: one contains two MMUs and one address translation table; the other contains seven MMUs and four address translation tables. These examples are given in sufficient detail to illustrate some of the major ideas in constructing memory-management systems around the Z8010 MMU. High-level block diagrams illustrate some of the major features of typical hardware configurations and short programs illustrate software techniques for using the MMU.

The first example system is the two-MMU configuration illustrated in Figure 10. The two MMUs are called MMU #1 and #2, and they are selected during a command cycle by AD_1 and AD_2 being Low, respectively. Since a Special I/O instruction is being used bit 0 must always be zero. Thus, when a low-order byte of a command is "%02," MMU #1 responds; when it is "%04," MMU #2 responds; and when it is "%06," both MMUs respond. (Note that AD_1 is inverted before attachment to the \bar{CS} pin.)

The A/D_1 line, which controls MMU #1 through the Chip Select input, is first com-

bined with the Reset line. This allows the Master Enable Flag to be set upon system initialization, so the logical addresses generated by the CPU are passed to the physical memory. This is done because—upon reset—the mode register is otherwise cleared, the Translate Flag is clear and addresses pass through the MMUs untranslated. The bootstrap program can therefore reside in absolute memory locations in the physical memory. If the Reset line is not an input to the Chip Select line, the Master Enable Flag would not be set during system initialization and the CPU would not be able to address memory through the MMUs.

Note that there is a direct path from the CPU and DMA to the system bus. This path is used during I/O and memory refresh because the MMUs are quiescent during these cycles. It is also used for data on memory reads and writes. Also, note that the Suppress line goes both to the memory, where it can be used to protect the memory from erroneous

Examples
(Continued)

writes, and back to the DMA device to save information upon the event of a DMA access error.

Of further interest in the example, address latches are used to buffer addresses between the Z8001 and a demultiplexed bus. This is required to demultiplex the address and data onto the bus. The address latch for AD₈-AD₁₅ may not be needed if the I/O device does not use separate address and data lines.

A detailed example indicates how such a system could be used. First, consider setting Segment Descriptor Register 65 to point to a read-only segment of 768 bytes starting at memory location %115200. The segment is to be accessed in Normal Mode. The Descriptor Register should be %115202 01. The first two bytes, %1152, indicate the starting location of the segment (note that the low-order byte of the memory address is all zeros and is not stored in the Descriptor Register). The third byte, %02, indicates that three blocks of 256 bytes have been allocated to this segment. The fourth byte, %01, indicates that only the read-only segment flag has been set.

To write this descriptor into the MMU, a copy of the descriptor should be created in primary memory and a Special I/O block transfer instruction used. The SOTIRB instruction can be used for this.

This instruction has the assembler syntax "SOTIRB destination, source, count register" where both the destination and source are registers. The destination register contains the command to the MMU, the memory location pointed to by the source register contains the first byte of the data to be transferred, and the Count Register contains the number of bytes to be transferred.

The opcode to load the Descriptor Register is "%0B". Segment Descriptor Register 65 is Segment Descriptor Register 1 of MMU #2, so the MMU command is "%0B04".

To specify which Segment Descriptor Register to write, it is necessary to load the Segment Address Register of MMU #2 with 1. The MMU opcode to do this is "%01" and so the command is "%0104." The segment number (in this case 65) is a parameter to the example routine, passed in register 0. The

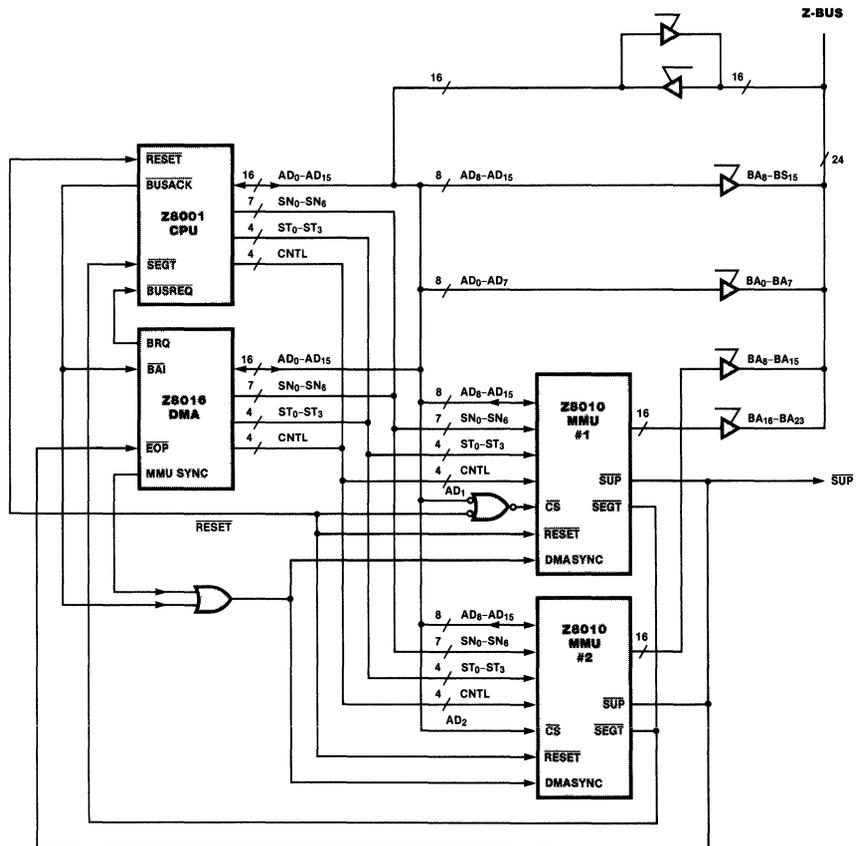


Figure 10. A Dual-MMU Configuration

Examples (Continued)	Instruction	Parameters	Description
	BIT	R0, #6	!Test to see if Descriptor Register is in MMU #1!
	JR	Z, OVER	!or MMU #2!
	SOUTB	%0104, RH0	!Set SAR in MMU #2!
	LD	R1, #%0B04	!Prepare to write descriptor!
	JR	NEXT	
OVER:	SOUTB	%0102, RH0	!Set SAR in MMU #1!
	LD	R1, #%0B02	!Prepare to write descriptor!
NEXT:	LD	R0, #4	!Load count field—4 bytes!
	SOTIRB	@R1, @RR2, R0	!Write descriptor!

descriptor to be written is another parameter to this routine: RR2 contains the address in memory where this information resides. The SOUTB instruction has a similar syntax to the SOTIRB instruction explained previously except that it writes one byte instead of a series of bytes, and the destination I/O address is in the instruction itself instead of in a register specified by the instruction.

The routine on this page initializes the Segment Descriptor. Its parameters are found in Register R0, which contains the segment number to be written, and in Register RR2, which points to the descriptor information in primary memory. Registers R0 through R3 are used by this routine.

Now suppose that the user tries to write into location <<65>>%9328. This causes a segment trap both because of the write to a read-only segment and because the access exceeds the segment limit. At the end of the instruction that has the illegal memory access, the CPU acknowledges the trap. During the trap acknowledge cycle, MMU #2 asserts AD₁₀ (assuming its ID field is "010") and this information is placed on the system stack for the

trap-handling routine.

The trap-handling routine reads the violation information registers from the MMU. The violation type register contains "%05" indicating both a length violation and a read-only violation. The Violation Bus Status Normal Register contains "%28". The first nibble indicates a write in Normal Mode was in progress and the second nibble indicates a memory data access cycle was in progress. The violation segment register contains "%41" indicating segment 1 of MMU #2 caused the violation (which is segment number 65), and the violation offset register contains "%93" indicating the high-order byte of the logical address offset. The operating system can then issue an error message to the user indicating a read-only violation to segment 65. Using the program counter that was stacked when the segment trap was acknowledged, the system can also indicate the next instruction that was to be executed. Note that in this system the low-order byte of the violation offset is lost. This condition is corrected in the next example system.

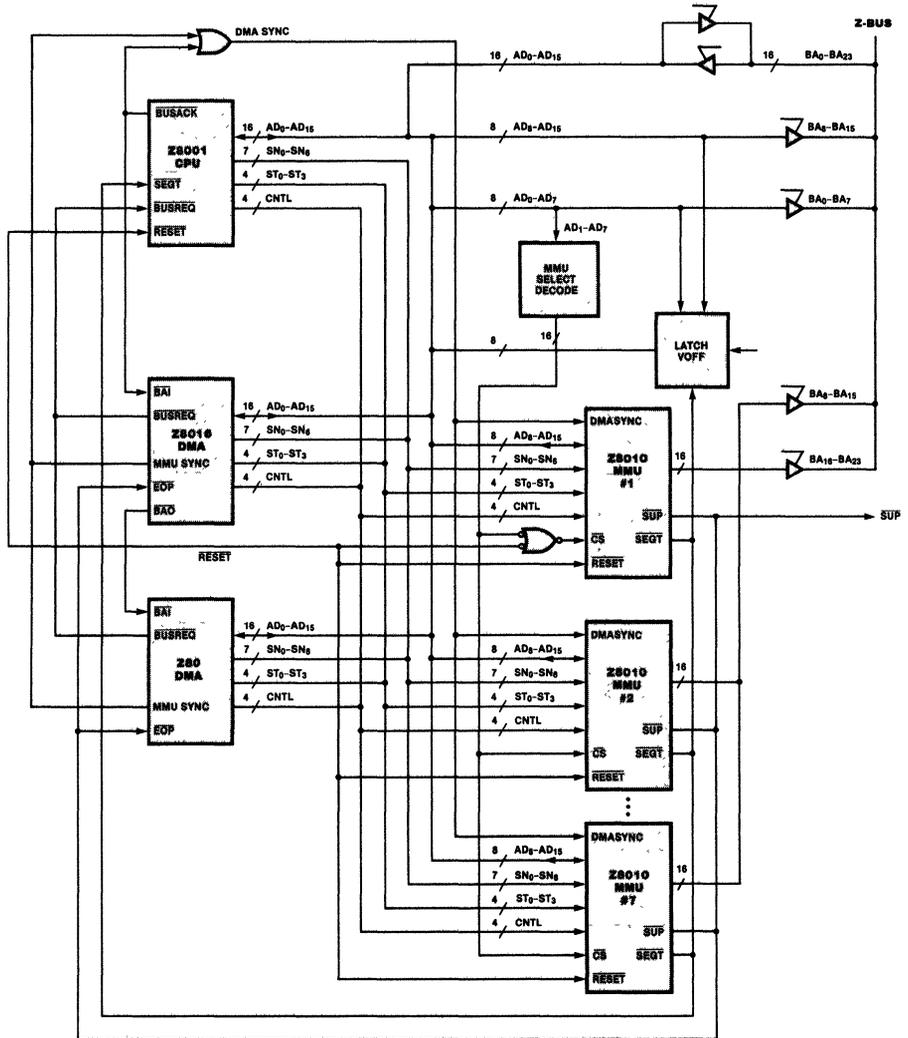


Figure 11. 16-MMU Configuration

Figure 11 gives a high-level diagram of the second system to be discussed. This configuration contains 16 MMUs, and the A/D lines select the appropriate MMU when in Command mode. The major innovation in this example, aside from the additional MMUs, is the latch that retains the least significant byte of an address offset when a violation is detected. This latch is enabled when a segment trap is generated by an MMU and holds the low-order byte of the address that generates an access violation.

In addition, external decoding logic for selecting one MMU Chip Select line is indicated. Seven MMUs is the limit in one configuration without additional decoding logic for selecting one MMU Chip Select line. (The reason why AD₀ cannot be used to control an eighth MMU is due to the Special I/O input

convention of the CPU. When the CPU inputs a byte of information and AD₀-AD₇ is asserted, the data is taken from AD₀-AD₇, which are not driven by the MMU.)

Switching Tables in a 16-MMU System.

The 16-MMU configuration can support a memory management system designed with two MMUs permanently allocated to the operating system and the others allocated in pairs to different user tasks. Thus, seven user tasks can have translation tables resident in the 14-user MMUs, and switching between active tasks requires the appropriate MMUs to be enabled and disabled. This selection process can be effected by manipulating the Master Enable (MSEN) flags in the mode registers of the appropriate MMUs.

Examples
(Continued)

The routine performs the selective enabling of MMUs required by a task swap. This routine disables all user MMUs (thus disabling the currently enabled user MMUs), then enables the appropriate pair. (The system pair is always enabled.) The code selecting the new task is passed in register R1; it contains %n, if task n is to be dispatched.

Two peculiarities of this example are worth noting. First, each user ID number corresponds to seven MMUs (for example, all upper-range user MMUs). The Segment Trap processing routine has to take this into account. Second, the Chip Select code is assumed to be as follows:

```

CLR    R0                !Clear R0!
SOUT   %00F8,R0         !Disable all user MMUs by clearing their mode registers!
SLA    R1,#1            !Multiply R1 by 2—the number of bytes in a memory word!
LD     R1,TABLE(R1)     !Get the command word (opcode always %00) for user n,
                        URS = 0!
LDA    RR2,DATA         !Get the new mode register bit pattern (%DA)!
SOUTIB @R1,@RR2,R0     !Send %DA to lower-range MMU and increment RR2 to
                        DATA + 1!
INC    R1,#8            !Command word for URS = 1!
SOUTIB @R1,@RR2,R0     !Send %FB to upper range MMU!
END:
DATA:  BYTES(%DA,%FB)  !Mode register bit patterns!
TABLE: WORDS (%8,%18,%28,%38,%48,%58,%68)

```

Program to Switch Tables

	AD ₀ -AD ₇	MMU Selected
System:	02	#1 ID=0, URS=0
	04	#2 ID=1, URS=1
User 0:	08	#3, ID=2, URS=0
	10	#4, ID=3, URS=1
User 1:	18	#5, ID=2, URS=0
	20	#6, ID=3, URS=1
User 2:	28	#7, ID=2, URS=0
	30	#8, ID=3, URS=1
	.	.
	.	.
User 6:	68	#15, ID=2, URS=0
	70	#16, ID=3, URS=1

It is also assumed that %F8 will select all user MMUs.

MMU Command Summary	Program to Switch Tables		Program to Switch Tables	
	Opcode	Operation	Opcode	Operation
	00	Read/Write Mode Register	0C	Read/Write Base Field And Increment SAR
	01	Read/Write Segment Address Register	0D	Read/Write Limit Field And Increment SAR
	02	Read Violation Type Register	0E	Read/Write Attribute Field And Increment SAR
	03	Read Violation Segment Number	0F	Read/Write Descriptor And Increment SAR
	04	Read Violation Offset (high byte)	10	Reserved
	05	Read Bus Cycle Status Register	11	Reset Violation Type Register
	06	Read Instruction Segment Number	12	Reserved
	07	Read Instruction Offset (high byte)	13	Reset SWW Flag In VTR
	08	Read/Write Base Field In Descriptor	14	Reset FATL Flag In VTR
	09	Read/Write Limit Field In Descriptor	15	Set All CPU-Inhibit Flags
	0A	Read/Write Attribute Field In Descriptor	16	Set All DMA-Inhibit Flags
	0B	Read/Write Descriptor (all fields)	17-1F	Reserved
			20	Read/Write Descriptor Selector Counter Register
			21-3F	Reserved



An introduction to memory management

Once used only on the largest computer systems, memory-management techniques will soon be used on a variety of high-level microprocessor-based systems

by D. Stevenson

The declining cost per bit of memory has led to systems with even larger memories, and the declining cost of logic has led to more powerful processors. Together, these two trends promote the sophisticated use of large memories, based on techniques commonly referred to as *memory management*. Automated memory-management systems date back to the Atlas computer project at Manchester University in the late 1950s. During the 1960s the concept was exploited in a number of time-sharing machines (e.g. the Scientific Data Systems 940, General Electric 645, Digital Equipment Co. PDP-10), and during the 1970s was highly publicised in its manifestation as *virtual memory* (IBM 370). Until fairly recently, memory management has been associated only with large mainframe computers, but with the 1978 introduction of Digital Equipment's VAX 11 'super mini', the concept has invaded the minicomputer market. Now, with the advent of single-chip memory-management units such as that available with the Zilog Z8000 processor, the concept is about to arrive in microprocessor-based systems.

Memory management has two functions: the efficient allocation and reallocation of memory space to executing tasks so as to optimise overall memory usage; and the protection of memory contents from unintended or unauthorised accesses by executing tasks. To keep overall memory usage optimised as demands on memory constantly change, dynamic relocation of tasks during their execution may be necessary, and this is accomplished by an address-translation mechanism. The restriction of memory access to prevent unintended or unauthorised accesses is accomplished by memory-attribute checking. Both operations occur with each memory access made during the

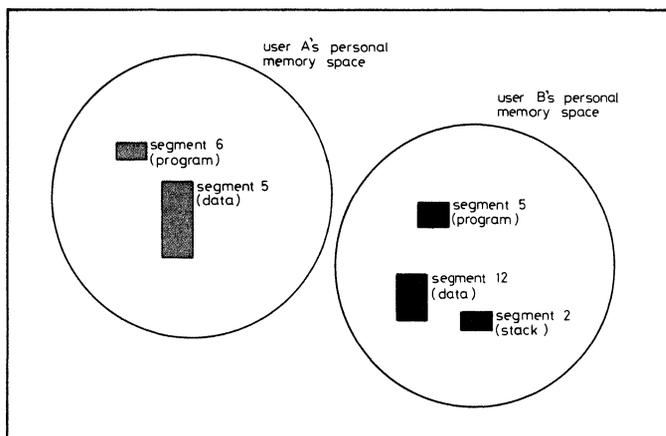
execution of a program, and both are transparent to the user.

Address translation simply means treating the memory addresses generated by the program as *logical* or *virtual* addresses to be *translated* into actual physical-memory addresses before dispatching the memory-access requests to the memory unit. Memory-attribute checking means that each area of memory has associated with it information as to which tasks can access it and what types of access can be made by each task. Each memory reference is checked to ensure that the task has the right to access that location in the given fashion (for example, to read the contents of the location or to write data to that location).

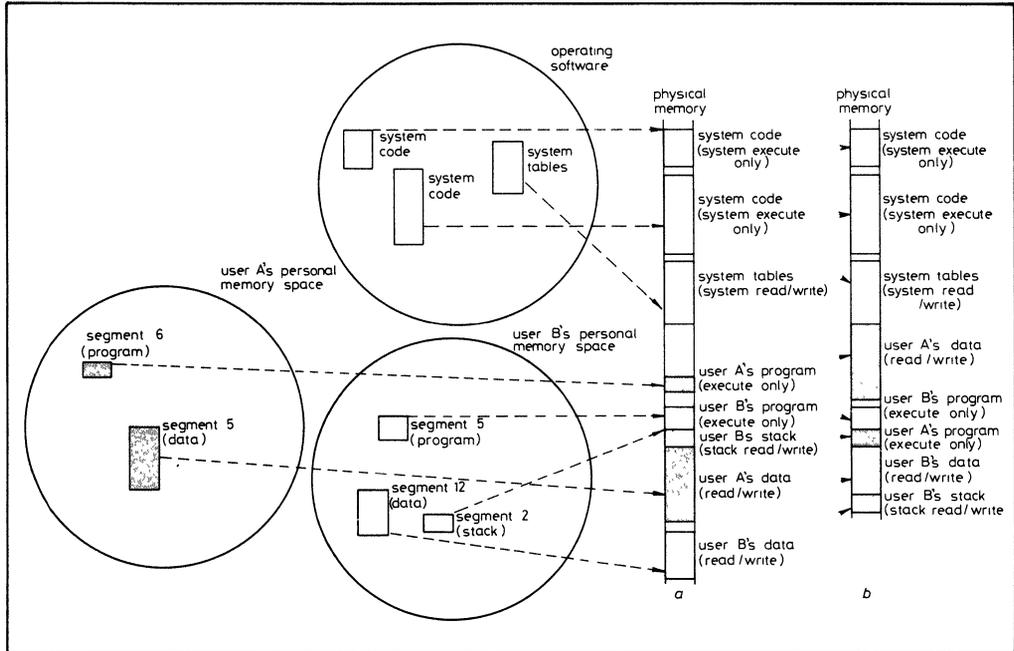
Instead of a conventional linear address space, more elaborate memory-

management systems simulate a hierarchical memory structure in which the memory consists of a collection of distinct memory areas, called segments. Access to this structured memory requires the specification of a segment and of an offset within that segment. Thus, instead of specifying, say, memory location 1050 in a linear address space, a task might specify memory location 5 in segment number 23. The actual location of the segment in the physical memory does not concern the task — the actual access is carried out via the address-translation mechanism, which is informed of the actual location of the segment by the operating software.

Generally, segments can be of variable size, within limits, and a user can specify the size of each segment to be used. Thus one user may be allocated



1 In a multiuser system, each user is aware of only those memory segments in his own 'personal' logical-memory space, and does not know where the segments are located in the system's physical memory



2 Transparently to the users, the system's operating software 'maps' each user's (and its own) logical memory space into the physical memory, also applying memory-protection attributes to each segment. If changing demands on memory space make the original mapping (a) no longer optimum, the system can dynamically relocate segments (b), again transparently to the users

two segments, one of 2000 words for his Fortran program, and the other of 10 000 words for his data. Another user might be allocated three segments, of 3000, 6000 and 2000 words, respectively, for her Pascal program, data, and runtime stack. If the first user called his data segment 'segment 5', then the first word in his data set would be accessed by the logical address (5,0), indicating segment 5, offset 0. The memory-management system then translates this symbolic name into the correct physical-memory address.

Fig. 1 gives a conceptual realisation of

these two users' logical program spaces. The first user, user A, has his program segment called 'segment 6' and his data segment called 'segment 5'. The second user, user B, has her program segment called 'segment 5', her data segment called 'segment 12' and her stack segment called 'segment 2'. Notice that both users have named one of their segments 'segment 5', but they refer to different entities. This causes no problem since the system keeps the two memory areas separate. The situation is analogous to both users having an integer variable called 'I' in their programs: the system

realises that these are two separate variables stored in different memory locations.

User A's data segment, 'segment 5', is 10 000 words long. If he tries to reference word 10 050 of segment 5, he gets an error message from the operating software indicating that he has exceeded the allocation limit for segment 5. Note that he does not accidentally access word 50 of segment 6; i.e. segments are logically distinct and unordered. A reference to one segment cannot inadvertently result in access to another segment. Thus, in this example,

A virtual end to microprocessor memory limits

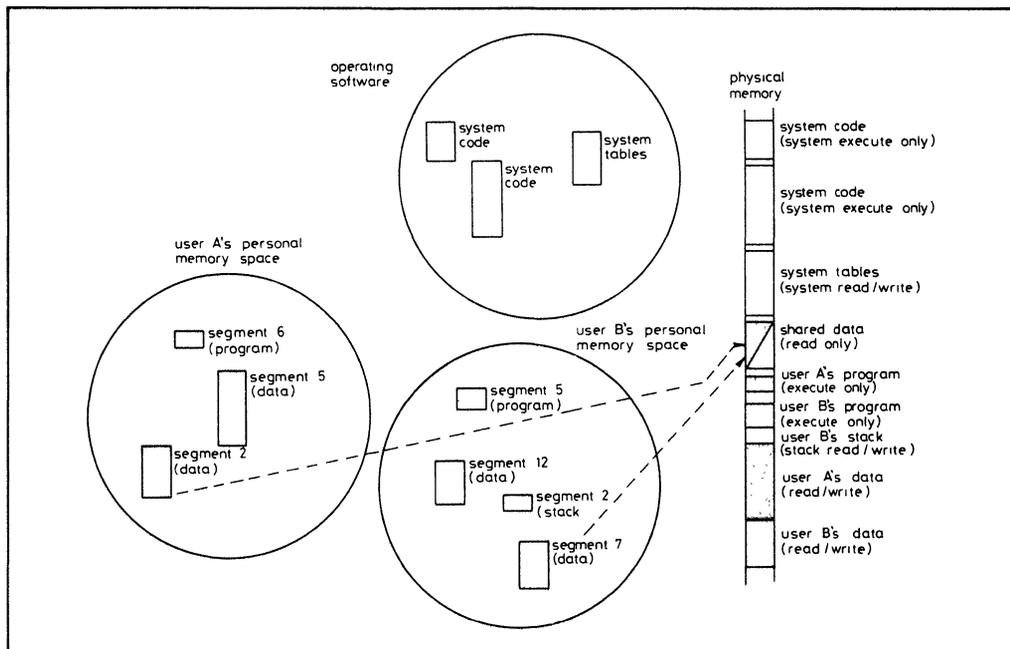
All the 'super microprocessor' designs are based on the need to reduce software costs by facilitating programming, and one of the major causes of difficulty in programming any computer-like device is limitations on the size of available memory. By removing the 64 kbyte limit imposed by earlier devices, the 'super micros' have eased this problem, but experience with larger, usually mainframe, computers suggests that the user's programming task will be simplified even more when, in the near future, microprocessors become capable of supporting *virtual-memory* operation, which effectively removes all practical limits on memory size.

Virtual-memory operation works by using relatively inexpensive *secondary* memory, such as that provided by magnetic discs, to supplement the system's *primary* memory, which is necessarily built from relatively expensive random-access-memory components. However, since any program or item of data can only be accessed by the processor when held in the random-access primary memory, this can only be done by continuously 'swapping' blocks of program code or data between the two memories. This swapping is carried out automatically by the processor's

hardware and operating software, so that the operation of the whole virtual-memory system is transparent to the user, who is aware only of having access to an extremely large personal memory space.

As example of the use of virtual-memory operation is given by Digital Equipment's VAX-11 advanced-architecture version of its PDP-11 minicomputer. As a full 32 bit machine, the VAX-11 is capable of addressing over 4.3×10^9 bytes of memory, and the virtual-memory system operates to effectively give each process, no matter how many are concurrently active, access to over 10^9 bytes of 'private' memory space. In practice, of course, not even the most demanding application requires any of its processes to have access to this huge amount of memory: thus the aim of removing all practical limitations on memory size has been achieved.

The VAX-11 processor implements its virtual-memory system by a relatively simple paging technique. The whole address space is divided into 512-byte 'pages' that can be swapped independently between primary and secondary memory. Each logical address used in the system is composed of a 23-bit *page number* and a 9-bit *offset*. At each attempted access,



3 The memory-management system can also allow segments to be shared between users, such as a shared data segment meant for input to more than one user program. To prevent any one user from altering the shared data, the system marks the segment 'read only'

user A is prevented from accidentally (or deliberately) accessing his program as though it were part of his data segment.

Fig. 2a illustrates one way that the operating software could arrange these segments in the physical memory. If demands on physical-memory space were to change, however, the operating software could dynamically relocate the segments (e.g. as shown in Fig. 2b), the relocation being completely transparent to the two tasks. In each case, the arrows indicate the address-translation or *memory-mapping* functions from the

logical-address space of the users to the physical-memory locations allocated to them. The Figure also indicates the access attributes associated with each user's segments. For example, program segments are 'execute only' and data segments are 'read/write'. Thus a user is prevented from executing a data segment or writing into a code segment.

Fig. 3 illustrates what happens when both users have access to the same data set in primary memory, say the results of a questionnaire that both intend to analyse. Each user has a logical name associated with that data set to specify

the segment in which the data set is to reside. Note that the two users have chosen to put the data set in different segments of their personal address spaces. The memory-mapping system translates these different segment names to the same physical memory locations. Thus user A's access to address (2,17) references the same physical memory location as user B's access to address (7,17). The shared data segment is marked 'read only' to prevent either user from deliberately or accidentally changing the data.

Before proceeding to the mechanism

the system's memory-management unit checks to see if the desired page is in the primary memory, and if so translates the logical address to the appropriate physical address. If the page is not in primary memory, it is 'swapped in' from the disc. All this is relatively straightforward — the complication comes in during the design of the 'paging algorithm' by which the operating software decides which page currently in primary memory can most readily be 'swapped out' to free space for the incoming page. The efficiency of the whole system depends critically on the choice of the correct swapping algorithm, and in computer-science terms this choice is 'non-trivial', or in other words extremely difficult.

With the forthcoming announcement of the National Semiconductor 16000 'super micro' range, virtual-memory operation of this kind will become feasible in microprocessor systems for the first time. The NS16082 memory-management unit (m.m.u.), which will act as a coprocessor to the NS16000 main processor, will support a paged system of virtual-memory operation rather like that used on the VAX-11. Because of the NS16000's use of 24-bit addresses, each virtual-memory space will be initially limited to only 16 Mbytes, but later expansion should increase this substantially. The m.m.u. will provide fast

associative storage for active address-translation tables within its internal memory, using a cache approach to ensure that only 5% of accesses require reference to the full translation tables stored in primary memory. On detecting that a required page is not in primary memory, it will send an 'abort' message to the main processor, which is equipped with a special hardware mechanism to 'roll back' its state to what it was at the start of the aborted instruction. National Semiconductor claims that, with these features, and with an appropriate operating system to control them, the design of a full virtual-memory system should not be significantly more difficult than the design of any other microprocessor-based system.

Will such virtual-memory microprocessor systems ever become widely used, however? One development that may make them extremely attractive is that of denser, less expensive, magnetic-bubble stores. A relatively inexpensive system could then be based on a 'super micro', a relatively small (say 128 kbyte) primary memory, and 1-2 Mbyte of fast non-volatile bubble storage. Such a system, occupying a single board, might well exhibit a performance approaching that of traditional mainframe systems.

DENNIS MORALEE

Memory management comes to micros

A survey of recent product announcements reveals that the microprocessor manufacturers all agree on the importance of providing memory-management facilities for the next generation of microprocessor-based systems. All the new-generation 'super microprocessors' have been designed with the use of memory management in mind, and existing microprocessor families, such as the Texas Instruments 9900 range, are being extended by the provision of 'add-on' memory-management units. Although it might at first seem that the use of memory-management techniques could only be justified in specialised high-end applications, the low cost of the large-scale-integration hardware and packaged operating software now becoming available may soon make sophisticated memory management a common feature of even relatively modest micro-based systems.

With the exception of Intel, all the microprocessor manufacturers have decided to use the traditional minicomputer approach to providing memory-management facilities, which involves the use of separate memory-management units (m m u s) located between the processor and memory. Using this approach, the m m u accepts memory-access requests from the processor on its input lines, performs address translation and memory-attribute checking as desired, then sends appropriately modified access requests to the memory via its output lines.

Typical of such m m u s is the Z8010 unit designed to provide memory-management facilities to systems based on the Zilog Z8000 microprocessor. In operation, the m m u receives 23-bit logical addresses from the processor, these addresses consisting of a 7-bit *segment number* and a 16-bit *offset*. These logical addresses are then translated into 24-bit physical addresses by using the segment number to address a 64-line table held within the m m u, adding the 16-bit *segment starting address* thus retrieved to the top 8 bits of the logical-address offset, and concatenating the lower 8 bits of the offset with the result of this addition (see Figure). The m m u then sends this 24-bit physical address to memory to complete the access.

The use of a 7-bit segment number enables the Z8000 to divide its memory into a maximum of 128 segments, a second m m u being used in parallel with the first if more than 64 segments are actually going to

be in use at any one time. These segments may be of variable length, up to the maximum of 64 kbytes imposed by the 16-bit size of the offset, and may be located freely within the overall 8 Mbyte memory, subject only to the restriction of the starting address being a multiple of 256 bytes, a restriction imposed by the 16-bit size of the stored segment-starting addresses. Relocation of segments is achieved by the processor changing the contents of the appropriate entries in the m m u's segment-address table, which it does by means of special input/output instructions.

As well as providing for address translation, the m m u also checks the attributes of the addressed segment, which are stored, along with its starting address, in the internal 64-line table. One attribute it always checks is the length of the segment, ensuring that the logical address provided does in fact lie within the declared segment boundaries. Other attributes relate to the type of access allowed, e.g. *execute-only*, *read-only* and *read/write*, and to check that the access being attempted does not violate these memory-protection attributes, the m m u needs to know for what purpose the processor is trying to access the specified segment. This it determines by monitoring four status lines connected to the processor, which indicate, *inter alia*, whether the processor is trying to fetch an instruction from memory, to access data from memory, or to manipulate a memory-based stack. If the attempted access is not allowed by the segment's attributes, the m m u interrupts the processor via a special 'segment trap' line.

A unique feature of the Z8000 design is that the use of the four processor-status lines could be used to divide the system's memory additionally into special-purpose areas each capable of holding only one type of data. Thus, completely separate memories could be provided for programs, data and stacks, and the distinction between user and system operation could double this to a total of six separate memories, each of which could be 8 Mbytes in length. Whether any user would actually want to partition his system's memory in this rather inflexible way, however, remains to be seen.

The operation of the m m u, although based on fast h m o s logic, inevitably results in each memory access suffering a certain additional delay. In the Z8000 system, this delay is minimised by arranging for the

of memory management, it is instructive to review the advantages of using this form of segmented address translation and attribute-based memory protection. The first advantage is that it permits the dynamic allocation of memory during the execution of tasks, i.e. tasks can be located anywhere in memory, and can be relocated as desired while their execution is suspended. The address-translation mechanism provides this flexibility because the task deals exclusively with logical addresses, and hence is independent of the addresses of the physical-memory locations it accesses. Moving the task to different physical-memory locations requires that the address-mapping function be changed to reflect the change in physical memory location, but the task's code need not be modified. Of course, this flexibility does incur the overheads involved in managing the various address-translation tables required by the operating software, but these are normally outweighed by the advantages.

The second advantage is that it allows

the sharing of common memory areas by different tasks. This is accomplished by mapping different logical areas in different tasks to the same physical-memory locations.

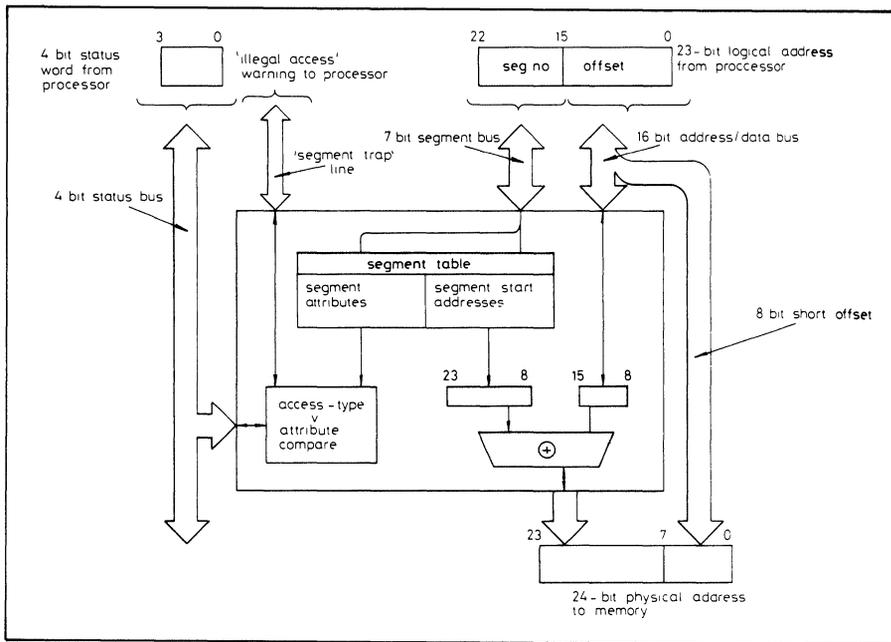
The third advantage is that it provides protection against certain types of memory access. This is accomplished by associating accessing attributes with each logical segment, and by checking the type of access to see if each access is permitted.

The fourth advantage is that it detects obvious execution errors related to memory accessing. This can be accomplished by checking each access to a segment to see whether the address falls within the physical-memory area allocated to that segment. It could also include affixing a read/write attribute to data to prevent a task from trying to execute a data segment, and affixing an execute-only attribute to code segments to prevent a task from trying to read or write data to this segment. Additionally, if a segment is used to hold a stack, the system could issue a warning to a task

when the stack approaches the allocated limit of the segment. The task could then request the operating software to allocate more memory to the stack before the stack overflows and creates a fatal error.

The final advantage of such memory-management systems is that they separate user functions from system functions. For processors that distinguish between a 'system' mode and a 'user' mode of operation, this goal can be accomplished by associating a system-only attribute with operating-system segments so users cannot directly access the operating software and its data tables.

As a final point, it should be noted how segmentation can be used to support the development and execution of large, complex programs and systems. The concept of segmentation corresponds to the concept of partitioning a large system into procedures and data structures, each procedure and data structure being associated with a separate segment. A task can then invoke a procedure or subtask, or access a



processor to put the 7-bit segment number on its output lines one cycle ahead of the rest of the address. This gives the m m u additional time to retrieve the appropriate segment-starting address and to check the appropriate segment attributes.

From this account of the m m u's action, it will be seen that the Z8000 processor handles 23-bit logical addresses directly, each logical address comprising a segment number and appropriate offset. These 23-bit addresses can be stored as 32-bit 'long words' in pairs of 16-bit registers or in adjacent 16-bit memory words, and can be manipulated by all the Z8000's built-in

'long word' operations. For more efficient manipulation of short (up to 256 byte) segments, shortened logical addresses consisting of 7-bit segment numbers and 8-bit offsets can also be used, these shortened addresses fitting within an ordinary 16-bit word.

Very similar memory-management facilities are said to be planned for Motorola's 68000 'super micro'. The Motorola device, however, uses 24-bit logical memories to give a larger 16 Mbyte addressing range.

DENNIS MORALEE

data structure, by referring to its logical-segment name. Access to these objects can be individually restricted by using the protection-checking mechanism of the memory-management system.

Virtual memory

With the memory-management systems considered so far, it has been assumed that the actual physical memory available is always large enough for all the users' logical-address space to be simultaneously mapped onto it. In fact, further advantages can result from making even this physical-memory space 'virtual', and from mapping it in turn into a two-level memory space, part of which is held in a relatively small 'true' physical memory, and part of which is held on a secondary-memory device such as a magnetic disc (Fig.4).

In operation, this *virtual-memory* arrangement relies on an extension of the address-translation scheme considered above. If a given segment is not currently in physical memory, the address-translation table indicates the fact, and

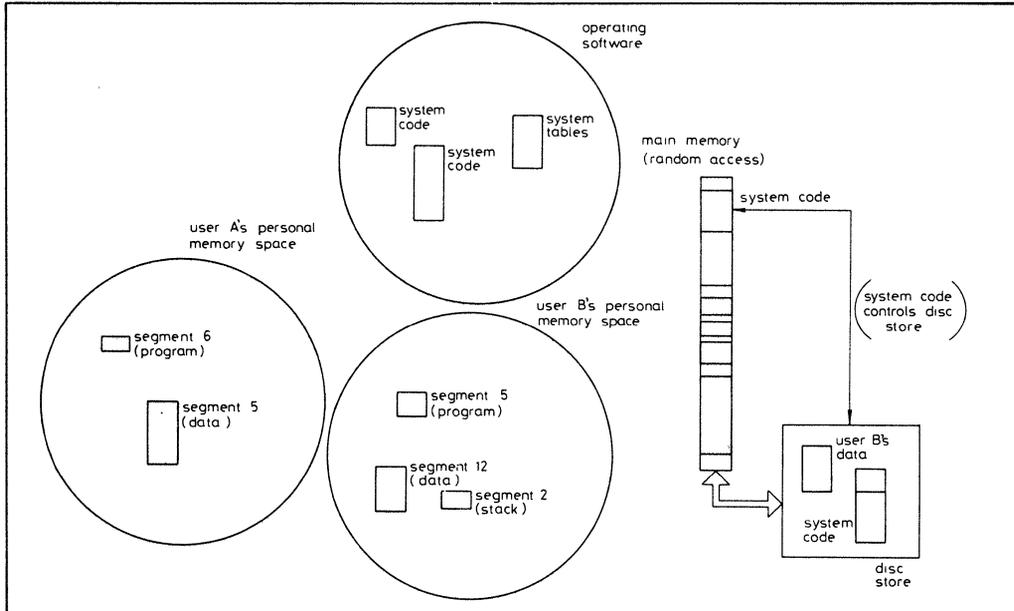
links to a routine forming part of the operating software and capable of fetching the segment from secondary memory when needed.

Whenever an access is made to a segment missing from physical memory, the instruction execution is held in abeyance until the segment can be brought into the physical memory, and then the instruction is allowed to proceed with the memory access. The address translation is then performed, access protection is checked, and the instruction proceeds as if the segment had been in the physical memory at the beginning of the instruction. Thus this technique of *demand swapping*, or *segmented virtual memory*, means that the segments will not in general reside in physical memory until a task actually tries to access it.

Another technique of virtual-memory management is paging, which is also a method of partitioning a user's logical-address space and mapping it onto a two-level physical memory. Essentially, a paging system divides the logical

memory into fixed-sized blocks, called pages. Like segments, the individual pages can be located anywhere in the physical memory, and a translation mechanism maps logical addresses to physical locations. There are two differences between paging and segmenting a logical memory. First, pages are of fixed size whereas segments are of various sizes. Second, under paging, the logical memory is still linear, i.e. a task accesses memory using a single number, rather than a pair as in segmentation.

The major advantage of paging is in treating memory as blocks of fixed sizes, which simplifies allocating memory to users and deciding where to place the logical pages in physical memory. The major disadvantage of paging is the difficulty of assigning different protection attributes to different areas in a user address space, because a paged memory appears homogeneous to the user and the operating system. Paging can, however, be combined with segmentation to produce a memory-management system with the advantages of both pag-



4 In a virtual-memory system, the system's 'physical memory' is split between a relatively small random-access 'main' memory and a secondary-memory disc store. If a program attempts to access a segment not currently stored in main memory, the operating software retrieves it from the disc, and processing continues transparently to the user

ing and segmentation, but at the cost of considerable extra complexity.

Mechanics of memory management

Essentially there are four issues in implementing a memory management system: how addresses are specified, how these addresses are translated, what attributes are checked for each access, and how the protection mechanism is implemented.

Two approaches have traditionally been taken for specifying addresses in a segmented memory (for simplicity, only addresses in instructions are discussed here). The first way puts all the addressing information in the instruction itself; i.e. each memory address in an instruction contains both the segment name and the offset within the segment. The alternative sets aside special registers that contain some of this information, for example, the segment name or the address in physical memory where the segment resides.

The advantage of the latter approach lies in the fact that fewer bits are needed in an instruction to specify addresses. Thus programs may be shorter. Also, because there is reduced traffic between the memory and the processor for fetching shorter instructions, a program may be executed faster.

On the other hand, these special registers must be manipulated to access more segments than there are registers, and this manipulation adds to the number of instructions, the program size and the execution time. In practice, these can destroy the advantages

described above. If the special registers contain physical memory locations, these must be protected from user access to maintain the integrity of the system, and changing segments requires system calls which can be time consuming if too few registers are supplied.

In either case, address translation is performed by adding the logical-segment offset to the address of the physical-memory location where the segment begins. Thus, when an address of the form (a,b) is presented to the translation mechanism, the segment name 'a' is used to determine where segment 'a' resides in memory. Assume that it resides in locations 10000 to 25000. Then the actual memory location (a,b) is memory location $10000 + b$. The major option in implementing this type of address translation is in determining the segment's location in physical memory. When special registers have been set aside to contain the starting location of the segment instead of putting all address information in the instruction, the addressing mechanism is similar to using the segment register as an index register or a base register.

When logical addresses are either completely specified in the instruction or when the special register contains the segment's symbolic name rather than its physical-memory location, a table must be used to translate the segment's name into its physical-memory location. The table may have an associative capability, i.e. the segment name is presented to the table and it automatically returns the physical-memory location where the segment begins. Alternatively, the table

could have one entry for every possible segment name, with the starting address of each segment in use stored as part of the table entry.

A number of other segment attributes can also be stored in the address-translation table and checked during each access. One of these is the allocated length of the segment, and each access is checked to see if it falls within the bounds of the segment.

Another type of attribute deals with ownership or class of ownership: tasks are grouped into classes, and only those in certain classes are permitted to own and therefore access a given segment. The simplest example is the 'system' versus 'user' classification, where tasks are either one or the other, and which they are determines whether or not they can access a given segment.

Other types of attributes that can be associated with a segment involve modes of accessing, for example 'read-only', 'read/write' or 'execute-only'. Attributes can be either permissive or prohibitive; for example the 'write' attribute can mean 'writing to this segment is permitted' or 'writing to this segment is prohibited'.

A final issue in the mechanics of memory-management systems is the implementation of the protection attributes. These may be associated either with the logical-address space or with the physical memory itself. Associating access attributes with the logical segment permits a more versatile memory-management scheme because different users can access the same physical segment and have different access attributes

associated with their accessing.

Other information that can be associated with each segment is associated not with the protection mechanism, but with other functions of the memory-management system. This information generally relates to the history of the segment; for example, whether a segment has been modified while resident in primary memory. If it has not been modified, and the system temporarily requires the memory space for another segment, the memory can be freed immediately; otherwise, the updated version of the segment must be

stored in secondary memory, and the primary memory is not available until the segment has been saved. Although not strictly necessary, such information can improve the performance of the overall memory-management system.

References

- 1 CORBATO, F.J., and VYSSOTSKY, V.A.: 'Introduction and overview of the multics system', AFIPS Conf. Proc., Fall Joint Computer Conf., 1965, **27**, pp.185-196
- 2 GLASER, E.L., *et al.*: 'System design of a computer for time-sharing applications', AFIPS Conf. Proc., Fall Joint Computer Conf., 1965, **27**, pp.297-302
- 3 WATSON, R.W.: 'Timesharing system design concepts', (McGraw-Hill, New York, 1970)

David Stevenson is with Zilog Inc., 10460 Bubb Road, Cupertino, Calif. 95014, USA

Z8000 vs. 68000 Concept Papers



Introduction

July 1981

Introduction

The Z8000 and 68000 are similar CPUs, but important differences exist between them. The following concept papers discuss several substantive differences that design engineers should consider when trying to choose between these two CPUs.

Both the Z8000 and 68000 are classified as 16-bit CPUs, although each offers many of the attributes of a 32-bit CPU; each was designed with provisions for compatible expansion to a full 32-bit architecture. Each of these CPUs has an address space two orders of magnitude larger than the largest 8-bit CPU. Each has 16 central registers designed for general use (see the concept paper on the differences in register architecture). Each has a powerful instruction set, powerful addressing modes, and great regularity in the association of instructions with addressing modes. Each has a protected "user" mode, privileged instructions, and separate system and user stack registers. Each has automatic vectoring of traps and interrupts, with CPU status saved on a stack.

Critical Issues

The following concept papers focus on a number of critical design issues. This section summarizes the issues discussed and lists the key criteria used in addressing the relative merits of the Z8000 and 68000 approaches.

Memory Addressing

There is a sharp contrast between the segmented addressing model of the Z8000 CPU and the purely linear addressing model of the 68000 CPU. In examining these approaches, the following desirable attributes for a memory addressing scheme should be recalled:

- An addressing model that mirrors program organization
- Provision for access protection
- Provision for memory mapping

- Support for dynamic relocation
- Support for sharing
- Support for stacks

I/O Addressing

The Z8000 CPU has separate address spaces for I/O and memory; the 68000 uses memory-mapped I/O. The designer evaluating an I/O addressing mechanism should consider:

- Naturalness of the programming model
- Protection of I/O references
- Complexity of external interfacing logic
- Potential for performance improvement
- Provision for the block I/O function

Address/Data Bus

The Z8000 and 68000 CPUs use asynchronous address/data bus protocols. The Z8000 time-multiplexes a single set of lines for addresses and data, whereas the 68000 uses separate lines for addresses and data. In choosing between these approaches, the designer must consider:

- Performance limitations
- Complexity of interface to peripherals chips
- Optimal use of CPU pins

Register Architecture

The Z8000 and 68000 CPUs are similar in their register architectures, but they differ in significant details. Points that should be considered are:

- General vs. special-purpose use of registers
- Availability of registers of all necessary sizes
- Addressability of subregisters
- Extensibility of the register set

Operating System Support

The Z8000 and 68000 both provide many architectural features designed to assist in implementing the "system" portions of large and small applications, but there is a difference in the degree to which this area was addressed in the two designs. The Z8000 designers gave careful consideration to a thorough, unified approach to operating system support. As a result, the Z8000 is much stronger in this area than the 68000. The discussion of this area covers all of the fol-

lowing architectural support features for operating systems:

- Restriction of access to CPU and memory
- Memory mapping
- Sharing of programs and data
- Program relocation
- Stacks
- Context switching
- I/O system and interrupts
- Distributed control
- Support for conventions

Z8000 vs 68000

Register Architecture

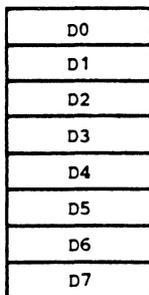


Concept Paper

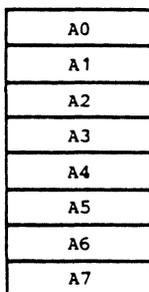
May 1981

The Z8000 and the 68000 take quite different approaches to register architecture. The principal points of difference are:

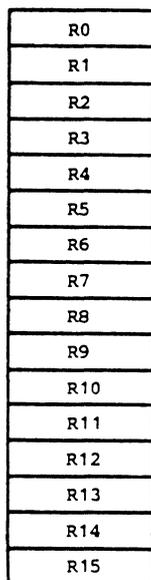
- General purpose vs. special purpose registers
- Pairing vs. telescoping of subregisters
- Extensibility of the register sets



68000 Data Registers



68000 Address Registers



Z8000 General Purpose Registers

Z8000 and 68000 Registers

GENERAL PURPOSE VS. SPECIAL PURPOSE REGISTERS

The Z8000 has a set of 16 16-bit general purpose registers. Each can be an address register, a

data register or an index register. (There are restrictions on the use of R0 imposed by the current instruction encoding.) The 68000 has two sets of 32-bit registers: eight address registers and eight data registers; either type can be used for indexing.

This difference in register architecture results in generally simpler programming of the Z8000 than of the 68000. Several aspects of this are:

- Information in a Z8000 register never has to be moved before being used as an address or in arithmetic operations.
- The Z8000 uses the same op codes for arguments in any of the registers. This is in contrast with the 68000's separate op codes (e.g., ADD and ADDA for operations on the two register sets).
- The Z8000 uses the same addressing modes for all of the registers. This is in contrast with separate 68000 addressing modes like "data register direct" and "address register direct."

The net effect of these differences is that with regard to register handling the job of the compiler writer is easier with the Z8000 than with the 68000 and that compiled code for the Z8000 is likely to be more efficient than code for the 68000.

PAIRING VS. TELESCOPING OF SUBREGISTERS

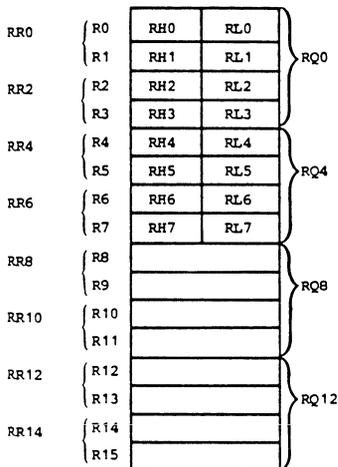
The Z8000 instructions refer to byte registers, 16-bit registers, 32-bit registers and (occasionally) 64-bit registers. The 68000 refers to 16-bit and 32-bit address registers and to 8-bit, 16-bit and 32-bit data registers. On both machines, every register, except for those of the largest size, is contained in a register of the next larger size. Thus, every byte register is contained within a 32-bit register, and so on. On the 68000, this is a one-to-one relationship. Each 32-bit register contains exactly one 16-bit register, each 16-bit data register contains exactly one byte register. In each case, the subregister is the rightmost half of the larger register.

Z8000 Register Hierarchy

On the Z8000 a different scheme is used. The 16 byte registers are packed into 8 16-bit registers. The other eight 16-bit registers contain no byte registers. Similarly, the sixteen 16-bit registers are packed into the eight 32-bit registers, and the eight 32-bit registers are packed into four 64-bit registers.

The 68000 arrangement facilitates the type-conversion operations that occur in higher level languages, since, for example, an 8-bit value stored in the rightmost eight bits of data register zero and sign extended to the whole 32 bits can then be referred to as the 32-bit R0, the 16-bit R0 or the 8-bit R0. On the Z8000, a similar situation is possible, but the names would be RR0, R1, RL1. The price that is paid for this one feature is that the 68000 register hierarchy is inconvenient to use, while the Z8000 register hierarchy is a great programming convenience. For example, a Z8000 programmer can allocate four byte registers inside of one 32-bit register. On the 68000 a programmer would have to tie up four 32-bit registers to store the same four byte quantities. That's half of the data register set to do what can be done on the Z8000 in one eighth of the general purpose register set. If the Z8000 programmer wishes to use 16 byte registers, this can be done using only half of the register set. On the 68000, the maximum number of byte registers available is eight, and this ties up the entire data register set.

Another advantage of the Z8000 register hierarchy is that half of the 16-bit registers and all of the 32-bit and 64-bit registers have addressable halves. Half of the 32-bit and 64-bit registers have addressable quarters. Half of the 64-bit registers have addressable quarters.



Z8000 General Purpose Registers

Smaller registers are paired to form larger registers.

Z8000 Register Hierarchy

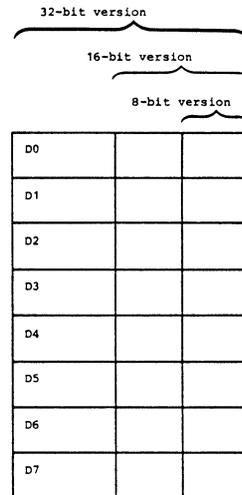
The availability of this feature facilitates many programming tasks. On the 68000, there is no way to address the left half or any of the three left-most quarters of any register. Such operations must be simulated with shift or rotate instructions.

EXTENSIBILITY OF THE REGISTER SET

The Z8000 instruction encoding uses 4-bit fields to designate registers; the 68000 uses 3-bit fields. This means that with no change in op codes and no change in instruction format, the Z8000 architecture will accommodate expansion of the general purpose register set to include 16 of each size of register. This means that eight 32-bit registers and twelve 64-bit registers can be added to the register set. The use of 3-bit fields and the telescoping of subregisters on the 68000 preclude a compatible extension of the number of registers of any given size and make introduction of 64-bit or larger registers extremely wasteful of register space.

SUMMARY

The Z8000 and 68000 register architectures are similar, but there are important differences. The 68000 uses special purpose address and data registers, the Z8000 uses a general purpose register file. The Z8000 uses pairing of smaller sized registers to make larger sized registers, the 68000 telescopes subregisters into the rightmost portions of larger registers. The Z8000 provides for compatible enlargement of the register file, the 68000 does not. In each case, the Z8000 approach is seen to be superior.



68000 Data Registers

8, 16 and 32-bit versions all have same name; each is rightmost subset of the 32-bit version

68000 Register Hierarchy

Memory-Mapped vs. Explicit I/O



Z8000 vs. 68000 Concept Paper

October 1980

The Z8000 and 68000 take very different approaches to the addressing of I/O transactions. In the 68000, I/O addresses and memory addresses share the same address range. This is called memory-mapped I/O. References to I/O addresses are made exactly like references to memory addresses, using the same instructions and addressing modes. The processor does not know, when it engages in a read or write, whether it is talking to memory or to an I/O device.

In the Z8000, there is a separate address range for I/O transactions, and separate instructions are used. The processor always knows which kind of transaction is being conducted. The same physical address/data lines are used for the two kinds of reference; the status lines ST_3 - ST_0 distinguish between them.

Several advantages have been claimed for memory-mapped I/O:

- Regularity - the same instructions and addressing modes are available for I/O as for memory.
- Simplicity - the size of the instruction set is reduced, since there are no I/O instructions.
- Ease of implementation - there is no need to design separate I/O bus protocols.

As to regularity, the kinds of operation performed on I/O ports are limited, as are the kinds of addressing that are useful in I/O operations. Furthermore, there are special needs of I/O operations that are different from those of memory operations (e.g., block transfers to a fixed address).

The 68000 design recognizes the fallacy of the regularity argument by introducing the MOVEP instruction--a block transfer of the bytes of a word or longword to consecutive even-addressed or consecutive odd-addressed bytes of memory. No 68000 instruction is provided for block transfers to a fixed address in memory.

The MOVEP instruction and the missing block I/O instruction also demolish the simplicity argument. Separate instructions are necessary because the two kinds of operation are different, and if the separation is not made explicit, an additional instruction will be necessary, as was done on the 68000.

In regard to the ease of implementation argument, I/O and memory transactions on the Z-Bus are only trivially different (I/O has an added cycle). The difference between the Z8000 and the 68000 bus protocols is not in ease of implementation. The difference is that the 68000 is locked into a single bus, while the Z8000 has the potential for future separation to improve performance.

Upon closer inspection, memory-mapped I/O has, in fact, many disadvantages.

- It makes protection of I/O references impossible at the instruction level--I/O instructions can't be privileged, because there are no I/O instructions.
- It creates "holes" in the memory address space, so that certain addresses--possibly localized, but potentially anywhere--cannot be used for memory addresses by any program.
- It prevents a compatible separation of I/O and memory buses--blocking an important path to performance improvement.

The question of protection is important in the design of operating systems. The I/O function is usually controlled by the system and prohibited to users, so it makes sense to make I/O instructions privileged. On the 68000, there are no I/O instructions (except for MOVEP, which is not privileged), so I/O instructions cannot be privileged. The only way to achieve this kind of protection on the 68000 is to assign to an external device the job of recognizing I/O addresses and preventing access to these addresses when the processor is executing in user mode.

The problem of "holes" in the memory address space can be partially alleviated by placing I/O addresses at one end or the other. (The 68000 sign extension of "short" addresses encourages this.) Nonetheless, the addresses are missing from the memory address range, and a runaway program could inadvertently store into these addresses, causing unpredictable results, including writing to tape or disk.

Finally, since no specific areas of the memory address range have been pre-assigned to I/O, the CPU has no way of knowing whether the transaction it is conducting is for I/O or memory. As a result, the potential performance improvement arising from separation of the I/O and memory buses is forever unavailable to the 68000. On the other hand, in keeping with the philosophy of "economy of means"--a major Z8000 design criterion --the

Z8000 offers both the economy of using one bus for both I/O and memory and the potential for future separation.

In summary, the Z8000 design, by recognizing the distinction between I/O and memory operations, has achieved the following advantages over the 68000 I/O architecture:

- A natural programming model that easily incorporates the important block I/O function and avoids awkward instructions like the 68000's MOVEP.
- Protection--through making I/O instructions privileged and through having a separate I/O address space that no wild memory access can reach.
- Potential for future performance improvement through the separation of I/O and memory buses or through different handling of I/O and memory transactions, even on the same bus.

The Address/Data Bus



Z8000 versus 68000 Concept Paper

October 1980

The Z8000 and 68000 address/data buses are similar in that both use asynchronous protocols. They differ in that the Z8000 time multiplexes one set of lines for address and data, while the 68000 uses two separate sets of lines. The trade-off involves the higher potential performance of separate, dedicated lines versus the more effective use of the limited numbers of pins available for chip packages.

Dedicated lines have a potential for improved performance when one device has access to both the address and the data and can send them out simultaneously. The principal occurrence of this situation is a write to memory. There are several reasons why this potential advantage is of little consequence in a comparison of the Z8000 and the 68000.

- Reads from memory (including instruction fetches) occur roughly eight times as often as writes. A read from memory does not benefit from the separation of lines, since the address must be sent from the CPU to the memory before the memory can retrieve the data or instruction in question and send it back to the CPU.
- In the case of writes to memory, most memory chips are incapable of simultaneously accepting both the address and the data to be stored.
- Even with a memory chip that is capable of accepting addresses and data simultaneously, the 68000 still achieves no performance benefit, since 68000 write instructions are two cycles longer than read instructions (six cycles for writes vs. four cycles for reads) in order to

allow the data bus to be turned around at the beginning and at the end of each write.

Considering the other side of this trade-off, the use of separate address and data lines results in the need for 16 (and, in the future, 32) pins that could be utilized to greater advantage. Looking just at the CPU, the 68000 faces all of the price, power and reliability problems of a 64-pin chip with no more capabilities than are provided by the Z8000's 48-pin package. When improved manufacturing technology allows economical and reliable expansion of the Z8000 to a 64-pin package, the 16 additional pins will provide greatly increased capabilities.

In addition to the more effective use of CPU pins, the multiplexing of address and data lines provides a means of addressing directly the internal registers of peripheral chips without the need to dedicate pins of the peripheral chip to separate address lines. Since at least eight data lines must generally go to a peripheral chip, these can be used during the addressing phase of an instruction to address a chip's internal registers (with the remaining eight I/O address lines possibly being decoded by external chip-select logic). This simplifies the programming of and access to peripheral chips by eliminating the separate address setup cycle required by an unmultiplexed peripheral interface.

In summary, the use of separate address and data lines gains little in performance, especially on the 68000 with its extra-long memory write instructions. It is wasteful of hard-to-come-by CPU pins and encourages a cumbersome interface for addressing peripheral chips.

**Z8000 vs. 68000
Segmented vs. Linear
Addressing**



Concept Paper

November 1980

INTRODUCTION

The Z8000 and the 68000 use fundamentally different models for memory addressing. The Z8000 uses segmented addressing. The 68000 uses linear addressing. We shall define these terms and explain why segmented addressing is a superior method.

Segmented addressing is a "higher-level language" for memory addressing. That is, it is a way for the programmer to think about and refer to the computer's memory in terms that are natural to programming rather than in terms of the memory's physical implementation. Linear addressing is the "machine language" of memory addressing. That is, with linear addressing, the programmer uses a model for the computer's memory that is very close to its actual hardware implementation. Before we state more specifically exactly what segmented addressing is and how it works, let's look at some of the memory addressing tasks that programmers face and see what kind of addressing model these tasks suggest.

MEMORY ADDRESSING

The programmer is concerned with a variety of programs, data areas, stacks, etc. (for all of which the general term "objects" is used) and with the interactions among these objects. What we mean by this is partly a question of how fine-grained our picture is to be. For example, we could say that a programmer deals with two objects: the program and the data. At the other end of the scale, we could say that the programmer deals with a multitude of objects--listing separately each instruction and datum. Between these alternatives there can usually be found for each programming situation a set of largely separate but interrelated objects. For example, for a Chess-playing program, the objects might include:

- Chessboard display program
- Current position representation
- Legal move generation program
- Move evaluation program
- File of previously evaluated positions
- Handling routines for previous position file
- Program to study published games

This program might run under control of an operating system that was also divided into objects, including:

- Task scheduler
- Memory allocator
- Secondary storage interface routines
- Terminal interaction routines
- Process status table
- System stack
- User process status tables

The example could be refined and enlarged, but these are good examples of what we mean by the objects that the programmer must deal with.

The traditional approach to dealing with these objects is to allocate portions of the computer's memory to each of them. A relocating loader might pack the programs together end to end and then allocate the data areas (of fixed sizes) end to end in the portion of memory not occupied by the programs. Since the only addressing model available with the earliest computers was linear addressing, each of the objects would receive an address directly related to (usually the same as) the actual memory address at which it was stored. These addresses were all numbers in the range 0 to $N-1$, where N was the total number of memory locations available. Every program that referred to any of these objects had to do so using this address.

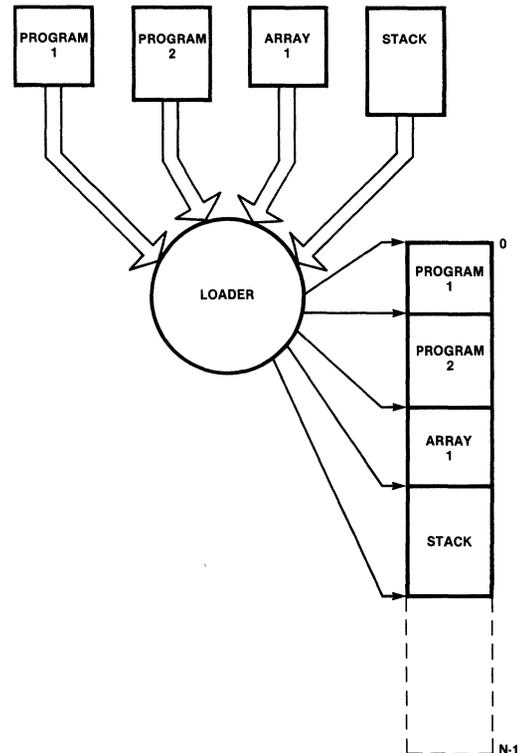


Figure 1. Traditional Approach to Memory Allocation

PROBLEMS WITH THE TRADITIONAL APPROACH

This approach always presented problems, and as systems grew larger the problems grew exponentially. We shall review these problems and look at some early solutions. The problems can be summarized under the following categories:

- Invalid accesses
- Difficulty of accommodating objects whose sizes vary--like stacks or lists

- Difficulty of creating and deleting objects dynamically--fragmentation of memory
- Difficulty of relocating objects after the loader has established linkages among them
- Difficulty of sharing objects among otherwise independent processes

Invalid Accesses

The problem of invalid accesses occurs even in the smallest systems and on the smallest computers. In its basic form, the problem occurs when a program erroneously uses an address as if it belonged to one object when it actually belongs to another. For example, if an array is 1024 bytes long and a program erroneously refers to its 1025th byte, then the reference will actually be to the first byte of the object stored in memory immediately following the 1024-byte array. If the erroneous access is a store operation, then the object following the array in memory will have been damaged.

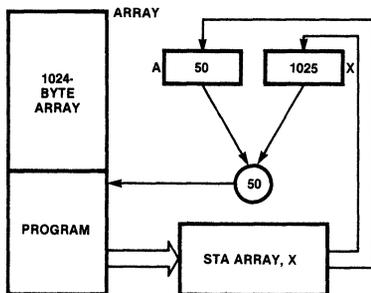


Figure 2. A Traditional Invalid Access

Another example of this problem concerns the use of stacks. A common approach to stack use in a single-user system is to allocate the "beginning" of memory to programs and data and the "end" to a stack, since the push and pop instructions on most computers are designed in such a way that stacks grow "backwards" in memory; that is, the first item placed on the stack is at the highest-numbered address, and the "top" of the stack is at the lowest-numbered address. If often happens that program changes cause the program and data areas to expand, so that less and less remains for the stack. Sooner or later, a stack push causes the stack to overflow the allotted area and eradicate the end of the area assigned to programs and data.

A frequently used approach to problems of the sort described above is to create an "envelope" around the accesses in question. Thus, for example, instead of using the computer's indexing capability to access arrays directly, the program might instead call a subroutine that accepts the index and the

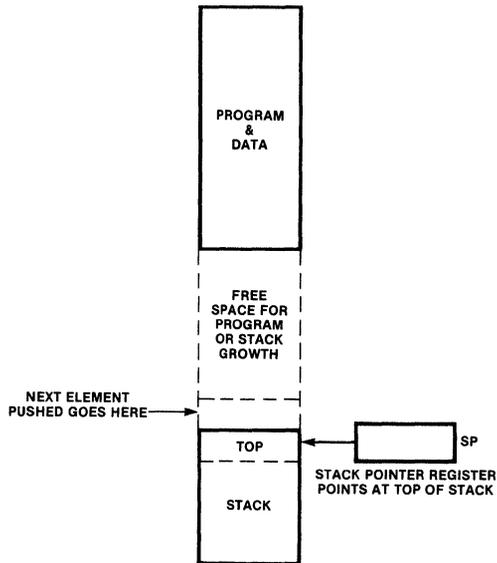


Figure 3. Allocation of Programs, Data and Stack Space

identity of the array as arguments and returns a validated memory address that the program can use for fetching or storing. The routine might also handle the actual fetching or storing, accepting data to be stored as another argument or returning the data fetched. In either case, the routine would validate the access by using the array identity as a key to a set of array attributes, including the array's length and location in memory.

For the stack example, a similar envelope would be placed around pushes and pops. Rather than using the machine's push and pop instructions, the program would call subroutines for these operations. Naturally, this approach entails a large software overhead.

Another type of invalid access occurs in even the most elementary systems, but it presents an urgent problem when several programs or sets of data--not necessarily related to one another--share memory simultaneously. This problem concerns the restriction of a program's accesses to those portions of the memory containing its own subroutines and data or--even more difficult--to portions of memory containing data or subroutines that it shares with another program and to which it is allowed only certain kinds of access (such as "read only" or "execute only").

The software envelopes discussed above can be extended to accommodate shared access to data, but it is difficult to place such envelopes around program accesses. Furthermore, these envelopes are voluntary; that is, a programmer who wishes to avoid them can

usually discover enough information to be able to make the accesses directly. For situations of this sort hardware solutions were introduced. One such solution was the use of limit registers. For example, the operating system might set registers that defined the limits of the program about to run to be locations 10000 through 19999. In this case, the program is free to make references of any sort so long as the address used lies within the given range. An attempt, for example, to call a subroutine at address 20000 results in a "trap," and control is returned to the operating system.

These examples are an indication of some of the ways in which the problem of invalid accesses can manifest itself, and they show how early system designers attempted to solve them. Shortly we shall see how segmentation provides a complete solution to this problem.

Objects of Varying Sizes

In our stack example above, we saw the kind of problem that can arise when an object varies in size. We showed how an envelope around pushes and pops can detect invalid accesses before they occur, but we are still faced with the problem of what to do about them. In the example given above, there was only one stack, and it didn't run out of

and irrelevant to the program using the stack. Unfortunately, the way that stacks are ordinarily used does not lend itself to this approach. Frequently a program is allocated a block of stack space, which it then accesses using based addressing. That is, the actual memory address of the first location of a block of stack space is kept in a register, and accesses into the block are made by adding an index (from a register or from an instruction) to the base addresses in the register. This common practice is incompatible with the existence of gaps in the set of addresses assigned to the stack.

The solution to this problem (before segmentation was invented) was to allocate a larger contiguous block of memory to the enlarged stack--either by moving the stack to another part of memory or by moving something else out of its way so that it could be expanded where it was. This approach has two inherent problems: the processing overhead to move objects around in memory and keep the unused memory all in one place and the "relocation" problem of changing all of the base addresses of blocks of stack space that the program has in registers or in storage. The second problem is almost insurmountable, except in the most elementary cases.

The problem of accommodating objects whose

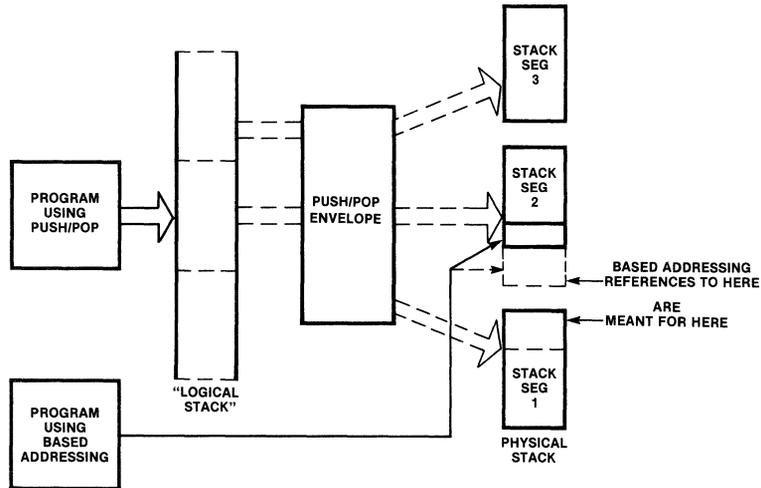


Figure 4. Why Stacks Must be in One Piece

memory until the entire memory of the computer was exhausted. However, if we had many stacks to manage, we might want to assign a small amount of memory to each and then expand those that were about to overflow. If all accesses to stacks are through the envelopes that surround the push and pop instruction, this is no problem. The stack can merely be "continued" elsewhere in memory, and the gap in the actual memory addresses between the last location of the original stack and the first location of the extension will be completely concealed from

sizes vary has as a special case the problem of creating and deleting objects dynamically. This problem arises in the simplest single-user systems--for example, "initialization" code might be abandoned after it is executed once and the space given to a large data array. As with our other examples, however, the difficulties mount rapidly as the system becomes more complex. In particular, because of the difficulty of "relocating" addresses, the moving of objects that would be necessary in order to keep the unused memory in one place is avoided. The unused memory soon

comes to be scattered about in small pieces, and it becomes increasingly difficult to find contiguous blocks of sufficient size to accommodate newly created or expanded objects, even when the total amount of unused memory is sufficient. This problem is known as "fragmentation" of memory.

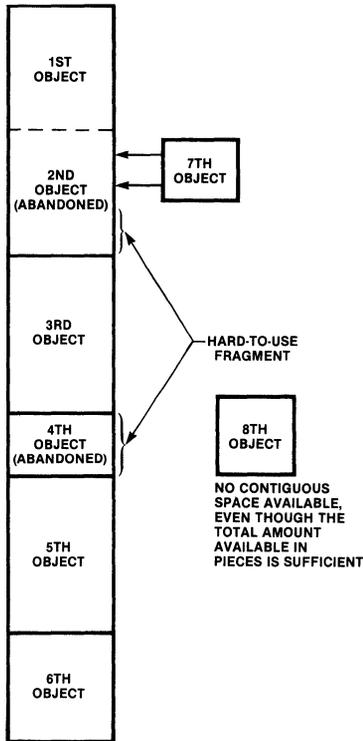


Figure 5. Fragmentation

Traditionally, there has really been no solution to this problem other than to leave management of the assigned memory to the user program. The user is provided with tools like "chaining" commands and overlay structures in certain systems, but by and large, the creation and deletion of objects is simply treated as part of the "algorithm" that the program implements. Soon we shall see how segmentation allows system control of this function.

Relocation

In discussing the expansion of stacks we alluded to the "relocation" problem that arose when a stack was moved: all of the pointers into it (the base register values for accesses to blocks of stack space) become invalid. This is a special case of the general problem of dynamic relocation. After the loader has established linkages among the parts of the program, it becomes almost impossible to move any of them. This is another problem that had to wait for a hardware solution. This solution has been

provided at several levels.

Dynamic relocation, that is, relocation that occurs after the initial load of the program, requires a mechanism that allows actual addresses to be determined at run time. The first approach to this is provided by various kinds of based addressing. Based addressing of program references is usually provided by PC-relative addressing: calls, jumps and loads of program constants are specified using an offset that is added to the actual program counter value to obtain the memory address. Based addressing of data references is also made using offsets--to be added to a stack pointer or other address register. Relocation effected through based addressing is called "user-controlled" relocation, since the setting of the stack pointer or other address register is under control of the running program. A better approach from the standpoint of reliability is "system-controlled" relocation. This kind of relocation can be provided using memory mapping.

Memory mapping is, in its simplest form, a translation mechanism that converts the addresses used by the running program (which now become called logical addresses) into the actual memory addresses (now called physical addresses). With memory mapping, the program always uses a fixed set of addresses, and relocation is achieved by a change to the translation mechanism. A simple example of this is provided by a mechanism similar to based addressing. A value is set into a base register, and the translation mechanism consists of automatically adding that value to any address used in the program. (The difference between this and based addressing is that with based addressing there is an

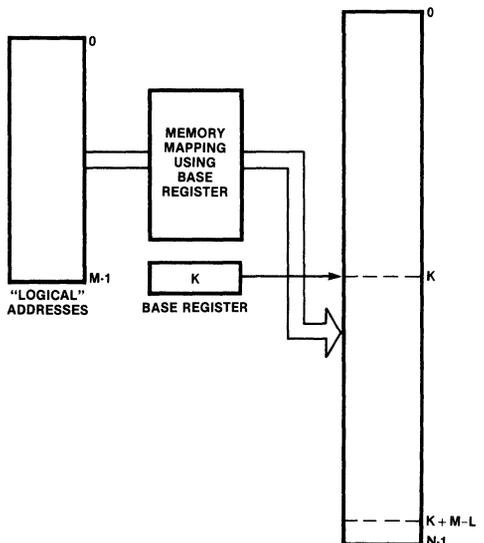


Figure 6. Memory Mapping with a Base Register

explicit reference to the base register in the instruction. With this mechanism, the base register is used independently of the program to translate the addresses that it generates.)

This very simple form of memory mapping quickly evolved in two directions: paging and segmentation. Paging is a natural extension of the linear addressing model (although it can also be applied to the linear addresses used within segments). We won't say any more about paging or segmentation at this point.

Sharing

A natural outgrowth of memory mapping is a mechanism for the sharing of objects among otherwise independent processes. Given a mapping mechanism (more sophisticated than the simple base register mentioned above) that allows different blocks of logical addresses to be mapped independently of one another, a program or data area in physical memory can correspond to different logical addresses for different processes. Thus, the shared program or data can reside at a convenient location in the logical address space of each process, and the mapping mechanism will cause references from each process to be mapped by that process' mapping scheme into the given physical locations.

We shall say more about memory mapping in conjunction with our discussion of segmentation. At this point, we should simply note that system controlled relocation and sharing through memory mapping alleviate one of the problems that tends to occur with user-controlled relocation and non-mapped sharing: fragmentation of the address space.

SOLUTIONS

We have now discussed the major problems with the use of the linear addressing model and have looked at some early attempts to solve them. Now we shall look at the abstract addressing model provided by segmentation, and we shall see how the Z8000 CPU and memory management unit have been designed to work together to provide an implementation of this model that incorporates memory mapping and access protection. We shall show how this unified approach alleviates all five of the major problems with linear addressing that we stated earlier.

Segmentation

Segmentation is the organization of the address space into a collection of independent objects. As we noted earlier, in each programming situation there can usually be identified a set of largely separate but interrelated objects. The segmented addressing model assigns to each of these objects a

"name" and a linear address space. The "name" is, of course, a binary number, but we call it a name to emphasize the fact that there is no relation between objects implied by a numerical relationship between their "names."

For example, in the example given above, the chessboard display program could be assigned the name 1, the current position representation could be 2, the legal move generation program could be 3 and so forth. The address of any location within the chessboard display program would then consist of the name, 1, and an address within object 1's linear address space. If this program occupied 2048 bytes, for example, then the addresses within object 1 would be (1,0), (1,1) ... (1,2047). One of the attributes of object 1 would be a length of 2048 bytes, and the mechanism responsible for the interpretation of segmented addresses would be aware of this attribute and would cause an appropriate error indication if an address of the form (1,N) with $N > 2048$ were ever used.

Now consider the case of the current position representation--object 2 in our example. Let's suppose that this representation takes the form of an array of 256 bytes. The addresses of these bytes would be (2,0), (2,1) ..., (2,255). One means of referring to items of this array involves the use of indexed addressing. The address of the item referred to would be specified by giving the array base address of (2,0) in one place--in the instruction or in a register--and an index (also called an offset) in a register. The index is simply a number to be added to the second component of the segmented address. Thus, if the index were 17, then the item address would be (2,17). That is, the address manipulation cannot affect the object name portion of the address--only the linear address within the object is affected.

Similarly, returning to the display program (object 1 in our example), the mechanism responsible for address interpretation performs a similar computation for PC-relative addressing. If the program contains a branch to "current location - 24" or a call to "current location + 1264" for example, then the offset given in the instruction is applied to the second part of the address. If the call were made from location (1,562), then 1264 would be added to 562, and the final address would be (1,1826).

Preventing Invalid Accesses

These examples show how segmented addressing helps to alleviate our first major problem with linear addressing: invalid accesses. Suppose, for example, that we had made a programming error that caused us to address the current position representation array by using an index value of 257. With a linear

addressing scheme, this would result in a reference to the second byte of whatever object follows the current position representation array in memory. Thus, we might overwrite the second byte of the legal move generation program if that happened to follow the array.

With segmented addressing, the address computation would result in an address of (2,257). The mechanism that interprets addresses would discover that this address is incompatible with the declared length of the array (256 bytes), so an appropriate error indication would be generated.

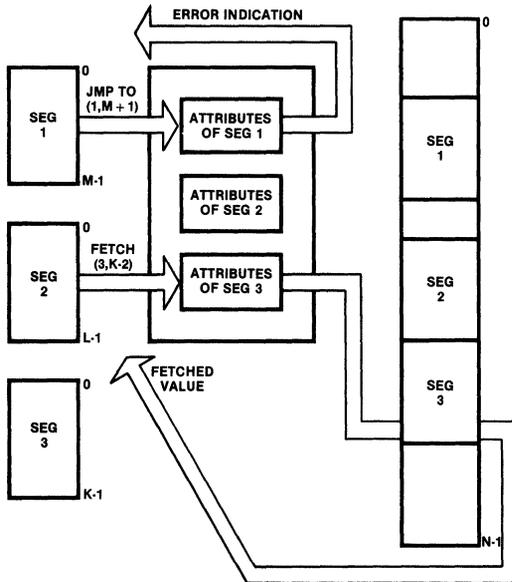


Figure 7. Attribute Checking for Segmented Addressing

Once the mechanism has been established for the checking of accesses against the declared size of an object, it is a small step to add the checking of other attributes of objects. The problems we mentioned earlier, such as protecting one process's data or programs from accesses by another process or allowing "read only" or "execute only" accesses to a section of data or program, can be solved by associating attributes with the objects in question and checking these attributes against properties of the access. A write into a "read-only" object, a user access to a "system-only" object and other such invalid accesses can be identified and prevented.

Sharing and System-Controlled Relocation

Since physical memories do not usually have a segmented organization, a segmented addressing scheme must include a plan for memory mapping. As noted earlier, memory mapping provides the means of dealing with two of our

other major problems with linear addressing: the difficulties of implementing system-controlled relocation and of sharing objects among otherwise independent processes. We shall see shortly the specific details of how this is accomplished on the Z8000.

Avoiding Fragmentation

Our other two major problems had to do with the difficulty of creating, deleting, shrinking or expanding objects dynamically. We saw that these operations could lead to fragmentation in a linear memory space and that there were additional problems when stacks were involved. It is easy to see that segmentation provides solutions for these problems, but rather than discussing them abstractly, we shall now look at how segmentation has been implemented on the Z8000 and how the Z8000 and the MMU work together. Then we shall see concretely how all of the major problems of linear addressing have been solved by segmentation.

THE Z8001 AND THE MMU

The Z8000 has been designed with a built-in segmented addressing model. Included within the 32-bit addresses used by the Z8001 are two fields: the segment name field and the "offset." The offset is an address within the linear address space of the segment. It is called an offset because in the interpretation of segmented addresses, the offset is added to the physical memory address of the "base" of the segment to obtain the physical address of the element in question. For example, if segment 5 has a base address in

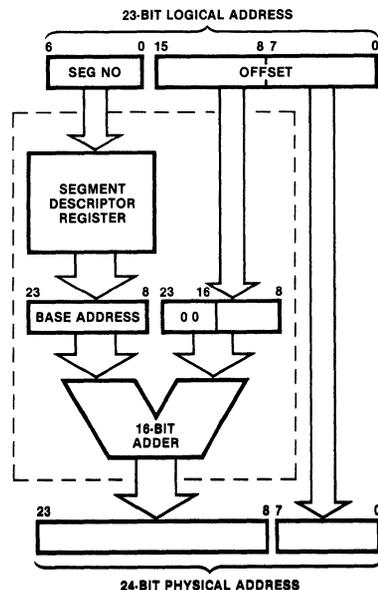


Figure 8. Z8000/MMU Address Translation

physical memory of 1024, then the physical memory location addressed by the segmented address (5,26) is 1050, because $1024 + 26 = 1050$.

The Z8001 is designed to work with an external circuit called a Memory Management Unit (MMU), which keeps track of the base addresses corresponding to the various segments and performs the computation of the actual physical addresses. This MMU can also associate a variety of attributes with each segment and can perform the corresponding access checking, generating an error interrupt (called a "segmentation trap") in the event of an invalid access.

Another feature of this implementation is that seven bits have been assigned to the segment name field and 16 bits have been assigned to the offset. This means that there are up to 128 segments, and each of them presents a linear address space of 64K bytes. Furthermore, the external MMU circuit is designed to translate only the uppermost eight bits of the offset; the low-order eight bits are passed directly to the physical memory untranslated. The practical effect of this is that a segment base address in physical memory must be a multiple of 256 (i.e., its low-order eight bits must be zeros), and the size of a segment (one of the attributes that the MMU checks) must also be a multiple of 256 bytes.

Implementing Structures Whose Size Exceeds 64K Bytes

Let's look at the effect of these implementation details on programming the Z8000 and on the efficiency of its operation. The most obvious effect is that no object can exceed 64K bytes in size; that is, any data structure that exceeds this size must consist of more than one segment. This is a genuine problem, although it rarely occurs. Fortunately, it can be solved through the use of software with very little overhead. For example, if you are dealing with an array of size greater than 64K bytes then you cannot use

```
LD RL1,RR2(R3)
```

to access the byte with index kept in R3 of the array whose base is in RR2. Rather, you must use a sequence like

```
ADD RR2, RR4    !add index to base!  
ADDB RH2, RL2  !add overflow to seg name!  
CLRB RL2       !clear "unused" bits!  
LD RL1, @RR2
```

to access the byte with the index kept in RR4 of the array whose base is in RR2. What you are doing in this case is placing several segments "end-to-end" and treating the segment name like a number. This approach is similar to "paging."

Speed of Address Translation with the MMU

A more positive aspect of the implementation details has to do with the computational overhead of using an external circuit for address translation and attribute checking. Two facts enter into this:

- Since the segment name field is not involved in the address computations of indexed, based or relative addressing, this field can be output to the MMU one cycle earlier than the offset portion of the address, so that the MMU gets a one-cycle head start on the address translation.
- The low-order eight bits of the offset, which go directly to the memory untranslated, are the bits needed first by the memory, so that the memory also gets a small head start on the transaction.

The combination of these two factors results in the use of an external MMU circuit that entails very little time penalty in memory accesses.

The point made in the previous paragraph needs to be stressed: the true independence of the segment name field from the offset in all address computations means that off-chip memory mapping can be achieved with very little overhead. This is an architectural advantage of the Z8000 that leads to an economical implementation. This can be seen by looking at how a nonsegmented CPU might achieve memory management. Undoubtedly, the approach will be a form of paging. In a paged system, the uppermost bits of the linear address are treated like a segment name field after the address computation is complete. Until the computation is complete, these bits are treated like part of a monolithic linear address--they can be changed in the course of the computation. Thus, while a paging scheme allows memory mapping and attribute checking, it suffers from many of the problems of linear addressing, and it cannot achieve the overlap of MMU and CPU computational time that is available with the Z8000 because of its true segmentation scheme. The only antidote to the computational overhead of an off-chip MMU for a linear addressed machine is to design an on-chip MMU, and with the current technology, that approach is likely to lead to a design that is short on features.

MMU Support for Stacks

One more point worth mentioning about the way that the Z8000/MMU combination implements segmented addressing concerns the use of stacks. Earlier, we noted some of the problems that are associated with dynamically expanding stack sizes. The most difficult of these problems concerned the correction of pointers into the stack when a stack was

moved to another location to accommodate a larger size. Naturally, this problem goes away with memory mapping, since the logical addresses of the locations already used on the stack don't change when it is physically relocated in memory. Furthermore, the MMU accepts as one of the attributes of a segment that it is to be used for a stack. This has two main consequences:

- There is a nonfatal stack warning interrupt that occurs when the stack is nearly full, i.e., when an access is made into the last 256 words allocated to the stack.
- The memory address computation and size specification method are altered to take account of the fact that stacks grow downward in memory from the highest addresses toward the lowest.

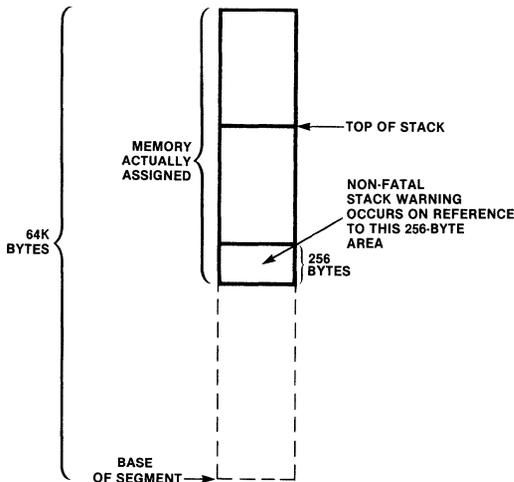


Figure 9. Stack Segments

CONCLUSION

This concludes our discussion of the specific details of the implementation of segmented addressing and memory mapping on the Z8000. We have discussed the many problems associated with linear addressing, the solution provided to these problems by the segmented addressing model and the details of segmentation on the Z8000. We have shown that segmented addressing is clearly superior to linear addressing.

Just as there are some who argue that higher level languages are "inefficient" and that they don't allow the programmer the total flexibility of assembly language programming, there are also those who adamantly reject segmentation and cling to linear addressing. The truth is that there is merit to their argument. Just as higher level languages may be inappropriate for very small systems, so also may segmentation represent "overkill" in a small memory space. The Z8000 answer to this problem is to provide large enough segments that small applications can be implemented completely within the bounds of one segment. The Z8000 CPU is provided with a mode in which addresses consist only of offsets, so that no references occur outside of the 64K byte linear address space of one segment. In fact, for applications of that size, a smaller package is provided that does not have the eight pins dedicated to the segment name output and segment error interrupt input; this smaller version cannot enter the segmented mode of operation at all.

It is a matter for subjective judgment to decide where to draw the lines between systems that are too small for segmentation, systems in which segmentation is desirable but inessential, and systems that are so large that segmentation is mandatory. The Z8000 architecture provides for a 16-bit linear address space but demands segmentation for any size above 16 bits. In its 23-bit address space, it is possible that clever, well disciplined programmers could manage to handle unrestricted linear addressing. In its ultimate 32-bit address space, there is no doubt that segmentation is the only viable approach.

This concern for the future expansion to 32-bit address spaces greatly influenced the decision to use segmented addressing in the 23-bit version. The Z8000 represents a break from the architecture of the Z80; it seems short-sighted to ask designers moving from 8-bit to 16-bit or 23-bit systems to face one architectural break today and another in a few years. This is in contrast with the situation of designers who adopt the 68000 today and who will have to face another architectural upheaval if true segmentation is introduced—that occurrence seems inevitable if the address space increases in size to 32 bits.

SUMMARY

The segmented addressing used by the Z8000 is a higher level language for memory addressing. This method is superior to linear addressing, the "machine language" of memory addressing, because it provides a model that is natural to programming.

The traditional approaches to memory allocation based upon the linear addressing model cannot solve the following problems:

- Invalid accesses
- Accommodating variable-sized objects
- Fragmentation arising from dynamic creation and deletion of objects

- Dynamic relocation
- Sharing

The Z8000/MMU combination provides a segmented addressing facility that solves the above problems and provides the following benefits:

- Built-in segmentation, memory mapping and attributes checking
- Efficient and cost effective operation
- Support for system-controlled relocation
- Support for stacks, with overflow checking
- Upward compatibility to 32-bit architecture

Operating System Support Features



Z8000 vs. 68000 Concept Paper

May 1981

One of the most striking differences between the architectures of the Z8000 and 68000 is the provision for operating system support. This area received careful consideration in the Z8000 design. The designers of the 68000 addressed most of their careful attention to other issues. This paper shows the importance of operating system support features, even in relatively small applications, and contrasts the designs of the Z8000 and the 68000 in regard to operating system support.

OPERATING SYSTEMS

Every computer application contains an operating system—either explicitly or implicitly. For the purpose of this paper the following definition of an operating system is used:

The portion (hardware and software) of a computer application that is devoted to managing hardware and software resources.

Most definitions of "operating system" are similar to this one. The idea of resource management is central to everyone's idea of an operating system. The resources of a computer application can be divided (approximately) into the following categories:

- Processing elements (e.g., CPUs, floating point chips, "intelligent" disk controllers)
- Storage elements (e.g., ROM, RAM, disks, tape)
- External interfaces (e.g., I/O ports, modems)
- Programs (e.g., compilers, application programs)

A process (also called a task) is the ongoing execution of a program by one or more processing elements. For example, a compilation is a process. The goals of a computer application can be viewed as the completion of processes. From this point of view, the job of the operating system is to "direct traffic" for as many processes as it makes sense to run concurrently, allocating re-

sources among these processes and resolving conflicts according to an externally selected policy. Directing traffic entails:

- Protection of the operating system and of each process from damage or invasion of privacy arising from the actions of any other process.
- Establishment, support, and enforcement, of protocols and conventions for the interactions of system elements.
- Facilitation of interprocess communication and sharing.

Thus, the responsibilities of an operating system are:

- Allocation and protection of processing and storage elements, external interfaces, and programs.
- Definition, facilitation, and enforcement of protocols and conventions.
- Communication and sharing.
- Policy enforcement.

ARCHITECTURAL SUPPORT FOR OPERATING SYSTEM RESPONSIBILITIES

The operating system responsibilities listed above differ from system to system. For example, the work of a small application may be carried on by a single process, although the system process that handles external device interrupts will share the CPU with the application process. There are several kinds of architectural support that facilitate the operating system's task in a wide range of applications:

- Restriction of access to CPU facilities
- Restriction of memory use
- Memory mapping
- Sharing of programs and data
- Program relocation
- Stacks
- Context switching
- I/O system and interrupts
- Distributed control
- Support for conventions

Each of these features deals with one of the four responsibilities listed above--allocation and protection, protocols and conventions, communication and sharing, and policy enforcement.

Restriction of Access to CPU Facilities

The operating system must allocate the CPU to a process while still protecting itself and other processes. That is, the operating system must be able to turn the CPU over to a process and be assured that the process does not perform potentially destructive actions. A key to solving this problem is some kind of restriction of CPU use. Most CPU designs introduce a restricted "user" mode, in which certain instructions (called privileged instructions) cannot be executed and key CPU registers (called control registers) cannot be accessed.

The existence of a user mode and privileged instructions does not solve the entire protection problem. The other half of the solution involves restriction of access to memory and I/O. In addition, the introduction of user mode brings another problem, namely the question of how the CPU passes between system and user modes (especially, how it gets out of user mode). A solution frequently provided is one or more System Call instructions. These instructions allow programs running in user mode to call system mode programs without allowing the user mode program to retain control of the CPU once it has left user mode.

Restriction of Memory Use

Most CPU designs call for some sort of comprehensive memory management facility, which provides a unified approach to restriction of memory use, memory mapping, program relocation, sharing of programs and data, and stack use.

Restriction of access to memory usually depends upon the use of sets of attributes associated with portions of the memory address range of the CPU. These attributes are checked against certain access rights associated (implicitly or explicitly) with each process. Then, for example, if a program in user mode attempts to access a memory address whose attributes don't match the program's access rights, the CPU will trap to a system routine designed to deal with such invalid accesses.

The portions of the memory address range to which sets of attributes can be assigned depend upon the CPU addressing scheme and the memory management facility. Typically, in a machine using two-dimensional (segmented) addressing, attributes are associated with a segment. In a machine with linear addressing, attributes are usually associated with fixed-size blocks of addresses called pages. (See the Z8000 vs. 68000 concept paper, "Segmented vs. Linear Addressing.")

Memory Mapping

As noted above, the operating system allocates memory and programs and facilitates sharing and interprocess communication. These tasks are aided by memory mapping.

Memory mapping is the establishment of a function that assigns to each address (now called a logical address) in the memory address range an address in the actual physical memory available to the application. Naturally, a completely arbitrary function would be difficult to specify and to alter, so the usual approach is to divide the logical address space into blocks of contiguous addresses and to map each block to a block of contiguous physical addresses. All that is required to specify such a mapping is to provide the base physical address for each of these blocks and, if not predetermined by the architecture, to provide the origins or sizes of the blocks of logical addresses.

In general, the blocks of logical addresses that can be mapped separately are the same as the blocks of memory that can be assigned attributes. The information about block size, base physical address, and attributes is usually grouped together into a segment or page descriptor.

Sharing of Programs and Data

Given memory mapping and access restriction, it is easy to see how the operating system can facilitate the sharing of programs and data among processes while still providing protection. For example, a block of physical memory containing a generally useful program can be "placed in the maps" of several processes. That is, each process can have a block of logical addresses (not necessarily the same addresses if the program is relocatable) that is mapped into the given block of physical memory. Similarly, a block of data can also be placed in the maps of several processes.

In the above examples of sharing, protection is provided by the access restriction mechanism discussed earlier. In the case of a program, for example, one of the attributes of the associated block of logical addresses can be that it is read only or even execute only. [This requires the existence of instructions and addressing modes that allow the generation of pure (i.e., not self-modifying) code.] Furthermore, the attributes can change, depending upon which process is running. For example, when the process responsible for a given block of data is running, the attributes of the associated block of logical addresses can be unrestricted, and when other processes are running, the attributes can include read only. The changing of attributes discussed here is part of the context switching accompanying the transfer of the CPU from one process to another. (In a multi-CPU system, the protection has to be provided by a difference in the access rights of the processes, not by a change in attributes.)

Program Relocation

There are three types of relocation: static relocation, dynamic logical address relocation, and dynamic physical address relocation. Static relocation is the kind provided by a relocating loader. Separate source files are assembled as if each were to begin at logical address zero, and a program combines these separate files into one large program designed to be run at fixed logical addresses. Dynamic logical address relocation is the process of changing the logical addresses at which a given program is to run. In most cases, this is possible only if the program has been designed to be independent of the logical addresses at which it runs. Dynamic physical address relocation is the process of changing the physical addresses at which a given program is to run, while leaving its logical addresses unchanged. By definition, dynamic physical address relocation makes sense only in connection with memory mapping.

Static relocation is possible regardless of the CPU architecture, whereas both kinds of dynamic relocation depend upon architectural support features. Both kinds of dynamic relocation help the operating system meet its responsibilities: Logical address relocation, as noted above, is important in program sharing, because it allows the logical address spaces of the processes sharing a given program to be managed independently. Physical address relocation is important in the allocation of memory, because it allows "garbage collection" to be performed easily and facilitates the implementation of virtual memory schemes.

Dynamic Logical Relocation. Dynamic logical relocation depends upon the ability to write programs that are independent of the logical addresses at which the program's instructions reside. Most CPUs provide some support for such programs. To understand this support, we must first understand what computer instructions do.

The CPU interprets instructions that are stored in memory. Each instruction must specify (explicitly or implicitly):

- The operation to be performed.
- The locations of the arguments.
- The location of the next instruction to be executed.

Computers have been designed (e.g., the IBM 650) in which all of these addresses are specified explicitly in each instruction. Obviously, no program on such a computer can be independent of the addresses at which the instructions reside. On most computers, however, position-independent means are available for specifying the locations of arguments and the sequence of instruction execution. These means all rely upon the use of registers and special addressing modes. The most fundamental of these registers is the Program Counter (PC), which is found in all modern com-

puters; the associated addressing mode is called PC-Relative (or simply Relative) Addressing.

The PC contains the address of the next instruction to be executed. As each instruction is fetched from memory, the PC is changed to contain the address of the first memory location following the fetched instruction. Thus, in most cases, the sequence of instruction execution is defined by the sequence in which the instructions are stored in memory. This leaves only the case of transfer-of-control instructions (i.e., instructions that change the value of the PC) to be considered. Most modern computers have transfer instructions that add a signed offset (usually contained in the instruction) to the PC value. That is, these transfer instructions use relative addressing.

Obviously, transfer instructions that use relative addressing can be represented independently of the addresses at which the program containing the instructions resides. Other position-independent means of changing the PC are available on many computers:

- Indirect Register Addressing--the PC is set to a value specified in a register.
- Based addressing--the PC is set to the sum of an address value specified in a register and an offset specified in the instruction or in a register. (Many variations of based addressing exist.)
- Popping from a stack--the PC is set to a value previously saved on a stack.

Position-independent argument specification is achieved similarly. Based addressing, stacks, register addressing (the analog for argument specification of Indirect Register mode for transfers), and even Relative Addressing (for program constants) are commonly used ways of specifying arguments, which are available on many computers.

Dynamic Physical Relocation. Changing the physical addresses at which a program resides without changing the logical addresses is only possible with a memory-mapping scheme. Given memory-mapping, this kind of relocation requires no further architectural support.

Physical relocation is helpful to an operating system that must allocate a limited amount of memory among a varying set of processes or among processes with varying memory needs. It helps avoid the fragmentation of memory that can occur when there is dynamic allocation and release of varying amounts of memory. Physical relocation is also the basis of any virtual memory system.

A virtual memory system is an addressing scheme in which the logical address space is larger than the physical memory. Parts of the logical address space correspond to blocks of secondary storage, which are brought into physical memory only when a program attempts to access them. A virtual memory system requires extensive CPU support, since it

involves, in effect, the transfer of information to or from secondary storage in the midst of executing an instruction.

Stacks

Stacks are an important tool for meeting the operating system's responsibilities. A stack is a variably sized last in, first out memory. Associated with a stack are two operations, pushing (adding an item) and popping (removing an item).

Stacks are used by the operating system (explicitly or implicitly) to allocate memory in a flexible way that, in connection with based addressing, allows programs that need nonregister storage to remain position independent. Special cases of this are the storage of return addresses for subroutine calls and machine state for interrupt processing.

Stacks provide an important application of dynamic physical relocation, because the way they are used makes logical relocation of stacks almost impossible. In order to provide flexible allocation of stack space, the operating system must be able to expand a stack upon demand. This sometimes entails physical relocation of the stack to a larger area of physical memory, since with based addressing, a stack must consist of contiguous logical addresses and since most memory-mapping schemes require contiguous logical address blocks (below a minimum size) to map into contiguous physical addresses.

Other architectural features desirable for stack support include:

- The ability to designate one or more stacks for program use.
- Single- and multiple-argument push and pop instructions.
- The ability to address items at locations defined relative to the top of a stack.
- Automatic warning (traps) of impending stack overflow or underflow.

Most architectures call for the implementation of stacks as linear arrays in memory with an address register marking the top of the stack and providing (through based addressing) access to items at other locations in the stack. The stack register is a dedicated (special-purpose) register in some architectures. In other architectures, any address register can be used as a stack register, although the program usually cannot specify which stack register is to be used for saving returns from a subroutine or the machine state on interrupts.

The implementation of stacks as arrays in memory and the use of general-purpose address registers for stack registers make the provision of overflow and underflow protection difficult. Architectures that provide stack limit protection usually do so through the use of the attribute specification associated with memory protection. Several archi-

tectures provide stack access protection by means of a rudimentary separation of stack and data address spaces through externally interpreted CPU status outputs. A better approach is provided by a two-dimensional (segmented) addressing scheme, in which distinct objects and not just stacks can be assigned to independent parts of the address space. (See "Segmented vs. Linear Addressing," Z8000 vs. 68000 Concept Paper.)

Context Switching

One of the difficulties of running several processes concurrently is the overhead associated with context switching. The context of a process is the portion of its state that occupies shared resources. For example, since most CPU architectures call for only a single Program Counter (PC), all processes must share this register, so the PC value of each process is part of its context. Most architectures also call for a single set of general-purpose registers, control registers, CPU status registers, and so forth. Thus, when the same CPU is allocated to more than one process, the process contexts must include the contents of any of these registers used by the processes.

Context switching is the saving of the context of one process and the recalling of the stored context of another process. Some architectural features for the support of context switching are desirable. These include automatic saving of CPU state on interrupts, single-instruction block register saving and restoring, and access to all necessary control registers.

All modern CPUs provide automatic saving of a portion of the CPU state on interrupts and access to all control registers that can form part of a process context. Block saving and restoring of registers is available with some CPUs. Either a starting register and the number of registers to be saved or a bit-encoded selection of registers to be saved provides some flexibility--not all registers need to be saved in every case. In most cases, the operating system saves registers on a stack.

I/O System and Interrupts

The operating system responsibilities pertaining to the I/O system and interrupts vary greatly with the type of application. The architecture of a general-purpose CPU must provide the flexibility necessary to accommodate the I/O requirements of a wide range of application types.

One of the operating system's most difficult tasks in this area is the control of access to I/O resources. Unlike memory, which can be divided into large, relatively homogeneous blocks, the elements of the I/O space require special-purpose management, protection, and access techniques. In addition, device timing requirements and externally set policies for conflict resolution make hardware support of I/O mechanisms mandatory.

Desirable architectural features for the support of the I/O system and interrupts include:

- A vectored interrupt scheme.
- Program-controlled specification of the CPU state to be established for each type of interrupt.
- A rapid, automatic context-switching mechanism for responding to interrupts.
- A means of defining conflict-resolution policies and "interruptibility" of interrupt processing.
- Block I/O instructions and DMA capabilities.
- Restricted access to I/O facilities.

A vectored interrupt scheme allows the CPU state to be switched immediately to an appropriate processing routine without the need for software to ascertain the interrupt type and call the appropriate routine. The port of connection or the contents of a vector supplied by the interrupting device are used to determine the new state.

A vectored interrupt scheme can be designed so that the new CPU state is specified in the hardware by the interrupting device, but in most architectures this is under program control. Most CPUs store this information in memory locations (often called interrupt vectors). In some CPUs a fixed block of addresses is devoted to storage of interrupt vectors, but a better approach is to allow any block of locations to serve and to have a CPU control register that points at the chosen block. One advantage of this approach is that the block of vectors can be assembled with the program and need not be set individually by initialization instructions. One disadvantage is that it discourages modular management of the vectors.

Every CPU with an interrupt facility has some kind of context-switching mechanism to support it, usually involving the use of a stack. In CPUs that support multiple stacks, the architecture designates one of these stacks for this use. Those parts of the machine state that the interrupt processing routine cannot easily save by itself are pushed onto the designated stack. This saved information must include the PC value, and it can include other items as well. Usually, CPU condition indicators and operating mode bits are saved, while general-purpose register contents are not. Such CPUs have interrupt return instructions to pop the saved CPU state off the stack, thus returning the CPU to its pre-interrupt state.

Conflict resolution is controlled by a policy that is set by the system designer and enforced by the system. The usual approach is to provide a small number of priority levels to which device interrupts can be assigned, by virtue of either the means of connection to the CPU or the setting of a priority level in the "vector" for each device. Then, when the CPU is processing a device interrupt of a given priority level, only higher-level interrupts can occur.

Block I/O instructions and direct memory access (DMA) capabilities are important features that improve performance. Block I/O instructions require careful implementation. In general, they must use general-purpose registers to save their ongoing state, so that they can be interrupted. DMA capabilities require the development of bus control protocols and a means of protecting partially loaded or saved memory blocks from unwanted access by concurrently executing programs.

Restriction of access to I/O facilities can take many forms. In a CPU with a user operating mode and privileged instructions, the I/O instructions can be privileged. This is the easiest and most natural approach. In a CPU that does not have I/O instructions and a separate I/O address space (see Z8000 vs. 68000 concept paper, "Memory Mapped vs. Explicit I/O"), a memory-protection approach must be taken.

Distributed Control

One of the recent advances in operating system design is the distribution of operating system functions among many separate processes. Such distributed systems present problems of inter-process synchronization.

When processes to which separate processing units may have been allocated share a common memory, the techniques of guarded commands and semaphores (developed by Dijkstra and others) are applicable. The basic architectural support for these techniques is the atomic Test and Set instruction, a CPU instruction that tests a memory location for the value "available" and simultaneously sets the value to "not available." The word "atomic" means that there can be no other access to the given memory location between the "test" and "set" portions of the instruction, so no two concurrently running processes can find the location set to "available" simultaneously. Implementation of a Test and Set instruction requires a bus-locking mechanism.

When processes do not share a common memory, a similar nonmemory exclusion mechanism must be provided. A separate bus can carry the signals needed to implement such a mechanism, and CPU instructions can be provided to manage the CPU's connection to that bus.

Support for Conventions

One of the issues that must be considered in the design of a CPU is whether its architecture should support all conventions equally, favoring none, or whether it should encourage, through special features, specific conventions. For example, should a CPU be designed with general support for high-level languages, or should it be designed to optimize Pascal, say, at the expense of making FORTRAN programming less efficient? Should it provide special features that make a subroutine

argument passing convention using the stack especially efficient at the cost of decreased efficiency for other argument passing conventions?

In practice, there are many cases in which the choice to support one method of accomplishing a task makes the designer's job easier, but discourages the use of other, equally valid approaches. If the other approaches are no better than the one supported, then support of one specific approach is a net advantage. But if the unsupported approach is preferable to the supported one in some applications, and if the special support feature makes the unsupported feature less efficient than it would otherwise have been, then there is a net disadvantage for those applications.

Another aspect of the system's support for conventions is the definition of the CPU's operating environment provided by a coherently designed family of components and a compatible interconnect bus. In most CPU architectures, this definition of the CPU's operating environment is not given much attention. Key points that should be considered are:

- The need for a staged, modular development--over many years--of a CPU and its component family.
- The importance of changing the distribution of function between the CPU and associated components with minimal impact on existing programs.
- The need for future enlargement of capabilities without substantial redesign of existing components or systems.

THE Z8000 APPROACH

The Z8000 designers were aware of all of the operating system support features mentioned in the preceding section, and an attempt was made to provide for these support features in the Z8000 architecture. Naturally, other design criteria were present and tradeoffs were made, but on the whole, a better and more unified approach to operating system support was taken with the Z8000 than with other CPUs in its class.

Restriction of CPU Use in the Z8000

The Z8000 has a system/normal bit in its Flag/Control Word register (FCW). When the bit is in the normal state, privileged instructions cannot be executed. Operating system tasks are expected to execute in the system mode. The privileged Z8000 instructions are:

- I/O instructions, including the interrupt return and nonmemory synchronization instructions.
- Control register manipulation instructions.
- The Halt instruction.

In addition to privileged instructions, another protection feature is associated with the system/normal bit. There are two copies of the implied stack register (the stack register used for interrupt and subroutine returns); one is used when the CPU is in system mode, the other when it is in normal mode. Programs executing in normal mode have no access to the system mode stack register.

Passing between system and normal modes requires a change to the FCW. This can only be accomplished through a privileged instruction (LDCTL, IRET, LDPS) or automatically in response to an interrupt or trap. A system call trap (a one-word instruction with eight programmable bits) allows a normal mode program to call one of 256 system mode programs.

Memory Management in the Z8000

The Z8000 design provides for a comprehensive memory management facility, which offers a unified approach to restriction of memory use, memory mapping, sharing of programs and data, program relocation, and stack use. This memory management facility is integrated with a segmented (two-dimensional) addressing scheme in the CPU. (For a discussion of this entire area, see the Z8000 vs. 68000 concept paper, "Segmented vs. Linear Addressing.") One of the many advantages of segmentation is that it is an ideal organization for a system (e.g., UNIX*) in which there are many small tasks.

Another feature of the Z8000 memory management facility is that it is designed to facilitate the implementation of virtual memory systems. Virtual memory is an important technique for handling the enormous address space of a CPU like the Z8000 with a reasonable amount of physical memory. Details of the implementation of Z8000 virtual memory systems are available in the articles on the Z8003/Z8004 by Calahan, Patel, and Stevenson and on the PMMU by Hu, Lai, and Stevenson (both to appear in "Electronics" in late summer 1981).

Context Switching in the Z8000

Z8000 interrupt and trap-handling provides an automatic, rapid context switch from the executing program to the interrupt-processing routine. The FCW and PC values and a "reason" are saved on the system mode stack, and new FCW and PC values are set from the program status area (PSA) entry corresponding to the interrupt type.

The Z8000 block register saving and restoring instructions facilitate context switching. These instructions can be used to simulate the pushing or popping of a block of registers to or from any stack.

* UNIX is a registered trademark of Bell Laboratories.

In some cases, the values of control registers are essential to the context of a process. (The normal stack register and the FLAGS register are obvious examples.) A load control register instruction allows the transfer of any of these registers to or from a general-purpose register, so they can be saved and restored like other registers.

The Z8000 Component Family and the Z-BUS

A fundamental concept in the Z8000 architecture is that of a family of components designed to work together. Because a CPU and its associated component family must be developed and introduced over a span of several years, the outlines of the family and the framework to support it must be established at the outset. In the case of the Z8000, the CPU chip was designed with many hooks in place, including segmented memory addressing, nonmemory synchronization instructions, block I/O instructions, a multiplexed address/data bus, an encoded 4-bit CPU status output, and extended processor instructions. The extended processor instructions are an especially good example of this planning. Processor instructions and a bus protocol allow for the development of "slave" processors (like a floating-point chip) that execute instructions taken from the CPU's instruction stream and have access to the CPU's addressing capabilities.

The ZCITM Z-BUS Component Interconnect provides the signal lines and protocols required to tie members of the Z8000 family together and provides the necessary interface specification for family members still to be developed.

An even wider environment for the CPU is defined by the ZBITM Z-BUS Backplane Interconnect, which is compatible with the ZCI and provides for expansion of the Z8000 to a full 32-bit architecture.

The Z8000 I/O System and Interrupts

The Z8000 uses a block of memory called the program status area (PSA) to store interrupt vectors (i.e., the new CPU status) for each type of interrupt and trap. In addition to separate lines for nonvectored and vectored interrupts and a non-maskable interrupt for situations that can't wait, there is a table of PC values to be indexed by an 8-bit vector placed on the address/data bus by the interrupting device. The block of memory used for the PSA is not fixed, as with some CPUs; it can be anywhere in memory, and a pointer to it (the PSAP register) can be set using the privileged LDCTL instruction.

Conflict resolution is done simply. The three kinds of interrupt (nonmaskable, nonvectored, and vectored) are assigned three levels of priority by the CPU. In addition, the vectored and nonvectored interrupt lines can be masked (using the privileged Disable/Enable Interrupt instruction),

so that interrupts are not processed until the unmasking of the associated line. When interrupts arrive on more than one line simultaneously, the priority determines which is processed first. The processing routine for any interrupt type can be interrupted by the routine for any other if the corresponding line has not been masked. Whether other lines are to be masked or not can be determined automatically by specifying the appropriate mask bit in the FCW portion of the PSA entry. Otherwise, the determination can be made by the program, which can bracket sensitive code between Disable Interrupt (DI) and Enable Interrupt (EI) instructions.

A daisy chain is used to determine the order of processing of interrupts from devices attached to the CPU on the same interrupt line. In this way, devices closer to the CPU can interrupt the processing of interrupts from devices farther away from the CPU, unless the given line is masked during all or part of the processing.

A key aspect of the Z8000 I/O system is the protection provided by privileged instructions. This protection allows an operating system to manage the I/O interfaces without interference from normal mode programs.

Distributed Control

The Z8000 architecture provides ways to synchronize processes that share memory and those that do not. The Test and Set instruction provides the basis for synchronization of processes that share memory. For nonmemory synchronization, the Z-BUS has a set of lines and a protocol for resolving simultaneous requests for shared resources, and the CPU provides instructions to support the bus connection and protocol.

Support for Conventions

The Z8000 design supports many conventions. Principal among these are

- Use of a segmented address scheme.
- Use of message passing for interprocess communication.
- Component and backplane bus protocols.
- Interrupt protocols for all components.

The only convention listed above that has not yet been discussed is message passing. A message is a set of characters sent by one process and received, asynchronously, by another. The processes do not need to know whether they have been allocated the same or different processing elements.

The Z8000 family architecture provides message passing support both in the CPU and in other components:

- Block I/O instructions in the Z8000 CPU support message passing.
- The Z-FIO (FIFO I/O unit) chip provides the asynchronous interprocessor connection necessary to a message-passing philosophy.
- The Z-UPC (Universal Peripheral Controller) chip accepts commands from and delivers messages to the master CPU in designated message registers in the Z-UPC.
- The DMA (Direct Memory Access) chip has been designed with a "flyby" mode that allows external devices (e.g., a Z-FIO chip) high-speed direct access to memory without storage of the data by the DMA chip.

THE 68000 APPROACH

The 68000 provides many of the architectural support features mentioned earlier, but there are several that did not receive the same careful attention given to the Z8000 design.

Restriction of CPU Use in the 68000

The 68000 has system and user modes and privileged instructions, just like the Z8000. The two main differences are:

- In the Z8000 I/O instructions are privileged, while in the 68000 ordinary memory reference instructions double as I/O instructions and thus cannot be privileged.
- The Z8000 has one System Call (SC) instruction with 8 programmable bits, while the 68000 has 16 separate System Call instructions, none of which has any programmable bits.

The 68000 operating system designer is forced to use an external memory management system to implement the protection of I/O operations. The Z8000 operating system designer can work with privileged instructions--a tool that is consistent with the tools used for protection of other key Z8000 functions. (For a detailed discussion see the Z8000 vs. 68000 concept paper "Memory-Mapped vs. Explicit I/O.")

The second point boils down to the number of distinct instructions available for system calls and the number of separate traps over which these instructions are distributed. The 68000 architecture provides for a total of 16 system calls and ties them all to separate traps. The Z8000 architecture provides for 256 system calls and ties them all to one trap, so that dispatch software is required to route the calls to the proper routines, but obviously, the Z8000 design can accommodate the addition of a hardware dispatch mechanism in the future with no change to user programs. Furthermore, the 68000 approach forces duplication of context-saving operations (e.g., register saving).

The key difference is that the Z8000 has 256 System Call instructions, while the 68000 has only 16. The Z8000's 256 calls will accommodate the

needs of most applications. The 68000's 16 calls will be too few for all but the simplest cases, so most 68000 applications will be unable to use the system and user modes in a natural way.

Memory Management in the 68000

The key point to understand about the 68000 memory management mechanism is that it is completely external to the CPU. Several facts follow from this:

- The 68000 cannot use segmentation without extensive software overhead.
- Because of the overhead that a fully general mapping facility would entail, the 68000 must use a "simple but powerful" scheme that limits mapping to the bitwise replacement of fields in the address.

Naturally, many benefits can be derived from the use of such a scheme, but it can never be more than a separate, after-the-fact transformation and checking mechanism, so that the 68000 is not relieved of the problems of linear addressing. (See the Z8000 vs. 68000 Concept Paper "Segmented vs. Linear Addressing".)

Context Switching in the 68000

There is very little difference between the Z8000 and the 68000 in their approaches to context switching.

The 68000 Component Family and Bus

The idea of a family of components designed to work together, which is so fundamental to the Z8000 philosophy, is not clearly evident in the 68000 design. While the Z-BUS forms the framework for the entire, expanding Z8000 Family, the 68000 seems to have been designed with an eye toward compatibility with the older 6800 family peripherals and with existing bus structures.

While the Z8000 is designed to include features to facilitate its integration into a family of components, the 68000 seems to have been designed before there was a clear conception of what its environment would be. For example, the Z8000 has provision for memory management integrated into the CPU; the 68000 memory management mechanism is entirely external to the CPU. The Z8000 has a nonmemory synchronization facility; no bus or processor provision for nonmemory synchronization exists in the 68000. The Z8000 was designed with a multiplexed address/data bus to accommodate the advanced programmable peripherals designed with it; the 68000 was designed with a nonmultiplexed address/bus (See the Z8000 vs. 68000 Concept Paper "Multiplexed vs. Non-Multiplexed Address/Data Bus"). The Z8000 was designed with block I/O instructions to facilitate the message passing protocols to be used with the Universal Peripheral Controller and, via the FIFO interface, with other

processors and devices; the 68000 does not seem to support any particular interprocess communication protocols and techniques.

These examples only serve to emphasize the key point: the Z8000 design is based on a family concept, the 68000 design is not.

The 68000 I/O System and Interrupts

The Z8000 and 68000 interrupt systems are similar, with roughly equivalent capabilities. Notable differences are:

- Location of the interrupt vectors. In the Z8000, the PSA can be anywhere in program memory. In the 68000, the interrupt vectors must be specific locations in data memory.
- Resolution of priority. The 68000 requires each device to supply a 3-bit interrupt request code that is used in determining the device's priority. In the Z8000 Family, devices can be used at any priority level without modification. Each device is attached to one of three interrupt request lines, and each line has a different priority. A daisy-chain protocol defines priorities among devices attached to the same line.

A key difference between the Z8000 and 68000 I/O systems is the use of explicit I/O instructions in the Z8000 and the use of memory mapped I/O in the 68000. (See the Z8000 vs. 68000 Concept Paper "Memory Mapped vs. Explicit I/O".) Among other problems, this forces the 68000 to use the coarse-grained protection of memory management to do what the Z8000 accomplishes with privileged I/O instructions.

Distributed Control

The 68000, like the Z8000, has a Test and Set instruction for synchronizing processes that share memory. The 68000 lacks a mechanism for nonmemory synchronization such as is provided by bus protocols and CPU instructions on the Z8000.

Support for Conventions

The only area in which the 68000 provides support for an operating system's definition of conventions is subroutine argument passing. By means of its Link and Unlink instructions and its stack-oriented addressing modes, the 68000 design en-

courages a stack-based argument-passing scheme. While the Z8000 also allows an efficient stack-based argument-passing convention, it provides equally good support for a register-based convention.

In the areas of memory addressing, component and backplane bus protocols and interprocess communication, the 68000 provides little support for the framework of conventions that an operating system must provide. Especially striking is the contrast between the Z8000's system-wide support of message passing for interprocess communication and the 68000's failure to provide any special support for interprocess communication.

SUMMARY

Operating systems are responsible for the allocation and protection of processing and storage elements, external interfaces and programs; for the definition, facilitation and enforcement of protocols and conventions; for communication and sharing; and for policy enforcement.

Several kinds of architectural support help operating system designers meet these responsibilities: restriction of access to CPU facilities, restriction of memory use, memory mapping, sharing of programs and data, program relocation, stacks, context switching, an I/O system and interrupts, distributed controls and support for conventions. Both the Z8000 and the 68000 provide these kinds of support, but the Z8000 approach is more integrated and far-reaching.

The most notable differences between the Z8000 and the 68000 are:

- The support for virtual memory in the Z8000.
- The lack of privileged I/O instructions and the scarcity of System Calls on the 68000.
- The lack of a family concept in the 68000 design, and the resulting lack of cohesion among the 68000 and its peripheral components.
- The Z8000's greater flexibility in the specification of interrupt vectors and the determination of device priorities.
- The lack of a provision for nonmemory process synchronization in the 68000 design.
- The absence of support for message passing (or any other interprocess communication scheme) in the 68000 design.

The Z8000 is seen to provide superior support for operating system functions.

A tale of four μ Ps: Benchmarks quantify performance

Aided by portions of the Carnegie-Mellon test package and with the cooperation of DEC, Intel, Motorola and Zilog, EDN compares the performance of four popular processors.

Robert D Grappel, Consultant,
and **Jack E Hemenway**, Consulting Editor

In this article, EDN proudly publishes the results of the first comprehensive benchmark study of four major 16-bit processors: the Digital Equipment Corp LSI-11/23, Intel 8086, Motorola 68000 and Zilog Z8000. You'll find these results highly interesting, and they should help you choose the best device for your application.

Don't assume, however, that limiting the study to four devices implies that they are the four "best" machines; we hope that future articles will add new processors to the comparisons. Nevertheless, any benchmark study must start somewhere, and these machines seem representative of those used in today's systems.

Why the need for a benchmark study at all? One sure way to start an argument among computer users is to compare each one's favorite machine with the others. Each machine has strong points and drawbacks, advantages and liabilities, but programmers can get used to one machine and see all the rest as inferior. Manufacturers sometimes don't help: Advertising and press releases often imply that each new machine is the ultimate in computer technology. Therefore, only a careful, complete and unbiased comparison brings order out of the chaos.

The benchmarking process isn't a new one; Special Features Editor Robert Cushman has explored much the same ground for older processors that this article does for newer ones (EDN, April 20, 1975, pg 41). The modern devices are more powerful, but the task of choosing the right one for a given application is no simpler.

Who chose the benchmarks?

Benchmarking anything as complex as a 16-bit processor is a very difficult task to perform fairly. The choice of benchmark programs can strongly affect the comparisons' outcome, so the benchmarker must choose the test cases with care. The programs used in this

article were compiled in 1976 by a group at Carnegie-Mellon University for use in benchmarking minicomputers and mainframes. The group presented the results of those benchmark tests at the 1977 National Computer Conference in the paper "Evaluation of

Benchmark A—I/O interrupt kernel

The test case chosen for this benchmark consists of one interrupt at each of the four levels. The times shown are the sum of the four interrupts, computed by counting the number of the instruction cycles by hand and including the time required for the processor to recognize and process an external interrupt.

Processor	Clock Speed (MHz)	Code Bytes	Execution Time (μ sec)
LSI-11/23	3.33	20	114
8086	10.00	55	126 (note)
68000	10.00	24	33
Z8000	6.00	18	42

Note that the 8086 implementation of this benchmark saves the complete machine context on the stack; it's the only implementation that does so.

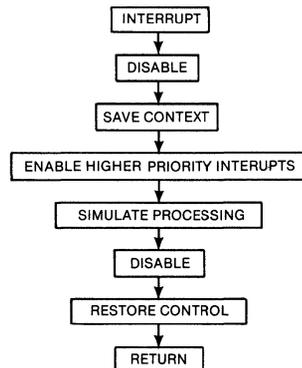


Fig 1—This benchmark tests a processor's basic interrupt capability. Some machines require external hardware, while others put limitations on the number of interrupt vectors supported.

When faster chips are developed, execution times will decrease

Computer Architectures via Test Programs," by S H Fuller, P Shaman, D Lamb and W E Burr.

The set of programs includes many common algorithms that appear frequently in real-world applications. They test the ability of a computer to handle data in chunks ranging from individual bits to 32-bit integers to floating-point numbers. The tests include interrupt handlers and character-string searches, bit manipulations and sorting. Taken as a set, they encompass a fair test of a computer's real power.

For our benchmark tests, we have chosen a subset of this Carnegie-Mellon set; we have not included the benchmarks dealing with floating-point math or virtual-memory handling. We excluded floating-point math because most of the 16-bit processors don't support floating-point operations; we would thus have ended up benchmarking their floating-point software or external math processors. Similarly, the processors don't provide virtual-memory support. We also excluded two benchmarks that require extensive number-crunching capability (Fourier transforms and Runge-Kutta integration) because their results would depend so heavily on the floating-point support used. The remaining seven benchmarks (labeled A, B, E, F, H, I and K in the Carnegie-Mellon literature) provide a sufficient test of each processor without handicapping any of the contestants.

Who coded the benchmarks?

Clearly, once you've chosen a benchmark set, coding it is the next critical task. We wanted to show each machine in the best possible light. Therefore, we asked a representative of each manufacturer to code the set for that manufacturer's machine, assuming that the manufacturer would best understand its machine's features.

To referee this process, EDN then reviewed each program. Additionally, we circulated copies of each program for comments to the programmers of each of the other computers. This process ensured careful proofreading and close adherence to the benchmark-test rules.

We did allow minor variations in the benchmark implementations where we felt they were justified. Two of the processors (the 8086 and Z8000) have special character-string instructions—we didn't want to penalize these devices by forcing their manufacturers to code unnecessary loops. Additionally, the 8086 has no general bit-manipulation instructions—a feature that sometimes produced differences among the bit-level benchmark implementations. The processors also differ widely in their extended-addressing capabilities, so we allowed some latitude in addressing mechanisms.

The manufacturers coded each benchmark in assembly language, for several reasons. First, of course,

Benchmark B—I/O kernel with FIFO processing

The test data for this benchmark consists of the following set of interrupts:

- Level 1 once
- Level 2 once
- Level 3 once
- Level 4 once
- Level 2 three times (forces queuing of interrupts)
- Level 3 five times.

The times shown are the sum of all of these interrupts. We hand-computed these times by counting instruction cycles, assuming worst-case arrival times.

Processor	Clock Speed (MHz)	Code Bytes	Execution Time (μsec)
LSI-11/23	3.33	86	1196
8086	10.00	85	348
68000	10.00	118	390
Z8000	6.00	106	436

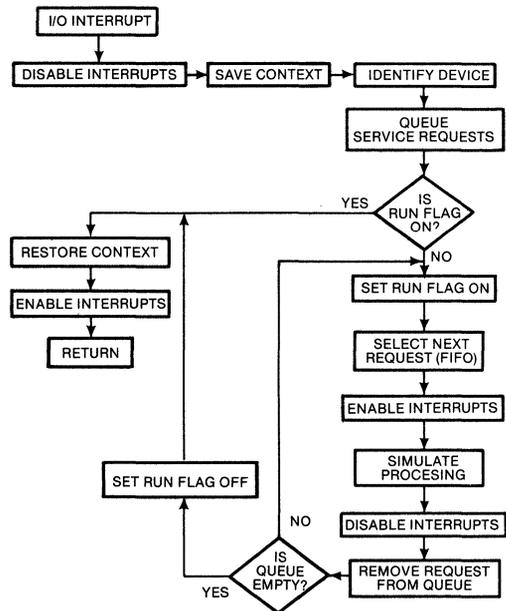


Fig 2—Extending Benchmark A, this test includes a FIFO buffer that queues interrupts as they arrive. The 68000 and Z8000 FIFO implementations trade a few words of memory for a fast and simple queue structure.

there is no mutually acceptable high-level language available on all the machines. But in any case, EDN wanted to benchmark the processors and not the capabilities of compiler writers. The documentation provided to each programmer was the Carnegie-Mellon specification, in the form of flowcharts or a PASCAL-like pseudocode. The representatives required several iterations to get all the programs to work and then to optimize them; the final code appears at the end of this

Each manufacturer coded the programs for its processor

and position independent.

Finally, Benchmark K (Fig 7) transposes a matrix of bits. This test further checks out a processor's bit-manipulation facilities, as well as exercising loop constructs. It assumes a tightly packed bit matrix starting on a word boundary, and its program must be re-entrant and position independent.

On your marks, get set,...

Before examining the programs themselves, pause and recall the μ P state of the art a scant few years ago. A programmer would have been hard pressed to write many of these benchmarks on the 8-bit μ Ps available then; for one thing, the requirements of position independence and re-entrancy would have caused nightmares. In this light, the power of all the processors described in this article is very impressive.

Turning now to the benchmark programs, we note that all of them appear to be properly coded and bug free. However, some differences among them deserve comment.

There's a major difference in "philosophy" between machine designers and programmers who favor registers and those who favor memory—a fact that's especially clear in the case of passing parameters to subroutines. Note in this regard that the various processor manufacturers are not especially consistent in this area. The LSI-11/23 programs use the stack for parameters; the 68000 and 8086 sometimes use the

stack and sometimes use registers; the Z8000 uses registers exclusively. Because our benchmark specifications merely say "re-entrant," either method is satisfactory, provided that the programs furnish register-save and -restore instructions. Using the stack is "cleaner," but registers are usually faster.

A problem of benchmark interpretation is apparent for the specification of Benchmark A: What does "Save Context" mean? One reading would suggest making a copy of the machine state (registers, program counter, condition codes, etc) while another might argue that all the spec calls for is saving the contents of any registers actually used (assuming that the actual interrupt-processing routines save and restore registers). Clearly, you get more compact code and faster execution times by choosing the "save only what is used" approach; only Intel performed an explicit and complete context-save operation.

Also observe that several of the interrupt benchmarks use special hardware. The 8086 version, for example, assumes that an 8259 PIC chip is available to field and prioritize interrupts. And the LSI-11/23 has fully vectored interrupt hardware. (Thus, the interrupt benchmarks would run slightly slower on an LSI-11/2, because the processor is without such hardware.) The Z8000 assumes device daisy chaining, while the 68000 uses its built-in vectoring.

Finally, note the ONTRACE and OFFTRACE instructions in some of the LSI-11/23 code. DEC has used these commands to trigger a logic analyzer at the start and end of each routine being timed. Other benchmark programs also contain instructions intended for timing purposes. We didn't include these instructions in our code-byte totals, though.

Benchmark F—Bit set, reset, test

The test data for this benchmark is an array of 125 bits consisting of alternating ZEROs and ONES. The array begins on a word boundary. The times shown are the sum of the following nine tests:

Test	Function	Bit Number
1	TEST	10
2	TEST	11
3	TEST	123
4	SET	10
5	SET	11
6	SET	123
7	RESET	10
8	RESET	11
9	RESET	123

Note that the processors should perform these tests in this order without resetting the bit string. Motorola and Intel hand-computed their times; the others come from actual computer runs with real-time clocks or logic analyzers.

Processor	Clock Speed (MHz)	Code Bytes	Execution Time (μ sec)
LSI-11/23	3.33	70	799
8086	10.00	46	122
68000	10.00	36	70
Z8000	6.00	44	123

```

procedure BITTEST(F,N,A1,RC,WORK)
integer ABIT,D

ABIT := A1+N/(word length)
D := N mod (word length)

if Dth bit at address ABIT-1
then RC := 1
else RC := 0
end-if

if F = 2
then Dth bit at address ABIT := 1
else if F = 3
then Dth bit at address ABIT := 0
end-if
end-if
    
```

Fig 4—Benchmark F exercises each processor's bit-manipulation capabilities. The 8086 and Z8000 use a somewhat different algorithm than the other two devices; they always test the specified bit, regardless of the function code. If they then find the bit to be ZERO and the function to be Reset, they merely return. On the other hand, if they find the bit to be ONE and the function Set, they also return. Finally if the function is Test, they can return regardless of the bit value. This version of the algorithm saves some execution time at the expense of code that's not as clear. The 8086 doesn't have the bit-manipulation instructions of the other processors, so it must use shifts and other instructions to perform the bit-manipulation operations.

Benchmark H—Linked-list insertion

This data set starts with an empty list, into which five records are inserted with keys (32-bit hexadecimal numbers), as shown. The timings are for the sum of all five insertions:

1. 12345
2. 12300
3. 13344
4. 12345
5. 34126

Motorola hand-computed the 68000 timings; the remainder come from real-time clocks or logic analyzers.

Processor	Clock Speed (MHz)	Code Bytes	Execution Time (μsec)
LSI-11/23	3.33	138	592
8086	10.00	94	—
68000	10.00	106	153
Z8000	6.00	96	237

```

procedure LISTINSERT (LISTCB, NEWENTRY)
  "the notation POINTER.FIELD is used to access a
  particular field of the structure pointed to by POINTER"

  pointer PRESENT
  if LISTCB.NUMENTRIES = 0
  then
    "list is empty, so initialize"

    LISTCB.HEAD := LISTCB.TAIL := NEWENTRY
    LISTCB.NUMENTRIES := 1
    NEWENTRY.NEXT := NEWENTRY.PREV := 0
  else
    "list not empty"

    PRESENT := LISTCB.HEAD
    LISTCB.NUMENTRIES := LISTCB.NUMENTRIES+1

    "determine position of new entry"

    while NEW.KEY >= PRESENT.KEY and PRESENT.NEXT <> 0 do
      PRESENT := PRESENT.NEXT

      if PRESENT.PREV = 0 and NEW.KEY < PRESENT.KEY
      then
        "new list head"

        LISTCB.HEAD := NEW
        NEW.PREV := 0
        PRESENT.PREV := NEW
        NEW.NEXT := PRESENT
      else
        if NEW.KEY >= PRESENT.KEY
        then
          "new list tail"

          PRESENT.NEXT := LISTCB.TAIL := NEW
          NEW.NEXT := 0
          NEW.PREV := PRESENT
        else
          "insert in middle"

          NEW.NEXT := PRESENT
          NEW.PREV := PRESENT.PREV
          PRESENT.PREV := NEW

          "back up and link predecessor"

          PRESENT := NEW.PREV
          PRESENT.NEXT := NEW
        end-if
      end-if
    end-if
  end-if
end-if

```

Fig 5—Linked-list insertion using a 32-bit key value tests many aspects of a 16-bit processor's architecture. This benchmark exercises the addressing modes of each device.

Memory limitations surface

A major feature of the new 16-bit processors is their ability to address large memories. Unfortunately, many of the benchmark programs were coded in a way that limits them to a 64k-byte range; only the Motorola 68000 programs are truly usable over the machine's full addressing range.

The LSI-11/23 and Z8000 benchmarks assume a 64k data space, because they use only 16-bit addressing. For example, in Benchmark E, the character-string search, neither of these machines can (with the coding shown) deal with the case where the search substring and the data string are not in the same 64k space. In that case, you'd require additional coding to handle the segment information, necessary to extend the programs to the processor's full addressing range.

In the same vein, the 8086 benchmark programs frequently assume that the calling program and the subroutine share data and stack segments—an assumption that also limits the subroutine's addressing range. For example, the character-string-search Benchmark (E) assumes that the string to be searched is in the extra segment (ES). This must be the case to make the compare-string (CMPS) instruction work properly; if the string were not already in the extra segment, you would need code to change the segment addressing.

The 8086 coding of the Quicksort (Benchmark I) uses a clever trick involving the 8086 segment registers to gain efficient indexing of the data records. Unfortun-

nately, this trick only works because the records are exactly 16 bytes long. Because the 8086 addressing system internally multiplies each segment by 16, putting a record number in the segment register automatically points to the appropriate address. Executing Quicksort for records of any other length, though, would require rewriting the Intel program. Modification of this routine for general record lengths would increase code size by an estimated 25% worst case (this also allows records extending over segment boundaries) while affecting performance by no more than an estimated 5%. The performance degradation occurs only for segment-boundary checks, record-length incrementing through the array (rather than segment-register incrementing) and segment-boundary transitions. The code expansion arises from segment-boundary-transition logic that's infrequently—if ever—invoked.

Note that the Zilog benchmarks shown are coded for the Z8002 (unsegmented) version of the Z8000. On the segmented (Z8001) version, these programs would be virtually identical: Except for the I/O-interrupt-kernel benchmarks, all of the programs use exactly the same number of bytes for both devices. (The I/O-interrupt-kernel routines use direct addressing for some variables.) Execution times for the Z8001 benchmarks would tend to be longer than the Z8002 times, though, because of such factors as

- 32-bit Load instructions for address moving

Tests attempt to measure μ Ps, not programmers' skills

- More registers to save and restore
- The longer execution time of the RET instruction
- The longer time required for direct addressing.

What did we measure?

Two statistics are important in computer benchmarks: program size and speed. A program's size is easy to measure—just add up the bytes. Our ground rule in this regard was “If you placed the program in ROM, how much ROM would be used?” We didn't count stack space. (There are no local variables, because the benchmarks are re-entrant.)

Speed values, on the other hand, are very difficult to get a handle on: It seems that the chip makers produce faster μ Ps weekly. The memory you use can also affect the execution speed, thanks to such factors as dynamic-memory refresh. Therefore, because it wasn't possible to obtain a consistent timing mechanism for all of the benchmarks, the timing information provided is merely what the programmers themselves measured

(or in Motorola's case, calculated). We do include data on the processors' clock rates, as well as on how the timings were obtained. And we also performed spot checks on the timing figures provided, using our experience in working with these processors to ensure that the times were reasonable.

Execution times for the 8086-, Z8000- and 68000-based single-board computers assume on-board memory-access operations. By contrast, results for the LSI-11/23 are based on the use of standard off-board dynamic-RAM systems and an asynchronous bus for instruction and data transfers—a configuration dictating the use of processor Wait states, which slowed speeds somewhat. DEC points out, however, that the LSI-11/23's performance figures reflect the actual operation of current board-level product offerings and that the data doesn't necessarily reflect a limitation of the board's processor chip set.

Finally, note that we list the clock speeds of the fastest boards currently available; ie, we have 10-MHz units from Intel and Motorola running in our lab. However, we expect that the manufacturers will build even faster machines in the future. For example, Zilog plans to introduce a 10-MHz version of the Z8000 within the next 3 months. Because faster processors obviously

Benchmark I—Quicksort

The test data for this benchmark consists of 102 (N=100) records, each 16 bytes long. Parameter M is set to nine. The records are initialized as follows:

```
Record 0 --- 00 00 00 00 00 00 00 00 -----
Record 1 --- FF 00 00 00 00 00 00 00 -----
Record 2 --- FE 00 00 00 00 00 00 00 -----
Record 3 --- FD 00 00 00 00 00 00 00 -----
.
.
.
Record 100 --- 9C 00 00 00 00 00 00 00 -----
Record 101 --- FF FF FF FF FF FF FF -----
```

Note that only the key values (bytes 3 to 9 in each record) are significant. All data values are hexadecimal bytes. As in the previous benchmarks, the 68000 times are hand computations; the others are the results of program runs. No data is available for the LSI-11/23 on this benchmark.

Processor	Clock Speed (MHz)	Code Bytes	Execution Time (μ sec)
LSI-11/23	3.33	—	—
8086	10.00	347	115,669
68000	10.00	266	33,527
Z8000	6.00	386	115,500

```
procedure QUICKSORT(N,REC,M,WORK)
integer L,R,I,J,K
integer array STACK[0:2*F(N)-1]
character string V

REC[N+1] := oo
L := 1; R := N
do forever
  I := L; J := R+1; V := REC[L]
  do forever
    do I := I+1 until REC[I] >= V end-do
    do J := J-1 until REC[J] <= V end-do
    if J > I
      then swap REC[I] with REC[J]
      else goto end-first
    end-if
  end-do
end-outer:
do for I from N-1 to 1 in steps of 1
  if REC[I] > REC[I+1] then
    V := REC[I]; J := J+1
    do forever
      REC[J-1] := REC[J]; J := J+1
      if REC[J] >= V then goto end-last end-if
    end-do
  end-last:
  REC[J-1] := V
end-if
end-do

else
  push lower and upper limits of larger
  subfile onto stack
  set L and R to limits of smaller subfile
end-if
end-if
end-do

end-outer:
do for I from N-1 to 1 in steps of 1
  if REC[I] > REC[I+1] then
    V := REC[I]; J := J+1
    do forever
      REC[J-1] := REC[J]; J := J+1
      if REC[J] >= V then goto end-last end-if
    end-do
  end-last:
  REC[J-1] := V
end-if
end-do

end-first:
swap REC[L] with REC[J]
if both subfile sizes (J-L and R-J) <= M
  then
    if stack is empty
      then goto end-outer
    else pop L and R from stack
    end-if
  else
    if smaller subfile size <= M
      then set L and R to lower and upper limits
        of larger subfile
    end-if
  end-if
end-if
end-do
```

Fig 6—The Quicksort requires most work from a processor of any of the benchmarks, and it's typical of a type of application that these devices must frequently serve. The 8086 implementation depends on the 16-byte length of each record and can't be used for general sorting of records of differing length.

Benchmark K—Bit-matrix transposition

The test data for this benchmark consists of 49 bits in a 7×7 array:

```
0100100
1010111
0010001
1101010
0101000
0000101
1100101
```

The array begins on a word boundary. Timing for the 68000 was hand-computed; the other times come from test runs.

Processor	Clock Speed (MHz)	Code Bytes	Execution Time (μsec)
LSI-11/23	3.33	152	1517
8086	10.00	88	820
68000	10.00	74	368
Z8000	6.00	110	646

```
procedure BMT(N,A1,A2)
  integer I,J
  boolean B[1:N,1:N] beginning at bit A2 of word A1
  do for all I and J such that (I <= J <= N) and (J+1 <= I <= N)
    swap B[I,J] and B[J,I]
  end-do
```

Fig 7—Transposition of a bit matrix is another exercise of a processor's bit-manipulation capabilities. The availability of instructions to dynamically test, set and clear individual bits in a word is an advantage in this case.

produce shorter execution times, keep such technological progress in mind if your design project has a long development time or can allow for future upgrading.

“Just the facts, ma'am”

The data in the accompanying boxes summarizes the benchmarking results in terms of code size and execution speed. And the program listings that follow this article illustrate the codings for each processor. As noted, we assume that the manufacturers have done a good job of optimizing the benchmarks; if they don't know how to write code for their own devices, who does?

If there are bugs in the code or ways to improve the coding, EDN would like to know. We have made every effort to check the benchmark programs for correctness and adherence to the specifications. And we thank each of the manufacturers for providing a substantial investment of time and manpower in coding, checking and documentation. We leave any conclusions to you, the reader.

(*Ed Note: Some of the benchmarks were not complete at the time this article was prepared. Specifically, the LSI-11/23 Quicksort (Benchmark I) was incomplete, and one of the 8086 timings remained to be determined. We have left the entries for these values blank.*) **EDN**

BENCHMARK A—LSI-11/23

```
1 .TITLE BENCHMARK A
2 .IDENT /OCT.22/
3 .ENABL LC
4
5 ; I/O INTERRUPT KERNEL, FOUR PRIORITY LEVELS
6 ;
7 ; Services interrupts from four levels, produces a count of interrupts
8 ; by level.
9 ;
10
11 ; The following "ASECT" or absolute program section will load up
12 ; four interrupt vectors.
13 ;
14 000000 .ASECT ; absolute
15
16 000300 . = 300
17 000300 000000 INT1
18 000302 000200 ; execute at priority 4
19
20 000302 . = 302
21 000302 000006 INT2
22 000304 000240 ; execute at priority 5
23
24 000304 . = 304
25 000304 000014 INT3
26 000306 000300 ; execute at priority 6
27
28 000306 . = 306
29 000306 000022 INT4
30 000310 000340 ; execute at priority 7
31
32 000000 .PSECT ; relocatable
33
34 ; Hardware saves context: program counter and processor status.
35 ; Hardware masks out lower level interrupts.
36 ; Hardware vectors to one of the four interrupt service routines.
37 ; Interrupt service routine increments counter.
38 ; RTI instruction restores processor status and program counter,
39 ; lower level interrupts are re-enabled.
40 ;
41 ; Note: If ROMability was a requirement, this program would be four words
42 ; longer!
43 ;
44 000000 INT1:
45 000000 INC (PC)+
46 000002 COUNT1: 0
```

Continued on pg 186

BENCHMARK A—68000

```

1          OPT      BRS,FRS
2
3          *
4          *
5          *           MC68000 EDN BENCHMARK A
6          *
7          *           PRIORITY I/O INTERRUPT KERNEL, FOUR PRIORITY LEVELS
8          *
9          *           NOTES:  1) FOUR AUTOVECTORS ARE ASSUMED INITIALIZED
10         *                TO POINT TO THE FOUR INTERRUPT ENTRY POINTS.
11         *                2) THE MC68000 INTERRUPT SEQUENCE TAKES 4.7
12         *                MICROSECONDS WITH AN ASSUMED INTERRUPT
13         *                ACKNOWLEDGE BUS CYCLE OF 4 CYCLES.
14         *                3) INTERRUPTS ARE TAKEN ANYWHERE WITHIN THE
15         *                MC68000 16 MEGABYTE ADDRESS SPACE.
16         *                4) THE MC68000 PROCESSES INTERRUPTS
17         *                IN PRIORITY ORDER WITHOUT REQUIRING THE
18         *                SUPPORT OF EXTERNAL CIRCUITRY.
19         *
20         *                LINES:  8
21         *                BYTES:  24
22         *
23         *           MC68000L10 BENCHMARK TIME:  33.600 MICROSECONDS
24         *
25         * INTERRUP HANDLERS
26 0 00000000 52780018 INTRPT1 ADD #1,COUNTER1      INCREMENT COUNTER FOR INTERRUPT 1
27 0 00000004 4E73      RTE                      RETURN FROM EXCEPTION
28
29 0 00000006 5278001A INTRPT2 ADD #1,COUNTER2      INCREMENT COUNTER FOR INTERRUPT 2
30 0 0000000A 4E73      RTE                      RETURN FROM EXCEPTION
31
32 0 0000000C 5278001C INTRPT3 ADD #1,COUNTER3      INCREMENT COUNTER FOR INTERRUPT 3
33 0 00000010 4E73      RTE                      RETURN FROM EXCEPTION
34
35 0 00000012 5278001E INTRPT4 ADD #1,COUNTER4      INCREMENT COUNTER FOR INTERRUPT 4
36 0 00000016 4E73      RTE                      RETURN FROM EXCEPTION
37
38         * INTERRUP COUNTERS
39 0 00000018 0000      COUNTER1 DC 0                COUNTER FOR INTERRUPT 1
40 0 0000001A 0000      COUNTER2 DC 0                COUNTER FOR INTERRUPT 2
41 0 0000001C 0000      COUNTER3 DC 0                COUNTER FOR INTERRUPT 3
42 0 0000001E 0000      COUNTER4 DC 0                COUNTER FOR INTERRUPT 4
43
44         *
45         *           END

```

***** TOTAL ERRORS 0-- 0

BENCHMARK A—Z8000

!Example A: I/O Interrupt Kernel, Four Priority Levels!

!Definitions for interrupt kernel programs: !

```

SYSTEM      := $5000          !Addresses for basic system uses!
SYSTACK     := SYSMEM + 256   !One word past highest stack adr!
SP          := R15           !Stack register (RR14 for Seg)!
SPOFF       := R15

```

```

REASON      := R1
QUEPTR      := R2           !RR2 for segmented !
QUENXT      := REASON
ADRLEN      := 2           !4 for segmented !
JUMP        := ADRLEN + 2
ENTOFF      := ADRLEN

```

! The four routines that follow are the processing routines for the four priority levels. VI0 is the highest priority routine, VI3 the lowest. Each of these routines is reached in response to an interrupt on the vectored interrupt (VI) line. Priority resolution is through a hardware protocol defined as part of the Z8000 family architecture. Each of the four devices assumed to be attached to the VI line places its own identifier on the bus when it interrupts, and this identifier is used by the CPU for automatic vectoring to the appropriate routine. The addresses of the routines appear in the program status area (see below). The flag/control word (FCW) value assembled into the PSA has the vectored interrupt enable (VIE) bit set, so that each processing routine is interruptible by other vectored interrupt devices. The hardware interconnection protocol assures that interrupts come only from higher priority devices.

```

0000 6900  VI0:  INC VICNT0
0002 0000'
0004 7B00      IRET
0006 6900  VI1:  INC VICNT1
0008 0002'
000A 7B00      IRET
000C 6900  VI2:  INC VICNT2
000E 0004'

```

Continued on pg 196

BENCHMARK A—Z8000

```
0010 7B00      IRET
0012 6900  VI3:  INC VICNT3
0014 0006'
0016 7B00      IRET
```

```
!Counters for simulated four priority level processing !
0000 0000  VICNT0: 0
0002 0000  VICNT1: 0
0004 0000  VICNT2: 0
0006 0000  VICNT3: 0
```

BENCHMARK B—LSI-11/23

```
1          .TITLE  BENCHMARK B
2          .IDENT  /OCT.22/
3          .ENABL  LC
4
5          ; I/O INTERRUPT KERNEL, FIFO PROCESSING
6          ;
7          ; Services interrupts from four levels, using a FIFO queue.
8          ; Each of the four devices has an interrupt vector set up as follows
9          ;
10         ;
11         ; . = vector address
12         ; .WORD  ROUTINE, 340
13         ; The vectored interrupt capability of the LSI-11 hardware is used
14         ; here to implicitly identify the device causing the interrupt.
15         ; Each interrupt will vector to a different service routine.
16         ; Hardware will save context (program counter = PC, and processor
17         ; status = PS) at the interrupt.
18
19 000000    .ASECT
20
21          ; Set up a vector for each device.
22          ;
23          . = 300
24
25 000300    000000'    DEV1          ; new PC
26 000302    000340    340          ; new PS
27
28 000304    000006'    DEV2
29 000306    000340    340
30
31 000310    000014'    DEV3
32 000312    000340    340
33
34 000314    000022'    DEV4
35 000316    000340    340
36
37          ; Each device operates at priority seven (340 octal in the PS) to disable
38          ; other interrupts.
39          ;
40          ; The queue contains a power of two number of words. It must begin on
41          ; an even multiple of the queue size, such that for all addresses in
42          ; the queue, (address AND queue size) is zero, and ((queue start +
43          ; queue size) AND queue size) is nonzero. QEND points to where the
44          ; next new entry will be made in the queue. QSTART points to where the
45          ; next entry will be removed from the queue. When they point to the
46          ; same place, the queue is empty. We assume the queue never overflows.
47          ;
48 000000    .PSECT  DATA
49
50          000040    QSIZE =      40          ; sixteen elements
51          ; QUEUE will be relocated to
52          ; address 1000(8) at LINK time
53 000000    QUEUE:   .BLKB  QSIZE
54 000040    000000'  QSTART:  QUEUE
55 000042    000000'  QEND:   QUEUE
56 000044    000000  RUNFLG:  0
57
58 000000    .PSECT  CODE
59
60          ; Each of the four devices has an interrupt routine as follows:
61          ;
62          ;ROUTINE:
63          ; any immediate processing
64          ; CALL  COMMON
65          ;CTR:  .WORD  0
66          ;
67          ; CTR is the counter that will be incremented by the FIFO processor.
68          ; In a real example, it would be the first instruction of the
69          ; interrupt service routine.
70          ;
71 000000    DEV1:
72 000000    004767  000024    CALL  COMMON
73 000004    000000    CTR1:  .WORD  0
74
75 000006    DEV2:
76 000006    004767  000016    CALL  COMMON
77 000012    000000    CTR2:  .WORD  0
78
79 000014    DEV3:
```

Continued on pg 200

BENCHMARK B—68000

```

70 0 00000036 3050          MOVE      QUELINK(A0),A0      LOAD NEXT IN LIST
71 0 00000038 B0F8004A      CMP       QUEIN,A0           TEST END OF LIST
72 0 0000003C 66EA         BNE      SELECT             PROCESS IT IF NOT
73 0 0000003E 42380074     CLR.B    FLAG              SHOW NO ELEMENTS QUEUED
74
75                          * RESTORE CONTEXT
76 0 00000042 4CDF0300     RETURN   MOVEM.L (SP)+,A0/A1  RELOAD USERS REGISTERS
77 0 00000046 588F         ADD.L    #4,SP             CLEAN COUNTER ADDRESS OFF STACK
78 0 00000048 4E73         RTE                               RETURN FROM EXCEPTION
79
80
81                          * QUEUE POINTER
82 0 0000004A 004C      QUEIN    DC              QUEUE          NEXT ENTRY TO USE
83
84                          * QUEUE PROPER
85 0 0000004C 00500000     QUEUE   DC              **+,0      QUEUE ENTRY ONE
86 0 00000050 00540000     DC      **+,0            QUEUE ENTRY TWO
87 0 00000054 00580000     DC      **+,0            *
88 0 00000058 005C0000     DC      **+,0            *
89 0 0000006C 00600000     DC      **+,0            *
90 0 00000066 00640000     DC      **+,0            *
91 0 00000064 00680000     DC      **+,0            *
92 0 00000068 006C0000     DC      **+,0            *
93 0 0000006C 00700000     DC      **+,0            *
94 0 00000070 004C0000     DC      QUEUE,0         QUEUE ENTRY TEN
95
96                          * INTERRUPT IN PROGRESS FLAG
97 0 00000074 00         FLAG    DC.B    0        INTERRUPT IN PROGRESS INDICATOR
98
99                          END
***** TOTAL ERRORS 0-- 0

```

BENCHMARK B—Z8000

```

!Example B: I/O Interrupt Kernel, FIFO Processing!
0000 2DF1  FIFO:  EX REASON,ESP      !Save the context!
0002 93F2          PUSH @SP,QUEPTR      !Save registers !
0004 6102          LD  QUEPTR,QUEIN    !Queue the request !
0006 000E!
0008 3321          LD  QUEPTR(@ENTOFF),REASON
000A 0002
000C 2121          LD  QUENXT,@QUEPTR
000E 6F01          LD  QUEIN,QUENXT    !Set pointer to next slot !
0010 000E!
0012 4C06          TSETB FLAG          !Can it be processed now? !
0014 003C!
0016 E50C          JR  MI,RESTOR      ! No !
0018 7C06  LOOP:  EI  NVI          !Yes, let more happen !
001A 3121          LD  REASON,QUEPTR(@ENTOFF) !Simulate processing by !
001C 0002
001E 6810          INCB COUNTS(REASON) ! bumping count !
0020 0008!
0022 7C02          DI  NVI          !Disable before dequeing !
0024 2122          LD  QUEPTR,@QUEPTR !Check on next !
0026 4B02          CP  QUEPTR,QUEIN   !Anything else in queue ? !
0028 000E!
002A EEF6          JR  NE,LOOP        ! Yes, do it !
002C 4C08          CLRB FLAG          ! No, clear flag !
002E 003C!
0030 97F2  RESTOR: POP QUEPTR,@SP      !Restore registers !
0032 2DF1          EX  REASON,ESP
0034 7B00          IRET              !And return !
0008          !Counters for FIFO interrupt processing !
          COUNTS array [5 byte]

          !Queue for FIFO interrupt processing !
000E 0010!
          QUEIN:  QUEUE

0010 0014!  QUEUE:  $+JUMP, 0,
0012 0000
0014 0018!  $+JUMP, 0,
0016 0000
0018 001C!  $+JUMP, 0,
001A 0000
001C 0020!  $+JUMP, 0,
001E 0000
0020 0024!  $+JUMP, 0,
0022 0000
0024 0028!  $+JUMP, 0,
0026 0000
0028 002C!  $+JUMP, 0,
002A 0000
002C 0030!  $+JUMP, 0,
002E 0000
0030 0034!  $+JUMP, 0,
0032 0000
0034 0038!  $+JUMP, 0,
0036 0000
0038 0010!  QUEUE, 0,
003A 0000
003C 00          FLAG:  0

```

BENCHMARK E—68000

```

1          OPT      BRS
2
3          *
4          *          MC68000 EDN BENCHMARK E
5          *
6          *          SUBSTRING CHARACTER SEARCH
7          *
8          * ATTRIBUTES: * 16 MEGABYTE ADDRESSING RANGE
9          *                * POSITION INDEPENDENT
10         *                * REENTRANT
11         *
12         * INPUT ARGUMENTS:
13         * D0 - SEARCH PATTERN LENGTH  A0 - SEARCH PATTERN STRING ADDRESS
14         * D1 - SEARCHED STRING LENGTH  A1 - SEARCHED STRING ADDRESS
15         *
16         * OUTPUT:
17         * D2 - RETURNED MATCHED OFFSET VALUE (-1 IF NO MATCH)
18         *
19         *          ALL OTHER REGISTERS ARE TRANSPARENT OVER THIS ROUTINE
20         *
21         *          LINES: 18
22         *          BYTES: 44
23         *
24         *          MC68000L10 BENCHMARK TIME: 244.000 MICROSECONDS
25         *
26         * REGISTER USAGE:
27         * D0 - COMPARE LOOP COUNTER  A0 - SEARCHED STRING ADDRESS
28         * D1 - POSITION LOOP COUNTER  A1 - PATTERN STRING ADDRESS
29         * D2 - TOTAL LENGTH SAVE    A2 - COMPARE TEMPORARY POINTER
30         * D3 - INITIAL CHARACTER    A3 - COMPARE TEMPORARY POINTER
31         * D4 - COMPARE LOOP TEMP
32
33         * INDEX FUNCTION SUBROUTINE
34 0 00000000 48E71830 INDEX MOVEM.L D3/D4/A2/A3,-(SP) SAVE WORK REGISTERS
35 0 00000004 9240 SUB D0,D1 FIND SEARCH LOOP COUNT (-1)
36 0 00000006 3401 MOVE D1,D2 ALSO USE FOR FINAL OFFSET COMPUTATION
37 0 00000008 5540 SUB #2,D0 ADJUST COUNT FOR DBRA LOOP
38 0 0000000A 1618 MOVE.B (A0)+,D3 D2=FIRST CHAR TO BE FOUND
39
40 0 0000000C 8619 * SEARCH FOR FIRST CHARACTER MATCH
41 0 0000000E 57C9FFFC FIND1 CMP.B (A1)+,D3 SEARCH STRING UNTIL FIRST CHAR
42 0 00000012 6612 COUNT1 DBEQ D1,FIND1 D1 GETS DECREMENTED FOR LOC CALCULATION
43 *          BNE NOTFOUND BRANCH IF NO FIRST CHAR
44 *          * PERFORM FULL PATTERN COMPARE
45 0 00000014 2448 MOVE.L A0,A2 A2=POINTER TO SUBSTRING
46 0 00000016 2649 MOVE.L A1,A3 A3=POINTER TO STRING
47 0 00000018 3800 MOVE D0,D4 D3=TEMP FOR SUBSTRING COUNTER
48 0 0000001A 6B08 BMI ONECHAR BRANCH IF ONLY ONE CHARACTER
49 0 0000001C 870A COMPARE CMP.B (A2)+,(A3)+ COMPARE REST OF SUBSTRING
50 0 0000001E 56CCFFFC DBNE D4,COMPARE LOOP IF STILL EQUAL AND MORE
51 0 00000022 66EA BNE COUNT1 BRANCH BACK IF REST NOT THE SAME
52 0 00000024 9441 ONECHAR SUB D1,D2 CALCULATE OFFSET OF MATCH
53 0 00000026 4CDF0C18 NOTFOUND MOVEM.L (SP)+,D3/D4/A2/A3 RESTORE WORK REGISTERS
54 0 0000002A 4E75 RTS RETURN TO CALLER
55
56         *          END

```

BENCHMARK E—Z8000

!Example E: Character Search!

```

!Arguments:
SRCHLNTH      := R0  !Length (in bytes) of SRCHSTR!
ARGLNTH       := R1  !Length (in bytes) of SRCHARG !
SRCHSTR       := R2  !Address of the string to be searched!
SRCHOFF       := R2  !offset portion of address!
SRCHARG       := R4  !Address of string sought!
ARGOFF        := R4  !offset portion of address!
LOC           := R6  !Return arg: char position (>=0) or !
FAILCODE      := -1  ! negative (FAILCODE) if no match!

```

```

!Workspace for the routine:
G             := RH6  !First char of sought string !
LCT          := R7   !Substring counter !
SCH          := R8   !Address register used with ARCHARG!
ARG          := R10  !Address register used with SRCHARG!
OFFSAVE      := R12  !Remembers original SRCHOFF value!
CT           := R13  !Count (bytes in srch string)!
WK1          := R7
NW1          := 7

```

```

0000 ABFD SEARCH: DEC SP,#2*NW1
0002 1CF9 LDM @SP,WK1,#NW1 !Save registers used !
0004 0706
0006 A107 LD LCT,SRCHLNTH !Compute number of substrings!
0008 8317 SUB LCT,ARGLNTH ! long enough to match!

```

BENCHMARK E—Z8000

```

000A A970          INC LCT
000C A12C          LD OFFSAVE,SRCHOFF      !Save initial SRCHOFF value!
000E 2046          LDB G,@SRCHARG          !First char to look for!
0010 A940          INC ARGOFF              !Set to compare remainder!
0012 AB10          DEC ARGLNGTH          !Chars in remainder!

!Check possible substrings from left to right!
CLOOP: CPIRB G,@SRCHSTR,LCT,EQ !Match first char!

        JR NZ,FAIL          !No match!
0014 BA24          TEST ARGLNGTH          !Any more chars in string?!
0016 0766          JR Z,MATCH          ! no - already finished!
0018 EEOA          LD CT,ARGLNGTH          !Set length and !
001A 8D14          LD SCH,SRCHSTR          ! string address!
001C E60B          LD ARG,SRCHARG          ! for block comparison!
001E A11D          CPSIRB @SCH,@ARG,CT,NE !Look for a mismatch !
0020 A128
0022 A14A
0024 BAA6
0026 OD8E
0028 EEO5          JR NZ,MATCH          !Strings match , byte for byte!
002A 8D74          TEST LCT              !No match - try next substring!
002C EEF3          JR NZ,CLOOP          ! (if any) !
002E 2106          FAIL: LD LOC,#FAILCODE !No more substrings-fail!
0030 FFFF
0032 E803          JR EXIT1
0034 A126          LD LOC,SRCHOFF          !Match - return index of !
0036 83C6          SUB LOC,OFFSAVE          ! substring matching initial !
0038 AB60          DEC LOC              ! search string!
003A 1CF1          EXIT1: LDM WK1,@SP,#NW1
003C 0706
003E A9FD          INC SP,#2*NW1          !Restore registers !
0040 9E08          RET

```

BENCHMARK F—LSI-11/23

```

1
2                .TITLE BENCHMARK F
3                .IDENT /OCT.22/
4                .ENABL LC
5
6                ; BIT TEST, SET, OR RESET
7                ;
8                ; Find a bit, check it, change it, bash it, smash it
9                ;
10               ; Assumes that bits are numbered from 0 from the right-hand end of a word.
11               ; This is the way a PDP-11 views words. Luckily left-to-right ordering
12               ; Was not a benchmark requirement! Arguments are passed on the stack.
13               ; Naturally, performance improvements could be made by passing the arguments
14               ; in registers.
15               ; Stack offsets assume 4 bytes for saved registers:
16               ;
17               F      =      6      ; function code
18               N      =      10     ; relative bit number
19               A1     =      12     ; address of bit string
20               RC      =      14     ; address of return code word
21               WORK   =      16     ; not used
22 000000
23 000000          ONTRACE::
24 000000          BTSR::
25 000002          MOV     R0, -(SP)      ; save registers
26 000004          MOV     R1, -(SP)
27 000010          CLR     @RC(SP)        ; assume bit is zero
28 000014          MOV     N(SP), R0     ; R0 = bit offset
29 000020          BIC     #'C<7>, R0   ; R0 = bit within byte
30 000024          MOV     #1, R1        ; K1 = 1
31 000026          ASH     R0, R1        ; shift the 1 in R1 left R0 times
32 000032          MOV     N(SP), R0     ; R0 = bit offset
33 000036          ASH     #-3, R0      ; R0 = byte offset into bit string
34 000042          BITB   K1, (R0)     ; R0 -> the byte address
35 000044          BEQ    10$           ; check out the bit
36 000046          INC     @RC(SP)      ; branch if zero
37 000052          CMP     F(SP), #2    ; else return code becomes one
38 000060          BNE    20$           ; if function code is two,
39 000062          BISR   K1, (R0)     ; if function code is two,
40 000064          BR     30$           ; set the bit
41
42 000066          CMP     F(SP), #3    ; if function code is three,
43 000074          HNE    30$           ;
44 000076          BICB  K1, (R0)     ; clear the bit
45 000100          30$: MOV     (SP)+, R1 ; restore registers
46 000102          MOV     (SP)+, R0
47 000104          OFFTRACB::
48 000104          RETURN
49
50
51 000001          .END

```

BENCHMARK F—68000

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29 0 00000000 2401
30 0 00000002 E68A
31 0 00000004 5540
32 0 00000006 670C
33 0 00000008 5340
34 0 0000000A 6710
35
36 0 0000000C 03302800
37 0 00000010 56C2
38 0 00000012 4E75
39
40 0 00000014 03F02800
41 0 00000018 56C2
42 0 0000001A 4E75
43
44 0 0000001C 03B02800
45 0 00000020 56C2
46 0 00000022 4E75
47
48
***** TOTAL ERRORS 0-- 0

```

OPT BRS

*
*
* MC68000 EDN BENCHMARK F
*
* BIT ARRAY TEST, SET, AND RESET
*
* ATTRIBUTES: * 16 MEGABYTE ADDRESSING RANGE
* * POSITION INDEPENDENT
* * REENTRANT
*
* INPUT:
* D0 - FUNCTION CODE A0 - BIT ARRAY BASE ADDRESS
* D1 - BIT SUBSCRIPT INDEX
*
* OUTPUT:
* D2 - RETURN CODE
*
* ALL OTHER REGISTERS ARE TRANSPARENT OVER THIS ROUTINE
*
* LINES: 15
* BYTES: 36
*
* MC68000L10 BENCHMARK TIME: 70.200 MICROSECONDS
*
* BIT TEST, SET, RESET SUBROUTINE
* BIT MOVE.L D1,D2 COPY BIT INDEX OVER
* LSR.L #3,D2 DIVIDE BY 8 FOR BYTE ADDRESS
* SUB #2,D0 OFFSET FUNCTION CODE DOWN 2
* BEQ SET BRANCH IF 2 TO SET
* SUB #1,D0 DOWN ANOTHER
* BEQ RESET BRANCH IF 3 TO RESET
* * TEST
* BTST D1,(A0,D2.L) CODE NOT 2 OR 3 - TEST BIT
* SNE D2 SET RETURN CODE SAME AS BIT
* RTS RETURN TO CALLER
* * SET
* BSET D1,(A0,D2.L) CODE 2 - SET BIT
* SNE D2 SET RETURN CODE FOR PREVIOUS SETTING
* RTS RETURN TO CALLER
* * RESET
* BCLR D1,(A0,D2.L) CODE 3 - CLEAR BIT
* SNE D2 SET RC ONE OR ZERO, SAME AS PREVIOUS
* RTS RETURN TO CALLER
* END

BENCHMARK F—Z8000

```

|Example F: Bit array manipulation routines|
|Arguments: |
F            := R0 |Function code - possible vaues are:|
FL          := R10
FH          := R80
TSTCODE    := 1    |Test the bit|
SETCODE    := 2    |Set the bit to 1|
RESCODE    := 3    |Reset the bit to 0|
N          := R1    |Index (from zero) of desired bit|
AL         := R2    |Address of bit array (RR2 for seg) |
ALOFF      := R2    |R3 for segmented |
RC         := R3    |Return arg: set to value of desired bit|

|Workspace for the routine:|
BYTNUM     := RC    |Byte offset of desired bit in A1 array|
CURBYTE    := FH    |Temp home of byte containing the bit|

0000 A113    BITMAN: LD BYTNUM,N            |Compute byte offset of |
0002 B339            SRA BYTNUM,#3        | desired byte (divide by 8) |
0004 FFFD
0006 8132            ADD ALOFF,BYTNUM            |Point at desired byte |
0008 2020            LDB CURBYTE,@AL            |Get the byte|
000A 2601            BITB CURBYTE,N        |Test the bit|
000C 0000
000E E607            JR Z,NOTSET
0010 BD31            LDK RC,#1                    |Set-indicate in RC|
0012 AA82            DECB FL,#RESCODE            |Should it be cleared?|
0014 9E0E            RET NE                      | No, exit |
0016 2201            RESB CURBYTE,N              | Yes, do it |
0018 0000
001A 2E20            LDB @AL,CURBYTE
001C 9E08            RET
001E BD30            NOTSET: LDK RC,#0             |Clear-indicate in RC|
0020 AA81            DECB FL,#SETCODE            |Should it be set?|
0022 9E0E            RET NE                      | No, exit |
0024 2401            SETB CURBYTE,N              | Yes, do it |
0026 0000
0028 2E20            LDB @AL,CURBYTE             |Save changed byte |
002A 9E08            RET

```

Continued on pg 229

BENCHMARK H—68000

```

80 0 0000004E 234A0004      MOVE.L A2,NEXT(A1)      SET NEW.NEXT TO CURRENT
81 0 00000052 236A00000008  MOVE.L PREV(A2),PREV(A1) SET CURRENT EARLIER TO NEW.PREV
82 0 00000058 25490008      MOVE.L A1,PREV(A2)     SET CURRENT.PREV TO NEW
83 0 0000005C 24690008      MOVE.L PREV(A1),A2     LOAD CURRENT EARLIER ADDRESS
84 0 00000060 25490004      FINISHL MOVE.L A1,NEXT(A2) SET EARLIER.NEXT TO NEW
85
86                               * RESTORE REGISTERS AND RETURN
87 0 00000064 4CDF0403      FINISH  MOVEM.L (SP)+,D0/D1/A2  RESTORE REGISTERS
88 0 00000068 4E75          RTS                    RETURN TO CALLER
89
90                               END

```

***** TOTAL ERRORS 0-- 0

BENCHMARK H—Z8000

!Example H: Insertion in a Doubly Linked list!

```

ADLEN  := 2                !Number of bytes in an address
                                (4 for segmented operation) !
!Arguments: !
LISTCB := R12              !Address of list control block
                                (RR12 for segmented) !
!Format of list control
block:
HEADF  := 0                !Adr of the list "head" (first entry)!
TAILF  := ADLEN            !Adr of "tail" (last entry)!
NUMF   := TAILF+ADLEN      !Number of entries in list!
NEWENTRY:= R10             !Address of entry to be inserted
                                (RR10 for segmented) !
NEWOFF := R10              !Offset portion of address!

!Format of an entry
KEYF   := 0                !Key portion of entry!
LKEY   := 4                !Number of bytes in a key!
NEXTF  := LKEY             !Pointer to "next" entry !
PREVF  := NEXTF+ADLEN     !Pointer to "previous" entry!

```

!Working storage for routine:!

```

KEY    := RRO
PTRS   := R2                !First of registers for NEXT and PREV!
NEXTAD := R2                ! (RR2 for segmented)!
PREVAD := R3                ! (RR4 for segmented)!
NPTRS  := ADLEN            !Registers in the block (2*ADLEN/2)!
NUM    := R4                !"Number of entries" from LISTCB !
                                ! (R6 for segmented) !

WK3    := RO
NW3    := NPTRS+3

0000 ABF9      LISTIN: DEC SP,#2*NW3
0002 1CF9      LDM @SP,WK3,#NW3                !Save registers used !
0004 0004
0006 14A0      LD  LDM @SP,#NW3                !Get the new entry's key
0008 31C4      LD  NUM,LISTCB(#NUMF)           !Count the new entry!
000A 0004
000C A940      INC  NUM
000E 33C4      LD  LISTCB(#NUMF),NUM
0010 0004
0012 BD30      LDK  PREVAD,#0                  !Zap PREVAD pointer!
0014 0B04      CP   NUM,#1                    !First entry?!
0016 0001
0018 EE05      JR  NE,NOTFRST                 ! no - go scan list!
001A BD20      LDK  NEXTAD,#0                  ! yes - zap "next" ptr!
001C 2FCA      LD  @LISTCB,NEWENTRY           !Set LISTCB "head" ptr!
001E 33CA      LD  LISTCB(#TAILF),NEWENTRY    !Set LISTCB "tail" ptr!
0020 0002
0022 E817      JR  UPNEW                       !Update new entry's ptrs
0024 21C2      NOTFRST:LD NEXTAD,@LISTCB      !Init "next" for scan!
0026 1020      SCANLP: CPL KEY,#NEXTAD        !Compare keys!
0028 E906      JR  GE,TRYNEXT                 ! not the place!
002A 8D34      TEST  PREVAD                   !Insert here. Head?!
002C EE0E      JR  NZ,UPMID                   ! no - update and exit
002E 2FCA      LD  @LISTCB,NEWENTRY           ! yes - adjust LISTCB!
0030 332A      LD  NEXTAD(#PREVF),NEWENTRY    !Update prev's "next"!
0032 0006
0034 E80E      JR  UPNEW                       !Update new entry's ptrs
0036 A123      TRYNEXT:LD PREVAD,NEXTAD       !Next in list!
0038 3132      LD  NEXTAD,PREVAD(#NEXTF)
003A 0004
003C 8D24      TEST  NEXTAD                   !New tail?!
003E EEF3      JR  NZ,SCANLP                  ! no - keep looking!
0040 333A      LD  PREVAD(#NEXTF),NEWENTRY    ! yes - set prev's "nxt"
0042 0004
0044 33CA      LD  LISTCB(#TAILF),NEWENTRY    !Set LISTCB "tail" ptr!
0046 0002
0048 E804      JR  UPNEW                       !Update new entry's ptrs
004A 332A      UPMID: LD NEXTAD(#PREVF),NEWENTRY !Update next's "prev"!
004C 0006
004E 333A      LD  PREVAD(#NEXTF),NEWENTRY    !Update prev's "next"!
0050 0004
0052 A9A3      UPNEW: INC  NEWENTRY,#LKEY      !Write entry pointer!
0054 1CA9      LDM  @NEWENTRY,PTRS,#NPTRS
0056 0201
0058 1CF1      LDM  WK3,@SP,#NW3
005A 0004
005C A9F9      INC  SP,#2*NW3                !Restore registers !
005E 9E08      RET

```

BENCHMARK I—68000

```

96 0 0000008A 48D3000F      MOVEM.L  D0-D3,(A3)      .          AND
97 0 0000008E 48D0000F      MOVEM.L  D4-D7,(A0)      .          REC(J)
98 0 00000092 2209          MOVE.L   A1,D1           D1 <- R
99 0 00000094 240B          MOVE.L   A3,D2           D2 <- J
100 0 00000096 9202          SUB.L    D2,D1           D1 <- R-J
101 0 00000098 9408          SUB.L    A0,D2           D2 <- J-L
102 0 0000009A 840E          CMP.L    A6,D2           COMPARE (J-L) <= MSIZE
103 0 0000009C 62C6          BHI     NEWLR0          BRANCH IF NO
104 0 0000009E 8208          CMP.L    A6,D1           COMPARE (R-J) <= MSIZE
105 0 000000A0 62D8          BHI     NEWLR1          BRANCH IF NO
106 0 000000A2 4CD0300      MOVEM.L  (SP)+,A0/A1     POP NEXT L AND R FROM STACK
107 0 000000A6 2008          MOVE.L   A0,D0          TEST IF STACK IS EMPTY
108 0 000000A8 6600FF6C      BNE     SORT           CONTINUE SORT IF NOT EMPTY
109
110
111 * FALL INTO INSERTION SORT AS ALL SUBFILES BELOW OR EQUAL M RECORDS
112 *          INSERTION SORT PHASE
113 *
114 * REGISTER USE: D0 - LOOP CONTROL          A0 - REC(I)
115 *          D1 - COUNTER AND SWAP REGISTER  A1 - REC(J)
116 *          D2/D4 - SWAP REGISTERS          A2/A3 - WORK REGISTERS
117 *          D5/D7 - "V" SAVE REGISTERS     A4 - REC(J-1)
118 *                                          A5 - "V" SAVE REGISTER
119 *                                          A6 - FRAME POINTER
120 *
121 * NOTE: STACK SPACE IS RESERVED FOR "V" KEY COMPARE RECORD COPIES
122 *
123
124 0 000000AC 4CDF0101      MOVEM.L  (SP)+,D0/A0     RELOAD RECORD COUNT AND TOP RECORD
125 0 000000B0 4E56FFFF      LINK     A6,#-ENTRYLEN  ALLOCATE "V" KEY COPY AREA ON STACK
126 0 000000B4 5540          SUB      #2,D0          D0 RANGES FROM N-2 THROUGH 0
127
128 0 000000B6 41E8FFFF      LOOPOUT LEA     -ENTRYLEN(A0),A0  I <- I-1
129 0 000000BA 45E80003      LEA     KEY(A0),A2      A2 -> KEY(I)
130 0 000000BE 47E80013      LEA     ENTRYLEN+KEY(A0),A3  A3 -> KEY(I+1)
131 0 000000C2 7206          MOVE.L  #KEYLEN-1,D1     LOOP COUNTER FOR COMPARE
132 0 000000C4 850B          CMP.B   (A3)+,(A2)+     COMPARE KEY(I)-KEY(I+1)
133 0 000000C6 56C9FFFF      DBNE   D1,CMP11        LOOP WHILE EQUAL
134 0 000000CA 6332          BLS     ENDIF          BRANCH IF KEY(I) <= KEY(I+1)
135
136 0 000000CC 4CD020E0      MOVEM.L  (A0),D5-D7/A5   V <- REC(I)
137 0 000000D0 48D700E0      MOVEM.L  D5-D7,(SP)     AND ON STACK FOR KEY COMPARE
138 0 000000D4 43E80010      LEA     ENTRYLEN(A0),A1  A1 -> REC(J) = REC(I+1)
139 0 000000D8 2848          MOVE.L  A0,A4          PRIME A4 -> REC(J-1)
140
141 0 000000DA 4CD1001E      LOOPIIN MOVEM.L  (A1),D1-D4     TEMP <- REC(J)
142 0 000000DE 48D4001E      MOVEM.L  D1-D4,(A4)     REC(J-1) <- TEMP
143 0 000000E2 2849          MOVE.L  A1,A4          A4 -> NEXT REC(J-1)
144 0 000000E4 43E90010      LEA     ENTRYLEN(A1),A1  J = J+1
145 0 000000E8 45EF0003      LEA     KEY(SP),A2      A2 -> KEY(V)
146 0 000000EC 47E90003      LEA     KEY(A1),A3      A3 -> KEY(J)
147 0 000000F0 7206          MOVE.L  #KEYLEN-1,D1     LOOP COUNTER IN D1
148 0 000000F2 850B          CMP.B   (A3)+,(A2)+     COMPARE KEY(V)-KEY(J)
149 0 000000F4 56C9FFFF      DBNE   D1,CMPVJ        LOOP WHILE EQUAL
150 0 000000F8 62E0          BHI     LOOPIIN        IF KEY(V) > KEY(J) CONTINUE LOOP
151
152 0 000000FA 48D420E0      MOVEM.L  D5-D7/A5,(A4)  REC(J-1) <- V
153
154 0 000000FE 51C8FFB6      ENDIF   DBRA    D0,LOOPOUT  CONTINUE LINEAR INSERT
155
156 0 00000102 4E5E          UNLK    A6             FREE AND RESTORE STACK
157 0 00000104 4CDF7FFF      MOVEM.L  (SP)+,D0-D7/A0-A6  RESTORE REGISTERS
158 0 00000108 4E75          RTS     RTS            RETURN TO CALER
159
160
161                          END

```

BENCHMARK I—Z8000

!Example I: Quicksort/Insertion Sort!

```

!Arguments !
N      := R0   !Number of records !
M      := R1   !Changeover point !
REC    := RR2  !Array base !
RECOFF := R3
RECSEG := RH2

!Working registers !
SCR1   := R0   !Scratch borrowed from argument registers !
SCR2   := R1
BIGM   := SCR1
ADR    := RR4; ADRHH := RH4; ADRHL := RL4; ADRL := R5
I      := RR6; IHI := R6; ILO := R7
J      := RR8; JHI := R8; JLO := R9
L      := RR10; LHI := R10; LLO := R11
U      := RR12; UHI := R12; ULO := R13

ITAD   := ADRL !Address of I-item !

```

BENCHMARK I—Z8000

```

JTAD   := LLO   !Address of J-item !
PIVOT  := SCRL
PIVHI  := SCR1
PIVLO  := SCR2

```

```

!Temporary registers for item moving !
DEST   := LLO
SRCE   := ADRL

```

```

!Other constants !

```

```

ESIZE  := 16   !Bytes per record!
KEYOFF := 3    !Index in record of first byte of key!
KEYBYTES:= 7   !Bytes per key!

0000 ABFF      SORT:  DEC SP,#16
0002 ABFB      DEC SP,#12
0004 1CF9      LDM @SP,RO,#14           !Save all registers !
0006 000D
0008 A10D      LD ULO,N                !Number of records !
000A B1CA      EXTS U
000C 190C      MULT U,#ESIZE           !Mult by size of records !
000E 0010
0010 140A      LDL L,#0                !Zero lower limit to start !
0012 0000
0014 0000
0016 91FC      PUSHL @SP,U
0018 1900      MULT BIGM,#ESIZE        !Adjust cutoff for record size
001A 0010
001C 91F0      PUSHL @SP,BIGM
001E DFB7      CALR QUICK
0020 95F0      POPL BIGM,@SP
0022 95F6      POPL I,@SP
0024 1206      SUBL I,#ESIZE
0026 0000
0028 0010
002A 9464      INSORT: LDL ADR,I
002C 1604      ADDL ADR,#ESIZE
002E 0000
0030 0010

0032 8135      ICALR ADCOMP!
0034 1604      ADD ADRL,RECOFF
0036 0000      ADDL ADR,#KEYOFF

0038 0003
003A 9440      LDL PIVOT,ADR           !PIVOT is adr of key of A(I+1) !
003C DF7D      CALR CPPI
003E EF2F      JR UGE,END1            !If A(I+1)>=A(I), end block !
0040 ABFF      DEC SP,#ESIZE
0042 8D08      CLR PIVHI
0044 A1F1      LD PIVLO,SP           !PIVLO has adr of V on stack !
0046 1600      ADDL PIVOT,#KEYOFF      !PIVOT points to key in V !
0048 0000
004A 0003
004C A1FB
004E 9464

0050 8135      LD DEST,SP
0052 BDA8      LDL ADR,I
0054 BB51      ICALR ADCOMP!           !ADRL is source address !
0056 0AB0      ADD ADRL,RECOFF
0058 9468      LDK R10,#(ESIZE/2)
005A 1608      LDIR @DEST,@SRCE,R10    !Save a(I) on stack !
005C 0000
005E 0010
0060 9484      LDL J,I
0062 1204      ADDL J,#ESIZE           !J = I + 1 !
0064 0000
0066 0010      AGN2:  LDL ADR,J
0068 8135      SUBL ADR,#ESIZE         !ADR = J - 1 !
006A 944A
006C 9484      ICALR ADCOMP!
006E 8135      ADD ADRL,RECOFF
0070 BDA8      LDL L,ADR
0072 BB51      LDL ADR,J
0074 0AB0      ICALR ADCOMP!
0076 1608      ADD ADRL,RECOFF
0078 0000      LDK R10,#(ESIZE/2)
007A 0010      LDIR @DEST,@SRCE,R10    !A(J-1) = A(J) !
007C DF9B      ADDL J,#ESIZE           !J = J + 1 !
007E E401
0080 EFEF      CALR CPPJ
0082 9484      JR OV,ENDLAST
0084 1204      JR UGE,AGN2
0086 0000      ENDLAST:LDL ADR,J
0088 0010      SUBL ADR,#ESIZE         !ADR = J - 1 !

008A 8135      ICALR ADCOMP!
008C 944A      ADD ADRL,RECOFF
008E 9404      LDL L,ADR
0090 1204      LDL ADR,PIVOT
0092 0000      SUBL ADR,#KEYOFF        !ADR = address of V again !
0094 0003
0096 BDA8      LDK R10,#(ESIZE/2)
0098 BB51      LDIR @DEST,@SRCE,R10    !A(J-1) = V !

```

BENCHMARK I—Z8000

```

009A 0AB0
009C A9FF
009E 1206
00A0 0000
00A2 0010
00A4 9C68
00A6 EEC1
00A8 1CF1
00AA 000D
00AC A9FF
00AE A9FB
00B0 9E08
                INC SP,#ESIZE
END1:  SUBL I,#ESIZE

                TESTL I
                JR NZ,INSORT
                LDM RO,@SP,#14

                INC SP,#16
                INC SP,#12
                RET
                IRestore registers I

!Subroutine Quicksort - after C. A. R. Hoare
CALL QUICK with  BASE = array address
                  U = offset of upper limit
                  L = offset of lower limit
                Semi-sorts elements at offsets between L and U (inclusive).
                The 23-bit integers L and U are in the range 0 to 8,388,607.
!
00B2 94C4
00B4 92A4
00B6 9004
00B8 9E02
00BA 91F0
QUICK:  LDL ADR,U
                SUBL ADR,L      !compute subfile size !
                CPL ADR,BIGM
                RET LE          !Return if subfile is <= M long !
                PUSHL @SP,BIGM
                ! Partition array segment between offsets L and U (inclusive)
                ! around a pivot element with index J. Returns the ranges:
                (L,J-1) in L,U
                (J+1,U) in I,J
!
00BC 94A4
PART:  LDL ADR,L      !ADR = L !
                ICALR ADCOMP!  !ADR = actual address of a(L) !
                ADD ADRL,RECOFF
                ADDL ADR,#KEYOFF !add in offset of key within record !
                LDL PIVOT,ADR  !PIVOT = actual address of pivot !
                LDL I,L
                LDL J,U
                ADDL J,#ESIZE  !J = J+1 !
!
                PUSHL @SP,L
                PUSHL @SP,U
                ISAVE L,U !
LPI:   CALR UPI      !Inc I until a(I) >= pivot value!
                CALR DOWNJ !Dec J until a(J) <= pivot value or J=<I
                LDL ADR,J
                ICALR ADCOMP!
                ADD ADRL,RECOFF
                LDL L,ADR      !L = actual address of a(J) !
                CPL J,I        !Compare J and I !
                JR LE,MOVPIV   !J <= I, exchange a(J) and pivot !
                CALR EXCHIJ    !Exchange a(I) and a(J) values!
                JR LPI
MOVPIV: CALR EXCHJ    !Exchange a(J) and pivot values!
                POPL U,@SP
                POPL L,@SP     !Restore L,U !
                LDL I,J !Put J in RR4 !
                LDL J,U !Put U in RR6 !
                LDL U,I !Copy of J into U also!
                SUBL U,#ESIZE  !L,U = (L,J-1) !
!
                ADDL I,#ESIZE  !I,J = (J+1,U) !
! Put shorter range in L,U, longer in I,J !
SHORT: LDL SCRL,J      !SCRL = U-L for first range!
                SUBL SCRL,I
                LDL ADR,SCRL !Save first U-L !
                LDL SCRL,U  !SCRL = U-L for second range!
                SUBL SCRL,L
                CPL SCRL,ADR !Compare lengths!
                JR LE,Q1     !Done if second U-L <= first U-L !
                EX IHI,LHI
                EX ILO,LLO
                EX JHI,UHI
                EX JLO,ULO
Q1:   POPL BIGM,@SP
                DEC SPOFF,#8 !Save I,J = longer (L,U) range!
                LDM @SP,IHI,#4
                CALR QUICK   !Recursive call to sort shorter range!
                LDM LHI,@SP,#4 !Restore longer range into L,U !
                INC SPOFF,#8
                CALR QUICK   !Recursive call to sort longer range!
                RET

!Subroutines for moving I and J
CALL UPI: Increment I until a(I) >= pivot value
CALL DOWNJ: Decrement J until a(J) <= pivot value
!
012A 1606
UPI:  ADDL I,#ESIZE !Increment I !

```

BENCHMARK I—Z8000

```

012C 0000
012E 0010
0130 DFF7          CALR CPPI          !Compare pivot value with a(I)!
0132 9E04          RET OV           !OV = 1 says pivot = a(I) !
0134 9E07          RET ULT          !Return if pivot value <= a(I)!
0136 E8F9          JR UPI

0138 1208          DOWNJ:  SUBL J,#ESIZE  !Decrement J !
013A 0000
013C 0010
013E DFFC          CALR CPPJ          !Return if pivot >= A(J) !
0140 9E0F          RET UGE
0142 E8FA          JR DOWNJ

!Pivot and exchange subroutines
CALL CPPI - compare pivot value and a(I). Set FLAGS.
CALL EXCHJP - exchange a(J) and pivot values
CALL EXCHIJ - exchange a(I) and a(J) values
Register use: as for PART
U          scratch
ADR        calling arg for and address returned by ADCOMP
L          actual address of a(J) for exchange routines.
I

0144 9464          CPPI:   LDL ADR,I          IADR = I!
0146 E801          JR IJM

0148 9484          CPPJ:   LDL ADR,J          IADR = J!
014A 8135          IJM:   !CALR ADCOMP!      IADR = adr of comparand!
014C 1604          ADD ADRL,RECOFF
014E 0000          ADDL ADR,#KEYOFF
0150 0003
0152 940A          LDL L,PIVOT          !L = actual pivot address !
0154 BDC7          LDK UHI,#KEYBYTES   !Number of bytes in key !
0156 BA56          CPSIRB @DEST,@SRCE,UHI,NE
0158 0CBE          RET
015A 9E08

015C 9464          EXCHIJ: LDL ADR,I          !CALR ADCOMP!
015E 8135          ADD ADRL,RECOFF
0160 E804          JR EXCH              !Exchange a(I) and a(J) !

0162 9404          EXCHJP: LDL ADR,PIVOT
0164 1204          SUBL ADR,#KEYOFF    IADR = address of a(J) !
0166 0000
0168 0003
016A BDC8          EXCH:   LDK UHI,#(ESIZE/2) !Record word count !
016C 215D          EXLOOP: LD ULO,@ITAD !Pick up pivot or A(I) !
016E 2DBD          EX      EX ULO,@JTAD !Exchange with A(J) !
0170 2F5D          LD @ITAD,ULO        ! a(I) or pivot = a(J)!
0172 A951          INC ADRL,#2
0174 A9B1          INC LLO,#2
0176 FCB6          DJNZ UHI,EXLOOP     !And repeat for whole record !
0178 9E08          RET

017A ACC4          ADCOMP: EXB ADRHH,ADRHL !Move high index to seg field !
017C 8135          ADD ADRL,RECOFF     !Add offset of REC to low index!
017E B424          ADCB ADRHH,RECSEG  !Add seg of REC (with C) to high
0180 9E08          RET                  ! part of index !

```

BENCHMARK K—LSI-11/23

```

1          .TITLE  BENCHMARK K
2          .IDENT  /OCT.23/
3          .ENABL  LC
4
5          ; BOOLEAN MATRIX TRANSPOSE
6          ;
7          ; Transpose a tightly-packed bit matrix
8          ;
9          ; Arguments are passed on the stack.
10         ; Offsets assume 14(8) bytes used for saving registers on stack.
11         ;
12         000016          N          =          16          ; size of matrix
13         000020          A1         =          20          ; pointer to a word of storage
14         000022          A2         =          22          ; bit offset of start of matrix
15
16 000000          ONTRACE::
17 000000          BMT::
18
19          ; save registers
20          ;
21 000000  010046          MOV      R0, -(SP)
22 000002  010146          MOV      R1, -(SP)
23 000004  010246          MOV      R2, -(SP)
24 000006  010346          MOV      R3, -(SP)
25 000010  010446          MOV      R4, -(SP)
26 000012  010546          MOV      R5, -(SP)
27

```

Continued on pg 256

BENCHMARK K—68000

```

69 0 0000003C B883      CMP.L   D3,D4          TEST FOR MEET AT DIAGONAL
70 0 0000003E 66D6      BNE     INNRLP        BRANCH IF NOT FOR ANOTHER SWAP
71
72 0 00000040 51CAFFCC   DBRA   D2,OUTRLP     LOOP UNTIL PAST 'N'
73
74 0 00000044 4CDF0738   MOVEM.L (SP)+,D3-D5/A0-A2  RESTORE REGISTERS
75 0 00000048 4E75      RTS          RETURN TO CALLER
76
77                          END
    
```

BENCHMARK K—Z8000

!Example K: Boolean Matrix Transpose!

!Arguments:!

```

NX      := R0  !Dimension of Matrix!
A2      := R1  !Bit (0<=A2<=15) at which matrix begins in A!
ALX     := R2  !Address of first word of Matrix!
    
```

!Working storage for the routine:!

```

WK5     := R4
NW5     := 9
IJBYTE  := RH4 !Byte containing a(I,J) !
JIYBYTE := RL4 !Byte containing a(J,I) !
TWOBITS := RH5 !Holds both bit values!
IJBP    := R6  !Bit number of a(I,J) !
JIBP    := R7  !Bit number of a(I,J) !
IJPTR   := R8  !Address of IJ byte!
JIPTR   := R9  !Address of JI byte!
IJBX    := R10 !IJBP for outer loop!
JIBX    := R11 !JIBP for outer loop!
LPCNT   := R12 !Counter for outer loop!
    
```

!Long registers for one-step loads!

```

BPL     := RR6
OFFL    := RR8
BXL     := RR10
    
```

```

!Code for Boolean matrix transpose
0000 ABFF  BMTRAN: DEC SP,#16
0002 ABF1  DEC SP,#2*NW5-16
0004 1CF9  LDM @SP,WK5,#NW5      !Save registers !
0006 0408
0008 A11A  LD IJBX,A2            ! I,J = J,I = 1,1 !
000A A11B  LD JIBX,A2
000C A10C  LD LPCNT,NX          !Execute outer loop !
000E ABC0  DEC LPCNT            ! N-1 times !
0010 A9A0  OUTLP: INC IJBX      !Increment row and !
0012 810B  ADD JIBX,NX          ! column for outer loop !
0014 94A6  LDJ BPL,BXL         !Init inner loop ptrs !
0016 9468  INLP: LDJ OFFL,BPL  !Compute byte addresses !
0018 B389  SRA IJPTR,#3
001A FFFD
001C 8128  ADD IJPTR,ALX
001E B399  SRA JIPTR,#3
0020 FFFD
0022 8129  ADD JIPTR,ALX
0024 8255  SUBB TWOBITS,TWOBITS !Init bit keeper!
0026 209C  LDB JIBYTE,@JIPTR   !Get JI byte!
0028 2607  BITB JIBYTE,JIBP    !Put bit into !
002A 0C00
002C AE5E  TCCB NZ,TWOBITS     ! TWOBITS, !
002E B254  RRB TWOBITS         ! sign bit !
0030 2084  LDB IJBYTE,@IJPTR   !Get IJ byte !
0032 2606  BITB IJBYTE,IJBP   !Get IJ bit !
0034 0400
0036 AE5E  TCCB NZ,TWOBITS     ! TWOBITS, 1.s.b. !
0038 B254  RRB TWOBITS         !Reset V if & only if !
003A E0CF  JR NOV,$3           ! bits equal, done !
003C ED07  JR PL,$1            !Is IJ bit set? !
003E 2407  SETB JIBYTE,JIBP   ! Yes, set JI !
0040 0C00
0042 2E9C  LDB @JIPTR,JIBYTE   !Store JI byte !
0044 2084  LDB IJBYTE,@IJPTR !Reread in case IJPTR = JIPTR !
0046 2206  RESB IJBYTE,IJBP !Reset IJ !
0048 0400
004A E806  JR $2
004C 2207  $1: RESB JIBYTE,JIBP ! No, reset JI !
004E 0C00
0050 2E9C  LDB @JIPTR,JIBYTE   !Store JI byte !
0052 2084  LDB IJBYTE,@IJPTR !Reread in case IJPTR = JIPTR !
0054 2406  SETB IJBYTE,IJBP !Set IJ !
0056 0400
0058 2E84  $2: LDB @IJPTR,IJBYTE !Store IJ byte!
005A 8106  $3: ADD IJBP,NX      !Increment row and column !
005C A970  INC JIBP            ! for inner loop!
005E 8B76  CP IJBP,JIBP       !At the diagonal yet? !
0060 EEDA  JR NE,INLP        ! No, keep swapping !
0062 FCAA  DJNZ LPCNT,OUTLP   ! Yes, do NEXT outer loop !
0064 1CF1  LDM WK5,@SP,#NW5
0066 04F8
0068 A9FF  INC SP,#16
006A A9F1  INC SP,#2*NW5-16    !Restore registers !
006C 9E08  RET
    
```

Architectural Concepts for Microprocessor Peripheral Families



Concept Paper

December 1980

Some fundamental constraints on microprocessor peripheral families have always existed, but some of the more severe constraints in the present 16-bit environment will be worse in future 32-bit environments. One of these restrictions is the number of signal lines available--usually corresponding to the number of pins on a package. Present packaging technology for mass-produced parts allows up to 64 pins, which is sufficient for a 16-bit microprocessor with an unmultiplexed address/data bus or a 32-bit microprocessor with a multiplexed address/data bus. Unfortunately, control of these wide buses uses most of the pins available with current packaging, so any device controlling the bus cannot have a wide, independent data path.

The key word here is "independent." It is certainly possible to design a device that could operate a local bus and, when necessary, switch modes to control a global bus. This mode of operation for multiple processor-type devices is inferior for several reasons. First, when the buses are linked, other processes experience longer delays in being serviced. Second, an architecture that allows multiple-processor devices access to most memory in the system is a difficult one in which to assure data and system integrity. A third difficulty is simply the number of devices necessary to link the buses. Typical implementations require six to eight packages.

A significant observation is that the only commercially available I/O devices that incorporate a DMA-type function are serial input/output devices and CRT controllers. Only these applications allow enough pins to properly implement the DMA function.

Fortunately, the same technology that enables the integration of 16- and 32-bit microprocessors also allows the integration of considerable intelligence and some buffer memory in the peripheral device. This is a very powerful combination, especially in conjunction with highly integrated CPU/DMA

combinations, and can be used to link multiple local buses to a main system bus at high speed and with little overhead.

Local buses are, in general, a very effective way to improve overall system performance. They allow significant parallel processing to occur and can improve system reliability by partitioning the tasks to make interference between processes less likely. Many of the problems with linking multiple buses can be avoided by adding buffer memory between the buses. In many of the new-generation I/O devices, this buffer memory can be included on the integrated circuit itself.

An example of the power of these techniques is the construction of a high-speed parallel/serial front-end processor for a high-end microcomputer system (Figure 1).

The key element in this system is the Z8038 FIO (FIFO Input/Output) device. This is a 128x8 FIFO buffer that has the necessary intelligence and flexibility to interface to a wide variety of microprocessors. It also has the ability to interrupt under a variety of conditions and can bypass the data FIFO by a separate path to pass control and status information from one processor to another.

Information is passed from one processor to the other on a message basis. A typical transfer begins with the main system processor sending a control byte through the FIO to the local processor via the bypass register. This control communication typically includes information about the data block length, the intended destination, and any other relevant parameters. At the same time, the main system DMA can be set up to begin transferring data into the FIO. Either of the two DMA controllers in this system can be eliminated with little loss in performance if the CPU has block memory-to-I/O move instructions available, as in the Z80 or Z8000. After initial setup of the FIO, the main system DMA is activated and quickly

fills the FIO's data buffer, if the local system DMA has not yet been activated. This is of little consequence, since the main system DMA will simply stop transfers when the RDY signal from the FIO goes inactive. Similarly, if a block move instruction is being used instead of a DMA, the FIO provides an "interrupt-on-full" interrupt, which allows the CPU to do other tasks until next interrupted by the FIO. This second interrupt occurs only when the contents of the FIO have been emptied to a predetermined programmable level.

Similarly, on the local bus side of the FIO, the DMA will be active only when there is data remaining in the FIO. To reduce the number of bus request cycles (or interrupts in the case of a block move instruction), the FIO can be programmed to request service from the local DMA only when the FIFO contains more than a certain programmable number of bytes. It will then transfer until the FIFO is empty and continue this burst cycling until the end of the block.

The combination of the block move instructions and the FIO is more powerful than the replacement of the DMA function. Unlike the DMA, which, by requesting the system bus, places itself at a higher priority than any interrupt in the system, the block move instructions can be interrupted. This means that a high-priority interrupt in either the local system or the main system can be serviced immediately, even though the CPU is involved in a very high speed transfer of

data through the FIO. If the interrupt routine is short, the other system may not even notice that the FIO was not being serviced for a short interval. If the interrupt is longer, the fact that the FIO may go empty is of little consequence. An interrupt on empty or an inactive RDY line will serve to temporarily suspend service of the FIO at the local end.

The FIO is sufficiently flexible to interface in four distinct applications:

- To a multiplexed address/data bus microprocessor.
- To an unmultiplexed address/data bus microprocessor.
- With handshake lines to most types of parallel-interface I/O devices.
- As a "high byte portion" of a 16- or 32-bit link between buses.

Figure 1 also shows the use of the FIO in a handshake application. One of the principal advantages of the FIO in this configuration is its ability to decrease interrupt handling overhead by more than two orders of magnitude, compared to the typical interrupt handling with a parallel I/O device. For example, if interfaced to a line printer, the CPU would be interrupted once per line rather than once per character. Another capability of the FIO is its ability to recognize special characters (or bits in a character). It can interrupt or stop DMA transfers when a special character comes through the FIFO, such as End of File.

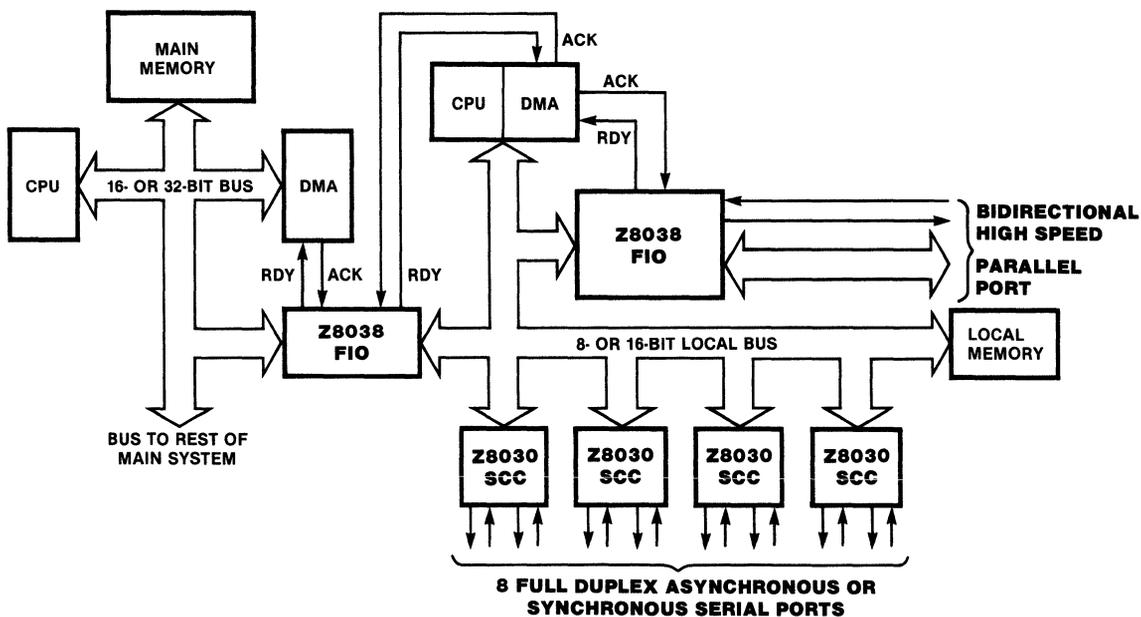


Figure 1. High-End Microcomputer System

The other device shown in the example is the Z8030 Z-SCC (Serial Communications Controller). This device can interface to nearly any type of serial device at up to a speed of 1 million bits per second. This includes all popular asynchronous formats and IBM Bisync (including Transparent mode), as well as the newer protocols such as X.21, X.25, SDLC, and HDLC. In its various modes, the Z-SCC can generate and check the two most popular CRCs (Cyclic Redundancy Codes). It also provides parity generation and checking and handles various lengths of characters.

One major advantage of the Z-SCC over previous serial communications devices is its ability to do all clock recovery and generation for most types of encoding. Specifically, it can encode and decode NRZI as well as FM encoded data with transitions being interpreted as either 1s or 0s. It can also recover both clock and data from Manchester encoded data.

In addition to its clock recovery capabilities, the Z-SCC has two timers for independent baud rate generation in each full duplex channel. The timing sources can be the Z-SCC control clock, an external clock source, or the output of either of the on-chip crystal oscillators. This extreme flexibility in timing allows complete on-chip local loopback testing. An Auto-Echo mode is also provided for modem and link testing.

In keeping with the trend of increased buffer memory, the Z-SCC has sufficient on-board buffering (four characters in the receiver) to allow time for interrupt response even at relatively high data rates. If DMA control becomes necessary for even faster data transfer, this can be accomplished in a full duplex manner in both channels.

Zillog
Zillog
Zillog
Zillog
Zillog

Interfacing to the Z6132 Intelligent Memory



Application Note

April 1981

Introduction

In memory applications where the requirements for byte-wide buffer storage are modest (2K to 32K bytes), the Z6132 offers a new concept in intelligent memory. The Z6132 features 4K bytes of RAM in a byte-wide, 28-pin package that conforms to the 2716/2732 JEDEC standard.

This application note discusses the basic features and operating modes of the Z6132, with application examples given for each of Zilog's microprocessors. In addition to a discussion of the interface requirements for the Z8™, Z80® and Z8000™ CPUs, an application example describes the design requirements for interchanging the Z6132 with 2716/2732-type EPROMs. Each interface design includes logic and timing diagrams. Other Zilog documents

that might be useful are referenced throughout the application note.

The application note is divided into four sections. The first section provides a general description of the Z6132 along with functional descriptions of each available type of memory operation. The self-refresh operation is discussed with the various refresh options available with the Z6132. The second section begins the application examples by providing interface circuitry for the Z80A CPU. The third and fourth sections provide interface circuitry and timing for the Z8002 and Z8 microprocessors. The section that discusses the Z8 memory interface also treats the design criteria for interchanging the Z6132 with either 2716- or 2732-type EPROMs.

General Description of the Z6132

The Zilog Z6132 is a +5 V, intelligent, MOS dynamic RAM organized into 4096 8-bit words. The Z6132 uses high-performance, depletion-load, double-poly, n-channel, silicon-gate MOS technology with a mixture of static and dynamic circuitry that provides a small memory cell and low power consumption. Internally, the Z6132 uses dynamic storage cells, but externally, the Z6132 functions as a static RAM because it controls and performs its own refresh. This eliminates the need for external refresh support circuitry and combines the convenience of a static RAM with the high density and low power consumption normally associated with dynamic RAMs.

The Z6132 is particularly well suited for microprocessor and minicomputer applications where its byte-wide organization, self-refresh, and single power-supply voltage result in a reduced parts count and a simplified design. The Z6132 supports both multiplexed and non-multiplexed address and data lines using the control signals Address Strobe (\overline{AS}) and Data Strobe (\overline{DS}) to latch address and data internal to the memory chip. The circuit is packaged in an industry-standard, 28-pin DIP and is pin

compatible with the proposed JEDEC standard. The Z6132 conforms with the Z-BUS specification used by the new generation of Zilog microprocessors, the Z8 and Z8000.

The Z6132 4K × 8 quasi-static RAM is organized as two separate memory-bit blocks. Each block has 128 sense amplifiers with 64 rows of memory bits on each side. Both blocks have separate row address buffers and decoders. The two sets of row address decoders are addressed either by the address inputs A_1 - A_7 or by the internal 7-bit refresh counter. The least significant address input (A_0) selects one of the two blocks for external access. While the selected block performs a read or write operation, the other memory block uses the refresh counter address to refresh one row. Details of the self-refresh mechanism are discussed in the next section.

A memory cycle starts when the rising edge of Address Clock (AC) clocks in Chip Select (\overline{CS}), A_0 , and Write Enable (WE). If the chip is not selected (\overline{CS} is High), all other inputs are ignored until the next rising edge of AC. If the chip is selected (\overline{CS} is Low), the 12 address bits and the Write Enable bit are

General Description of the Z6132
(Continued)

clocked into their respective internal registers. The block addressed by A_1-A_{11} is determined by A_0 ; the other block is refreshed by the 7-bit refresh counter.

The Chip Select and address inputs must be held valid for only a short time after the rising edge of AC. This supports the multiplexing of address and data and allows enough setup time for the multiplexed data lines to settle with respect to the input control signal Data Strobe.

A read cycle is initiated by the rising edge of AC while \overline{CS} is Low and \overline{WE} is High. A Low

on the \overline{DS} input activates the data outputs after a specified delay. During a read operation, \overline{DS} is only a static Output Enable signal.

Write cycle is initiated by the rising edge of AC while both \overline{CS} and \overline{WE} are Low. The \overline{WE} input is checked again on the falling edge of \overline{DS} . If \overline{WE} is still Low, the falling edge of \overline{DS} strobes the data on the D_0-D_7 inputs into the addressed memory location. Data must be valid for only a short hold time after the falling edge of \overline{DS} .

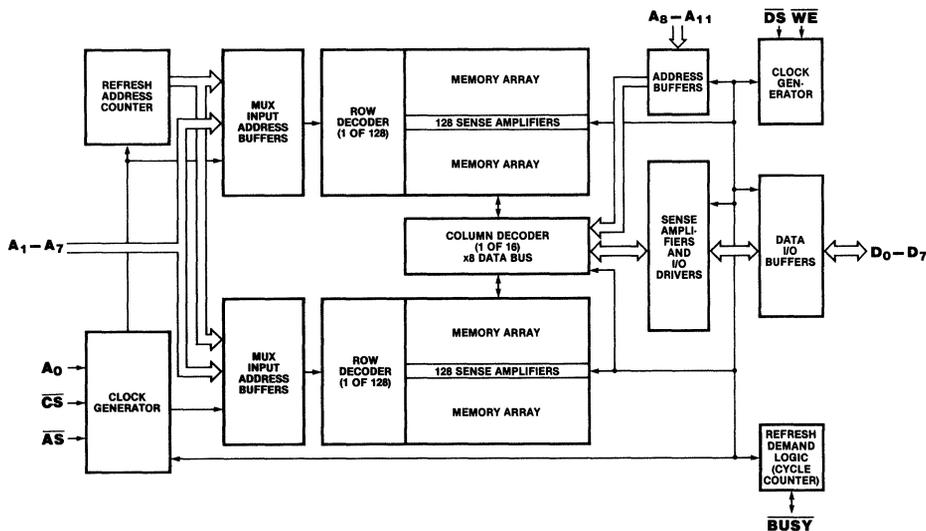


Figure 1. Block Diagram

Self-Refresh Operation

The Z6132 stores data in a single-transistor dynamic cells that must be refreshed at least every 2 ms. Each of the two memory blocks contains 16,384 cells and requires 128 refresh cycles to completely refresh the array. The Z6132 operates in one of two user-selectable self-refresh modes, each satisfying the refresh time requirements. On the basis of the available memory cycle time, the user can decide to use either the Long Cycle-Time Refresh mode or the Short Cycle-Time Refresh mode. The Long Cycle-Time Refresh mode is the simplest self-refresh mode and is enabled by permanently grounding the \overline{BUSY} output pin of the Z6132. Every memory cycle in this

mode consists of a memory operation followed by a refresh operation on both blocks, after which the refresh counter is incremented. Internally, the complete cycle consists of a four-phase sequence:

1. Memory read, write, or write inhibit
2. Precharge
3. Refresh
4. Precharge

These internal operations are automatic and transparent to the user. When the chip is not selected (\overline{CS} is High when AC goes High), the first two phases are omitted. There are two important requirements: the memory cycle

Self-Refresh Operation

(Continued)

times must always be longer than the TC (minimum memory cycle time) value specified when $\overline{\text{BUSY}}$ is Low, and there must be at least 128 Address Clocks in any 2 ms period.

The Long Cycle-Time Refresh mode is most practical for microprocessor applications where the read and write cycle times are in the range of 650–750 ns. The Short Cycle-Time Refresh mode is a more sophisticated self-refresh mode that is activated by pulling the $\overline{\text{BUSY}}$ output pin High through a pullup resistor (typically 1 k Ω) to V_{CC} . The $\overline{\text{BUSY}}$ outputs of several Z6132 chips can be OR-wired together. In this mode, the Z6132 always performs a refresh operation on the memory block that is not being addressed from the outside.

If the chip is selected ($\overline{\text{CS}}$ is Low when AC goes High), the refresh counter refreshes the block that is not addressed by A_0 . The refresh counter is incremented after both an even and an odd address have occurred. This self-refresh scheme takes advantage of the sequential nature of most memory addressing. If the chip is deselected ($\overline{\text{CS}}$ is High when AC goes High), both blocks are refreshed and the refresh counter is incremented after every

cycle. Hence, the addressing of PROM or I/O can also be used to refresh the Z6132 by allowing it to receive Address Clocks without Chip Select.

Under normal conditions, the deselected and odd/even self-refresh mechanisms step through 128 refresh addresses in less than 2 ms. To guarantee proper refresh operation, even in the exceptional case of the memory being continually selected and addressed by a long string of all even or all odd addresses, a built-in cycle counter activates the $\overline{\text{BUSY}}$ output and requests a lengthened memory cycle to append a refresh operation. This internal cycle counter is reset whenever the refresh counter is incremented. The cycle counter then counts memory cycles and activates the $\overline{\text{BUSY}}$ output when it reaches a count of 17.

$\overline{\text{BUSY}}$ is fed into the WAIT input of most microprocessors and is a request to the CPU for a longer memory cycle. The $\overline{\text{BUSY}}$ line is held Low by the Z6132 until the refresh cycle has started. $\overline{\text{BUSY}}$ becomes active only when the Z6132 has been selected and addressed with all odd or all even addresses for 17 consecutive Address Clocks.

Interfacing the Z6132 to the Z80A CPU

The Z6132 was designed to interface with Z-BUS™-compatible microprocessors such as the Z8 and Z8000. Although the Z80 does not directly produce Z-BUS-compatible memory signals, only three commonly available integrated circuits are required to interface the Z6132 with the Z80A CPU. The interface logic, circuit description, and timing diagrams for each important processor cycle are discussed later. Further information on the Z6132 and Z80A CPU can be obtained from the *Z6132 Product Specification* (document number 00-2028-A) and the *Z80B CPU AC Characteristics* (document number 00-2005-A).

The M1 or opcode fetch cycle of the Z80A CPU represents the shortest memory cycle and must be given careful consideration when designing memory interface logic. Figure 2 shows the Z80A CPU M1 cycle in detail along with worst-case delay timings for the important control signals. The maximum access time allowed for an opcode fetch (under ideal conditions) is 500 ns in clock cycles T_1 and T_2 . Considering worst-case Z80A CPU data setup time (35 ns in T_2) and worst-case opcode

address stable time (110 ns in T_1), the maximum access time available for a memory fetch is reduced to 355 ns.

To keep the interface logic for the Z6132 to a minimum and still use commonly available parts, the Z6132-5 (300 ns access time) is exemplified. Timing edges provided by the Z80A CPU clock are used to activate the Z6132 Address Clock (signal AC shown in Figures 2, 3, 4, and 5). Figure 7 shows the logic for the Z80A-to-Z6132 interface. The 74S00 NAND gate has a maximum delay of 5 ns, the 74LS04 inverter has a maximum delay of 15 ns, and the 74S74 has a maximum clock to output delay of 9 ns. The clear-to-Q output Low delay is 8 ns for the 74S74. These numbers are displayed in the timing diagrams for the Z6132 control signals $\overline{\text{CS}}$, AC, $\overline{\text{DS}}$, and $\overline{\text{WE}}$ (Figures 2-6).

The following description of a memory fetch cycle illustrates how each of the important Z6132 timing parameters is met. The M1 cycle begins with the activation of Z80A CPU control signal $\overline{\text{M1}}$ in clock cycle T_1 . Since the maximum delay for $\overline{\text{M1}}$ is 100 ns (Figure 2) and the

Interfacing the Z6132 to the Z80A CPU
(Continued)

maximum delay from the rising edge of T_1 until addresses are stable is 110 ns, the control path that gates $\overline{M1}$ and CLK to clear the 74S74 flip-flop is used to force AC High.

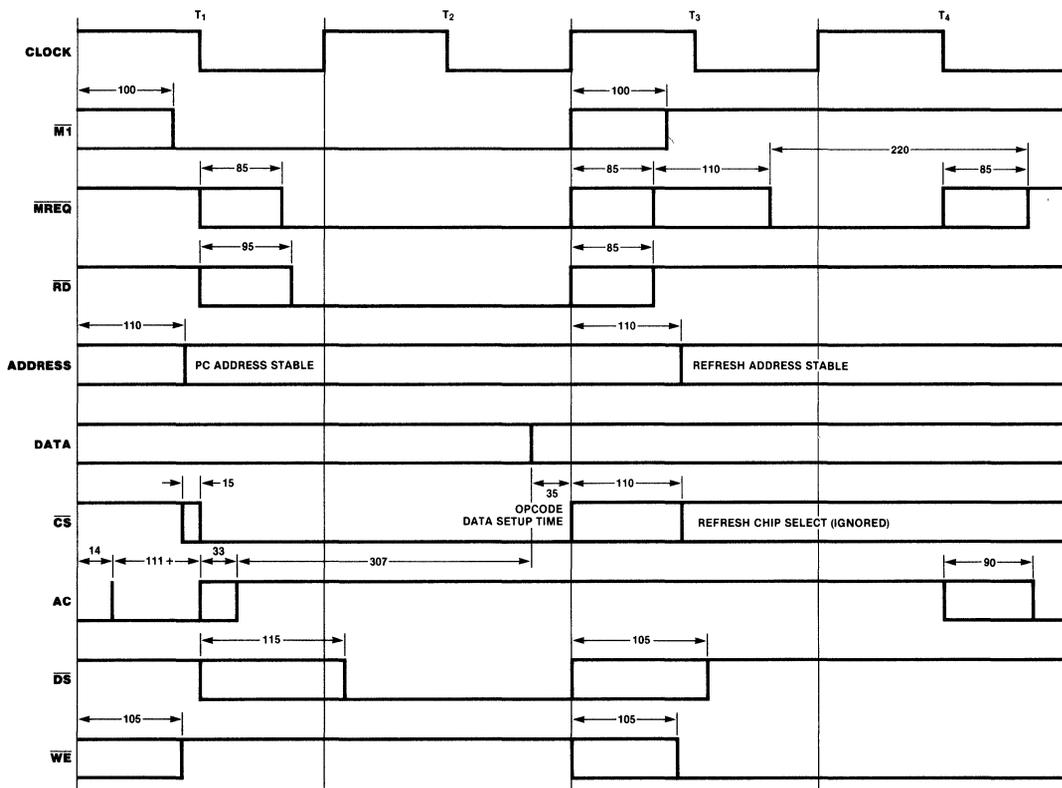
The delay of 33 ns shown in Figure 2 for AC from the falling edge of T_1 was derived from the collective delays of the 74LS04 (15 ns), the 74S00 (5 ns), the 74S74 clear (8 ns), and the final 74S00 gate (5 ns). Thus, under the worst conditions possible, a memory cycle begins with the rising edge of AC 158 ns after the rising edge of clock cycle T_1 .

As a reminder, the $M1$ machine cycle is a 2-clock-cycle instruction fetch, which requires the data fetched to meet the specified setup time (35 ns) before the rising edge of clock cycle T_3 . With 35 ns required for worst-case data setup time, the remaining time in T_1 and T_2 for memory access is:

$$500 \text{ ns} - (158 \text{ ns} + 35 \text{ ns}) = 307 \text{ ns}$$

This allows the use of 300 ns access time RAMs even under worst-case conditions.

The Z6132-5 has a guaranteed access time of



NOTE:
Two wait states automatically inserted here by CPU. Each wait state is one clock cycle long.

Figure 2. Z80A Opcode Fetch Cycle Timing

Interfacing the Z6132 to the Z80A CPU
(Continued)

300 ns and is recommended for use with the Z80A CPU to simplify interface circuitry. This mode takes advantage of the self-refresh feature of the Z6132 so that interfacing the Z80A CPU refresh control signals is not required.

The 74S74 flip-flop is useful for two reasons. The Z80A CPU refresh cycle, with its accompanying $\overline{\text{MREQ}}$, is effectively blocked by the 74S74 during an M1 cycle. This is required because the refresh cycle during machine

cycle M1 generates an $\overline{\text{MREQ}}$ signal that violates the AC timing requirements of the Z6132. The second purpose of the 74S74 is realized during an interrupt acknowledge cycle. The Z80A CPU uses the simultaneous occurrence of $\overline{\text{M1}}$ active with $\overline{\text{IORQ}}$ active to indicate that an interrupt acknowledge cycle is in progress. If the 74S74 flip-flop is removed, the Address Clock becomes active during every clock cycle time (425 ns) for the Z6132-5. Figure 3 illustrates memory timing for

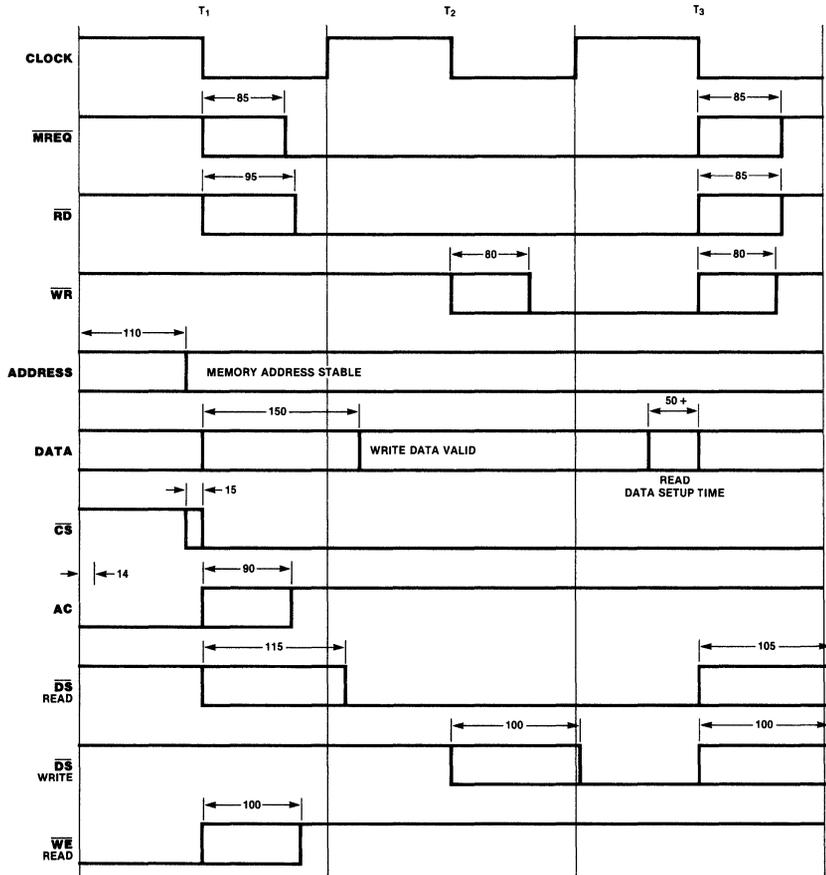


Figure 3. Z80A Memory Cycle Timing

Interfacing the Z6132 to the Z80A CPU
(Continued)

the Z80A CPU memory read or write cycle. In this cycle, \overline{MREQ} is issued by the Z80A CPU to initiate a memory operation. The Z80A CPU control signals, \overline{MREQ} and \overline{RD} , closely track each other over the guaranteed temperature range. Were this not the case, \overline{DS} could potentially become active before \overline{AC} becomes true. The three 74LS04 inverters in the \overline{DS} path help to insure that \overline{DS} will become active only after \overline{AC} has become true. Figure 3 shows \overline{WE} in a memory read cycle. Only the occurrence of $\overline{M1}$ (indicating an opcode fetch or an interrupt acknowledge) or the occurrence of \overline{RD} (indicating $\overline{M1}$ or memory read) inhibit \overline{WE} from becoming active. During a memory read, the close tracking of \overline{MREQ} and \overline{RD} insures that \overline{WE} setup time to \overline{AC} High (-10 ns) is met.

Figure 4 shows a Z80A CPU I/O cycle along with the corresponding active Z6132 memory control signals. Since \overline{AC} never makes a positive transition during this I/O cycle, the

other memory control signals (such as \overline{CS} and \overline{DS}) do not affect operation of the Z6132.

Figure 5 shows a Z80A CPU interrupt acknowledge cycle. Although \overline{AC} makes a positive transition and \overline{CS} could be true (depending on the Z80A CPU's current PC), the memory control signal \overline{DS} never becomes active during an interrupt acknowledge cycle. This cycle appears to be an aborted read cycle to the Z6132 and has no harmful effect.

Thus, with only three commonly available 14-pin packages, a simple interface between the Z80A CPU and the Z6132 can be constructed. The Z80A was chosen for this application example because it allows 4 MHz operation while using relatively inexpensive (300 ns) memory. Operation of the Z80B CPU (6 MHz) provides for a maximum memory access time of 210 ns in the opcode fetch cycle (not including memory interface logic) under worst-case conditions. Figure 6 shows the timing for the Z80B opcode fetch cycle with its associated

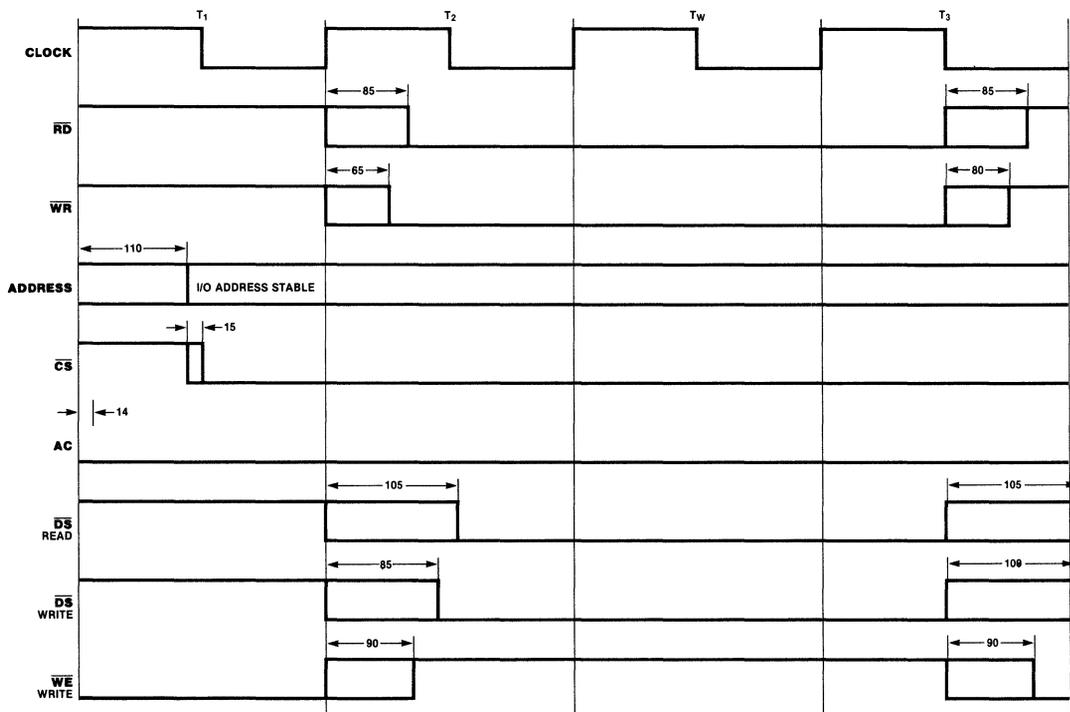


Figure 4. Z80A I/O Cycle Timing

Interfacing the Z6132 to the Z80A CPU
(Continued)

maximum delays. In this configuration, one wait state can be inserted to increase the available access time to 375 ns. In systems that require higher performance, the Z80B CPU (even with one wait state included in opcode

fetch cycles) can increase processor execution efficiency. The Z80 CPU (2.5 MHz) is also easily interfaced with the Z6132 family. Here, as with the Z80A CPU, no additional wait states need to be added.

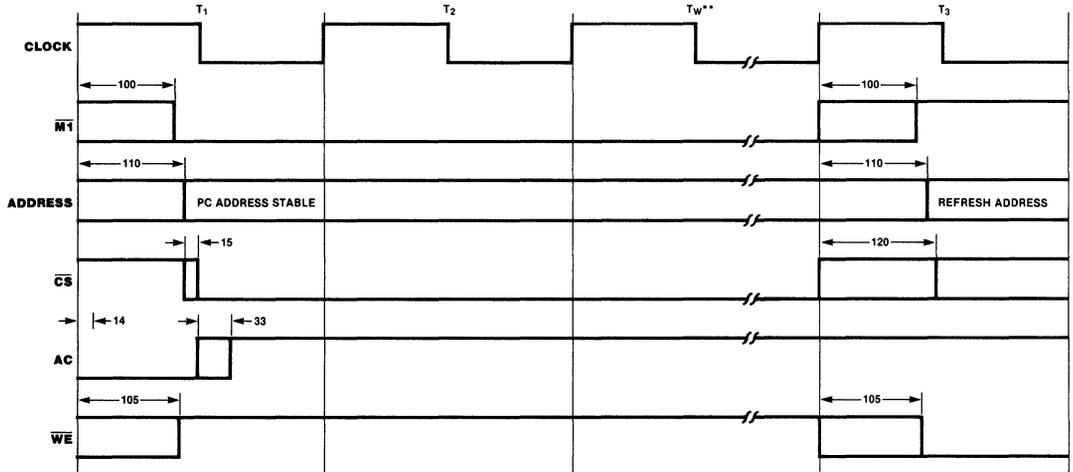


Figure 5. Z80A Interrupt Acknowledge Cycle Timing

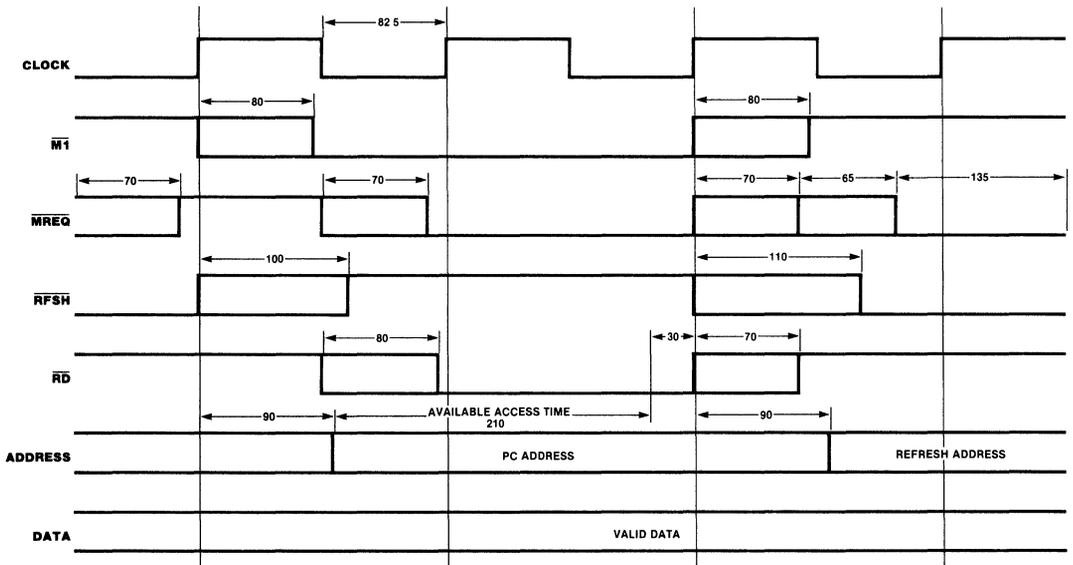


Figure 6. Z80B Opcode Fetch Timing

**Interfacing
the Z6132 to
the Z80A CPU**
(Continued)

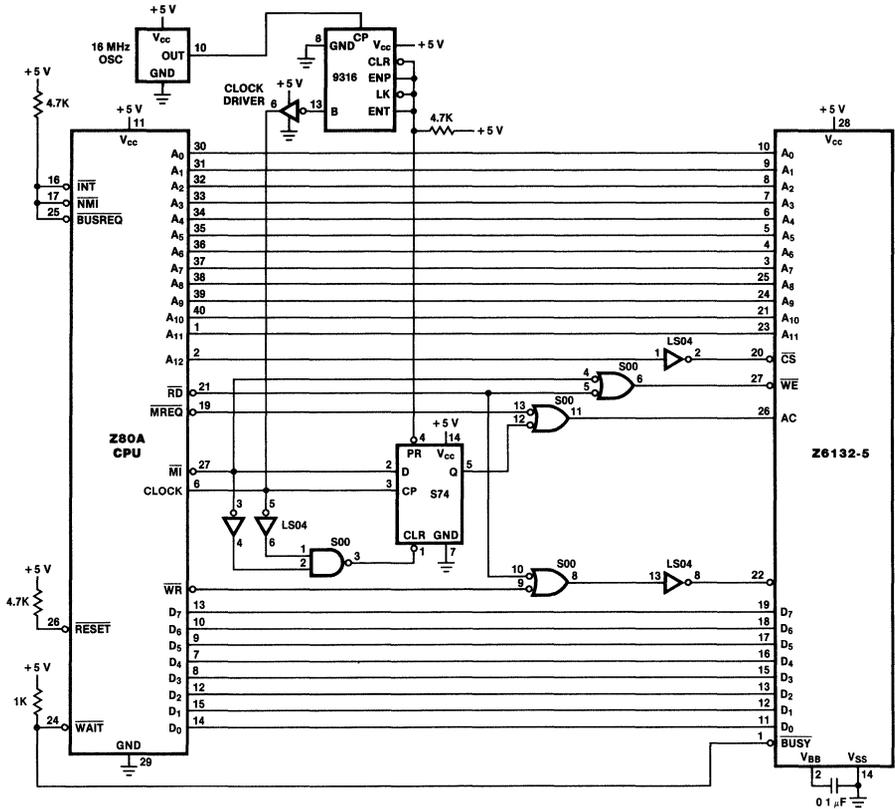


Figure 7. Z80A/Z6132 Interface Logic

**Interfacing
the Z6132 to
the Z8002 CPU**

Two Z6132s are interfaced to a Z8002 (non-segmented Z8000) in this example to provide 4K words (16 bits wide) of buffer storage. Three external TTL packages provide all address chip select and byte/word decoding to the Z6132s. The timing diagrams (Figures 8-10), the interface logic (Figure 11), and the circuit description are discussed later. Information on the Z8002 CPU can be obtained from the *Z8000 CPU Product Specification* (document number 00-2045-A), the *Z8001/Z8002 CPU AC Characteristics* (document number 00-2004-A) and from the *Z8000*

CPU Technical Manual (document number 00-2010-C). A Z8002 running at 4 MHz was chosen to provide high throughput while still providing a generous memory access time of 360 ns for the Z6132s (Figures 8-10). The Z6132-6 chosen for this example has a maximum access time of 350 ns. All Z8002 memory transactions are three clock cycles long and conform to the Zilog Z-BUS timing specifications. More information on the Zilog Z-BUS can be found in the *Z-BUS Summary* (document number 00-2031-A).

Interfacing the Z6132 to the Z8002 CPU
(Continued)

The Z8002 uses a multiplexed address/data bus to provide for memory addressing and data transfer. The rising edge of Address Strobe (\overline{AS}) guarantees that addresses from the Z8002 are stable. This signal (\overline{AS}) is fed directly to the Z6132s as the Address Clock (AC) input clocks in memory addresses and initiates a memory cycle. The Z6132 samples its Chip Select (\overline{CS}) pin with the rising edge of AC to determine whether the bus transaction is intended for it. If \overline{CS} is found Low on the rising edge of AC, the Z6132 begins a read or write operation, depending on the state of its Write Enable (\overline{WE}) pin. The Z6132 samples \overline{WE} again on the falling edge of Data Strobe (\overline{DS}). If \overline{WE} is still Low, the write cycle is continued. If \overline{WE} has returned to the High state, the memory write cycle to the Z6132 is aborted. This feature of the Z6132 allows memory write cycles to be suppressed if determined undesirable, without paying an access-time penalty. The R/ \overline{W} signal is fed directly from the Z8002 to the Z6132 \overline{WE} pin. The signal \overline{DS} from the Z8002 indicates when valid data is available on the multiplexed address/data bus. This signal indicates if valid CPU data is available to the Z6132 during a write cycle and enables the Z6132 output buffers during a CPU read cycle. The \overline{DS} signal from the CPU is fed directly to the \overline{DS} input of the Z6132. The only interface circuitry between the Z8002 and the Z6132 is the decoding of required byte/word, read/write, and high-byte/low-byte Z8002 memory control functions (Figure 11). A 74LS157 dual multiplexer is used to provide

enable signals for the even and odd banks of Z6132s. The truth table for this multiplexer follows. Both even and odd banks are enabled except during byte operations. During byte write operations, only one bank of Z6132s is enabled. This bank is determined by AD_0 .

INPUTS			OUTPUTS	
R/ \overline{W} (Enable)	AD_0 (B/A Select)	B/ \overline{W} (1A, 2B)	EVEN	ODD
0	X	0	0	0
0	0	1	0	1
0	1	1	1	0
1	X	X	0	0

X = don't care

When the Z8002 performs a read operation, 16 bits of memory data are returned to the CPU. For byte read transactions, the appropriate (odd or even) byte is selected internally to the Z8002. The enable input for the 74LS157 is active Low. When the R/ \overline{W} output of the Z8002 is High (indicating a read operation), the 74LS157 is disabled, forcing the even and odd outputs Low. During a write operation, the 74LS157 is enabled and the even and odd outputs are determined by the states of the B/ \overline{W} and AD_0 CPU outputs. During a word-write operation, both even and odd outputs are enabled. During a byte-write operation, the enabled even or odd bank is determined by the least significant address bit (AD_0). A byte-write to an even address (AD_0 is 0) corresponds to an even enable. When this byte is

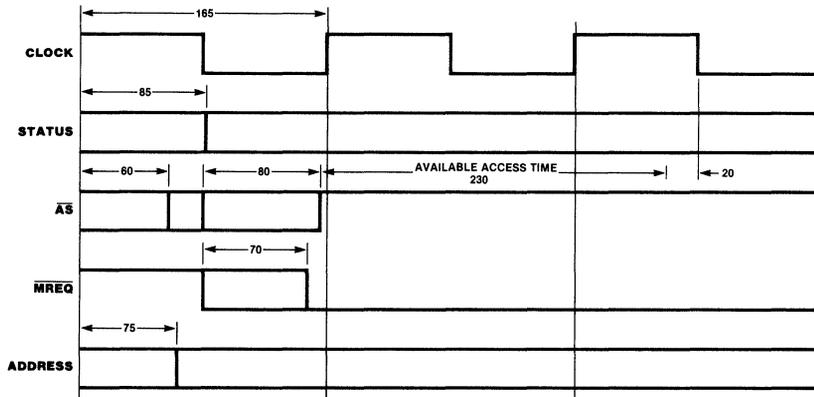


Figure 8. Z8000 Memory Transaction (6.0 MHz)

Interfacing the Z6132 to the Z8002 CPU
(Continued)

read back from the Z6132, the Z8002 expects it to appear on the upper eight data bits (AD₈-AD₁₅). The lower eight data bits are connected to the odd bank, and the upper eight data bits are connected to the even bank. The least significant address bit (AD₀) is not connected to the Z6132 (although it still functions

as a data bit). It is used instead in the selection of even or odd Z6132 banks. A 74LS138 is used to further decode even and odd addresses into individual even and odd Chip Selects for the Z6132s. Memory transactions (excluding refresh operations) are reflected by status bit 03 (High) of the Z8002 CPU. This bit is fed to

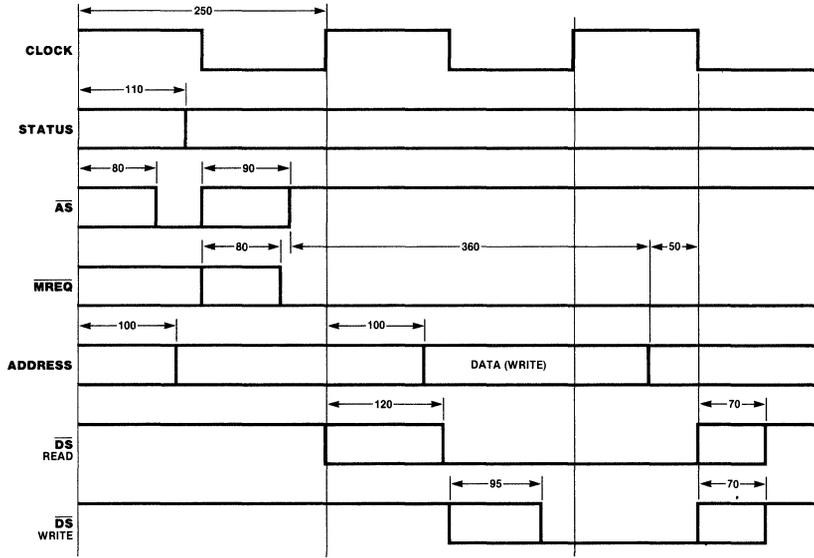


Figure 9. Z8000 Memory Transaction (4.0 MHz)

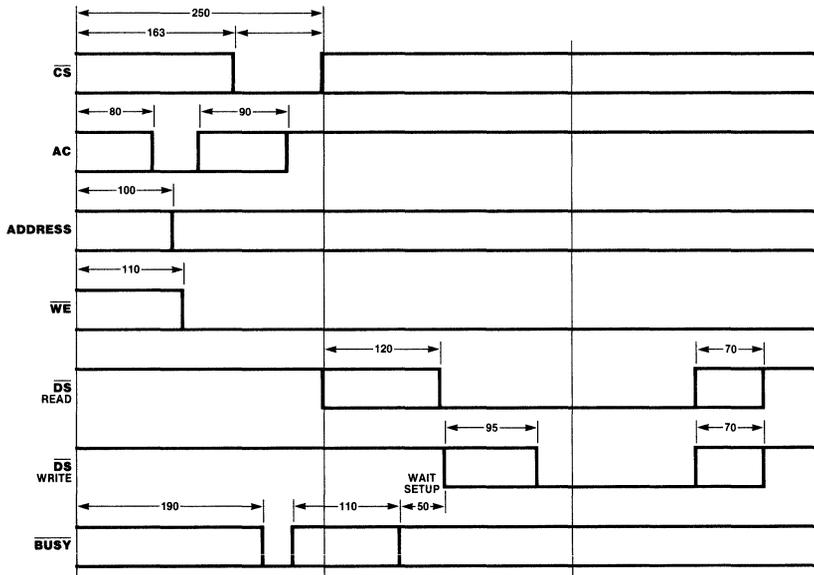


Figure 10. Z6132-6 Interface Timing (4.0 MHz)

**Interfacing
the Z6132
to the Z8**

In the following example, a Z6132-5 (300 ns) is interfaced to a Z8 operating at 7.3728 MHz. Timing for interfacing the Z8 to a Z6132-4 (250 ns) is discussed for 8-MHz Z8 operation. In addition, the example describes 2716 and 2732 EPROM interchangeability with the Z6132. Timing diagrams and circuit drawings have been included for Z8 memory interface timing and are discussed in this section.

The Z8 is an 8-bit, general-purpose micro-computer chip that can be configured under software control. The Z8 features regular architecture with 144 on-chip registers, 2K bytes of on-chip ROM, and 32 I/O lines configured for conventional I/O or for external memory. Detailed information on the Z8 can be found in the *Z8 Microcomputer Technical Manual* (document number 03-3047-02) and the *Z8601/2/3 MCU Microcomputer Product Specification* (document number 00-2037-A). The Z8 uses Port 1 (eight bits wide) as a

multiplexed address/data bus and Port 0 as the upper byte of a 16-bit address bus. Before external memory references to the Z6132 can be made by any instruction, the user must configure Ports 0 and 1 appropriately. Instruction pipelining mandates that after setting the modes of Ports 0 and 1 for external memory operation, the next two bytes are fetched from internal program memory. Two single-byte instructions, such as NOPs, can be used to accomplish this. On-board ROM in the Z8 is available from 0000-07FF (Hex). This application locates the external Z6132 in the Z8 address space from 1000-1FFF (Hex).

All Z8 timing references are made with respect to the output signals \overline{AS} and \overline{DS} . The control signal \overline{AS} indicates when the Z8 address bus is valid, while the control signal \overline{DS} controls the flow of data. The Z8 status signal R/W (Read/Write) indicates the direction of data flow. The Z8 indicates when a

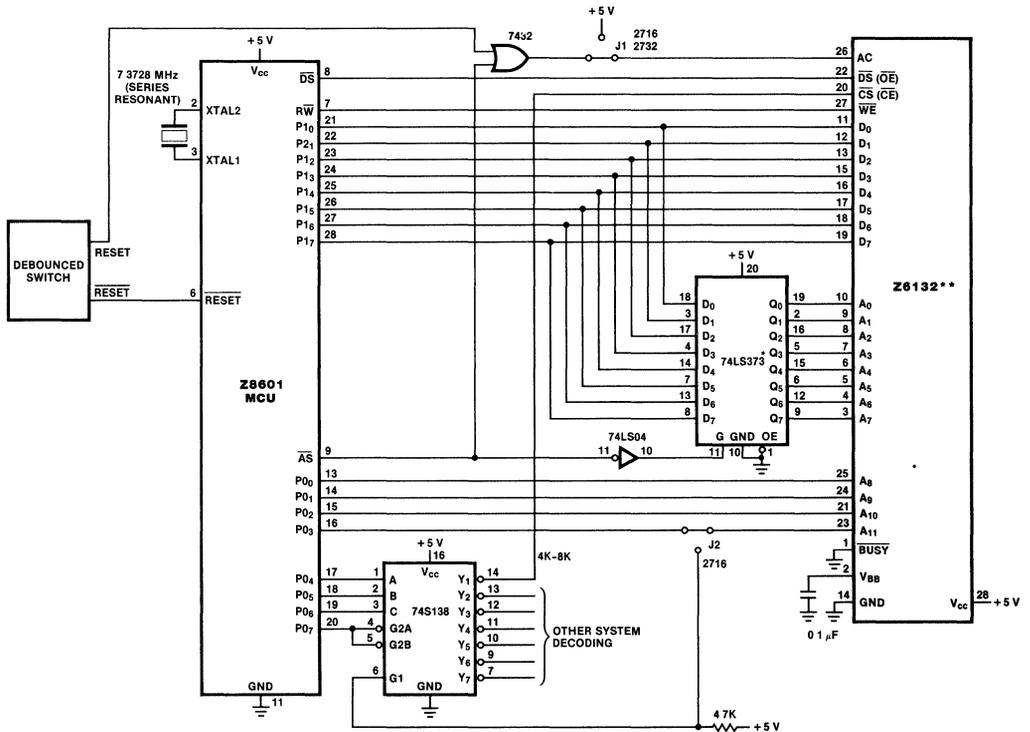


Figure 12. Z8601/Z6132 Interface Logic

Interfacing the Z6132 to the Z8

(Continued)

hardware reset operation is in progress by activating \overline{DS} while outputting \overline{AS} at the internal clock rate. Since the internal clock has a cycle time period of 250 ns, it is necessary to inhibit \overline{AS} during hardware reset operations so that the minimum memory cycle time for the Z6132 is not violated. This is easily accomplished by using the reset line to the Z8 as an inhibit line to the AC input of the Z6132 (Figure 12). The 74LS32 OR gate delays the Address Clock to the Z6132 a maximum of 22 ns.

The basic memory cycle of the Z8 is six clock cycles (810 ns at 7.3728 MHz). An extended cycle mode is available under software control that lengthens memory access by one clock cycle. At 8 MHz, this cycle is reduced to 750 ns. In either case, the Z6132 timing parameters TC (read or write cycle time), \overline{TwACh} (AC width High), and $\overline{TdDS(AC)}$ (Low to AC High) all allow the Z6132 to be used in the Long Cycle-Time Refresh mode. For this reason, the Z6132 \overline{BUSY} line is permanently grounded.

EPROM Compatibility

The Z6132 is packaged in an industry-standard, 28-pin DIP and is pin compatible with the proposed JEDEC standard. This allows the substitution of other 28-pin DIPs that conform to the proposed JEDEC standard (namely the 2732 and 2716 EPROMs). The 2732 EPROM requires only that +5 V (V_{CC}) be substituted for AC (pin 26 on the Z6132). This substitution can be accomplished easily with a jumper (Figure 12). Interfacing a 2716 requires one additional jumper change. The 2716 EPROM is only 2K bytes, and hence requires only 11 address bits for full addressing capability. A second jumper pad for 2716 selection can be

included to tie pin 23 to a pullup resistor as required for reading a 2716.

Since the Z8 multiplexes addresses and data on Port 1, it is necessary to latch the low-order address byte with the Z8 control signal \overline{AS} . This latch is unnecessary for systems without 2716/2732 EPROM capability, since the address to the Z6132 may change after the specified address hold time (60 ns for the Z6132-5). The 2716 and 2732 EPROMs are 24-pin packages, and the Z6132 is a 28-pin package. This requires the EPROMs to be physically justified so that pin 1 of the 2716/2732 is aligned with pin 3 of the Z6132.

Theory of Operation

Figure 12 shows the circuit diagram for a small Z8 system. In this configuration, a series resonant crystal (7.3728 MHz) provides all system timing. Port 1 is configured for multiplexed address and data, and Port 0 is configured to provide the upper address byte to complete the 12-bit address bus required by the Z6132 and to provide four bits of address decoding. The upper bits of Port 0 ($P0_4$ to $P0_7$) are decoded by a 74S138 to provide eight blocks, each 4K bytes long. The first block is discarded because it overlaps with internal Z8 ROM. The second segment is used to generate \overline{CS} for the Z6132, and the last six segments are free for other system chip select decoding, such as additional memory or external I/O ports. A 74LS373 is used to latch addresses from the multiplexed address/data bus of Port 1. This latch is enabled when \overline{AS} is active

(Low) and retains the addresses after \overline{AS} has returned High. *The Z6132 does not require addresses to be stable throughout the entire memory cycle, so this latch is used only with systems that provide the option of using the 2716 and 2732 EPROMs.* Addresses are latched internally to the Z6132 on the rising edge of AC. Jumpers J1 and J2 are connected as shown for Z6132 operation. To substitute a 2732 for the Z6132, the existing jumper (J1) must be cut from the Z6132 pin 26 to the Z8 pin 9, and Z6132 pin 26 is connected to V_{CC} . To substitute a 2716, one additional jumper change must be made. Jumper J2 is shown connected for Z6132 and 2732 operation. To substitute a 2716, the existing jumper is cut from the Z6132 pin 23 to the Z8 pin 16, and the jumper at J2 from the Z8 pin 23 is connected to the 4.7K pullup resistor.

Timing

The important control signals for memory interface to the Z8 have been reproduced in Figures 13-16. In this design example, a crystal frequency of 7.3728 MHz was selected for overall system timing. The Z8 product specifications provide timing specifications at 8 MHz. To calculate the timing parameters for frequencies other than 8 MHz, the timing parameters are derated by a factor based on the difference in clock period. For instance, the timing parameter $T_{dA}(\overline{AS})$ is given as 30 ns (min) for a clock input of 8 MHz. To

calculate the timing value for a clock input of 7.3728 MHz, the difference in clock periods ($135.6 \text{ ns} - 125.0 \text{ ns} = 10.6 \text{ ns}$) must be added to the value given in the Z8 product specifications. Hence, the delay time for $T_{dA}(\overline{AS})$ with a 7.3728 MHz clock is 40.6 ns ($30 \text{ ns} + 10.6 \text{ ns} = 40.6 \text{ ns}$). The \overline{AS} signal has a guaranteed minimum width of 70.6 ns at 7.3728 MHz. The Z8 guarantees that addresses will be stable 40.6 ns before the rising edge of \overline{AS} . With the additional maximum delay of 22 ns for the 74LS32, the resultant signal (\overline{AS})

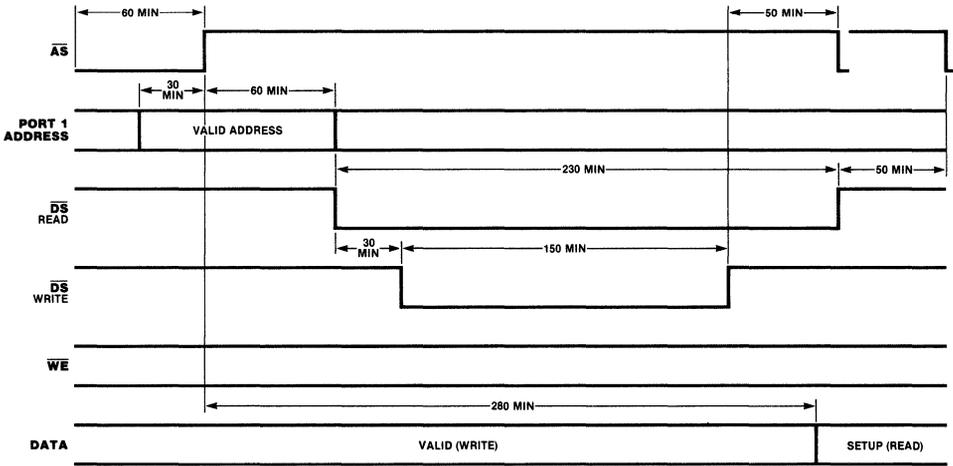


Figure 13. Z6132 Memory Timing (8.0 MHz)

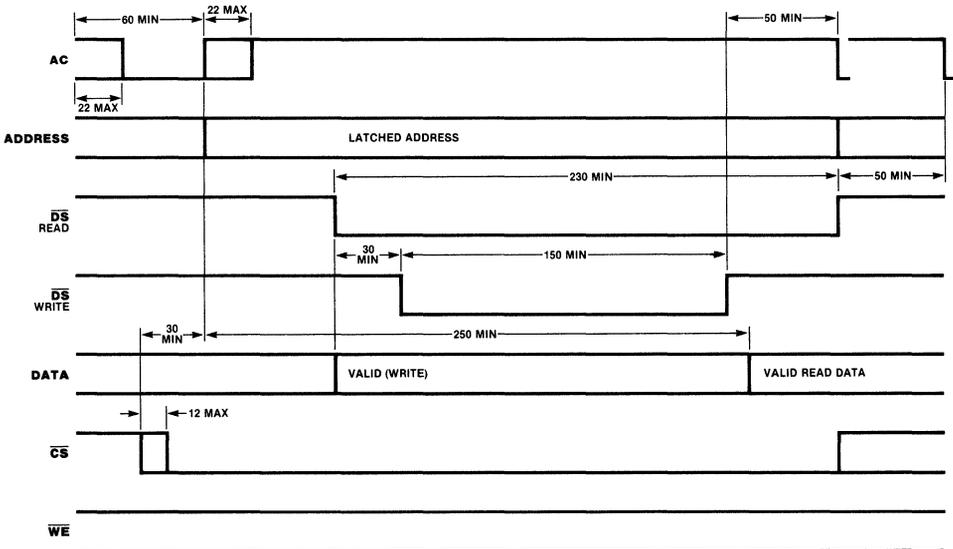


Figure 14. External Memory Timing (8.0 MHz)

Timing
(Continued)

is fed directly to the Address Clock input of the Z6132. The low-byte address encounters a maximum delay of 30 ns through the 74LS373 latch. The status signal R/W and the data bus control signal \overline{DS} are fed directly to the Z6132. The status signal R/W is available to the Z6132 40.6 ns before the rising edge of AC. The maximum delay for \overline{CS} through the 74S138 is 12 ns. This still leaves 27.4 ns setup time for \overline{CS} to AC, although 0 ns is the minimum requirement. The maximum access time for an external memory operation at 7.3728 MHz is calculated to be 322.4 ns (Figure 14). This

access time begins with the rising edge of AC and includes the data setup time to the Z8 CPU. This access time allows the use of low-speed Z6132-5 (300 ns) RAMs. For systems that require higher performance, the Z6132-4 can be used with an 8-MHz Z8 CPU. Timing for the Z8 at 8 MHz has been included in Figure 13. The maximum access time allowed for external RAM by the Z8 when operating at 8 MHz is 280 ns. The Z6132-4 has an access time of 250 ns, making it directly compatible with an 8-MHz Z8.

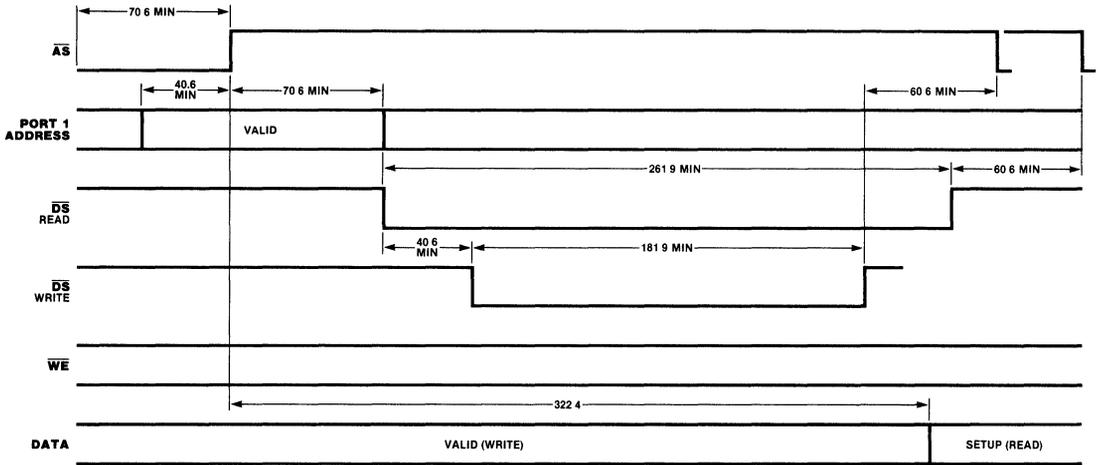


Figure 15. Z6132 Memory Timing (7.3728 MHz)

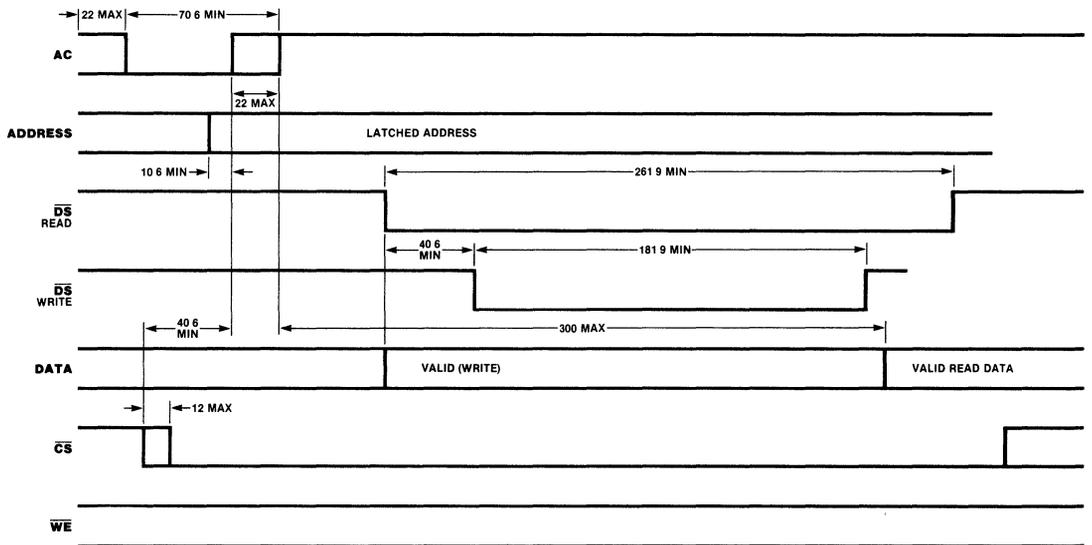


Figure 16. External Memory Timing (7.3728 MHz)

A Minimum Z8 System

Figure 17 illustrates the simplicity with which a Z8601/Z6132 system is reduced to a minimum chip count. The expansion bus of the Z8601 and the interface to the Z6132 are Z-BUS compatible. As a result, the two parts connect

directly without additional logic. As mentioned in the previous section, the access time of the Z6132-4 meets the requirements of an 8-MHz Z8.

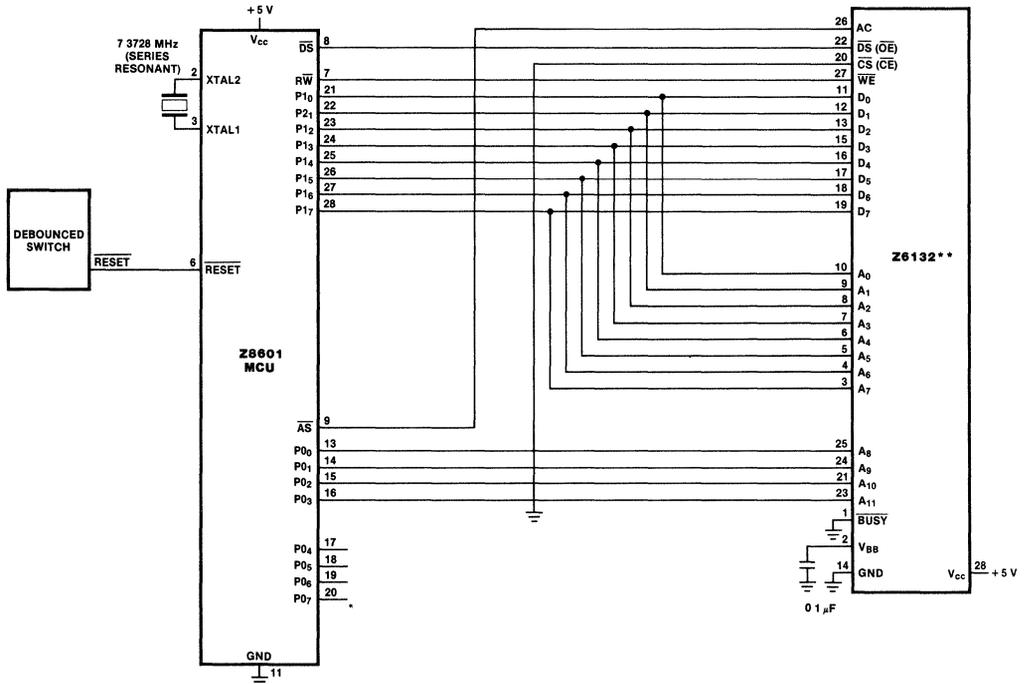


Figure 17. Z8601/Z6132 Minimum System

Summary

The Z6132 is a versatile, intelligent byte-wide RAM, which provides an attractive solution for primary buffer storage. Because the Z6132 provides two modes of self-refresh, the user can select between executing a refresh after each memory access or taking advantage of the inherent sequential access of most

memory systems. The Z6132 is an industry-standard, 28-pin DIP that conforms to the JEDEC recommended pinout and is interchangeable with 2716/2732-type EPROMs. The Z6132 is Z-BUS compatible and interfaces easily with the Z8, Z80, and Z8000 Families of microprocessors.

Zilog
Zilog
Zilog
Zilog
Zilog

Z-BUS™ Component Interconnect



Summary

March 1981

Features

- Multiplexed address/data bus shared by memory and I/O transfers.
- 16 or more memory address bits; 16-bit I/O addresses; 8 or 16 data bits.
- Supports polling and vectored or non-vectored interrupts.
- Daisy-chain interrupt structure services interrupts without a separate priority controller.
- Direct addressing of registers within a peripheral facilitates I/O programming.
- Bus signals allow asynchronous CPU and peripheral clocks.
- Daisy-chain bus-request structure supports distributed control of the bus.
- Shared resources can be managed by a general-purpose, distributed resource-request mechanism.

General Description

The Z-BUS is a high-speed parallel shared bus that links components of the Z8000 Family. It provides family members with a common communication interface that supports the following kinds of interactions:

- *Data Transfer.* Data can be moved between bus controllers (such as a CPU) and memories or peripherals.
- *Interrupts.* Interrupts can be generated by peripherals and serviced by CPUs over the bus.
- *Resource Control.* Distributed management of shared resources (including the bus itself) is supported by a daisy-chain priority mechanism.

The heart of the Z-BUS is a set of multiplexed address/data lines and the signals that control these lines. Multiplexing data and addresses onto the same lines makes more efficient use of pins and facilitates expansion of the number of data and address bits. Multiplexing also allows straightforward addressing of a peripheral's internal registers, which greatly simplifies I/O programming.

A daisy-chained priority mechanism resolves interrupt and resource requests, thus allowing distributed control of the bus and eliminating the need for separate priority controllers. The resource-control daisy chain allows wide physical separation of components.

The Z-BUS is asynchronous in the sense that peripherals do not need to be synchronized with the CPU clock. All timing information is provided by Z-BUS signals.

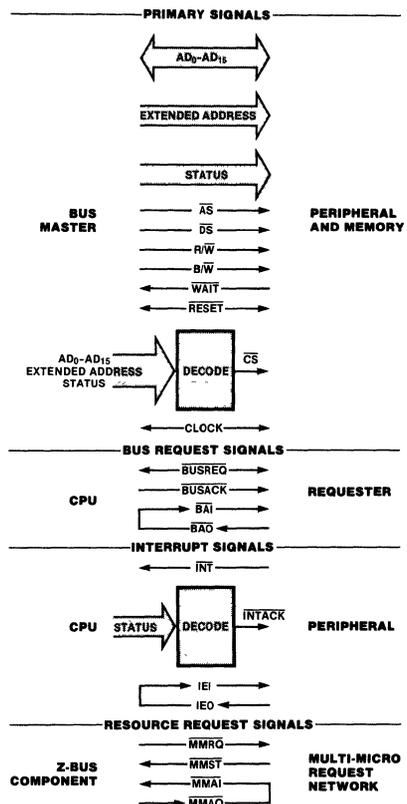


Figure 1. Z-BUS Signals

**Z-BUS
Components**

A Z-BUS component is one that uses Z-BUS signals and protocols, and meets the specified ac and dc characteristics. Most components in the Z8000 Family are Z-BUS components. The four categories of Z-BUS components are as follows:

CPUs. A Z-BUS system contains one CPU, and this CPU has default control of the bus and typically initiates most bus transactions. Besides generating bus transactions, it handles interrupt and bus-control requests. The Z8001 Segmented CPU and Z8002 Non-Segmented CPU are Z-BUS CPUs.

Peripherals. A Z-BUS peripheral is a component capable of responding to I/O transactions and generating interrupt requests. The Z8036 Counter Input/Output Circuit (Z-CIO),

Z8038 FIFO Input/Output, Interface Unit (Z-FIO), the Z8030 Serial Communication Controller (Z-SCC), the Z8090 Universal Peripheral Controller (Z-UPC), and the Z8052 CRT Controller (Z-CRT) are all Z-BUS peripherals.

Requesters. A Z-BUS requester is any component capable of requesting control of the bus and initiating transactions on the bus. A Z-BUS requester is usually also a peripheral. The Z8016 DMA Transfer Controller (Z-DTC) is a Z-BUS requester and a peripheral.

Memories. A Z-BUS memory is one that interfaces directly to the Z-BUS and is capable of fetching and storing data in response to Z-BUS memory transactions. The Z6132 Quasi-Static RAM is a Z-BUS memory.

**Other
Components**

The Z8 Microcomputer—in its micro-processor configuration—conforms to Z-BUS timing (which allows it to use Z-BUS peripherals and memories), but is missing a wait input and certain status outputs.

The Z8010 Memory Management Unit (Z-MMU) is a Z8000 CPU support component that interfaces with part of the Z-BUS on the CPU side and provides demultiplexed

addresses on the memory side.

The Z8060 First-In-First-Out Buffer (Z-FIFO) is not a Z-BUS component; rather, it is used to expand the buffer depth of the Z-FIO or to interface the I/O ports of the Z-UPC, Z-CIO, or Z-FIO to user equipment.

Z-80 Family components, while not Z-BUS compatible, are easily interfaced to Z-BUS CPUs.

Operation

Two kinds of operations can occur on the Z-BUS: transactions and requests. At any given time, one device (either the CPU or a bus requester) has control of the Z-BUS and is known as the *bus master*. A transaction is initiated by a bus master and is responded to by some other device on the bus. Four kinds of transactions occur in Z-BUS systems:

- **Memory.** Transfers 8 or 16 bits of data to or from a memory location.
- **I/O.** Transfers 8 or 16 bits of data to or from a peripheral.
- **Interrupt Acknowledge.** Acknowledges an interrupt and transfers an identification/status vector from the interrupting peripheral.
- **Null.** Does not transfer data. Typically used for refreshing memory.

Only one transaction can proceed on the bus

at a time, and it must be initiated by the bus master. A request, however, may be initiated by a component that does not have control of the bus. There are three kinds of requests:

- **Interrupt.** Requests the attention of the Z-BUS CPU.
- **Bus.** Requests control of the Z-BUS to initiate transactions.
- **Resource.** Requests control of a particular resource.

When a request is made, it is answered according to its type: for interrupt requests an interrupt-acknowledge transaction is initiated; for bus and resource requests an acknowledge signal is sent. In all cases a daisy-chain priority mechanism provides arbitration between simultaneous requests.

Signal Lines

The Z-BUS consists of a set of common signal lines that interconnect bus components (Figure 1). The signals on these lines can be grouped into four categories, depending on how they are used in transactions and requests.

Primary Signals. These signals provide timing, control, and data transfer for Z-BUS transactions.

AD_0 - AD_{15} . Address/Data (active High). These multiplexed data and address lines carry I/O addresses, memory addresses, and data during Z-BUS transactions. A Z-BUS may have 8 or 16 bits of data depending on the type of CPU. In the case of an 8-bit Z-BUS, data is transferred on AD_0 - AD_7 .

Extended Address. (active High). These lines extend AD_0 - AD_{15} to support memory addresses greater than 16 bits. The number of lines and the type of address information carried is dependent on the CPU.

Status. (active High). These lines designate the kind of transaction occurring on the bus and certain additional information about the transaction (such as program or data memory access or System versus Normal Mode).

\overline{AS} . Address Strobe (active Low). The rising edge of \overline{AS} indicates the beginning of a transaction and that the Address, Status, R/\overline{W} , and B/\overline{W} signals are valid.

\overline{DS} . Data Strobe (active Low). \overline{DS} provides timing for data movement to or from the bus master.

R/\overline{W} . Read/Write (Low = write). This signal determines the direction of data transfer for memory or I/O transactions.

B/\overline{W} . Byte/Word (Low = word). This signal indicates whether a byte or word of data is to

be transmitted on a 16-bit bus. This signal is not present on an 8-bit bus.

\overline{WAIT} . (active Low). A Low on this line indicates that the responding device needs more time to complete a transaction.

\overline{RESET} . (active Low). A Low on this line resets the CPU and bus users. Peripherals may be reset by \overline{RESET} or by holding \overline{AS} and \overline{DS} Low simultaneously.

\overline{CS} . Chip Select (active Low). Each peripheral or memory component has a \overline{CS} line that is decoded from the address and status lines. A Low on this line indicates that the peripheral or memory component is being addressed by a transaction. The Chip Select information is latched on the rising edge of \overline{AS} .

CLOCK. This signal provides basic timing for bus transactions. Bus masters must provide all signals synchronously to the clock. Peripherals and memories do not need to be synchronized to the clock.

Bus Request Signals. These signals make bus requests and establish which component should obtain control of the bus.

\overline{BUSREQ} . Bus Request (active Low). This line is driven by all bus requesters. A Low indicates that a bus requester has or is trying to obtain control of the bus.

\overline{BUSACK} . Bus Acknowledge (active Low). A Low on this line indicates that the Z-BUS CPU has relinquished control of the bus in response to a bus request.

\overline{BAI} , \overline{BAO} . Bus Acknowledge In, Bus Acknowledge Out (active Low). These signals form the bus-request daisy chain.

Z-BUS Connections	Signal	CPU	Requester	Peripheral	Memory
	AD ₀ -AD ₁₅	Bidirectional ² 3-state	Bidirectional ² 3-state	Bidirectional ¹ 3-state	Bidirectional ² 3-state
	Extended Address ⁸	Output 3-state	Output 3-state	□	Input
	Status	Output 3-state	Output 3-state	Input ¹⁰	□
	R/W	Output 3-state	Output 3-state	Input	Input
	B/W ⁹	Output	Output	Input ³	Input
	WAIT	Input	Input	Output ⁸ Open Drain	Output ⁸ Open Drain
	AS	Output 3-state	Output 3-state	Input	Input
	DS	Output 3-state	Output 3-state	Input	Input
	CS ⁴	□	□	Input	Input
	RESET	Input	Input ¹³	Input ⁵	□
	CLOCK ¹⁴	Input	Input	Input ⁸	Input ⁸
	BUSREQ	Input	Bidirectional Open Drain	□	□
	BUSACK	Output	□	□	□
	BAI ⁷	□	Input	□	□
	BAO ⁷	□	Output	□	□
	INT	Input	□	Output Open Drain	□
	INTACK ⁶	□	□	Input ¹¹	□
	IEI ⁷	□	□	Input	□
	IEO ⁷	□	□	Output	□
	MMRQ ¹²	Output Open Drain			
	MMST ¹²	Input			
	MMAI ^{7, 12}	Input			
	MMAO ^{7, 12}	Output			

1. Only AD₀-AD₇, unless peripheral is 16-Bit.

2. For an 8-bit bus, only AD₀-AD₇ are bidirectional.

3. Only for a 16-bit peripheral.

4. Derived signal, one for each peripheral or memory; decoded from status and address lines.

5. Optional—peripherals are typically reset by AS and DS being Low simultaneously; however, they can have a reset input.

6. Derived signal; decoded from status lines.

7. Daisy-chain lines.

8. Optional signal(s).

9. For 16-bit data bus only.

10. Optional—usually only input on peripherals that are also requesters.

11. May be omitted if peripheral inputs status lines.

12. Optional signal; any component may attach to the resource request lines.

13. Optional signal; a bus requester may also be reset by AS and DS going Low and BAI being High simultaneously.

14. This signal is optional if there are no requesters on the bus. CPU timing can be provided by alternate means such as crystal oscillator inputs.

□ No Connection

Table 1. Z-BUS Component Connections to Signal Lines. This table shows how the various Z-BUS components attach to each signal line. When a device is both a bus requester and a

peripheral, the attributes in both columns of the table should be combined (e.g., input combined with output and 3-state becomes bidirectional and 3-state.)

Signal Lines

(Continued)

Interrupt Signals. These signals are used for interrupt requests and for determining which interrupting component is to respond to an acknowledge. To support more than one type of interrupt, the lines carrying these signals can be replicated. (The Z8000 CPU supports three types of interrupts: non-maskable, vectored, and non-vectored.)

***INT.* Interrupt (active Low).** This signal can be driven by any peripheral capable of generating an interrupt. A Low on *INT* indicates that an interrupt request is being made.

***INTACK.* Interrupt Acknowledge (active Low).** This signal is decoded from the status lines. A Low indicates an interrupt acknowledge transaction is in progress. This signal is latched by the peripheral on the rising edge of *AS*.

***IEI, IEO.* Interrupt Enable In, Interrupt Enable Out (active High).** These signals form the interrupt daisy chain.

Resource Request Signals. These signals are used for resource requests. To manage more than one resource, the lines carrying these signals can be replicated. (The Z8000 supports one set of resource request lines.)

***MMRQ.* Multi-Micro Request (active Low).** This line is driven by any device that can use the shared resource. A Low indicates that a request for the resource has been made or granted.

***MMST.* Multi-Micro Status (active Low).** This pin allows a device to observe the value of the *MMRQ* line. An input pin other than *MMRQ* facilitates the use of line drivers for *MMRQ*.

***MMAI, MMAO.* Multi-Micro Acknowledge In, Multi-Micro Acknowledge Out (active Low).** These lines form the resource-request daisy chain.

Transactions

All transactions start with Address Strobe being driven Low and then raised High by the bus master (Figure 2). The Status lines are valid on the rising edge of Address Strobe and indicate the type of transactions being initiated. If the transaction requires an address, it must also be valid on the rising edge of Address Strobe.

For all transactions except null transactions (which do nothing beyond this point), data is then transferred to or from the bus master. The bus master uses Data Strobe to time the movement of data. For a read (R/\bar{W} = High), the

bus master makes AD_0 - AD_{15} inactive before driving Data Strobe Low so that the addressed memory or peripheral can put its data on the bus. The bus master samples this data just before raising Data Strobe High. For a write (R/\bar{W} = Low), the bus master puts the data to be written on AD_0 - AD_{15} before forcing Data Strobe Low.

For an 8-bit Z-BUS, data is transferred on AD_0 - AD_7 . Address bits may remain on AD_8 - AD_{15} while *DS* is Low.

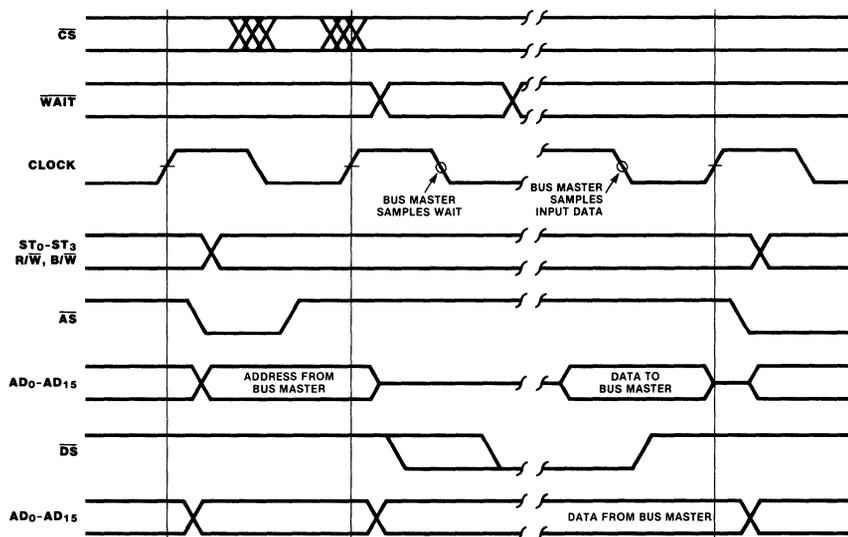


Figure 2. Typical Transaction Timing

Memory Transactions

For a memory transaction, the Status lines distinguish among various address spaces, such as program and data or system and normal, as well as indicating the type of transaction. The memory address is put on AD₀-AD₁₅ and on the extended address lines.

For a Z-BUS with 16-bit data, the memory is organized as two banks of eight bits each (Figure 3). One bank contains all the upper

bytes of all the addressable 16-bit words. The other bank contains all the lower bytes. When a single byte is written (R/W = Low, B/W = High), only the bank indicated by address bit A₀ is enabled for writing.

For a Z-BUS with 8-bit data, the memory is organized as one bank which contains all bytes. This bank always inputs and outputs its data on AD₀-AD₇.

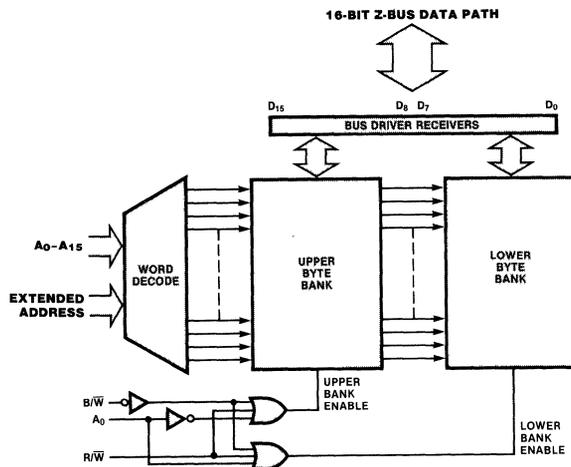


Figure 3. Byte/Word Memory Organization

I/O Transactions

I/O transactions are similar to memory transactions with two important differences. The first is that I/O transactions take an extra clock cycle to allow for slow peripheral operation. The second is that byte data (indicated by B/W High on a 16-bit bus) is always trans-

mitted on AD₀-AD₇, regardless of the I/O address. (AD₈-AD₁₅ contain arbitrary data in this case.) For an I/O transaction, the address indicates a peripheral and a particular register or function within that peripheral.

Null Transactions

The two kinds of null transactions are distinguished by the Status lines: internal operation and memory refresh. Both transactions look like a memory read transaction except that Data Strobe remains High and no data is transferred.

For an internal operation transaction, the Address lines contain arbitrary data when Address Strobe goes High. This transaction is initiated to maintain a minimum transaction rate when a bus master is doing a long internal

operation (to support memories which generate refresh cycles from Address Strobe).

For a memory refresh transaction, the Address lines contain a refresh address when Address Strobe goes High. This transaction is used to refresh a row of a dynamic memory.

Any memory or I/O transaction can be suppressed (effectively turning it into a null transaction) by keeping Data Strobe High throughout the transaction.

Interrupts

A complete interrupt cycle consists of an interrupt request followed by an interrupt-acknowledge transaction. The request, which consists of INT pulled Low by a peripheral, notifies the CPU that an interrupt is pending. The interrupt-acknowledge transaction, which is initiated by the CPU as a result of the request, performs two functions: it selects the peripheral whose interrupt is to be acknowledged, and it obtains a vector that identifies the selected device and cause of interrupt.

A peripheral can have one or more sources of interrupt. Each interrupt source has three

bits that control how it generates interrupts. These bits are an Interrupt Pending bit (IP), and Interrupt Enable bit (IE), and an Interrupt Under Service bit (IUS).

A peripheral may also have one or more vectors for identifying the source of an interrupt during an interrupt-acknowledge transaction. Each interrupt source is associated with one interrupt vector and each interrupt vector can have one or more interrupt sources associated with it. Each vector has a Vector Includes Status bit (VIS) controlling its use.

Finally, each peripheral has three bits for

Interrupts (Continued)

controlling interrupt behavior for the whole device. These are a Master Interrupt Enable bit (MIE), a Disable Lower Chain bit (DLC), and a No Vector bit (NV).

Peripherals are connected together via an interrupt daisy chain formed with their IEI and IEO pins (Figure 4). The interrupt sources within a device are similarly connected into this chain with the overall effect being a daisy chain connecting the interrupt sources. The daisy chain has two functions: during an interrupt-acknowledge transaction, it determines which interrupt source is being acknowledged; at all other times it determines which interrupt sources can initiate an interrupt request.

Figure 5 is a state diagram for interrupt processing for an interrupt source (assuming its IE bit is 1). An interrupt source with an interrupt pending (IP = 1) makes an interrupt request (by pulling $\overline{\text{INT}}$ Low) if, and only if, it is enabled (IE = 1, MIE = 1), it does not have an interrupt under service (IUS = 0), no higher priority interrupt is being serviced (IEI = High), and no interrupt-acknowledge transaction is in progress (as indicated by $\overline{\text{INTACK}}$ at the last rising edge of $\overline{\text{AS}}$). IEO is not pulled down by the interrupt source at this time; IEO continues to follow IEI until an interrupt-acknowledge transaction occurs.

Some time after $\overline{\text{INT}}$ has been pulled Low, the CPU initiates an interrupt-acknowledge

transaction (indicated by $\overline{\text{INTACK}}$ Low). Between the rising edge of $\overline{\text{AS}}$ and the falling edge of $\overline{\text{DS}}$, the IEI/IEO daisy chain settles. Any interrupt source with an interrupt pending (IP = 1, IE = 1, MIE = 1) or under service (IUS = 1) holds its IEO line Low; all other interrupt sources make IEO follow IEI. When $\overline{\text{DS}}$ falls, only the highest priority interrupt source with a pending interrupt (IP = 1) has its IEI input High, its IE bit set to 1, and its IUS bit set to 0. This is the interrupt source being acknowledged, and at this point it sets its IUS bit to 1, and, if the peripheral's NV bit is 0, identifies itself by placing the vector on $\text{AD}_0\text{-AD}_7$. If the NV bit is 1, then the peripheral's $\text{AD}_0\text{-AD}_7$ pins remain floating, thus allowing external circuitry to supply the vector. (All interrupts, including the Z8000's non-vectorized interrupt, need a vector for identifying the source of an interrupt.) If the vector's VIS bit is 1, the vector will also contain status information further identifying the source of the interrupt. If the VIS bit is 0, the vector held in the peripheral will be output without modification.

While an interrupt source has an interrupt under service (IUS = 1), it prevents all lower priority interrupt sources from requesting interrupts by forcing IEO Low. When interrupt servicing is complete, the CPU must reset the IUS bit and, in most cases, the IP bit (by means of an I/O transaction).

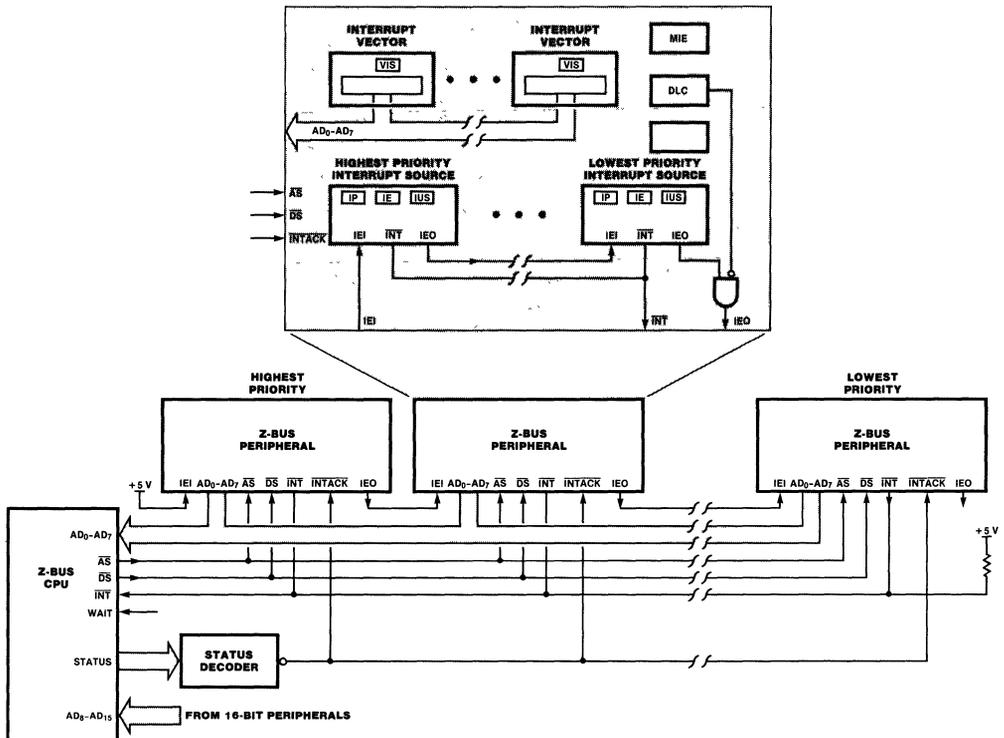


Figure 4. Interrupt Connections

Interrupts
(Continued)

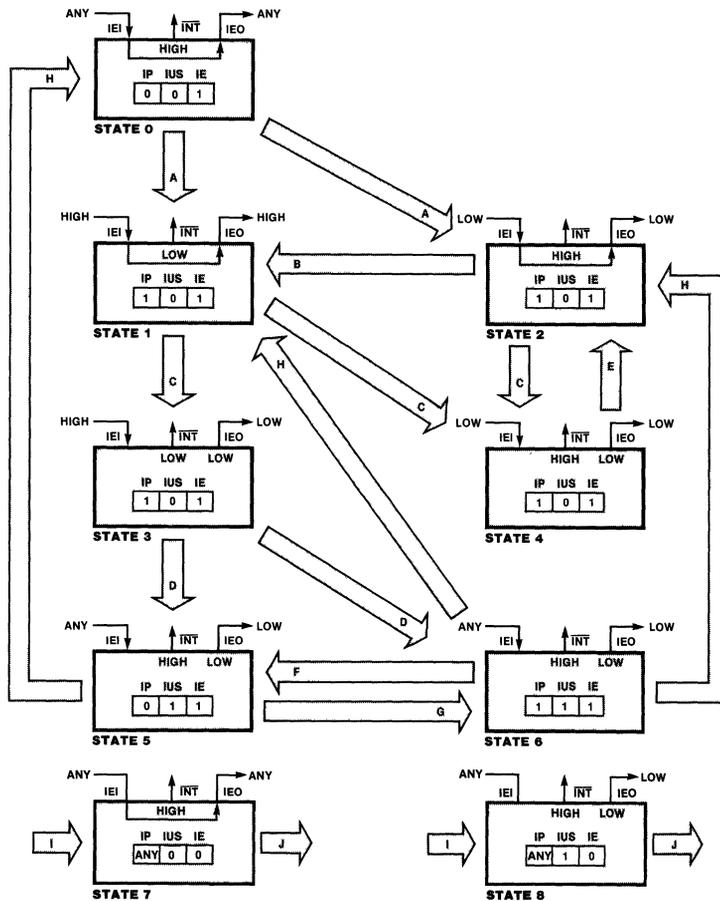


Figure 5. State Diagram for an Interrupt Source

Transition Legend

- A** The peripheral detects an interrupt condition and sets Interrupt Pending.
- B** All higher priority peripherals finish interrupt service, thus allowing IEI to go High.
- C** An interrupt-acknowledge transaction starts, and the IEI/IEO daisy chain settles.
- D** The interrupt-acknowledge transaction terminates with the peripheral selected. Interrupt Under Service (IUS) is set to 1, and Interrupt Pending (IP) may or may not be reset.
- E** The interrupt-acknowledge transaction terminates with a higher priority device having been selected.
- F** The Interrupt Pending bit in the peripheral is reset by an I/O operation.
- G** A new interrupt condition is detected by the peripheral, causing IP to be set again.
- H** Interrupt service is terminated for the peripheral by resetting IUS.
- I** IE is reset to zero, causing interrupts to be disabled.
- J** IE is set to one, re-enabling interrupts.

State Legend

- 0** No interrupts are pending or under service for this peripheral.
- 1** An interrupt is pending, and an interrupt request has been made by pulling INT Low.
- 2** An interrupt is pending, but no interrupt request has been made because a higher priority peripheral has an interrupt under service, and this has forced IEI Low.
- 3** An interrupt-acknowledge sequence is in progress, and no higher priority peripheral has a pending interrupt.
- 4** An interrupt-acknowledge sequence is in progress, but a higher priority peripheral has a pending interrupt, forcing IEI Low.
- 5** The peripheral has an interrupt under service. Service may be temporarily suspended (indicated by IEI going Low) if a higher priority device generates an interrupt.
- 6** This is the same as State 5 except that an interrupt is also pending in the peripheral.
- 7** Interrupts are disabled from this source because IE = 0.
- 8** Interrupts are disabled from this source and lower priority sources because IE = 0 and IUS = 1.

1. This diagram assumes MIE = 1. The effect of MIE = 0 is the same as that of setting IE = 0.
 2. The DLC bit does not affect the states of individual interrupt sources. Its only effect is on the IEO output of a whole peripheral.

3. Transition I to state 6 or 7 can occur from any state except 3 or 4 (which only occur during interrupt acknowledge).
 4. Transition J from state 6 or 7 can be to any state except 3 or 4, depending on the value of IEI, IP, and IUS.

Interrupts
(Continued)

A peripheral's Master Interrupt Enable bit (MIE) and Disable Lower Chain bit (DLC) can modify the behavior of the peripheral's interrupt sources in the following way: if the MIE bit is 0, the effect is as if every Interrupt Enable bit (IE) in the peripheral were 0; thus all interrupts from the peripheral are disabled. If the DLC bit is 1, the effect is to force the peripheral's IEO output Low, thus disabling all lower priority devices from initiating interrupt

Bus Requests

To generate transactions on the bus, a bus requester must gain control of the bus by making a bus request. This is done by forcing $\overline{\text{BUSREQ}}$ Low (Figure 6). A bus request can be made only if $\overline{\text{BUSREQ}}$ is initially High (and has been for two clock cycles), indicating that the bus is controlled by the CPU and no other device is requesting it.

After $\overline{\text{BUSREQ}}$ is pulled Low, the Z-BUS CPU relinquishes the bus and indicates this condition by making $\overline{\text{BUSACK}}$ Low. The Low on $\overline{\text{BUSACK}}$ is propagated through the $\overline{\text{BAI}}/\overline{\text{BAO}}$ daisy chain (Figure 6). $\overline{\text{BAI}}$ follows $\overline{\text{BAO}}$ for components not requesting the bus, and any component requesting the bus holds

requests.

Polling can be done by disabling interrupts (using MIE and DLC) and by reading peripherals to detect pending interrupts. Each Z-BUS peripheral has a single directly addressable register that can be read to determine if there is an interrupt pending in the device and, if so, what interrupt source it is from.

its $\overline{\text{BAO}}$ High, thereby locking out all lower priority users.

A bus requester gains control of the bus when its $\overline{\text{BAI}}$ input goes Low. When it is ready to relinquish the bus, it stops pulling $\overline{\text{BUSREQ}}$ Low and allows $\overline{\text{BAO}}$ to follow $\overline{\text{BAI}}$. This permits lower priority devices that made simultaneous requests to gain control of the bus. When all simultaneously requesting devices have relinquished the bus, $\overline{\text{BUSREQ}}$ goes High, returning control of the bus to the CPU and allowing other devices to request it.

The protocol to be followed in making a bus request is shown in Figure 7.

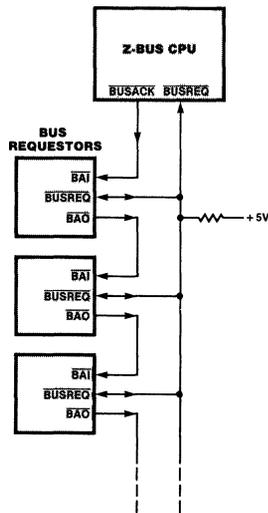


Figure 6. Bus Request Connections

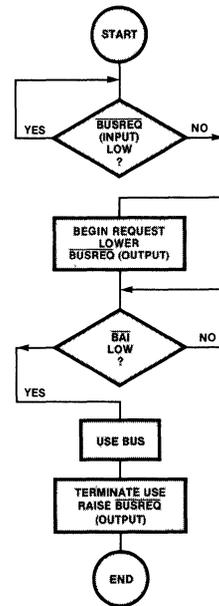


Figure 7. Bus Request Protocol

Resource Requests

Resource requests are used to obtain control of a resource that is shared between several users. The resource can be a common bus, a common memory or any other resource. The requestor can be any component capable of implementing the request protocol.

Unlike the Z-BUS itself, no component has control of a general resource by default; every device must acquire the resource before using it. All devices sharing the general resource drive the $\overline{\text{MMRQ}}$ line (Figure 8). When Low, the $\overline{\text{MMRQ}}$ line indicates that the resource is being acquired or used by some device. The $\overline{\text{MMST}}$ pin allows each device to observe the state of the $\overline{\text{MMRQ}}$ line.

When $\overline{\text{MMRQ}}$ is High, a device may initiate a resource request by pulling $\overline{\text{MMRQ}}$ Low (Figure 9). The resulting Low on $\overline{\text{MMRQ}}$ is propagated through the $\overline{\text{MMAI}}/\overline{\text{MMAO}}$ daisy chain. If a device is not requesting the resource, its $\overline{\text{MMAO}}$ output follows its $\overline{\text{MMAI}}$ input. Any device making a resource request forces its $\overline{\text{MMAO}}$ output High to deny use of the resource to lower priority devices.

A device gains control of the resource if its $\overline{\text{MMAI}}$ input is Low (and its $\overline{\text{MMAO}}$ output is High) after a sufficient delay to let the daisy chain settle. If the device does not obtain the resource after this short delay, it must stop pulling $\overline{\text{MMRQ}}$ Low and make another request at some later time when $\overline{\text{MMRQ}}$ is again High. When a device that has gained control of a resource is finished, it releases the resource by allowing $\overline{\text{MMRQ}}$ to go High.

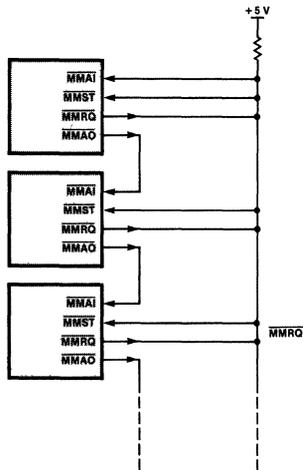


Figure 8. Resource Request Connections

The four unidirectional lines of the resource request chain allow the use of line drivers, thus facilitating connection of components separated by some distance. In the case of the Z8000 CPU, the four resource request lines may be mapped into the CPU $\overline{\text{MI}}$ and $\overline{\text{MO}}$ pins using the logic shown in Figure 10. With this configuration, the Multi-Micro Request Instruction (MREQ) performs a resource request.

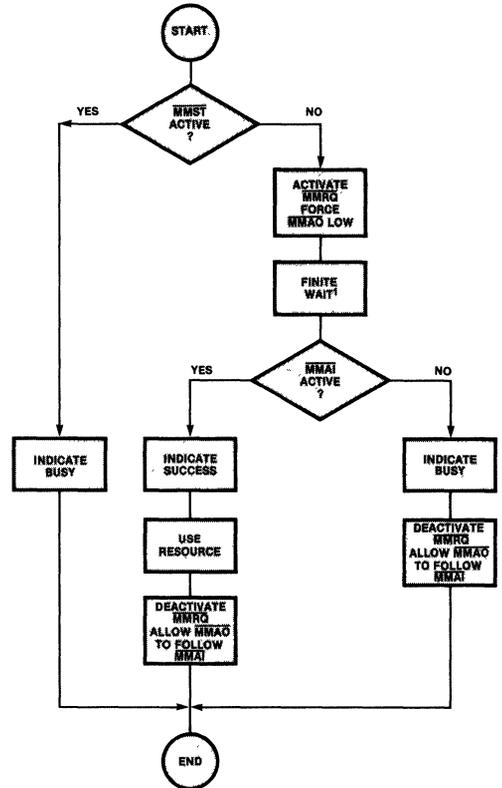


Figure 9. Resource Request Protocol

1. For any resource requested, this wait time must be less than the minimum wait time plus resource usage time of all other requesters.

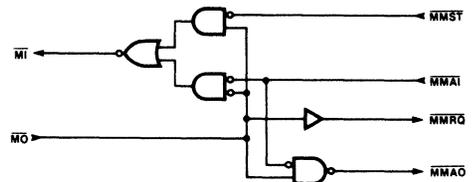
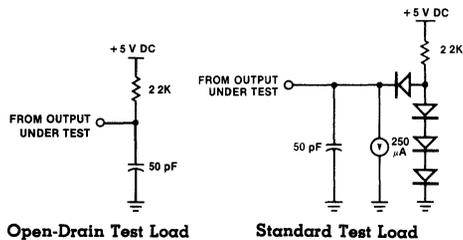


Figure 10. Bus Request Logic for Z8000

Test Conditions The timing characteristics given in this document reference 2.0 V as High and 0.8 V as Low. The following test load circuit is assumed. The effect of larger capacitive loadings can be calculated by delaying output signal transitions by 10 ns for each additional 50 pF of load up to a maximum 200 pF.



The following table states the dc characteristics for the input and output pins of Z-BUS

Symbol	Parameter	Min	Max	Unit	Test Condition
V_{IL}	Input Low Voltage	-0.3	0.8	V	
V_{IH}	Input High Voltage	2.0	$V_{CC} + 0.3$	V	
$V_{IHRESET}$	Input High Voltage on <u>RESET</u> pin	2.4	V_{CC} to 0.3	V	
V_{OL}	Output Low Voltage		0.4	V	$I_{OL} = 2.0mA$
V_{OH}	Output High Voltage	2.4		V	$I_{OH} = 250\mu A$
I_{IL}	Input Leakage Current	-10	+10	μA	$V_{IN} = 0.4$ to 2.4 V
I_{OL}	3-State Output Leakage Current in Float	-10	+10	μA	$V_{OUT} = 0.4$ to 2.4 V

components. All voltages are relative to ground.

Capacitance The following table gives maximum pin capacitance for Z-BUS components. Capacitance is specified at a frequency of 1 MHz over the temperature range of the component. Unused pins are returned to ground.

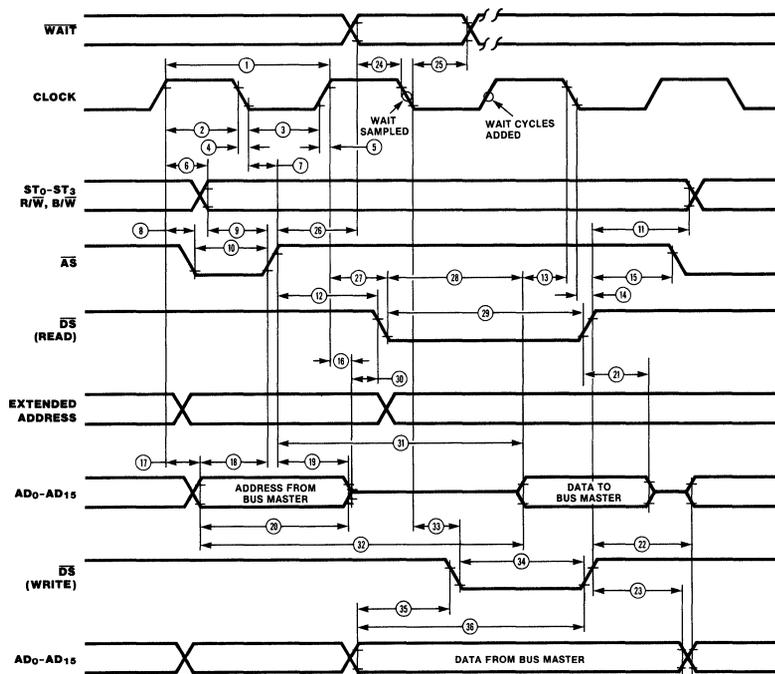
Symbol	Parameter	Max (pF)
C_{IN}	Input Capacitance	10
C_{OUT}	Output Capacitance	15
$C_{I/O}$	Bidirectional Capacitance	15

Timing Diagrams The following diagrams and tables give the timing for each kind of transaction (except null transactions). Timings are given separately for bus masters and for peripherals and memories and are intended to give the minimum timing requirements which a Z-BUS component must meet. An individual component will have more detailed and sometimes more stringent timing specifications. The differences between bus master timing and peripheral and memory timing allow for buffer and decoding circuit

delays and for signal skew. The timing given for memories is a constraint on bus-compatible memories (like the Z6132 Quasi-Static RAM) and is not intended to constrain memory subsystems constructed from conventional components.

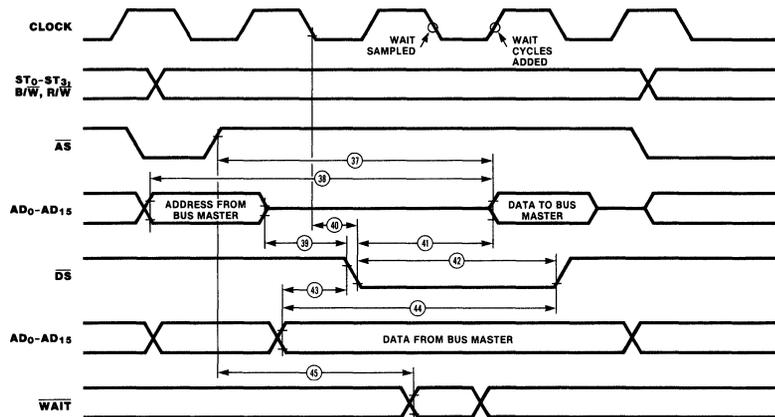
Besides these timings, there is a requirement that at least 128 transactions be initiated in any 2 ms period. This accommodates memories that generate refresh cycles from Address Strobe.

Bus Master Timing

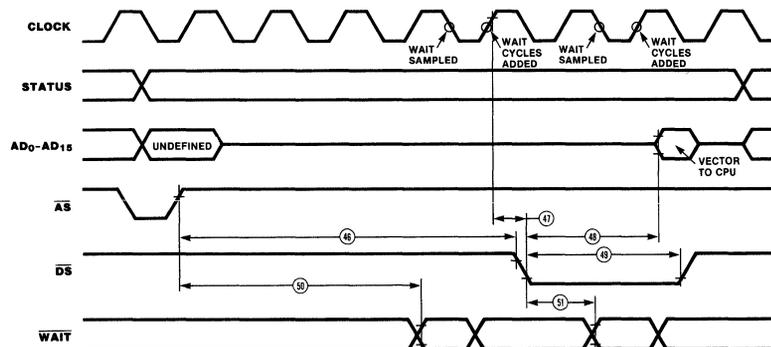


Parameters 1-25 are common to all transactions.

I/O Transaction Timing



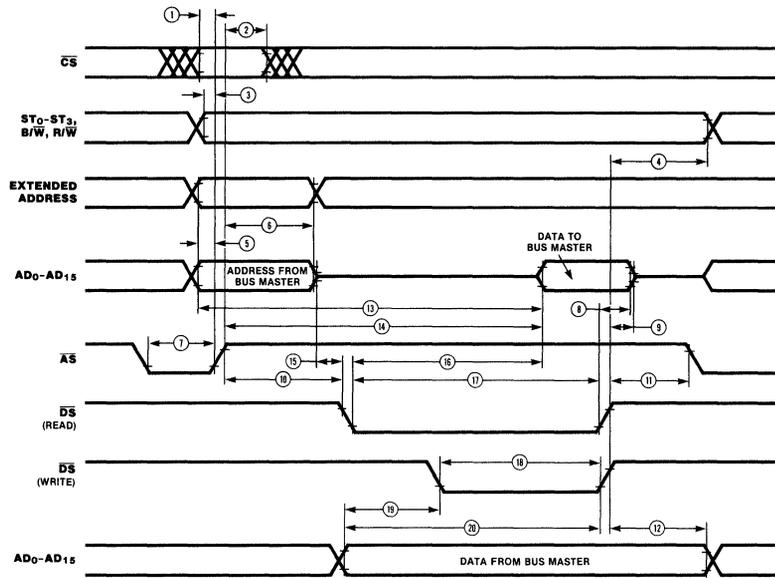
Interrupt Acknowledge Timing



Bus Master Timing Parameters	Number	Symbol	Parameter	Min (ns)	Max (ns)	Notes
All Transactions						
	1	TpC	Clock Period	250	2000	
	2	TwCh	Clock High Width	105	1895	
	3	TwCl	Clock Low Width	105	1895	
	4	TfC	Clock Fall Time		20	
	5	TrC	Clock Rise Time		20	
	6	TdC(S)	Clock ↑ To Status Valid Delay		100	
	7	TdC(ASr)	Clock ↓ To \overline{AS} ↑ Delay		90	
	8	TdC(ASf)	Clock ↑ To \overline{AS} ↓ Delay		80	
	9	TdS(AS)	Status Valid To \overline{AS} ↑ Delay	50		
	10	TwAS	\overline{AS} Low Width	80		
	11	TdDS(S)	\overline{DS} ↑ To Status Not Valid Delay	80		
	12	TdAS(DS)	\overline{AS} ↑ To \overline{DS} ↓ Delay	70	2095	3
	13	TsDR(C)	Read Data To Clock ↓ Setup Time	50		
	14	TdC(DS)	Clock ↓ To \overline{DS} ↑ Delay		70	
	15	TdDS(AS)	\overline{DS} ↑ To \overline{AS} ↓ Delay	70		
	16	TdC(Az)	Clock ↑ To Address Float Delay		65	
	17	TdC(A)	Clock ↑ To Address Valid Delay		90	
	18	TdA(AS)	Address Valid To \overline{AS} ↑ Delay	50		1
	19	TdAS(A)	\overline{AS} ↑ To Address Not Valid Delay	60		1
	20	TwA	Address Valid Width	150		
	21	ThDR(DS)	Read Data To \overline{DS} ↑ Hold Time	0		
	22	TdDS(A)	\overline{DS} ↑ To Address Active Delay	80		
	23	TdDS(DW)	\overline{DS} ↑ To Write Data Not Valid Delay	80		
	24	TsW(C)	\overline{WAIT} To Clock ↓ Setup Time	50		2,5
	25	ThW(C)	\overline{WAIT} To Clock ↓ Hold Time	0		2,5
Memory Transactions						
	26	TdAS(W)	\overline{AS} ↑ To \overline{WAIT} Required Valid		90	
	27	TdC(DSR)	Clock ↓ To \overline{DS} (Read) ↓ Delay		120	
	28	TdDSR(DR)	\overline{DS} (Read) ↑ To Read Data Required Valid		185	
	29	TwDSR	\overline{DS} (Read) Low Width		250	
	30	TdAz(DSR)	Address Float to \overline{DS} (Read) ↓ Delay	0		
	31	TdAS(DR)	\overline{AS} ↑ To Read Data Required Valid		320	
	32	TdA(DR)	Address Valid To Read Data Required Valid		400	
	33	TdC(DSW)	Clock ↓ To \overline{DS} (Write) ↓ Delay		95	
	34	TwDSW	\overline{DS} (Write) Low Width	160		
	35	TdDW(DSWf)	Write Data Valid To \overline{DS} (Write) ↓ Delay	50		
	36	TdDW(DSWr)	Write Data Valid To \overline{DS} (Write) ↑ Delay	230		
I/O Transactions						
	37	TdAS(DR)	\overline{AS} ↑ To Read Data Required Valid		570	
	38	TdA(DR)	Address Valid To Read Data Required Valid		650	
	39	TdAz(DSI)	Address Float To \overline{DS} (I/O) ↓	0		
	40	TdC(DSI)	Clock ↓ To \overline{DS} (I/O) ↓		120	
	41	TdDSI(DR)	\overline{DS} (I/O) ↑ To Read Data Required Valid		320	
	42	TwDSI	\overline{DS} (I/O) Low Width	400		
	43	TdDW(DSIf)	Write Data To \overline{DS} (I/O) ↓ Delay	50		
	44	TdDW(DSIr)	Write Data To \overline{DS} (I/O) ↑ Delay	480		
	45	TdAS(W)	\overline{AS} ↑ To \overline{WAIT} Required Valid		340	
Interrupt-Acknowledge Transactions						
	46	TdAS(DSA)	\overline{AS} ↑ To \overline{DS} (Acknowledge) ↓ Delay	960		
	47	TdC(DSA)	Clock ↑ To \overline{DS} (Acknowledge) ↓ Delay		120	
	48	TdDSA(DR)	\overline{DS} (Acknowledge) ↑ To Read Data Required Valid		420	
	49	TwDSA	\overline{DS} (Acknowledge) Low Width	485		
	50	TdAS(W)	\overline{AS} ↑ To Wait Required Valid		840	
	51	TdDSA(W)	\overline{DS} (Acknowledge) ↓ To Wait Required Valid		130	

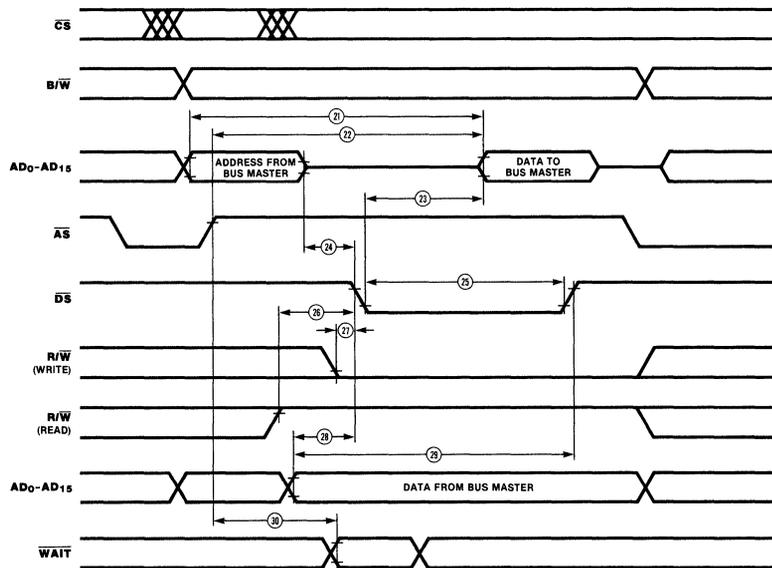
- Timing for extended addresses is CPU dependent; however, extended addresses must be valid at least as soon as addresses are valid on AD₀-AD₁₅ and must remain valid at least as long as addresses are valid on AD₀-AD₁₅.
- The exact clock cycle that wait is sampled on depends on the type of transaction; however, wait always has the given setup and hold times to the clock.
- The maximum value for TdAS(DS) does not apply to Interrupt-Acknowledge Transactions.
- Except where otherwise stated, maximum rise and fall times for inputs are 200 ns.
- The setup and hold times for \overline{WAIT} to the clock must be met. If \overline{WAIT} is generated asynchronously to the clock, it must be synchronized before input to a bus master.

Memory and Peripheral Timing

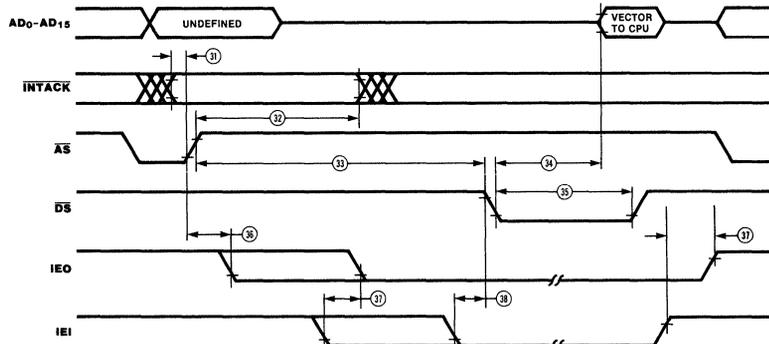


Parameters 1-12 are common to all transactions.

I/O Transaction Timing



Interrupt Acknowledge Timing



Memory and Peripheral Timing Parameters	Number	Symbol	Parameter	Min (ns)	Max (ns)	Notes
All Transactions						
	1	TsCS(AS)	\overline{CS} To \overline{AS} ↑ Setup Time	0		1
	2	ThCS(AS)	\overline{CS} To \overline{AS} ↑ Hold Time	60		1
	3	TsS(AS)	Status To \overline{AS} ↑ Setup Time	20		2
	4	ThS(DS)	Status To \overline{DS} ↑ Hold Time	60		
	5	TsA(AS)	Address To \overline{AS} ↑ Setup Time	10		1
	6	ThA(AS)	Address To \overline{AS} ↑ Hold Time	50		1
	7	TwAS	\overline{AS} Low Width	70		
	8	TdDS(DR)	\overline{DS} ↑ To Read Data Not Valid Delay	0		
	9	TdDS(DRz)	\overline{DS} ↑ To Read Data Float Delay		70	
	10	TdAS(DS)	\overline{AS} ↑ To \overline{DS} ↓ Delay	60	2095	5
	11	TdDS(AS)	\overline{DS} ↑ To \overline{AS} ↓ Delay	50		
	12	ThDW(DS)	Write Data To \overline{DS} ↑ Hold Time	30		1
Memory Transactions						
	13	TdA(DR)	Address Required Valid To Read Data Valid Delay		340	
	14	TdAS(DR)	\overline{AS} ↑ To Read Data Valid Delay		230	
	15	TdAz(DSR)	Address Float To \overline{DS} (Read) ↓ Delay	0		
	16	TdDSR(DR)	\overline{DS} (Read) ↓ To Read Data Valid Delay		95	
	17	TwDSR	\overline{DS} (Read) Low Width	240		
	18	TwDSW	\overline{DS} (Write) Low Width	150		
	19	TsDW(DSWf)	Write Data To \overline{DS} (Write) ↓ Setup Time	30		
	20	TsDW(DSWr)	Write Data To \overline{DS} (Write) ↑ Setup Time	210		
I/O Transactions						
	21	TdA(DR)	Address Required Valid To Read Data Valid Delay		590	
	22	TdAS(DR)	\overline{AS} ↑ To Read Data Valid Delay		480	
	23	TdDSI(DR)	\overline{DS} (I/O) ↓ To Read Data Valid Delay		255	
	24	TdAz(DSI)	Address Float To \overline{DS} (I/O) ↓ Delay	0		
	25	TwDSI	\overline{DS} (I/O) Low Width	390		
	26	TsRWR(DSI)	R/ \overline{W} (Read) To \overline{DS} (I/O) ↓ Setup Time	100		
	27	TsRWW(DSI)	R/ \overline{W} (Write) To \overline{DS} (I/O) ↓ Setup Time	0		
	28	TsDW(DSIf)	Write Data To \overline{DS} (I/O) ↓ Setup Time	30		
	29	TsDW(DSIf)	Write Data To \overline{DS} (I/O) ↑ Setup Time	460		
	30	TdAS(W)	\overline{AS} ↑ To \overline{WAIT} Valid Delay	195		
Interrupt-Acknowledge Transactions						
	31	TsIA(AS)	\overline{INTACK} To \overline{AS} ↑ Setup Time	0		
	32	ThIA(AS)	\overline{INTACK} To \overline{AS} ↑ Hold Time	250		
	33	TdAS(DSA)	\overline{AS} ↑ To \overline{DS} (Acknowledge) ↓ Delay	940		
	34	TdDSA(DR)	\overline{DS} (Acknowledge) ↓ To Read Data Valid Delay	360		
	35	TwDSA	\overline{DS} (Acknowledge) Low Width	475		
	36	TdAS(IEO)	\overline{AS} ↓ To IEO ↓ Delay			3, 4
	37	TdIEIf(IEO)	IEI To IEO Delay			4
	38	TsIEI(DSA)	IEI To \overline{DS} (Acknowledge) ↓ Setup Time			4

1 Parameter does not apply to Interrupt-Acknowledge Transactions

2 Does not cover R/ \overline{W} for I/O Transactions

3 Applies only to a peripheral which is pulling \overline{INT} Low at the beginning of the Interrupt-Acknowledge Transaction

4 These parameters are device dependent. The parameters for the devices in any particular daisy chain must meet the following constraint: for any two peripherals in the daisy chain, TdAS(DSA) must be greater than the sum of TdAS(IEO) for the higher priority peripheral, TsIEI(DSA) for the lower priority peripheral, and TdIEIf(IEO) for each peripheral separating them in the daisy chain.

5 The maximum value for TdAS(DS) does not apply to Interrupt-Acknowledge Transactions

6 Except where stated otherwise, maximum rise and fall times for inputs are 200 ns.

Z-bus and peripheral support packages tie distributed computer systems together

To couple support circuits to Z8000 microprocessors in an organized manner, an interconnection philosophy is needed. To this end, Zilog has developed the shared Z-bus—not a device, but a concept—to allow the construction of complex configurations of peripherals with program interfaces. This article takes the reader a step beyond basic interfacing circuits (ELECTRONIC DESIGN, Oct. 25, 1979, p. 90) and introduces both the Z-bus concept and a new family of peripheral packages, designed especially for the Z8000 μ Ps. Future articles will explore Z8000 software.

The Z-bus logically and efficiently organizes interconnections and transactions between Zilog's Z8000 microprocessors and their peripherals. The signals in transactions between microprocessors and peripherals inherently provide all the necessary timing, allowing asynchronous operation, so that the peripheral devices can be independent of the processor's speed and clock frequencies. In addition, the bus has a simple scheme—the daisy chain—for establishing sequential priority, as when a common system resource must be shared by several processors and peripherals.

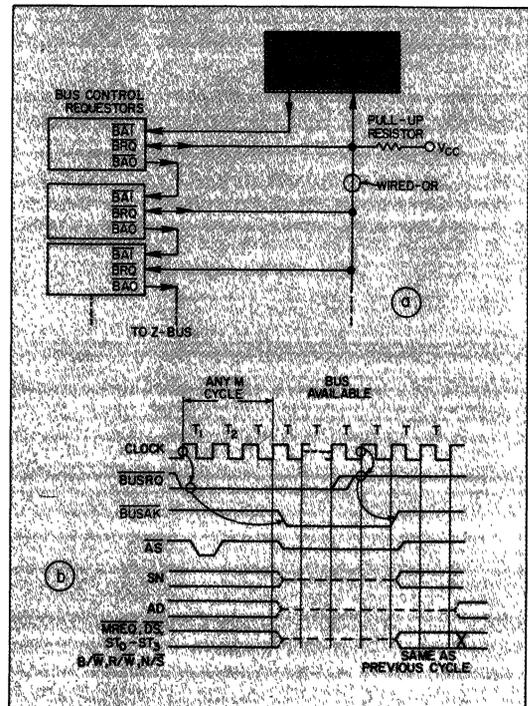
Processors and peripherals engage in five types of transactions through the Z-bus:

- Memory transfers.
- I/O transfers
- Interrupts requests, to interrupt the Z-bus processor.
- Bus requests, to gain control of the bus for both memory and I/O transfers
- Resource requests, to gain access to a general resource.

Although memory and I/O transfers are usually between the Z-bus processor and the memory or a peripheral, some Z-bus-system peripherals, such as a direct-memory-access controller can initiate transfers after making a successful bus request.

The Z-bus system depends on strobe, request and acknowledge signals to provide the timing information

between processor and peripherals (see Z-bus signal-description table). The multiplexed address/data lines, when combined with a low \overline{AS} (address strobe) signal, carry the addresses of memory or internal registers within peripheral-interface packages or peripheral devices. When combined with a low \overline{DS} (data strobe) the multiplexed address/data lines transfer data from or to the registers, depending on the state of the R/\overline{W} (Read/Write) line.



1. Bus-request signals to the master processor from all requestors are OR-wired to the BRQ line, and their BAI/BAO lines are daisy chained to provide a priority sequence (a). The daisy-chain signal is derived from the BUSACK signal delivered by the master processor (b).

Data can be formatted in 8 or 16-bit groups, with memory and I/O addresses that are 16-bits long (memory addresses in the Z8001 segmented version can be as long as 24 bits).

Peripherals get on the bus

For a peripheral to get control of the bus, the $\overline{\text{BUSRQ}}$ line of the Z-bus processor in the system must be driven low. When there are several peripherals, the easiest way to generate $\overline{\text{BUSRQ}}$ is to wire-OR all potential bus request signals together (Fig. 1a) via the Z-bus $\overline{\text{BRQ}}$ line, which then becomes $\overline{\text{BUSRQ}}$ at the processor port.

With $\overline{\text{BUSRQ}}$ low after the completion of any system cycle, the processor generates a $\overline{\text{BUSACK}}$ low output (Fig. 1b) to acknowledge the release of the bus. At this time, all the processor outputs go into a high-impedance state to avoid affecting other signals on the bus. Meanwhile, $\overline{\text{BUSACK}}$'s low propagates through a daisy-chain hookup (Fig. 1a) among the bus requestors—the low enters each unit's $\overline{\text{BAI}}$ and leaves via its $\overline{\text{BAO}}$ port. The device that requested control of the bus begins to use it; but the device's $\overline{\text{BAO}}$ remains high, preventing lower priority bus requesters from using the bus and providing a signal that identifies it as the requestor. When the device completes its use of the bus, $\overline{\text{BUSRQ}}$ returns to high, followed one cycle later by a high $\overline{\text{BUSACK}}$ and $\overline{\text{BAI}}$. This indicates that the Z-bus processor again controls the bus.

Clearly, the Z-bus processor occupies a special place on the bus, even though the processor, like any other device in the system, must wait until the bus is released before regaining control.

However, when peripherals that have intelligence and programmability approaching that of a processor share the bus, the management protocol must be more equitable than in a master-slave relationship. A protocol should be available that allows any intelligent component on the bus to seize a common resource of the system—a peripheral, memory, modem, display, etc.

An equal-opportunity protocol

Unlike the bus-request protocol, the resource-request chain is not dominated by a single system component. To acquire a resource, a component must issue a request signal, $\overline{\text{MMRQ}}$ low. All $\overline{\text{MMRQ}}$ signals for a given resource are wire-ORed to a common bus line (Fig. 2a). Nevertheless, the resource-requesting devices are daisy-chain connected, so that a low on the $\overline{\text{MMRQ}}$ line propagates through the chain—into each $\overline{\text{MMAI}}$ and out of each $\overline{\text{MMAO}}$. However, the $\overline{\text{MMAO}}$ of the requesting device remains high. Thus, the combination of $\overline{\text{MMRQ}}$ low and $\overline{\text{MMAO}}$ high in a device identifies it as the temporary controller of the resource.

Before a component makes a resource request, it first checks the $\overline{\text{MMST}}$ (resource-status) line to see if

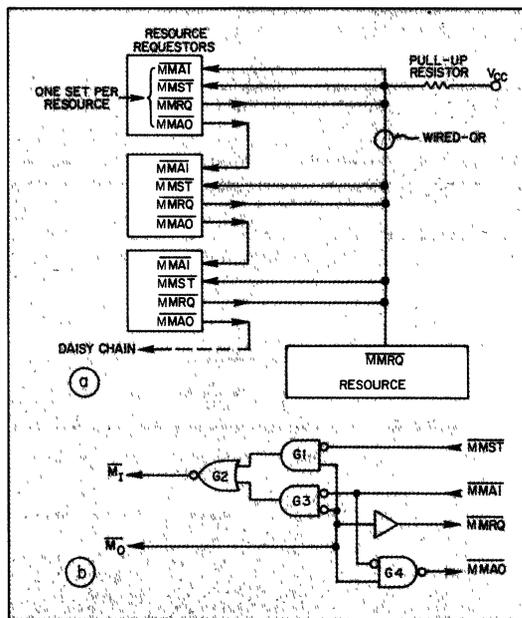
the resource is busy. A low $\overline{\text{MMST}}$ line indicates busy, and additional $\overline{\text{MMRQ}}$ s are blocked. No requestor can preempt another, but when simultaneous requests are made for the same resource, the requestor highest on the daisy chain will seize the resource first, all else being equal.

If $\overline{\text{MMST}}$ is high, however, $\overline{\text{MMRQ}}$ activates the line. After a finite delay, if $\overline{\text{MMAI}}$ also goes low, the resource has been seized successfully and the intended transaction can begin. Otherwise, the request is aborted—because another requestor higher on the daisy chain had already seized the resource. The preempted requestor may retry immediately or after some delay.

A simple logic circuit can take advantage of a requestor's low $\overline{\text{MMRQ}}$ and $\overline{\text{MMAI}}$ and its high $\overline{\text{MMAO}}$ to enforce access protection for both the resource and the requestor that has successfully seized the resource.

At this point, the designer might notice that, although four lines are used on the Z-bus to control resource requests, the Z8001/8002 processors provide just two pins M_0 and M_1 . On its Multimicro Output (M_0) pin, the μP issues a low signal to request a resource; its Multimicro Input (M_1) pin tests to determine the state of the resource.

To get onto the daisy-chain with other requestors, $\overline{\text{MMAI}}$ and $\overline{\text{MMAO}}$ pins are also needed. A logic circuit, as in Fig. 2b, can provide the interface for the

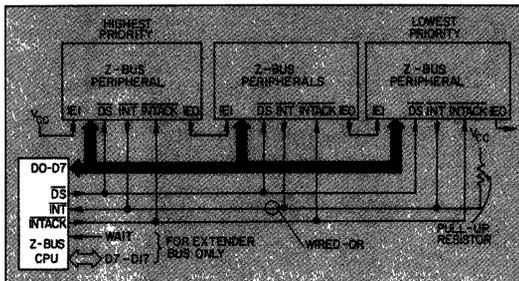


2. The resource-request chain, like the bus-request, also OR-wires the request signals, in this case $\overline{\text{MMRQ}}$, and daisy-chains for priority (a). Several gates, however, are needed to interface a Z8000, which has just two resource-request ports, M_0 and M_1 , with the daisy chain (b).

Z8001/8002 processors: \overline{MMAI} passes through gate G_4 to \overline{MMAO} as long as M_0 is high (not requesting the resource). While M_0 is high (before making a request), the state of the MMST line passes through G_1 and G_2 to M_T . With M_1 high (\overline{MMST} is not busy), M_0 can issue a request. But if another requestor higher had seized the resource first, \overline{MMAI} would be low and would pass through G_1 and G_2 to M_1 to abort the μP 's request, until it could try again.

Interrupts also are daisy chained

In the interrupt protocol (as in both the bus-request scheme and the resource-request scheme), the device's physical position in the daisy-chain—in at IEI, out at IEO—determines its priority. Also, like bus requests, interrupt requests are directed to the processor—in this case, to one of its three interrupt input ports—NMI, VI, or NVI. A separate set of interrupt-protocol signals— \overline{INT} , \overline{INTACK} , IEI and IEO—control each μP interrupt mode that is used. The peripheral \overline{INT} ports receive the same treatment as \overline{BRQ} —the \overline{INT} lines for one of a processor's interrupt modes are all wire-ORed together (Fig. 3a). The appropriate acknowledgement, decoded from the four status lines of the μP (Fig. 3b), returns via the Z-bus' \overline{INTACK} line to all the daisy-chained peripheral requestors. This procedure temporarily inhibits further interrupt requests.



(a)

Z8000 status-line codes			
ST ₃ -ST ₀	Definition	ST ₃ -ST ₀	Definition
0 0 0 0	Internal operation	1 0 0 0	Data memory request
0 0 0 1	Memory refresh	1 0 0 1	Stack memory request
0 0 1 0	I/O reference	1 0 1 0	Reserved
0 0 1 1	Special I/O reference (e.g., to an MMU)	1 0 1 1	Reserved
0 1 0 0	Segment trap acknowledge	1 1 0 0	Program reference, nth word
0 1 0 1	Nonmaskable interrupt acknowledge	1 1 0 1	Instruction fetch, first word
0 1 1 0	Nonvectored interrupt acknowledge	1 1 1 0	Reserved
0 1 1 1	Vectored interrupt acknowledge	1 1 1 1	Reserved

(b)

3. Microprocessor interrupts from peripherals also use an or-wired line (\overline{INT}) for initiating the sequence and a daisy chain to establish peripheral priorities for a given type

of interrupt (a). The properly decoded status line of that interrupt from the processor then becomes the \overline{INTACK} signal on the line (b).

Although more than one peripheral may have issued an interrupt request simultaneously, the request highest on the daisy chain prevails: Its IEO remains low, aborting any other interrupt requests further down the chain, until IEO drops low. Three Wait cycles occur after the leading edge of \overline{INTACK} to allow the daisy chain to settle (or more, if a peripheral device asks for it via the \overline{WAIT} line). Then, a \overline{DS} from the μP stimulates the interrupting peripheral to place its data on the bus. \overline{INTACK} returns high two (or more) Wait cycles later, after completion of the transaction for which the interrupt was initiated.

After \overline{INTACK} returns high, any requestor on the daisy chain can issue an interrupt; lower-priority devices are locked out until higher priority interrupts have been serviced.

I/O is main transaction

The main purpose of an interrupt request is to perform a transfer of information in or out of the processor. This I/O transaction is distinguished from every other by the μP 's status-lines code 0010, designated I/O Reference.

The bus R/\overline{W} line determines the direction in which the information flows: The processor reads from the requestor device when R/\overline{W} is high or writes into the device when R/\overline{W} is low. Information flows via the AD_0 to AD_{15} lines of the μP .

When \overline{AS} is low, the information being transferred is addresses; when \overline{DS} is low, the information is data. Word or byte formats are identified by the B/\overline{W} line—word format, when low—allowing 16 or 8-bit data elements (Fig. 4).

This early-status information, which defines the transaction ahead of the actual process, allows the enabling of bidirectional drivers and other interface hardware elements. The enabling action is a distinct benefit, which simplifies interfacing peripherals.

Indeed, the Z8000 processors distinguish between I/O-transaction and memory/processor-interchange, modes only by using different status-line codes; otherwise, the two modes work almost the same way. The address/data bus, strobe lines \overline{AS} and \overline{DS} , and the R/\overline{W} , B/\overline{W} , and N/\overline{S} lines are shared by both I/O and memory transactions; therefore the interface buffers can be shared by substantially fewer processor pins.

One difference in the modes—an extended address capability to 23 bits—applies only to memory, when the segmented Z8001 version of the processor is used.

Memory is organized into two 8-bit-wide banks. One bank contains the most-significant bytes of the addressable words; the other contains the lower bytes. The banks can be activated together or separately by a B/\overline{W} low signal (Fig. 4).

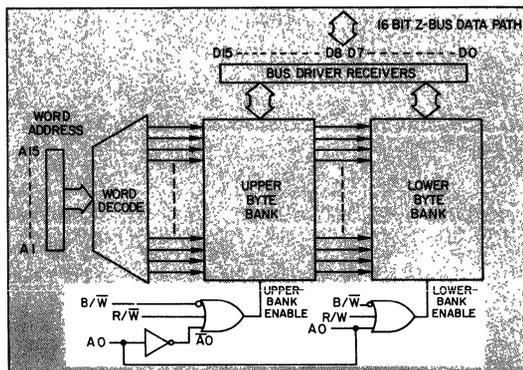
Memory and I/O functions must then be done sequentially, but the high-speed of the processor transactions can handle most applications adequately. If necessary, the $WAIT$ line can be called upon to extend a transaction for I/O and memory, because the device (or memory) is not ready or cannot work fast enough to keep up with the processor.

Help for the busy processor

When the processor gets too busy to handle all its peripherals efficiently, then Zilog's Universal Peripheral Controller (Z-UPC), one of several support packages that will soon be available, can step in and help out (Fig. 5). With pin functions \overline{AS} and \overline{DS} , R/\overline{W} and $WAIT$, IEI , and IEO , \overline{INT} and \overline{INTACK} , Z-UPC can plug right into the Z-bus and serve as a complete slave microcomputer for distributed processing. It can:

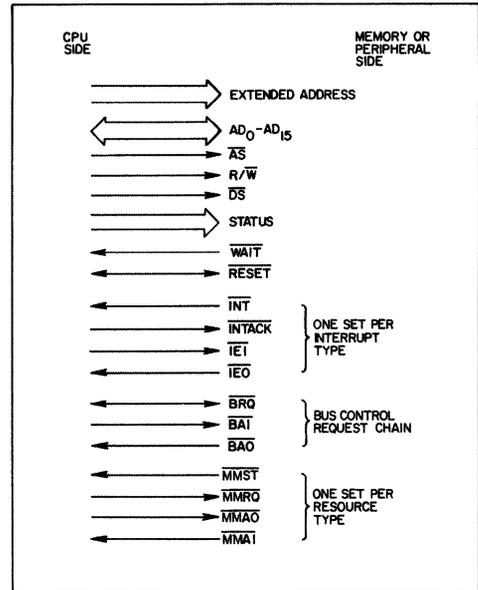
- Control peripheral devices with internal ROM or RAM instructions.
- Manipulate data arithmetically or format buffer data in internal registers.

Based on the Z8 microprocessor architecture and instruction set, the Z-UPC is an intelligent device that can unburden the main processor and greatly increase



4. The byte/word and read/write organization of words is handled almost the same way for I/O transactions between peripherals and processor as for I/O transactions between memory and processors.

Z-Bus signal descriptions



- AD_0-AD_{15} **Address/Data Lines.** The multiplexed address/data lines are used for both I/O and memory transfers.
- AS **Address Strobe.** The rising edge of AS indicates addresses are valid.
- \overline{BRQ} , \overline{BAI} , \overline{BAO} **Bus Request, Bus Acknowledge Input, Bus Acknowledge Output.** Other Z-Bus masters, such as the Z-DMA, use this bus control request chain to take control from the CPU.
- \overline{DS} **Data Strobe.** \overline{DS} times the data in and out of the CPU.
- EXTENDED ADDRESSES The number, type and nature of these lines depend on the CPU used.
- \overline{INT} , \overline{INTACK} , IEI , IEO **Interrupt, Interrupt Acknowledge, Interrupt Enable Input, Interrupt Enable Output.** This set of lines is used for interrupt daisy control and the interrupt daisy chain for each type of interrupt.
- \overline{RESET} **Reset.** A Low on this line resets the system.
- R/\overline{W} **Read/Write.** R/\overline{W} indicates the CPU is reading or writing.
- Status Lines The status lines distinguish the different kinds of bus transactions, such as I/O or memory.
- $WAIT$ **Wait.** This line indicates to the bus controller that the responder is not ready for data transfer.
- \overline{MMST} , \overline{MMRO} , \overline{MMAO} , \overline{MMAI} **Multi-Micro Status, Multi-Micro Request, Multi-Micro Acknowledge Output, Multi-Micro Acknowledge Input.** This resource-request chain controls access to common resources.

overall system efficiency and speed. It generates almost any control signal that a peripheral device might need. Operating on the same 4-MHz clock as the Z8000 μ Ps, it executes instructions in an average of just 2 μ s.

Not only speed, but flexibility is attained. An extensive register file of 256 byte-registers, organized into 16 groups of 16 working registers each make the Z-UPC very versatile. Short-format instructions expedite the access to any group. The file includes 234 general-purpose, 19 status-and-control (including two 16-bit counter/timer) and three I/O-port registers. Add six levels of priority interrupts and the Z-UPC is indeed a flexible support package.

Any general-purpose register can be used as an accumulator, address pointer, index register or stack for the the Z-UPC's program. All unused general-purpose registers can then act as data buffers between the master processor and the peripheral device. In addition, communications between the master processor and the Z-UPC takes place via one of the groups of 16 registers, which are accessed directly by the master processor over the Z-bus Address/Data (AD_x) lines.

Examining the I/O ports

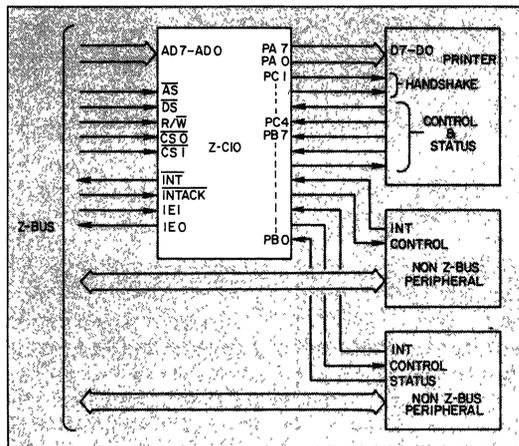
The Z-UPC's three I/O ports also allow great flexibility. Two of the I/O ports are 8-bits each; the third has 8 bits for I/O that can be shared between I/O and control lines, as determined by the program. In fact, all the I/O ports can be programmed in many combinations as input, output or bidirectional lines, with or without a handshake protocol.

When its P3₀, P3₂, P3₅ and P3₇ pins are programmed as IEL/IEO, \overline{INTACK} and \overline{INT} lines, the Z-UPC fits easily

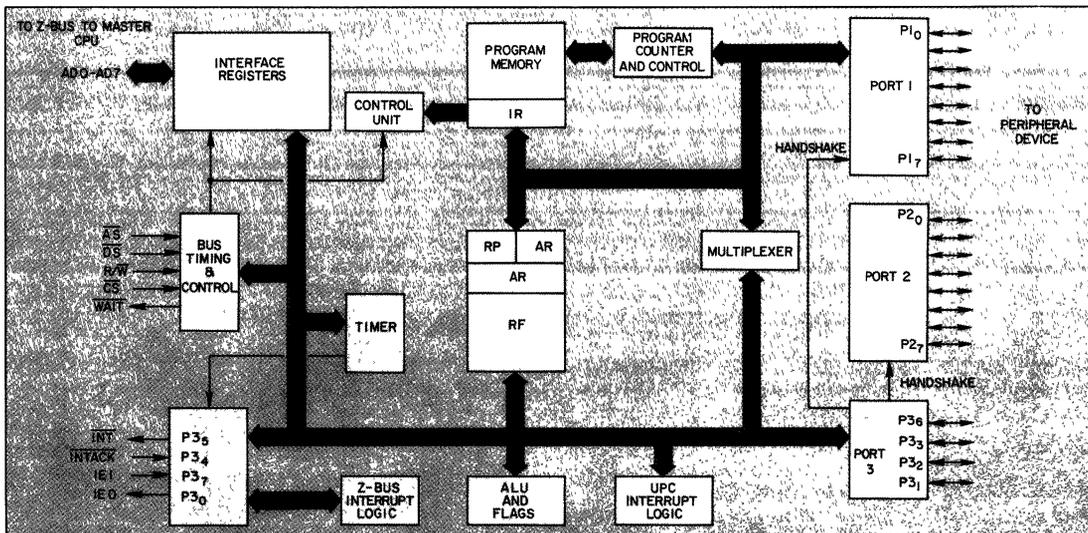
into the Z-bus's daisy-chain priority system. As an alternative, the controller can be programmed to operate with a polled system—a concept that is also compatible with the Z-bus.

Not all peripheral interfacing tasks need all the intelligence and flexibility that the Z-UPC possesses. Zilog's Counter/Timer and Parallel I/O (Z-CIO), with its two independent 8-bit bidirectional I/O ports and special-purpose 4-bit I/O port, can satisfy most ordinary needs for parallel I/O interfacing and counting/timing (Fig. 6).

Either of the Z-CIO's two identical 8-bit I/O ports can operate in a handshake-byte or bit-by-bit mode.

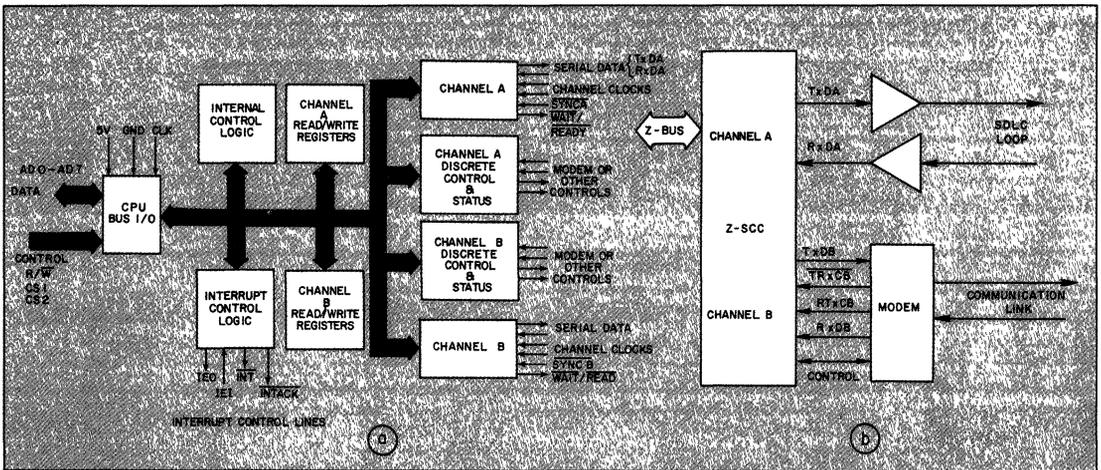


6. By taking care of parallel I/O interfacing and counting, the Z-CIO peripheral interfacing circuit chip can remove a heavy burden from the processor in a complex Z8000 system.



5. A universal peripheral controller (Z-UPC) can take a great amount of the load off a microprocessor, especially

when the processor is interfacing peripheral devices that demand a lot of detailed attention.



7. For long-distance serial communications with a processor, the Z-SCC converts parallel-to-serial data and

then serial-to-parallel for either synchronous or asynchronous data links.

In the later mode, the direction of each bit can be individually programmed. Like the universal controller, the two ports can perform in the handshake mode, as inputs, outputs or bidirectional lines; also, they can be linked into one 16-bit port. In addition, each of the 8-bit ports includes pattern-recognition logic to generate an interrupt when a specified pattern is detected.

provides the full complement of bus control signals and daisy-chain priority pins (IEI/IEO).

Four handshake protocols are available: the IEEE-488, an interlocked (with another Z-CIO or Z-UPC), a strobed and a pulsed.

Serial unit supports many protocols

Also supporting the Z-bus family is the Z-SCC, Serial Communications Controller, a peripheral-interfacing package for serial communications or data-transfer applications (for example, with disks and cassettes). The package contains two independent full-duplex channels, each with its own quartz-crystal oscillator, baud-rate generator and digital phase-locked loop for clock recovery (to 1 Mbit/s). Each channel also provides facilities for modem/control (Fig. 7a).

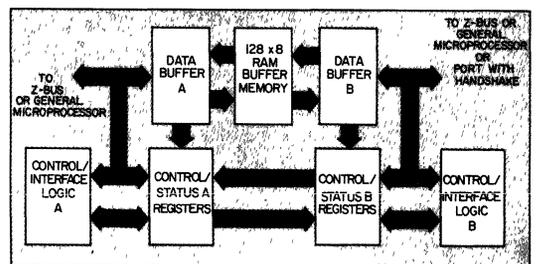
The pulsed handshake connects one of the Z-CIO's counters with logic, to interface a mechanical device such as a printer. The special-purpose 4-bit I/O port provides the handshake controls: a Wait/Request line for high-speed data transfer or general-purpose I/O. The programming and status for all the control features reside in 12 registers provided for each port.

The Z-SCC is programmable for NRZ, NRZI or FM-data encoding. A channel in an asynchronous mode can operate with 5 to 8-bit codes per character plus 1, 1-1/2 or 2 bits per character as stop bits. In addition the package provides such features as break detection and generation, and parity, overrun and framing error detection.

The Z-CIO's three, independent, identical 16-bit counter/timers (two of the counters can be programmed to form a 32-bit counter/timer) can help to control a device. Each counter/timer consists of a 16-bit register, to hold the initial value (called the Time Constant), which is loaded into the down-counter; another 16-bit register, to hold a current down-count output, when strobed; and two 8-bit registers to hold mode, control and status information.

In the synchronous mode, the Z-SCC handles such protocols as IBM Bisync or bit-oriented HDLC and

Either the counting or the timing function can be programmed for single-cycle (one-shot) or continuous operation with a pulse or square-wave output. Up to four control lines—for each counter/timer—can act as the counter input, enable input, trigger input and counter/timer output, as required.



Whether counting or parallel interfacing, the Z-CIO can substantially unburden the master processor in a computer system, especially when complex peripherals demanding high service must be handled. The Z-CIO is also fully compatible with the Z-bus and

8. The Z-FIO general-purpose bidirectional buffer can interconnect devices operating at different speeds. It is not limited to Z8000 configurations, but can handle almost any general-purpose μ P system.

SDLC with frame-level control, automatic zero-insertion and deletion, I-field residue handling, abort generation and detection, CRC generation and checking and loop-mode operation. Parity and overrun features also apply to synchronous operation

Fig. 7b shows one of the Z-SCC's channels connected as a synchronous data-link—the loop (SDLC) mode. Note the absence of clock lines. With NRZI or FM data, no clock lines are needed, since the clock can be recovered at the receive end from the bit stream by the Z-SCC's digital phase-locked loop. The other channel, via a modem under control of the Z-SCC, is shown servicing an asynchronous serial port.

Basically the Z-SCC functions as a parallel-to-serial and serial-to-parallel converter, but it does more: Its sophisticated repertoire of internal functions greatly reduces the amount of external supporting logic needed for a wide variety of serial-communications applications in distributed-processing systems.

Another great saver of external assorted logic in distributed-processor operation is the Z-FIO general-purpose bidirectional buffer.

First-in, first-out

The Z-FIO can interconnect components or sub-systems (of almost any μP including the Z8000) operating at different speeds. It can accept 128 bytes of data, which it then holds until they are called for by another device in the system. In this way, interrupt servicing time can be cut two orders of magnitude in most I/O transactions. Moreover, the capability of moving variable-sized blocks under either direct-memory access or interrupt control greatly facilitates system throughput, which is especially important with fast peripheral circuits.

The internal functions of the Z-FIO are shown in Fig. 8. Its two sets of Address/Data ports are identical except for programming. The A set (programmed by pins M_0 and M_1) and the B set (programmed by bits SL_0 and SL_1) have in common a 128×128 RAM for data storage, two 7-bit counters and several registers.

The RAM can read and write both simultaneously and independently: The A set can write a byte of data into the RAM without disturbing a simultaneous read operation at the B set. The counters address the RAM and, by means of a subtractor, determine the number of bytes remaining in the memory. This number can be read from a status register dedicated to each set.

When compared internally with the memory-status register, a programmable register generates an interrupt for starting and stopping DMA transfers. Another pair of registers permits direct communication between the ports by bypassing the main buffer memory. ■■

Zilog
Zilog
Zilog
Zilog
Zilog
Zilog

Zilog



Application note

SOFTWARE SERIES

**AN OPTIMISING DRIVER
FOR NEC SPINWRITER
AND DIABLO PRINTERS**

APPLICATION NOTE No. 101G
MARCH 1979

An Optimising Driver For NEC Spinwriter And Diablo Printers

Revision 2.1

CONTENTS

	Page
1 INTRODUCTION	6-6
2 THE DRIVER FROM A NON-TECHNICAL USER'S VIEWPOINT	6-7
2.1 Loading and initialising the driver	6-7
2.2 Using the driver	6-8
2.3 Automatic Formfeed	6-9
2.4 Use of Cut Sheets	6-10
2.5 Character Attributes	6-10
3 THE DRIVER FROM A TECHNICAL USER'S VIEWPOINT	6-12
3.1 The Interface Hardware	6-12
3.2 The Driver Software - General	6-14
3.3 The Modules	6-15
4 CONCLUSION	6-17
5 APPENDIX - MODULE LISTINGS	6-18
PRINTER.DRIVER.0	6-18
PRINTER.DRIVER.1	6-25
PRINTER.DRIVER.2	6-30
SPINWRITER	6-34
DIABLO	6-42
CALL.SYSTEM	6-49
RIO.IO.INTERFACE	6-50
SIB	6-52
PLZ.INTERFACE.MACROS	6-57

1 INTRODUCTION

This printer driver was written to suit a variety of applications, but in particular, it was prepared to be used as a companion to the Zilog text formatting utility, ZFORM.

It is also well suited to applications requiring the printing of long lines, such as are found in connection with business orientated data base management programs.

It provides compatibility with both of the most commonly used high performance, high quality printers, the NEC SPINWRITER, and the DIABLO.

The features provided by the driver ensure that the printer operates at its maximum possible speed under all circumstances. Adjacent spaces and tabs are merged into single head movements, and the printing direction is fully optimised to minimise unnecessary head travel. This has the effect not only of raising print speeds, but of considerably reducing noise and vibration, although this effect will not be obvious unless two printers are operating near to each other under different control algorithms.

Several further features are controlled by "attributes" which could be easily extended, but as currently implemented allow for underlining, BOLD character printing, RED printing, ^{super}scripting, and _{sub}scripting, or any combination of these features, which can even be invoked at the character level within words.

The driver can optionally skip to top-of-page at a given page line count, to prevent printing over fold lines in continuous stationery, and, if required will allow the operator to load a fresh sheet of cut paper after ejecting the current "page". In this case, a message is sent to the operator console, together with a bell-code, as indications that operator intervention is required. When the new sheet has been loaded, the operator presses a key on the console in order to continue printing. In this situation, the operator is able to give a specific response in order to inform the driver that pauses are no longer needed between pages. This response is given by pressing either upper-case, or lower-case C (for Continue).

The facility for using cut sheets is essential in the context of preparing documents such as letters, which will usually be printed on headed paper for the first sheet, and plain paper for continuations.

The printing operation may be aborted at any time by hitting the <ESC> key on the console, and may be halted temporarily at any time by raising the cover on the printer itself.

If the printer is equipped with a detector for end-of-ribbon, which is a standard feature of the SPINWRITER, it will automatically pause for the operator to load a new cartridge. Printing will be suspended, and an audible alarm given to indicate that operator

intervention is required. After the cartridge has been replaced, printing resumes without any visible discontinuity.

2 THE DRIVER FROM A NON TECHNICAL USER'S VIEWPOINT

2.1 LOADING AND INITIALISING THE DRIVER

The driver is loaded by RIO, the operating system executive, either in response to a direct command from the operator entered at the console keyboard, or to a command included in a file containing commands, such as the file OS.INIT .

In either case, the operation involves the use of the RIO command ACTIVATE.

Assuming that a NEC Spinwriter is in use, and the files of the disc supplied have been used without modification, the actual device driver will be known as \$NEC. The command is therefore

```
%ACTIVATE $NEC
^
          RIO prompt
          ~~~~~
          Operator command
```

Optionally, the operator may make the printer driver known by a 'Logical Unit Number', preferably 3, which is used by convention for printers. This can be achieved by the additional command :-

```
%DEFINE 3 $NEC
^
          RIO prompt
          ~~~~~
          Operator command
```

There are several RIO utilities, such as PRINT, and CAT, which automatically send their output to whatever device has been associated with LUN=3, and it is recommended that the procedure given above be used. This adds significantly to the general convenience of using the system.

When the driver receives an Initialisation request from RIO, it performs some general housekeeping operations, such as preparing the electrical characteristics of the hardware interface to the printer, and placing the printhead in a known position, and then it sends a message to the operator, asking whether it is to operate in the manner needed if cut sheets of paper are to be used, and whether it is to automatically perform paper throws to prevent printing over folds in continuous stationery. These two functions are separately controllable, as some programs which use the printer driver perform similar functions themselves. In that case, irritating interactions could possibly occur, resulting for example, in alternate pages being

completely blank. Initialisation requests are sent to the printer driver whenever it is ACTIVATED, or, if required, can be issued specifically on demand by the operator by using the command:-

```
%I $NEC
^
^^^^^^      RIO prompt
              Operator command
```

In either event, the driver will introduce itself by a message on the system console, giving its revision level, and will then ask the operator two questions. The responses are entered on the console keyboard as single characters. The response character Y in either upper, or lower case, specifies 'YES' to the question.

The dialogue takes the following form:-

```
%ACTIVATE $NEC

Printer Driver rev. 2.1

Cut sheets ?    Y
Auto formfeed ? Y

%
```

In the above example, the two responses for Yes are shown. Any other character response apart from Y is interpreted as NO.

If the requirements of the driver change during a session, all that is needed to redefine the characteristics is for the operator to give the command to re-initialise the driver, as exemplified previously. The two questions will be re-asked. If this is done, it does NOT alter the previously defined association between the driver and a RIO LUN.

2.2 USING THE DRIVER

RIO can be told to pass data (ie. text) to the printer in essentially two different ways. One is much simpler to use than the other, and relies on having DEFINED that the driver is known to RIO as LUN=3.

Assuming that a file is known to exist, such as PRINTER.DRIVER.MANUAL and that the printer driver is indeed known as LUN=3, all that the operator needs to enter is:-

```
%PRINT PRINTER.DRIVER.MANUAL
```

and the contents of that file will be printed. Obviously the text actually appearing on the paper will largely reflect exactly what is contained in the file, but pagination can be affected by whether the operator has selected automatic formfeed. The use of cut sheets does

not affect the printed output, merely the type of paper stock which can conveniently be used.

Alternatively it is often possible to send text direct to \$NEC as it is being generated, instead perhaps, of preparing a file which would have to be printed subsequently.

This can be achieved in different ways depending on the program which is generating the text. As an example, we will show the use of ZFORM.

ZFORM outputs its formatted text to the system console unless the operator specifies an alternative. This is particularly convenient, as most operators would require to view the fully formatted text on their VDU more frequently than actually printing it.

An alternative is specified by giving an "O=" option, which can define either the name of a file which is to be prepared to contain the formatted output text, or a driver, such as \$NEC. In this case, the formatted text would be sent directly to \$NEC, and therefore to the printer, as it is being generated.

For example:

```
%ZFORM PRINTER.DRIVER.MANUAL O=$NEC
```

2.3 AUTOMATIC FORMFEED

If the operator has selected the automatic formfeed option, after printing a given number of lines on a page, the driver will tell the printer to perform a paper-throw, ie. formfeed operation, which prepares it to start printing on a new sheet.

The driver maintains a count of the number of lines it has printed since the last time it started a new sheet, and when it reaches 63 (this can be varied if necessary) it requests a formfeed. As most paper sheets have a length equal to 66 lines, this means that every page would have at least three blank lines, and they would normally be positioned to bracket the folds in continuous stationery.

However, if the driver finds a formfeed code in the text being printed, it requests a paper throw, and zeros its line counter.

Most files of text which are prepared by Zilog utility programs do contain embedded formfeed codes, and it is for this reason that the driver usually does not need to insert any automatically. Interaction could occur if the text being printed were paginated using the same line count per page as the printer driver. Blank alternate pages would result.

It is very useful, however, to be able to neatly print, and paginate, files which are prepared directly by an operator using a text editor,

such as input files for use by ZFORM for example, and it is then that the auto formfeed option is likely to be used.

2.4 USE OF CUT SHEETS

If the operator has selected the cut sheet option, whenever the driver has sent a paper throw instruction to the printer, which will cause the current sheet to be ejected, it will then cause the console's buzzer to be sounded and send a message to the operator at the system console. The message represents a request to load a new sheet of paper, and then to press a key on the keyboard as an instruction to the driver that printing can resume. If any key other than either an upper case, or a lower case C is pressed, the driver will continue to request the loading of fresh sheets whenever it has ejected the current sheet. However, if the letter C is used in response, printing will continue in the expectation that continuous stationery is then in use.

2.5 CHARACTER ATTRIBUTES

As implemented herein, the attributes of BOLD, UNDERLINE, RED, SUPERSCRIPIT, and SUBSCRIPT, are invoked by two-character "ESCAPE code sequences" in the text being printed, employing codes which do not correspond to printable characters.

This technique ensures that text files which contain the necessary control codes for these functions can be printed by printers such as TALLY, CENTRONICS etc. and by conventional Visual Display Units when handled by standard Zilog drivers, with total compatibility. Obviously these peripherals cannot produce the effects obtainable when using a Spinwriter, but the text format and content would be the same. It would perhaps be regarded as a draft quality output which can be prepared at very high speed.

The attribute control sequences are as follows:

BOLD	<ESC> CTRL-B
UNDERLINE	<ESC> CTRL-U
RED	<ESC> CTRL-R
POSITIVE HALF-LINEFEED (for SUBSCRIPTING)	<ESC> CTRL-F
NEGATIVE HALF-LINEFEED (for SUPERSCRIPITING)	<ESC> CTRL-N

The first three attribute control sequences operate in the manner of 'toggles'. ie. if, for example, the printer is outputting in black, then the sequence <ESC> CTRL-R would switch it to red, and a further <ESC> CTRL-R would switch it back to black again.

The receipt of a formfeed request, whether internally generated, or received as part of the text being printed, cancels out any current

superscripting or subscripting. ie. the top line of a new page will always start in the same place relative to the top of the paper.

When preparing an original text file, the method for embedding the control code sequences will depend on exactly what software utility is being used at that time, however, as an example, we will illustrate the use of the standard RIO TEXT EDITOR.

As this editor makes a special use of the code generated by depressing the 'ESCAPE' key on the console keyboard, the following example illustrates perhaps the most involved way of incorporating attribute control sequences into the text.

The editor can be told to pass the code from the ESCAPE key into its output file by preceding it with the key labeled '\'. This prevents the editor from interpreting the code from ESCAPE for its special function.

Now, let's see exactly what keystroke sequences would have to be used in order to print the following line of text:-

This illustrates **BOLD** printing.

Using the terminology that <ESC> means the ESCAPE key, CTRL-B means pressing the 'B' key whilst holding down the CONTROL-SHIFT key, the operator would use the following key sequence:-

This illustrates\

Notice that that there must not be any character between the <ESC> and the following attribute control code.

Character attributes can be individually controlled. The inclusion of an attribute control sequence within the text is really interpreted as an instruction to make use of the currently set attribute(s) until redefined.

3 THE DRIVER FROM A TECHNICAL USER'S VIEWPOINT

3.1 THE INTERFACE HARDWARE

The printer is interfaced to the host system by the use of one of the four serial channels provided by an SIB (Serial Interface Board).

There is one other major software utility which currently employs an SIB channel, ie. the Asynchronous Communications Package.

The printer driver and the communications package are totally compatible with each other, and can successfully co-operate within a system.

This driver assumes the use of channel 2 of the SIB, which is installed in an MCZ-1/nn style Zilog system without modification. Channels 2 and 3 are pre-wired in all systems currently shipped. The comms package uses channel 3.

SIB modules have a great many link areas enabling the characteristics of each channel to be tailored to precisely suit the user's needs. The following link definitions are specific to channel 2, and the printer driver when operating with either a NEC Spinwriter model 5510/5520, or a DIABLO model 1610/1620.

Clock distribution:-

J3-6 to J3-12

J2-3 to J2-15

J2-3 to J2-16

Connections between USART-2 and the printer:-

J7-1 to J7-13

J7-2 to J7-14

J7-3 to J7-11

J7-4 to J7-12

J7-5 to J7-10

J7-6 to J7-9

The above links configure channel 2 of the SIB to communicate with a 'terminal', and interface the following signals at the 25 way socket with which the printer is to be connected. In most MCZ-1/nn systems this is labeled 'J102'.

The actual interface signals are as follows:-

J102-7	Ground
J102-8	'spare'
J102-5	Clear To Send -> To printer
J102-6	Data Set Ready -> To printer
J102-20	Data Terminal Ready <- From printer
J102-4	Request To Send <- From printer
J102-3	Received data <- From printer
J102-2	Transmitted data -> To printer

The printer driver assumes that a rate of 1200 bauds will be used, and switches in the printer must be appropriately set. As those settings depend upon the exact printer model number, it is impracticable to give them here.

All other links on the SIB are either to be left as when delivered, or set as required for defining the characteristics of the channels which are not used by the printer driver

3.2 THE DRIVER SOFTWARE - GENERAL

This driver operates by placing character codes into a line buffer in locations which correspond to columns of the output line. ie. the content of the line buffer is columnated.

Before being written into, the line buffer is cleared to contain space codes in bits 0 -> 6. Bit 7 is handled independently, and, when set, defines the location of a tab-stop. Each time the line buffer is cleared, the setting of bit 7 is left unaffected in each location.

The tab bits are defined after a call is made to determine the current tab locations in use by the system console driver. This is done each time an initialisation request is received by the printer driver.

A second columnated buffer is used in addition to the line buffer. This contains character attributes and effectively extends each character code to include bits which independently define whether the character is to be printed in red, bold, underlined, or as a superscript or subscript. As currently implemented, there are three spare attribute bits, which could easily be allocated for specific extensions to the driver's capabilities.

The attribute buffer is loaded according to the attribute control sequences embedded in the input text. These are used to directly control the value of the variable NEXT_ATTRIBUTE, which is copied into the attribute buffer when a character is placed into the line buffer.

After the line and attribute buffers have been loaded, the driver decides whether the current printhead position is nearer to the left or right end of the line about to be printed, and is therefore able to perform an absolute tab to the nearer end, and output the line in the appropriate direction.

Notice that the driver never issues a carriage-return code to the printer. It always sends absolute tabs and linefeeds. This is due to the danger of accidentally locking the "Auto Linefeed" switch of some printers, which is sometimes located near to frequently used controls. Its use would destroy carefully defined formats.

It is expected that any programmer who wishes to understand, or modify, the driver will be able to do so easily after reading the module listings. Therefore a blow-by-blow descriptions of the operation is considered unnecessary.

3.3 THE MODULES

The driver software consists of a number of modules, each being written in the language chosen to be most appropriate for the function it performs.

The modules perform the following general functions:-

MODULE	FUNCTION
PRINTER.DRIVER.0 (PLZSYS)	Receives the RIO request vector from RIO.IO.INTERFACE and interprets the request code. Makes a call back to RIO in order to determine the TABSTOP locations within the standard RIO console driver so as to be able to use the same locations itself.
PRINTER.DRIVER.1 (PLZSYS)	Contains the procedures for building the LINE and ATTRIBUTE buffers.
PRINTER.DRIVER.2 (PLZSYS)	Contains the procedures for optimising print direction and removing data from the LINE and ATTRIBUTE buffers during actual printing.
DIABLO and SPINWRITER (PLZSYS)	These modules contain printer-dependent procedures for selecting print direction, absolute tabbing, selecting ribbon colour, requesting positive or negative half linefeeds for subscripting and superscripting, and management of ETX/ACK protocol for maintaining control of the buffer in the printer itself. This is the only module of the driver which is not common to both the Spinwriter and the Diablo.

RIO.IO.INTERFACE (ASSEMBLER)

This very simple module merely converts the IY register content received from RIO as the request-vector-pointer into a PLZSYS procedure parameter. It then passes control to the main procedure in PRINTER.DRIVER.0

Return to RIO is through this module so that IY can be restored.

SIB (ASSEMBLER)

This module contains the routines to set up the basic I/O interface to the printer. All routines may be called direct from PLZSYS code.

All character level I/O is performed by this module.

CALL.SYSTEM (ASSEMBLER)

This module receives a pointer as a parameter from a PLZSYS program, and passes it as an RIO I/O request vector pointer to RIO. In this driver it is used when requesting the status of \$CON to determine if an abort has been requested, for getting tab locations from \$CON, also for issuing messages to, and obtaining operator responses from \$CON.

PLZ.INTERFACE.MACROS (ASSEMBLER)

For convenience only.

4 CONCLUSION

Hopefully this note will have given the reader a few ideas about the use of PLZSYS in association with Assembly Language for I/O driver writing. The author cannot realistically recommend its use if memory space is at a premium, but certainly does recommend it wholeheartedly if an objective is to produce intelligible, easily adaptable code quickly. The entire driver described herein took less than 30 man-hours to design, implement and test.

The source code files for all modules are available from Zilog's franchised distributors as part of the software library.

Any users' improvements to this driver would be warmly welcomed if contributed to the Software Library.

PLZSYS 2.02

```

1  PRINTER_DRIVER_0  MODULE
2
3  !   Extended 3/3/79 to allow for subscripting and superscripting !
4  !   Also for operator controlled page-waits and auto formfeed !
5
6
7  TYPE
8
9      RIO_REQUEST_VECTOR      RECORD [   LUN      BYTE
10                                     REQ      BYTE
11                                     DTA      ^BYTE
12                                     DTL      WORD
13                                     CRA      WORD
14                                     ERA      WORD
15                                     CCOD     BYTE
16                                     SPV_ADD  WORD   ]
17
18  CONSTANT
19
20      INITIALISE              := 0
21      ASSIGN                  := %02
22      OPEN                    := %04
23      CLOSE                   := %06
24      WRITE_BINARY            := %0E
25      WRITE_LINE              := %10
26      READ_LINE               := %0C
27      READ_STATUS             := %40
28      WRITE_STATUS            := %42
29      DEACTIVATE              := %44
30      INVALID_OPERATION_REQUEST := %C1
31      PROGRAMME_ABORT         := %49
32      GOOD_RETURN             := %80
33      CONIN                   := 1
34      CONOUT                  := 2
35      ASCII_SPACE             := ' '
36      ASCII_CR                := '%R'
37      ASCII_LF                := '%L'
38      ASCII_FF                := '%P'
39      ASCII_BELL              := %07
40      BLACK                   := %01
41      TRUE                    := 1
42      FALSE                   := 0
43      INTERRUPT_REQUEST_MASK := %FE
44      TAB_BIT                 := %80
45
46  EXTERNAL
47
48      REQUEST_BLACK           PROCEDURE
49      GET_CODE                PROCEDURE
50      SETCH2                  PROCEDURE
51      ABSOLUTE_TAB           PROCEDURE ( BYTE )
52      FORMFEED                PROCEDURE

```

```

53          PRINT_LINE_BUFFER          PROCEDURE
54          CALRIO                      PROCEDURE ( ^BYTE )
55                                          RETURNS ( BYTE )
56          CLEAR_LINE_BUFFER          PROCEDURE
57
58          LINE_CONTAINS_PRINTABLE_CHAR  BYTE
59          CODE                        BYTE
60          ATTRIBUTE_SEQUENCE_FLAG      BYTE
61          NEXT_ATTRIBUTE              BYTE
62          LINE_FINISHED_FLAG          BYTE
63          EOF_FLAG                    BYTE
64          BYTES_TAKEN_FROM_SOURCE      WORD
65          BYTE_COUNT                  BYTE
66          CONSOLE_STATUS_BUFFER        ARRAY [ 5 BYTE ]
67          LINE_BUFFER                  ARRAY [ 163 BYTE ]
68          LINE_BUFFER_PTR              ^BYTE
69
70
71          INTERNAL
72
73          INPUT_CHAR                   BYTE
74
75          date_code                    ARRAY [ * BYTE ] := '%RPrinter Driver rev. 2.1'R'
76          NEW_SHEET_MSG                ARRAY [ * BYTE ] := 'Load new sheet, hit a key :'
77          BELL_STRING                  ARRAY [ 1 BYTE ] := [ ASCII_BELL ]
78          CUT_SHT_QUES                 ARRAY [ * BYTE ] := '%RCut sheets ? '
79          AUTO_FM_FEED_QUES            ARRAY [ * BYTE ] := 'Auto formfeed ? '
80          NL_ARRAY                     ARRAY [ * BYTE ] := '%R'
81
82          GENERAL_RIO_CALL_VECTOR      RIO_REQUEST_VECTOR
83
84          GLOBAL
85
86          REQUEST_CODE                  BYTE
87          SOURCE_PTR                    ^BYTE
88          DATA_LENGTH                  WORD
89          ABORT_FLAG                    BYTE
90          AUTO_FF_FLAG                  BYTE
91          PAGE_WAIT_FLAG                BYTE
92
93
94          CALL_RIO      PROCEDURE ( UNIT          BYTE
95                          REQUEST              BYTE
96                          DATA_ADDRESS        ^BYTE
97                          DATA_LENGTH        WORD )
98
99          LOCAL      RETURN_CODE BYTE
100
101          ENTRY
102          GENERAL_RIO_CALL_VECTOR.LUN        := UNIT
103          GENERAL_RIO_CALL_VECTOR.REQ        := REQUEST
104          GENERAL_RIO_CALL_VECTOR.DTA        := DATA_ADDRESS
105          GENERAL_RIO_CALL_VECTOR.DTL        := DATA_LENGTH

```

```
106         GENERAL_RIO_CALL_VECTOR.CRA           := 0
107         GENERAL_RIO_CALL_VECTOR.ERA           := 0
108         GENERAL_RIO_CALL_VECTOR.CCOD          := 0
109         GENERAL_RIO_CALL_VECTOR.SPV_ADD       := 0
110
111         RETURN_CODE := CALRIO ( #GENERAL_RIO_CALL_VECTOR.LUN )
112
113         IF RETURN_CODE <> GOOD_RETURN
114             THEN
115                 ABORT_FLAG := TRUE
116         FI
117
118     END CALL_RIO
119
120
121     MAYBE_ABORT           PROCEDURE
122
123     ENTRY
124
125         CALL_RIO (  CONIN
126                   READ_STATUS
127                   #CONSOLE_STATUS_BUFFER[0]
128                   1 )
129
130         IF ( CONSOLE_STATUS_BUFFER[0] AND %20 ) = 0
131             THEN
132                 ABORT_FLAG := TRUE
133         FI
134
135     END MAYBE_ABORT
136
137
138     NEWLINE           PROCEDURE
139
140     ENTRY
141         CALL_RIO (  CONOUT
142                   WRITE_BINARY
143                   #NL_ARRAY[0]
144                   SIZEOF NL_ARRAY )
145
146     END NEWLINE
147
148
149     GET_CHAR           PROCEDURE
150
151     ENTRY
152         CALL_RIO (  CONIN
153                   READ_LINE
154                   #INPUT_CHAR
155                   1 )
156
157         IF INPUT_CHAR <> '%R' THEN NEWLINE FI
158
```

```

159     END GET_CHAR
160
161
162     PAGE_WAIT      PROCEDURE
163
164     ENTRY
165         CALL_RIO (  CONOUT
166                   WRITE_BINARY
167                   #NEW_SHEET_MSG[0]
168                   SIZEOF NEW_SHEET_MSG + SIZEOF BELL_STRING )
169
170     GET_CHAR
171     IF INPUT_CHAR
172         CASE 'C' 'c'
173         THEN
174             PAGE_WAIT_FLAG := FALSE
175             NEWLINE
176         FI
177     END PAGE_WAIT
178
179
180     GET_FLAGS      PROCEDURE
181
182     ENTRY
183         CALL_RIO (  CONOUT
184                   WRITE_BINARY
185                   #date_code[0]
186                   SIZEOF date_code )
187
188         CALL_RIO (  CONOUT
189                   WRITE_BINARY
190                   #CUT_SHT_QUES[0]
191                   SIZEOF CUT_SHT_QUES )
192
193     GET_CHAR
194     PAGE_WAIT_FLAG := FALSE
195     IF INPUT_CHAR
196         CASE 'Y' 'y'
197         THEN PAGE_WAIT_FLAG := TRUE
198     FI
199     CALL_RIO (  CONOUT
200               WRITE_BINARY
201               #AUTO_FM_FEED_QUES[0]
202               SIZEOF AUTO_FM_FEED_QUES )
203
204     GET_CHAR
205     AUTO_FF_FLAG := FALSE
206     IF INPUT_CHAR
207         CASE 'Y' 'y'
208         THEN AUTO_FF_FLAG := TRUE
209     FI
210     NEWLINE
211     NEWLINE

```

```

212
213  END GET_FLAGS
214
215
216  GET_TAB_LOCATIONS  PROCEDURE
217
218      LOCAL  COUNTER BYTE
219
220      ENTRY
221          CALL_RIO (  CONIN
222                      READ_STATUS
223                      #CONSOLE_STATUS_BUFFER[0]
224                      139 )
225
226          COUNTER := 0
227          LINE_BUFFER_PTR := # LINE_BUFFER [0]
228          DO
229              IF COUNTER = 163 THEN EXIT FI
230              IF COUNTER < 134
231                  THEN
232                      IF LINE_BUFFER_PTR^ <> 0
233                          THEN
234                              LINE_BUFFER_PTR^ := TAB_BIT
235                          ELSE
236                              LINE_BUFFER_PTR^ := 0
237                          FI
238                      ELSE
239                          LINE_BUFFER_PTR^ := TAB_BIT
240                      FI
241              LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
242              COUNTER += 1
243          OD
244
245          CLEAR_LINE_BUFFER
246
247  END GET_TAB_LOCATIONS
248
249
250
251
252  EJECT_PAGE          PROCEDURE
253
254      ENTRY
255          IF PAGE_WAIT_FLAG = TRUE
256              THEN
257                  PAGE_WAIT_FLAG := FALSE
258                  FORMFEED
259                  PAGE_WAIT_FLAG := TRUE
260              ELSE
261                  FORMFEED
262              FI
263
264  END EJECT_PAGE

```

```

265
266
267     PLZDVR                PROCEDURE ( VECTOR_PTR ^RIO_REQUEST_VECTOR )
268
269     ENTRY
270         REQUEST_CODE := VECTOR_PTR^.REQ AND INTERRUPT_REQUEST_MASK
271         SOURCE_PTR := VECTOR_PTR^.DTA
272         DATA_LENGTH := VECTOR_PTR^.DTL
273
274         VECTOR_PTR^.CCOD := GOOD_RETURN
275         VECTOR_PTR^.DTL := 0
276
277         BYTES_TAKEN_FROM_SOURCE := 0
278         EOF_FLAG := FALSE
279         ABORT_FLAG := FALSE
280
281         IF REQUEST_CODE
282
283             CASE INITIALISE
284                 THEN
285                     SETCH2
286                     BYTE_COUNT := 0
287                     ABSOLUTE_TAB (1)
288                     EJECT_PAGE
289                     ATTRIBUTE_SEQUENCE_FLAG := FALSE
290                     NEXT_ATTRIBUTE := BLACK
291                     REQUEST_BLACK
292                     GET_TAB_LOCATIONS
293                     GET_FLAGS
294
295                     RETURN
296
297             CASE ASSIGN
298                 THEN
299                     RETURN
300
301             CASE OPEN
302                 THEN
303                     GET_TAB_LOCATIONS
304                     RETURN
305
306             CASE CLOSE,DEACTIVATE
307                 THEN
308                     ABSOLUTE_TAB (1)
309                     EJECT_PAGE
310                     RETURN
311
312             CASE WRITE_BINARY
313                 THEN
314                 DO
315                     IF DATA_LENGTH = BYTES_TAKEN_FROM_SOURCE
316                         THEN
317                             EXIT

```

```

318             FI
319             GET_CODE
320             IF EOF_FLAG = TRUE
321                 THEN
322                 EXIT
323             FI
324             IF ABORT_FLAG = TRUE
325                 THEN
326                 VECTOR_PTR^.CCOD := PROGRAMME_ABORT
327                 EJECT_PAGE
328                 EXIT
329             FI
330         OD
331
332         VECTOR_PTR^.DTL := BYTES_TAKEN_FROM_SOURCE
333         RETURN
334
335     CASE WRITE_LINE
336     THEN
337         LINE_CONTAINS_PRINTABLE_CHAR := FALSE
338         DO
339             IF DATA_LENGTH = BYTES_TAKEN_FROM_SOURCE
340                 THEN
341                 PRINT_LINE_BUFFER
342                 EXIT
343             FI
344
345             GET_CODE
346             IF CODE = ASCII_CR THEN EXIT FI
347             IF ABORT_FLAG = TRUE
348                 THEN
349                 VECTOR_PTR^.CCOD := PROGRAMME_ABORT
350                 EJECT_PAGE
351                 EXIT
352             FI
353         OD
354
355         VECTOR_PTR^.DTL := BYTES_TAKEN_FROM_SOURCE
356
357         RETURN
358     ELSE
359
360         VECTOR_PTR^.CCOD := INVALID_OPERATION_REQUEST
361
362     FI
363
364     END PLZDVR
365     END PRINTER_DRIVER_0
366
367     END OF ZCODE GENERATION
    0 ERROR(S)      0 WARNING(S)

```

Spinwriter/Diablo driver

PLZSYS 2.02

```

1  PRINTER_DRIVER_1      MODULE
2
3  !   Extended 3/3/79 to allow for subscripting and superscripting
4
5
6  CONSTANT
7
8      TRUE                := 1
9      FALSE               := 0
10
11     ASCII_SPACE         := ' '
12     ASCII_TAB           := %09
13     ASCII_BS            := %08
14     ASCII_ESC           := %1B
15     ASCII_FF            := %0C
16     ASCII_CR            := '%R'
17     ASCII_LF            := '%L'
18     ASCII_CONTROL_R     := %12 ! COLOUR CHANGE !
19     ASCII_CONTROL_B     := %02 ! BOLD !
20     ASCII_CONTROL_U     := %15 ! UNDERLINE !
21     ASCII_CONTROL_N     := %0E ! SUPERSCRIPT !
22     ASCII_CONTROL_F     := %06 ! SUBSCRIPT !
23
24     BLACK                := %01
25     RED                  := NOT BLACK
26     BOLD                 := %02
27     NOT_BOLD             := NOT BOLD
28     UNDERLINE           := %04
29     NOT_UNDERLINE       := NOT UNDERLINE
30     SUPERSCRIP          := %08
31     NOT_SUPERSCRIP      := NOT SUPERSCRIP
32     SUBSCRIPT           := %10
33     NOT_SUBSCRIPT       := NOT SUBSCRIPT
34
35     TAB_MASK             := %80
36     PARITY_MASK         := %7F
37
38     EXTERNAL
39
40     PRINTER_WIDTH       BYTE
41     PRINT_LINE_BUFFER   PROCEDURE
42     FORMFEED            PROCEDURE
43     LINEFEED            PROCEDURE
44     MAYBE_ABORT         PROCEDURE
45
46     SOURCE_PTR          ^BYTE
47
48
49     GLOBAL
50
51     CODE                BYTE
52     BYTES_TAKEN_FROM_SOURCE WORD

```

```

53      CONSOLE_STATUS_BUFFER      ARRAY [ 5 BYTE ]
54      LINE_BUFFER                 ARRAY [ 163 BYTE ]
55      ATTRIBUTE_BUFFER            ARRAY [ 163 BYTE ]
56      COLUMN_NO                   BYTE
57      LINE_CONTAINS_PRINTABLE_CHAR BYTE
58      CHARS_IN_LINE_BUFFER        BYTE
59      ATTRIBUTE                   BYTE
60      NEXT_ATTRIBUTE              BYTE
61      ATTRIBUTE_BUFFER_PTR        ^BYTE
62      LEFTMOST_PRINTABLE_COLUMN   BYTE
63      RIGHTMOST_PRINTABLE_COLUMN  BYTE
64      LINE_BUFFER_PTR             ^BYTE
65      ATTRIBUTE_SEQUENCE_FLAG     BYTE
66      EOF_FLAG                    BYTE
67      LINE_FINISHED_FLAG          BYTE
68
69      CLEAR_LINE_BUFFER            PROCEDURE
70
71      ENTRY
72          LINE_BUFFER_PTR := #LINE_BUFFER [0]
73          COLUMN_NO := 1
74          LINE_CONTAINS_PRINTABLE_CHAR := FALSE
75          LEFTMOST_PRINTABLE_COLUMN := 1
76          RIGHTMOST_PRINTABLE_COLUMN := 1
77
78      DO
79          IF COLUMN_NO > PRINTER_WIDTH
80              THEN
81                  COLUMN_NO := 1
82                  LINE_BUFFER_PTR := #LINE_BUFFER [0]
83                  ATTRIBUTE_BUFFER_PTR := #ATTRIBUTE_BUFFER [0]
84                  RETURN
85          FI
86
87          LINE_BUFFER_PTR^ := ( LINE_BUFFER_PTR^ AND TAB_MASK )
88                          OR ASCII_SPACE
89          LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
90          COLUMN_NO += 1
91      OD
92
93      END CLEAR_LINE_BUFFER
94
95
96      PUT_CODE_INTO_LINE_BUFFER    PROCEDURE
97
98      ENTRY
99          LINE_BUFFER_PTR^ := ( LINE_BUFFER_PTR^ AND TAB_MASK )
100                          OR CODE
101          ATTRIBUTE_BUFFER_PTR^ := NEXT_ATTRIBUTE
102
103          IF LINE_CONTAINS_PRINTABLE_CHAR = FALSE
104              THEN
105                  LEFTMOST_PRINTABLE_COLUMN := COLUMN_NO

```

```

106             LINE_CONTAINS_PRINTABLE_CHAR := TRUE
107         FI
108
109         RIGHTMOST_PRINTABLE_COLUMN := COLUMN_NO
110         COLUMN_NO += 1
111         LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
112         ATTRIBUTE_BUFFER_PTR := INC ATTRIBUTE_BUFFER_PTR
113
114     END PUT_CODE_INTO_LINE_BUFFER
115
116
117     GOT_TAB                PROCEDURE
118
119     ENTRY
120     DO
121         LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
122         ATTRIBUTE_BUFFER_PTR^ := NEXT_ATTRIBUTE
123         ATTRIBUTE_BUFFER_PTR := INC ATTRIBUTE_BUFFER_PTR
124         COLUMN_NO += 1
125         IF ( LINE_BUFFER_PTR^ AND TAB_MASK ) <> 0
126             THEN
127                 RETURN
128         FI
129     OD
130
131     END GOT_TAB
132
133
134     FETCH_ATTRIBUTE        PROCEDURE
135
136     ENTRY
137         ATTRIBUTE_SEQUENCE_FLAG := FALSE
138
139     IF CODE
140
141         CASE ASCII_CONTROL_R
142         THEN
143             IF NEXT_ATTRIBUTE AND BLACK <> 0
144                 THEN
145                     NEXT_ATTRIBUTE := NEXT_ATTRIBUTE AND RED
146                 ELSE
147                     NEXT_ATTRIBUTE := NEXT_ATTRIBUTE OR BLACK
148             FI
149
150         CASE ASCII_CONTROL_B
151         THEN
152             IF NEXT_ATTRIBUTE AND BOLD <> 0
153                 THEN
154                     NEXT_ATTRIBUTE := NEXT_ATTRIBUTE AND NOT_BOLD
155                 ELSE
156                     NEXT_ATTRIBUTE := NEXT_ATTRIBUTE OR BOLD
157             FI
158

```

```
159         CASE ASCII_CONTROL_U
160             THEN
161                 IF NEXT_ATTRIBUTE AND UNDERLINE <> 0
162                     THEN
163                         NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
164                             AND NOT_UNDERLINE
165                     ELSE
166                         NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
167                             OR UNDERLINE
168                 FI
169
170         CASE ASCII_CONTROL_N
171             THEN
172                 IF NEXT_ATTRIBUTE AND SUBSCRIPT <> 0
173                     THEN
174                         NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
175                             AND NOT_SUBSCRIPT
176                     ELSE
177                         NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
178                             OR SUPERSCRIPT
179                 FI
180
181         CASE ASCII_CONTROL_F
182             THEN
183                 IF NEXT_ATTRIBUTE AND SUPERSCRIPT <> 0
184                     THEN
185                         NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
186                             AND NOT_SUPERSCRIPT
187                     ELSE
188                         NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
189                             OR SUBSCRIPT
190                 FI
191
192         FI
193
194     END FETCH_ATTRIBUTE
195
196
197     GET_CODE           PROCEDURE
198
199         ENTRY
200             CODE := SOURCE_PTR^
201
202             IF CODE = %FF
203                 THEN
204                     EOF_FLAG := TRUE
205                     RETURN
206             FI
207
208             CODE := CODE AND PARITY_MASK
209             SOURCE_PTR := INC SOURCE_PTR
210             BYTES_TAKEN_FROM_SOURCE += 1
211             IF ATTRIBUTE_SEQUENCE_FLAG = TRUE
```

```
212         THEN
213             FETCH_ATTRIBUTE
214             RETURN
215     FI
216     IF CODE > ASCII_SPACE
217         THEN
218             PUT_CODE_INTO_LINE_BUFFER
219             RETURN
220     FI
221     IF CODE
222
223         CASE ASCII_SPACE
224             THEN
225                 LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
226                 ATTRIBUTE_BUFFER_PTR^ := NEXT_ATTRIBUTE
227                 ATTRIBUTE_BUFFER_PTR := INC ATTRIBUTE_BUFFER_PTR
228                 COLUMN_NO += 1
229
230         CASE ASCII_CR
231             THEN
232                 PRINT_LINE_BUFFER
233                 LINEFEED
234                 MAYBE_ABORT
235
236         CASE ASCII_FF
237             THEN
238                 PRINT_LINE_BUFFER
239                 FORMFEED
240                 MAYBE_ABORT
241
242         CASE ASCII_LF
243             THEN
244                 PRINT_LINE_BUFFER
245                 LINEFEED
246                 MAYBE_ABORT
247
248         CASE ASCII_TAB
249             THEN
250                 GOT_TAB
251
252         CASE ASCII_ESC
253             THEN
254                 ATTRIBUTE_SEQUENCE_FLAG := TRUE
255
256     FI
257
258     END GET_CODE
259
260
261     END PRINTER_DRIVER_1
END OF ZCODE GENERATION
    0 ERROR(S)      0 WARNING(S)
```

PLZSYS 2.02

```

1  PRINTER_DRIVER_2  MODULE
2
3  CONSTANT
4
5      ASCII_SPACE           := ' '
6
7      PARITY_MASK           := %7F
8      FORWARD               := 1
9      BACKWARD              := 0
10
11     TRUE                   := 1
12     FALSE                  := 0
13
14     WRITE_LINE             := %10
15
16     EXTERNAL
17
18     LEFTMOST_PRINTABLE_COLUMN  BYTE
19     RIGHTMOST_PRINTABLE_COLUMN BYTE
20     PRESENT_COLUMN             BYTE
21     DIRECTION                  BYTE
22     CHARS_IN_LINE_BUFFER      BYTE
23     LINE_BUFFER_PTR            ^BYTE
24     ATTRIBUTE_BUFFER_PTR      ^BYTE
25     ATTRIBUTE                  BYTE
26     REQUEST_CODE               BYTE
27     LINE_CONTAINS_PRINTABLE_CHAR BYTE
28
29
30     LINE_BUFFER                ARRAY [ 163 BYTE ]
31     ATTRIBUTE_BUFFER           ARRAY [ 163 BYTE ]
32
33
34     ABSOLUTE_TAB               PROCEDURE ( BYTE )
35     SEND                       PROCEDURE ( BYTE )
36     PRINT                      PROCEDURE ( BYTE BYTE )
37                               ! CHAR, ATTRIBUTE !
38
39     REQUEST_FORWARD            PROCEDURE
40     REQUEST_BACKWARD          PROCEDURE
41     WAIT_FOR_ACK              PROCEDURE
42     CLEAR_LINE_BUFFER         PROCEDURE
43
44
45     INTERNAL
46
47     CHAR                       BYTE
48     COLUMN_NO                  BYTE
49     NEXT_COLUMN_NO            BYTE
50     SPACE_COUNT                BYTE
51     SPACE_SKIP_FLAG           BYTE
52

```

```

53
54 SET_UP_DIRECTION          PROCEDURE
55
56     ENTRY
57
58         IF LEFTMOST_PRINTABLE_COLUMN > PRESENT_COLUMN
59             THEN
60                 ABSOLUTE_TAB ( LEFTMOST_PRINTABLE_COLUMN )
61                 REQUEST_FORWARD
62                 RETURN
63         FI
64
65         IF PRESENT_COLUMN > RIGHTMOST_PRINTABLE_COLUMN
66             THEN
67                 ABSOLUTE_TAB ( RIGHTMOST_PRINTABLE_COLUMN )
68                 REQUEST_BACKWARD
69                 RETURN
70         FI
71
72         IF ( RIGHTMOST_PRINTABLE_COLUMN - PRESENT_COLUMN )
73             > ( PRESENT_COLUMN - LEFTMOST_PRINTABLE_COLUMN )
74             THEN
75                 ABSOLUTE_TAB ( LEFTMOST_PRINTABLE_COLUMN )
76                 REQUEST_FORWARD
77                 RETURN
78         FI
79
80         ABSOLUTE_TAB ( RIGHTMOST_PRINTABLE_COLUMN )
81         REQUEST_BACKWARD
82
83     END SET_UP_DIRECTION
84
85
86     GLOBAL
87
88
89     PRINT_LINE_BUFFER      PROCEDURE
90
91     ENTRY
92
93         IF LINE_CONTAINS_PRINTABLE_CHAR = FALSE
94             THEN
95                 CLEAR_LINE_BUFFER      RETURN
96         ELSE
97             CHARS_IN_LINE_BUFFER := RIGHTMOST_PRINTABLE_COLUMN
98                                     - LEFTMOST_PRINTABLE_COLUMN + 1
99         FI
100
101     SET_UP_DIRECTION
102     LINE_BUFFER_PTR := #LINE_BUFFER [ PRESENT_COLUMN-1 ]
103     ATTRIBUTE_BUFFER_PTR := #ATTRIBUTE_BUFFER [ PRESENT_COLUMN-1 ]
104     NEXT_COLUMN_NO := PRESENT_COLUMN
105     SPACE_SKIP_FLAG := FALSE

```

```

106         SPACE_COUNT := 0
107         DO
108
109             IF CHARS_IN_LINE_BUFFER = 0
110                 THEN
111                     CLEAR_LINE_BUFFER
112                     RETURN
113             FI
114
115
116             COLUMN_NO := NEXT_COLUMN_NO
117             CHAR := LINE_BUFFER_PTR^ AND PARITY_MASK
118             ATTRIBUTE := ATTRIBUTE_BUFFER_PTR^
119             IF DIRECTION = BACKWARD
120                 THEN
121                     LINE_BUFFER_PTR := DEC LINE_BUFFER_PTR
122                     ATTRIBUTE_BUFFER_PTR := DEC ATTRIBUTE_BUFFER_PTR
123                     NEXT_COLUMN_NO -= 1
124                 ELSE
125                     LINE_BUFFER_PTR := INC LINE_BUFFER_PTR /
126                     ATTRIBUTE_BUFFER_PTR := INC ATTRIBUTE_BUFFER_PTR
127                     NEXT_COLUMN_NO += 1
128             FI
129
130             CHARS_IN_LINE_BUFFER -= 1
131
132             IF CHAR = ASCII_SPACE
133                 THEN
134                     IF SPACE_SKIP_FLAG = FALSE
135                         THEN
136                             SPACE_SKIP_FLAG := TRUE
137                             SPACE_COUNT := 1
138                             REPEAT
139                     FI
140
141                     SPACE_COUNT += 1
142                     REPEAT
143             FI
144
145             IF SPACE_SKIP_FLAG = TRUE
146                 THEN
147                     SPACE_SKIP_FLAG := FALSE
148                     IF SPACE_COUNT >= 3
149                         THEN
150                             ABSOLUTE_TAB ( COLUMN_NO )
151                     ELSE
152                         DO
153                             IF SPACE_COUNT = 0 THEN EXIT FI
154                             PRINT ( ASCII_SPACE ATTRIBUTE )
155
156                             SPACE_COUNT -= 1
157                         OD
158             FI

```

```
159             FI
160
161             PRINT ( CHAR ATTRIBUTE )
162
163             OD
164
165     END PRINT_LINE_BUFFER
166
167
168
169     END PRINTER_DRIVER_2
END OF ZCODE GENERATION
  0 ERROR(S)      0 WARNING(S)
```

PLZSYS 2.02

```

1  SPINWRITER  MODULE
2
3  !   Extended 3/3/79 to allow for subscripting and superscripting !
4  !   Also for page-waits and controllable auto-formfeed !
5
6
7  CONSTANT
8
9
10     PRINTER_BUFFER_SIZE      := 256
11     PITCH                    := 12
12     SS                       := 120/PITCH
13
14     ASCII_ETX                 := %03
15     ASCII_ACK                 := %06
16     ASCII_ESC                 := %1B
17     ASCII_FF                  := %0C
18     ASCII_BS                  := %08
19     ASCII_UL                  := '_'
20     ASCII_SPACE               := ' '
21
22     PARITY_MASK               := %7F
23
24     FORWARD                   := 1
25     BACKWARD                  := 0
26
27     BLACK                     := %01
28     BOLD                      := %02
29     UNDERLINE                 := %04
30     SUPERSCRIP                := %08
31     NOT_SUPERSCRIP            := NOT SUPERSCRIP
32     SUBSCRIPT                 := %10
33     NOT_SUBSCRIPT             := NOT SUBSCRIPT
34
35     TRUE                      := 1
36     FALSE                     := 0
37
38
39     INTERNAL
40
41     LINE_COUNT                BYTE
42     COLOUR                    BYTE
43     HMI                       BYTE
44     SUBSCRIPT_FLAG            BYTE
45     SUPERSCRIP_FLAG           BYTE
46
47     LAST_SCRIPT_STATE         BYTE
48
49
50     EXTERNAL
51
52     OUTCH2                    PROCEDURE ( BYTE )

```

```

53             INCH2                PROCEDURE RETURNS ( BYTE )
54
55             PAGE_WAIT             PROCEDURE
56
57             NEXT_ATTRIBUTE        BYTE
58
59             PAGE_WAIT_FLAG        BYTE
60             AUTO_FF_FLAG         BYTE
61
62
63             GLOBAL
64
65             PRINTER_WIDTH        BYTE := 163
66             DIRECTION            BYTE
67             PRESENT_COLUMN        BYTE
68             AUTO_FF_LINE_COUNT   BYTE := 63
69             BYTE_COUNT           BYTE
70
71
72             SEND_ETX              PROCEDURE
73             ENTRY
74                 OUTCH2 ( ASCII_ETX )
75                 BYTE_COUNT := 0
76             END SEND_ETX
77
78
79             WAIT_FOR_ACK          PROCEDURE
80             ENTRY
81                 DO
82                     IF ( INCH2 AND PARITY_MASK ) = ASCII_ACK
83                         THEN
84                             RETURN
85                     FI
86                 OD
87             END WAIT_FOR_ACK
88
89
90             SEND                  PROCEDURE ( CODE BYTE )
91             ENTRY
92                 IF BYTE_COUNT > PRINTER_BUFFER_SIZE - 10
93                     THEN
94                         SEND_ETX
95                         WAIT_FOR_ACK
96                     FI
97                 BYTE_COUNT += 1
98                 OUTCH2 ( CODE )
99
100            END SEND
101
102
103            REQUEST_HALF_LINEFEED_PITCH    PROCEDURE
104            ENTRY
105                SEND ( ASCII_ESC )

```

```
106         SEND ( ']' )
107         SEND ( 'R' )
108     END REQUEST_HALF_LINEFEED_PITCH
109
110
111     REQUEST_STANDARD_LINEFEED_PITCH PROCEDURE
112     ENTRY
113         SEND ( ASCII_ESC )
114         SEND ( ']' )
115         SEND ( 'W' )
116     END REQUEST_STANDARD_LINEFEED_PITCH
117
118
119     REQUEST_POS_HALF_LINE           PROCEDURE
120     ENTRY
121         REQUEST_HALF_LINEFEED_PITCH
122         SEND ( '%L' )
123         REQUEST_STANDARD_LINEFEED_PITCH
124     END REQUEST_POS_HALF_LINE
125
126
127     REQUEST_NEG_HALF_LINE           PROCEDURE
128     ENTRY
129         REQUEST_HALF_LINEFEED_PITCH
130         SEND ( ASCII_ESC )
131         SEND ( '9' )
132         REQUEST_STANDARD_LINEFEED_PITCH
133     END REQUEST_NEG_HALF_LINE
134
135
136     FORMFEED                         PROCEDURE
137     ENTRY
138         SEND ( ASCII_FF )
139         LINE_COUNT := 0
140         LAST_SCRIPT_STATE := 0
141         SUPERSCRIP_FLAG := FALSE
142         SUBSCRIP_FLAG := FALSE
143
144         IF PAGE_WAIT_FLAG = TRUE
145             THEN
146                 PAGE_WAIT
147         FI
148
149     END FORMFEED
150
151
152     LINEFEED                         PROCEDURE
153     ENTRY
154         IF AUTO_FF_FLAG = TRUE
155             THEN
156                 IF LINE_COUNT >= AUTO_FF_LINE_COUNT
157                     THEN
158                         FORMFEED
```

```

159                                     RETURN
160                                     FI
161             FI
162
163             SEND ( '%L' )
164             LINE_COUNT += 1
165     END LINEFEED
166
167
168     REQUEST_FORWARD                     PROCEDURE
169     ENTRY
170         IF DIRECTION = FORWARD THEN RETURN FI
171         SEND ( ASCII_ESC )
172         SEND ( '>' )
173         DIRECTION := FORWARD
174     END REQUEST_FORWARD
175
176
177     REQUEST_BACKWARD                   PROCEDURE
178     ENTRY
179         IF DIRECTION = BACKWARD THEN RETURN FI
180         SEND ( ASCII_ESC )
181         SEND ( '<' )
182         DIRECTION := BACKWARD
183     END REQUEST_BACKWARD
184
185
186     REQUEST_BLACK                       PROCEDURE
187     ENTRY
188         SEND ( ASCII_ESC )
189         SEND ( '4' )
190         COLOUR := BLACK
191     END REQUEST_BLACK
192
193
194     REQUEST_RED                         PROCEDURE
195     ENTRY
196         SEND ( ASCII_ESC )
197         SEND ( '3' )
198         COLOUR := 0
199     END REQUEST_RED
200
201
202     DEFINE_HMI                          PROCEDURE ( SPACING BYTE )
203     ENTRY
204         SEND ( ASCII_ESC )
205         SEND ( ']' )
206         SEND ( SPACING + %40 )
207     END DEFINE_HMI
208
209
210     SEND_BOLD_CHAR                      PROCEDURE ( CHAR BYTE ATTRIBUTE BYTE )
211     ENTRY

```

```
212         DEFINE_HMI ( 0 )
213         SEND ( CHAR )
214         IF ( ATTRIBUTE AND UNDERLINE ) <> 0
215             THEN
216                 SEND ( ASCII_UL )
217         FI
218         DEFINE_HMI ( 1 )
219         SEND ( CHAR )
220         DEFINE_HMI ( SS-1 )
221         SEND ( CHAR )
222         DEFINE_HMI ( SS )
223
224     END SEND_BOLD_CHAR
225
226
227     ABSOLUTE_TAB                PROCEDURE ( COLUMN BYTE )
228     ENTRY
229
230         IF COLUMN > 162 THEN COLUMN := 162 FI
231         SEND ( ASCII_ESC )
232
233         PRESENT_COLUMN := COLUMN
234
235         IF COLUMN < 33
236             THEN
237                 SEND ( 'P' )
238                 SEND ( %3F + COLUMN )
239                 RETURN
240         FI
241
242         IF COLUMN < 65
243             THEN
244                 SEND ( 'Q' )
245                 SEND ( %1F + COLUMN )
246                 RETURN
247         FI
248
249         IF COLUMN < 97
250             THEN
251                 SEND ( 'R' )
252                 SEND ( COLUMN - 1 )
253                 RETURN
254         FI
255
256         IF COLUMN < 129
257             THEN
258                 SEND ( 'S' )
259                 SEND ( COLUMN - %21 )
260                 RETURN
261         FI
262
263         IF COLUMN < 161
264             THEN
```

```

265             SEND ( 'T' )
266             SEND ( COLUMN - %41 )
267             RETURN
268         FI
269
270             SEND ( 'U' )
271             SEND ( COLUMN - %61 )
272
273     END ABSOLUTE_TAB
274
275
276     PRINT             PROCEDURE ( CHAR BYTE ATTRIBUTE BYTE )
277     ENTRY
278
279         DO
280             IF ATTRIBUTE AND (SUBSCRIPT OR SUPERSCRIP)
281                 = LAST_SCRIPT_STATE
282             THEN
283                 EXIT
284             FI
285
286             IF ATTRIBUTE AND SUPERSCRIP <> 0
287             THEN
288                 IF SUBSCRIPT_FLAG = TRUE
289                 THEN
290                     REQUEST_NEG_HALF_LINE
291                     SUBSCRIPT_FLAG := FALSE
292                 FI
293                 REQUEST_NEG_HALF_LINE
294                 SUPERSCRIP_FLAG := TRUE
295                 EXIT
296             FI
297
298             IF ATTRIBUTE AND SUBSCRIPT <> 0
299             THEN
300                 IF SUPERSCRIP_FLAG = TRUE
301                 THEN
302                     REQUEST_POS_HALF_LINE
303                     SUPERSCRIP_FLAG := FALSE
304                 FI
305                 REQUEST_POS_HALF_LINE
306                 SUBSCRIPT_FLAG := TRUE
307                 EXIT
308             FI
309
310             IF SUPERSCRIP_FLAG = TRUE
311             THEN
312                 REQUEST_POS_HALF_LINE
313                 SUPERSCRIP_FLAG := FALSE
314                 EXIT
315             FI
316
317             REQUEST_NEG_HALF_LINE

```

```

318         SUBSCRIPT_FLAG := FALSE
319         EXIT
320
321     OD
322
323     LAST_SCRIPT_STATE := ATTRIBUTE
324     AND ( SUPERSCRIPIT OR SUBSCRIPT )
325
326     IF ( ATTRIBUTE AND BLACK ) <> COLOUR
327     THEN
328         IF ( ATTRIBUTE AND BLACK ) = BLACK
329         THEN
330             REQUEST_BLACK
331         ELSE
332             REQUEST_RED
333         FI
334     FI
335
336     IF CHAR > ASCII_SPACE
337     THEN
338         IF ATTRIBUTE AND BOLD <> 0
339         THEN
340             SEND_BOLD_CHAR ( CHAR ATTRIBUTE )
341         ELSE
342             IF ( ATTRIBUTE AND UNDERLINE ) <> 0
343             THEN
344                 DEFINE_HMI ( 0 )
345                 SEND ( CHAR )
346                 DEFINE_HMI ( SS )
347                 SEND ( ASCII_UL )
348             ELSE
349                 SEND ( CHAR )
350             FI
351         FI
352     ELSE
353         SEND ( CHAR ) ! SPACES WILL NOT BE UNDERLINED !
354     FI
355
356     IF DIRECTION = BACKWARD
357     THEN
358         PRESENT_COLUMN -= 1
359     ELSE
360         PRESENT_COLUMN += 1
361     FI
362
363     IF PRESENT_COLUMN = 0 THEN PRESENT_COLUMN += 1 FI
364
365
366
367     END PRINT
368
369
370

```

```
371     END SPINWRITER
END OF ZCODE GENERATION
    0 ERROR(S)      0 WARNING(S)
```

PLZSYS 2.02

```

1  DIABLO  MODULE
2
3  !   Extended 3/3/79 to allow for subscripting and superscripting !
4  !   Also for operator controlled page_waits, and auto formfeeds.  !
5
6  CONSTANT
7
8
9          PRINTER_BUFFER_SIZE      := 158
10         PITCH                     := 12
11         SS                         := 120/PITCH
12
13         ASCII_ETX                  := %03
14         ASCII_ACK                  := %06
15         ASCII_ESC                  := %1B
16         ASCII_FF                   := %0C
17         ASCII_BS                   := %08
18         ASCII_UL                   := '_'
19         ASCII_SPACE                := ' '
20         ASCII_US                   := %1F
21         ASCII_TAB                  := %09
22
23         PARITY_MASK                := %7F
24
25         FORWARD                   := 1
26         BACKWARD                   := 0
27
28         BLACK                      := %01
29         BOLD                       := %02
30         UNDERLINE                  := %04
31
32         SUPERSCRIP                  := %08
33         NOT_SUPERSCRIP              := NOT SUPERSCRIP
34         SUBSCRIPT                  := %10
35         NOT_SUBSCRIPT               := NOT SUBSCRIPT
36
37         TRUE                       := 1
38         FALSE                      := 0
39
40     INTERNAL
41
42         LINE_COUNT                  BYTE
43         COLOUR                      BYTE
44         HMI                        BYTE
45
46         SUPERSCRIP_FLAG             BYTE
47         SUBSCRIPT_FLAG              BYTE
48
49         LAST_SCRIPT_STATE           BYTE
50
51
52     EXTERNAL

```

```

53
54         OUTCH2                PROCEDURE ( BYTE )
55         INCH2                  PROCEDURE RETURNS ( BYTE )
56
57         PAGE_WAIT              PROCEDURE
58
59         PAGE_WAIT_FLAG         BYTE
60         AUTO_FF_FLAG          BYTE
61
62     GLOBAL
63
64         PRINTER_WIDTH          BYTE := 158
65         DIRECTION              BYTE
66         PRESENT_COLUMN         BYTE
67         AUTO_FF_LINE_COUNT     BYTE := 63
68         BYTE_COUNT             BYTE
69
70
71
72     SEND_ETX                    PROCEDURE
73     ENTRY
74         OUTCH2 ( ASCII_ETX )
75         BYTE_COUNT := 0
76     END SEND_ETX
77
78
79     WAIT_FOR_ACK                PROCEDURE
80     ENTRY
81         DO
82             IF ( INCH2 AND PARITY_MASK ) = ASCII_ACK
83                 THEN
84                     RETURN
85             FI
86         OD
87     END WAIT_FOR_ACK
88
89
90     SYNCH                       PROCEDURE
91     ENTRY
92         IF BYTE_COUNT < PRINTER_BUFFER_SIZE - 10 THEN RETURN FI
93         SEND_ETX
94         WAIT_FOR_ACK
95     END SYNCH
96
97
98     SEND                        PROCEDURE ( CODE BYTE )
99     ENTRY
100        IF BYTE_COUNT > PRINTER_BUFFER_SIZE - 10
101            THEN
102                SEND_ETX
103                WAIT_FOR_ACK
104            FI
105        BYTE_COUNT += 1

```

```
106          OUTCH2 ( CODE )
107  END SEND
108
109
110  FORMFEED          PROCEDURE
111  ENTRY
112      SEND ( ASCII_FF )
113      LINE_COUNT := 0
114      LAST_SCRIPT_STATE := 0
115      SUPERSCRIPIT_FLAG := FALSE
116      SUBSCRIPT_FLAG := FALSE
117
118      IF PAGE_WAIT_FLAG = TRUE THEN PAGE_WAIT FI
119
120  END FORMFEED
121
122
123  REQUEST_FORWARD   PROCEDURE
124  ENTRY
125      IF DIRECTION = FORWARD THEN RETURN FI
126      SYNCH
127      OUTCH2 ( ASCII_ESC )
128      OUTCH2 ( '5' )
129      BYTE_COUNT += 2
130      DIRECTION := FORWARD
131  END REQUEST_FORWARD
132
133
134  REQUEST_BACKWARD  PROCEDURE
135  ENTRY
136      IF DIRECTION = BACKWARD THEN RETURN FI
137      SYNCH
138      OUTCH2 ( ASCII_ESC )
139      OUTCH2 ( '6' )
140      BYTE_COUNT += 2
141      DIRECTION := BACKWARD
142  END REQUEST_BACKWARD
143
144
145  REQUEST_BLACK      PROCEDURE
146  ENTRY
147      SYNCH
148      OUTCH2 ( ASCII_ESC )
149      OUTCH2 ( 'B' )
150      BYTE_COUNT += 2
151      COLOUR := BLACK
152  END REQUEST_BLACK
153
154
155  REQUEST_RED        PROCEDURE
156  ENTRY
157      SYNCH
158      OUTCH2 ( ASCII_ESC )
```

```

159         OUTCH2 ( 'A' )
160         BYTE_COUNT += 2
161         COLOUR := 0
162     END REQUEST_RED
163
164
165     LINEFEED                                PROCEDURE
166     ENTRY
167
168         IF AUTO_FF_FLAG = TRUE
169             THEN
170                 IF LINE_COUNT >= AUTO_FF_LINE_COUNT
171                     THEN
172                         FORMFEED
173                         RETURN
174                 FI
175             FI
176             SEND ( '%L' )
177             LINE_COUNT += 1
178     END LINEFEED
179
180
181     REQUEST_POS_HALF_LINE                    PROCEDURE
182     ENTRY
183         SYNCH
184         OUTCH2 ( ASCII_ESC )
185         OUTCH2 ( 'U' )
186         BYTE_COUNT += 2
187     END REQUEST_POS_HALF_LINE
188
189
190     REQUEST_NEG_HALF_LINE                    PROCEDURE
191     ENTRY
192         SYNCH
193         OUTCH2 ( ASCII_ESC )
194         OUTCH2 ( 'D' )
195         BYTE_COUNT += 2
196     END REQUEST_NEG_HALF_LINE
197
198
199     DEFINE_HMI                                PROCEDURE ( SPACING BYTE )
200     ENTRY
201         SYNCH
202         OUTCH2 ( ASCII_ESC )
203         OUTCH2 ( ASCII_US )
204         OUTCH2 ( SPACING + 1 )
205         BYTE_COUNT += 3
206     END DEFINE_HMI
207
208
209     SEND_BOLD_CHAR                            PROCEDURE ( CHAR BYTE ATTRIBUTE BYTE )
210     ENTRY
211         DEFINE_HMI ( 0 )

```

```

212         SEND ( CHAR )
213         IF ( ATTRIBUTE AND UNDERLINE ) <> 0
214             THEN
215                 SEND ( ASCII_UL )
216         FI
217         DEFINE_HMI ( 1 )
218         SEND ( CHAR )
219         DEFINE_HMI ( SS-1 )
220         SEND ( CHAR )
221         DEFINE_HMI ( SS )
222
223     END SEND_BOLD_CHAR
224
225     ABSOLUTE_TAB                PROCEDURE ( COLUMN BYTE )
226
227         LOCAL  RESIDUE BYTE
228             DIR BYTE
229
230
231     ENTRY
232
233         IF COLUMN > 156 THEN COLUMN := 156 FI
234         SYNCH
235         OUTCH2 ( ASCII_ESC )
236         OUTCH2 ( ASCII_TAB )
237         BYTE_COUNT += 2
238
239         PRESENT_COLUMN := COLUMN
240
241         IF COLUMN <= 126
242             THEN
243                 OUTCH2 ( COLUMN )
244                 BYTE_COUNT += 1
245                 RETURN
246         FI
247
248         RESIDUE := COLUMN - 126
249         OUTCH2 ( 126 )
250         BYTE_COUNT += 1
251         DIR := DIRECTION
252         REQUEST_FORWARD
253
254     DO
255         IF RESIDUE = 0
256             THEN
257                 IF DIR = BACKWARD
258                     THEN
259                         REQUEST_BACKWARD
260                 FI
261                 RETURN
262             FI
263         FI
264         SEND ( ASCII_SPACE )

```

```
265             RESIDUE -= 1
266         OD
267
268     END ABSOLUTE_TAB
269
270
271     PRINT             PROCEDURE ( CHAR BYTE ATTRIBUTE BYTE )
272     ENTRY
273
274
275     DO
276         IF ATTRIBUTE AND (SUBSCRIPT OR SUPERSCRIPIT)
277             = LAST_SCRIPT_STATE
278             THEN
279                 EXIT
280         FI
281
282         IF ATTRIBUTE AND SUPERSCRIPIT <> 0
283             THEN
284                 IF SUBSCRIPT_FLAG = TRUE
285                     THEN
286                         REQUEST_NEG_HALF_LINE
287                         SUBSCRIPT_FLAG := FALSE
288                 FI
289                 REQUEST_NEG_HALF_LINE
290                 SUPERSCRIPIT_FLAG := TRUE
291                 EXIT
292         FI
293
294         IF ATTRIBUTE AND SUBSCRIPT <> 0
295             THEN
296                 IF SUPERSCRIPIT_FLAG = TRUE
297                     THEN
298                         REQUEST_POS_HALF_LINE
299                         SUPERSCRIPIT_FLAG := FALSE
300                 FI
301                 REQUEST_POS_HALF_LINE
302                 SUBSCRIPT_FLAG := TRUE
303                 EXIT
304         FI
305
306         IF SUPERSCRIPIT_FLAG = TRUE
307             THEN
308                 REQUEST_POS_HALF_LINE
309                 SUPERSCRIPIT_FLAG := FALSE
310                 EXIT
311         FI
312
313         REQUEST_NEG_HALF_LINE
314         SUBSCRIPT_FLAG := FALSE
315         EXIT
316     OD
317
```

```

318
319     LAST_SCRIPT_STATE
320         := ATTRIBUTE AND ( SUPERSCRIPIT OR SUBSCRIPT )
321     IF ( ATTRIBUTE AND BLACK ) <> COLOUR
322     THEN
323         IF ( ATTRIBUTE AND BLACK ) = BLACK
324         THEN
325             REQUEST_BLACK
326         ELSE
327             REQUEST_RED
328         FI
329     FI
330
331     IF CHAR > ASCII_SPACE
332     THEN
333         IF ATTRIBUTE AND BOLD <> 0
334         THEN
335             SEND_BOLD_CHAR ( CHAR ATTRIBUTE )
336         ELSE
337             IF ( ATTRIBUTE AND UNDERLINE ) <> 0
338             THEN
339                 DEFINE_HMI ( 0 )
340                 SEND ( CHAR )
341                 DEFINE_HMI ( SS )
342                 SEND ( ASCII_UL )
343             ELSE
344                 SEND ( CHAR )
345             FI
346         FI
347     ELSE
348         SEND ( CHAR ) ! SPACES WILL NOT BE UNDERLINED !
349     FI
350
351     IF DIRECTION = BACKWARD
352     THEN
353         PRESENT_COLUMN -= 1
354     ELSE
355         PRESENT_COLUMN += 1
356     FI
357
358     IF PRESENT_COLUMN = 0 THEN PRESENT_COLUMN += 1 FI
359
360
361     END PRINT
362
363
364
365     END DIABLO
366     END OF ZCODE GENERATION
        0 ERROR(S)      0 WARNING(S)

```

Call to SYSTEM	CALL.SYSTEM	PAGE 1
LOC OBJ CODE M STMT	SOURCE STATEMENT	ASM 5.8
	1 *H Call to SYSTEM	
	2	
	3 *I PLZ.INTERFACE.MACROS	
	133 *LIST ON	
	134 *MACLIST OFF	
	135 ; Date_code:- October 22nd. 1978.	
	136	
	137 ; This interface module allows a PLZ programme to make	
	138 ; calls to RIO.	
	139	
	140 ; Declare CALRIO PROCEDURE (VECTOR_PTR ^byte)	
	141 ; RETURNS (COMPLETION_CODE byte)	
	142	
	143 ; There must be a standard RIO request vector stored	
	144 ; starting at the location beginning VECTOR_PTR^	
	145	
	146 global CALRIO calrio	
	147	
	148 SYSTEM EQU 1403H	
	149	
	150	
	151 CALRIO	
	152 calrio	
0000	153 ENT 0 ; no locals	
	154	
0008	155 LDHL 4 ; put RIO vector address into hl	
000E E5	156 push hl	
000F FDE1	157 pop iy ; and then where it should be	
	158	
0011 DDE5	159 push ix ; save it	
0013 CD0314	160 call SYSTEM ; go and do the necessary	
0016 DDE1	161 pop ix ; restore it	
	162	
0018 FD7E0A	163 ld a,(iy+10) ; get the completion code	
001B	164 STA 6 ; and place it as return parameter	
	165	
001E	166 RTN 0 2 ; return to caller. 0 locals,2 I/P param bytes	
	167	
	168	
	169 END	

RIO.IO.INTERFACE				RIO.IO.INTERFACE		PAGE 1
LOC	OBJ	CODE	M	STMT	SOURCE STATEMENT	ASM 5.8
				1	*H RIO.IO.INTERFACE	
				2		
				3		
				4	; Date_code:- October 31st. 1978	
				5		
				6	; This interface module receives I/O calls from RIO, and	
				7	; passes the IY register value to the called programme	
				8	; as a single parameter.	
				9		
				10	; The intention of the module is to act as an interface	
				11	; to enable I/O drivers to be written largely in PLZ.	
				12		
				13		
				14	EXTERNAL PLZDVR	
				15		
				16	GLOBAL ENTRY entry	
				17		
				18	ENTRY	
				19	entry	
				20		
0000	FDE5			21	push iy	; the actual parameter
				22		
0002	FD223000	R		23	LD (IY_SAV),IY	; save iy
				24		
0006	CD0000	X		25	call PLZDVR	; pass control to the driver proper
				26		
0009	FD2A3000	R		27	LD IY,(IY_SAV)	; restore iy
				28		
000D	FDCB0146			29	bit 0,(iy+1)	; was it int.req ?
0011	FD7EOA			30	ld a,(iy+10)	; get c_code
0014	200E			31	jr nz,intreq	
0016	FE80			32	cp 80h	; was it good ?
0018	C8			33	ret z	; if so, go back quietly
				34	getera	
0019	FD6609			35	ld h,(iy+9)	
001C	FD6E08			36	ld l,(iy+8)	
				37	jmpret	
001F	7C			38	ld a,h	
0020	B5			39	or l	
0021	C8			40	ret z	; rtn add field was zero
0022	C1			41	pop bc	; balance stack
0023	E9			42	jp (hl)	
0024	FE80			43	intreq cp 80h	
0026	20F1			44	jr nz,getera	
0028	FD6607			45	ld h,(iy+7)	
002B	FD6E06			46	ld l,(iy+6)	
002E	18EF			47	jr jmpret	; check cra field
				48		
				49		
				50	IY_SAV	

```
0030          51      defs 2
          52
          53      end
```

SIB CH2 input/output	SIB	PAGE 1
LOC OBJ CODE M STMT SOURCE STATEMENT		ASM 5.8
1		*H SIB CH2 input/output
2		*P 50
3		
4		*I PLZ.INTERFACE.MACROS
134		*LIST ON
135		*MACLIST OFF
136		
137		; Date_code:- October 23rd. 1978.
138		
139		; This module contains the routines which enable a PLZ
140		; programme to be involved with I/O through channel 2
141		; of an SIB installed in the system.
142		
143		; The calling programme should contain the following
144		; declarations:-
145		
146		; EXTERNAL
147		;
148		; OUTCH2 PROCEDURE (BYTE)
149		; ! send, with wait ready !
150		
151		; INCH2 PROCEDURE RETURNS (BYTE)
152		; ! input, with wait ready !
153		
154		; INCH2E PROCEDURE RETURNS (BYTE)
155		; ! input, wait ready, echo !
156		
157		; STACH2 PROCEDURE RETURNS (BYTE)
158		; ! read status of USART2 !
159		
160		; SNDCH2 PROCEDURE (BYTE)
161		; ! send, without wait ready !
162		
163		; RDCH2 PROCEDURE RETURNS (BYTE)
164		; ! read, without wait ready !
165		
166		; SETCH2 PROCEDURE
167		; ! set up USART2 + CTC baud rate !
168		
169		
170		global OUTCH2 INCH2 INCH2E
171		global STACH2 SNDCH2 RDCH2 SETCH2
172		global outch2 inch2 inch2e
173		global stach2 sndch2 rdch2 setch2

SIB CH2 input/output				SIB	PAGE 2	
LOC	OBJ	CODE	M	STMT	SOURCE STATEMENT	ASM 5.8
				174	*E	
				175		
				176	SETCH2	
				177	setch2	
0000				178	ENT 0	
				179		
0008	CD9400	R		180	call BAUDR	; set up the baud rate
000B	CD7B00	R		181	call SUART2	; set up USART-2
				182		
000E				183	RTN 0 0	; no locals, no IN parameters
				184		
				185		
				186	OUTCH2	
				187	outch2	
0011				188	ENT 0	
				189		
0019				190	LDA 4	; get the code for issuing
001C	CDA600	R		191	call OUSIB2	; send it with wait ready
				192		
001F				193	RTN 0 2	; no locals, 2 bytes IN
				194		
				195		
				196	INCH2	
				197	inch2	
0024				198	ENT 0	
				199		
002C	CD9D00	R		200	call INSIB2	; get the code, with wait ready
002F				201	STA 4	; place it as return parameter
				202		
0032				203	RTN 0 0	; no locals, no IN parameters
				204		
				205		
				206	INCH2E	
				207	inch2e	
0035				208	ENT 0	
				209		
003D	CD9D00	R		210	call INSIB2	; get the code, with wait ready
0040	CDA600	R		211	call OUSIB2	; echo it, with wait ready
0043				212	STA 4	; place code as return parameter
				213		
0046				214	RTN 0 0	; no locals, no IN parameters
				215		
				216		
				217	STACH2	
				218	stach2	
0049				219	ENT 0	
				220		
0051	DB91			221	in a,(USART2+1)	; get the status reg contents
0053				222	STA 4	; place it as return parameter
				223		

SIB CH2 input/output		SIB		PAGE 3
LOC	OBJ CODE M STMT	SOURCE	STATEMENT	ASM 5.8
0056		224	RTN 0 0	; no locals, no IN parameters
		225		
		226		
		227	SNDCH2	
		228	sndch2	
0059		229	ENT 0	
		230		
0061		231	LDA 4	; get the code to be sent
0064	D390	232	out (USART2),a	; send it immediately
		233		
0066		234	RTN 0 2	; no locals, 2 bytes IN parameters
		235		
		236		
		237	RDCH2	
		238	rdch2	
006B		239	ENT 0	
		240		
0073	DB90	241	in a,(USART2)	; get data reg contents
0075		242	STA 4	; place it as return parameter
		243		
0078		244	RTN 0 0	; no locals, no IN parameters
		245		
		246	*I SIB.CONTROL	

SIB CONTROL		SIB		PAGE 4
LOC	OBJ CODE M STMT	SOURCE	STATEMENT	ASM 5.8
		247	*H SIB CONTROL	
		248		
		249		
		250		
		251	*****	
		252	; RESET AND SET UP USART2	
		253	*****	
		254		
007B	0E91	255	SUART2 LD C,USART2+1 ; USART2 - CONTROL REGISTER	
007D	AF	256	XOR A	
007E	ED79	257	OUT (C),A ; THREE INTERNAL RESETS = EXT RESET	
0080	ED79	258	OUT (C),A	
0082	ED79	259	OUT (C),A	
0084	3E40	260	LD A,40H ; INT RESET	
		261	; ENTER MODE INSTRUCTION FORMAT	
0086	ED79	262	OUT (C),A	
0088	3ECE	263	LD A,OCEH ; 8xDATA + 2xSTOP BITS	
		264	; NO PARITY, 16xBAUD RATE	
008A	ED79	265	OUT (C),A	
008C	3E37	266	LD A,37H ; RTS, ERROR RESET	
		267	; REC ENABLE, DTR, XMIT ENABLE	
008E	ED79	268	OUT (C),A	
0090	0D	269	DEC C	
0091	ED78	270	IN A,(C) ; CLEAR OUT GARBAGE CHARACTER	
0093	C9	271	RET	
		272		
		273		
		274	USART0 EQU 8CH	
		275	USART1 EQU 8EH	
		276	USART2 EQU 90H	
		277	USART3 EQU 92H	
		278		
		279		
		280		
		281		
		282	*****	
		283	; SET UP CTC1 TO GENERATE THE BAUD RATE	
		284	*****	
		285		
0094	3E07	286	BAUDR LD A,TIMMOD ; TIMER MODE ETC	
0096	D381	287	OUT (CTC1),A	
0098	3E04	288	LD A,RATEO ; WITH RATE IN RATEO	
009A	D381	289	OUT (CTC1),A	
009C	C9	290	RET	
		291		
		292		
		293	CTC0 EQU 80H ; ADDRESS OF CTC0	
		294	CTC1 EQU 81H ; ADDRESS OF CTC1	
		295	CTC2 EQU 82H ; ADDRESS OF CTC2	
		296	CTC3 EQU 83H ; ADDRESS OF CTC3	

SIB CONTROL		SIB		PAGE 5
LOC	OBJ CODE M STMT	SOURCE	STATEMENT	ASM 5.8
		297		
		298	TIMMOD EQU 07H	; TIMER MODE, PRESCALER=16
		299		; INT.DISABLED, RESET
		300		
		301	RATE0 EQU 4	; FOR 1200 BAUDS
		302		
		303		
		304		
		305		;*****
		306		; ELEMENTARY CHARACTER LEVEL I/O
		307		;*****
		308		
009D	DB91	309	INSIB2 IN A,(USART2+1)	; GET STATUS REGISTER
009F	CB4F	310	BIT RXRDY,A	; IS THE RECEIVER FLAG SET
00A1	28FA	311	JR Z,INSIB2	; IF NOT, WAIT FOR IT
00A3	DB90	312	IN A,(USART2)	; GET CONTENT OF DATA REGISTER
00A5	C9	313	RET	
		314		
		315		
00A6	F5	316	OUSIB2 PUSH AF	; SAVE CHARACTER
00A7	DB91	317	BZY IN A,(USART2+1)	; GET STATUS REGISTER
00A9	CB47	318	BIT TXRDY,A	; TEST THE TRANSMITTER READY FLAG
00AB	28FA	319	JR Z,BZY	; IF UNREADY THEN WAIT
00AD	F1	320	POP AF	; RESTORE CHARACTER CODE
00AE	D390	321	OUT (USART2),A	; SEND IT
00B0	C9	322	RET	
		323		
		324		
		325	RXRDY EQU 1	; RECEIVER READY BIT
		326	TXRDY EQU 0	; TRANSMITTER READY BIT
		327		

```
                                PLZ.INTERFACE.MACROS                PAGE 1
LOC  OBJ CODE M STMT SOURCE STATEMENT                               ASM 5.8

    1 ;*LIST OFF
    2
    3
    4 ; Mark-stack macro:
    5
    6 ; Allocate room on stack for out parameters
    7 ; before a procedure call.
    8
    9 ; Optimise the code when 0,1,or 2 parameters
   10 ; ie. 0,2 or 4 bytes.
   11
   12 MST      macro #n          ; #n is in BYTES ***
   13          cond (#n=2).or.(#n=4)
   14          push hl
   15          cond #n=4
   16          push hl
   17          ende
   18          cond .not.(#n=0).and..not.(#n=2).and..not.(#n=4)
   19          ld hl,-#n
   20          add hl,sp
   21          ld sp,hl
   22          ende
   23          endm
```

```
                                PLZ.INTERFACE.MACROS                PAGE 2
LOC  OBJ CODE M STMT SOURCE STATEMENT                            ASM 5.8

24  *E
25
26  ; Procedure entry:
27
28  ; Allocate locals on stack ( No. of bytes )
29
30  ; Optimise when 0,2,or 4 bytes.
31
32  ENT    macro #n          ; #n is in BYTES ***
33        push ix
34        ld ix,0
35        add ix,sp
36        cond (#n=2).or.(#n=4)
37        push hl
38        cond #n=4
39        push hl
40        endc
41        cond .not.(#n=0).and..not.(#n=2).and..not.(#n=4)
42        ld hl,-#n
43        add hl,sp
44        ld sp,hl
45        endc
46        endm
```

```

                                PLZ.INTERFACE.MACROS
                                PAGE 3
LOC  OBJ CODE M STMT SOURCE STATEMENT                                ASM 5.8

47  *E
48
49  ; Procedure return:
50
51  ; Deallocate locals ( bytes ) and IN parameters.
52
53  ; Optimise when 0,2,or 4 bytes.
54
55  RTN      macro #L, #n      ; #L, #n are in BYTES ***
56
57          cond #L
58          ld sp,ix
59          endc
60
61          pop ix
62
63          cond #n=0
64          ret
65          endc
66
67          cond (#n=2).or.(#n=4)
68          pop hl
69          pop de
70
71          cond #n=4
72          pop de
73          endc
74
75          cond (#n=2).or.(#n=4)
76          jp (hl)
77          endc
78
79          cond .not.(#n=0).and..not.(#n=2).and..not.(#n=4)
80          pop de
81          ld hl,#n
82          add hl,sp
83          ld sp,hl
84          ex de,hl
85          jp (hl)
86          endc
87
88          endm

```

```

                                PLZ.INTERFACE.MACROS
                                PAGE 4
LOC  OBJ CODE M STMT SOURCE STATEMENT                                ASM 5.8

89  *E
90
91  ; Macros for accessing locals and parameters
92  ; from the stack.
93
94  ; This is only a small selection.
95
96
97
98  ; Load hl from #n ( offset of word variable from ix )
99
100 LDHL  macro #n
101        ld l,(ix+#n)
102        ld h,(ix+#n+1)
103        endm
104
105
106
107 ; Store hl into #n ( offset of word variable from ix )
108
109 STHL  macro #n
110        ld (ix+#n),l
111        ld (ix+#n+1),h
112        endm
113
114
115
116 ; Load A from #n ( offset of byte variable from ix )
117
118 LDA   macro #n
119        ld a,(ix+#n)
120        endm
121
122
123
124 ;Store A into #n ( offset of byte variable from ix )
125
126 STA   macro #n
127        ld (ix+#n),a
128        endm
129
130 *LIST ON
```

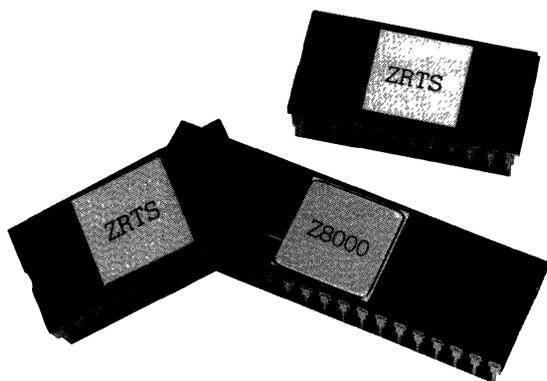
ZRTS™ 8000 Zilog Real-Time Software for the Z8000 Microprocessor



Product Description

Preliminary

June 1981



■ Real-time Multi-tasking Software Components

- Synchronization of multiple tasks
- Interrupt-driven priority scheduling
- Real-time response
- Dynamic memory allocation

■ Modular and Flexible Design

- Efficient memory utilization
- 4K byte PROMable kernel
- Support for Z8001 and Z8002 16-bit microprocessors
- Configurable via linkable modules

■ Versatile Base for Z8000™ System Designs

- Segmented/non-segmented tasks
- System/normal mode tasks
- Uses standard Zilog calling conventions

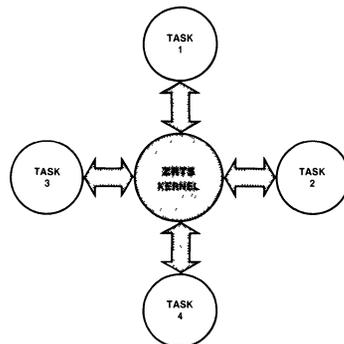
■ Easy-To-Use System Generator

- High-level configuration language
- Supports a wide variety of hardware configurations
- Easily changed control parameters allow system optimization
- Eliminates the requirement for intimate knowledge of system internal structure

OVERVIEW

Zilog's Real Time Software (ZRTS) provides a set of modular software components that allows quick and easy implementation of customized operating systems for all members of the Z8000 16-bit microprocessor family. In effect, ZRTS extends the instruction set of the Z8000, adding easy-to-use commands that give the Z8000 the capability for managing real-time, multi-tasking applications.

The ZRTS package consists of a small real-time, multi-tasking executive program, the Kernel, and a System Configurator. The Kernel provides synchronization and control of multiple events occurring in a real-time environment. All major real-time functions are available—task synchronization, interrupt-driven priority scheduling, intertask communication, real-time response, and dynamic memory allocation. The System Configurator is a language processor that allows the target operating system to be defined in high-level terms using the ZRTS Configuration Language (ZCL).



These functions greatly simplify the tasks of the designer, allowing development efforts to be concentrated on the application, instead of on real-time coordination, task management problems, and complicated system generations. ZRTS provides a modular and flexible development tool that serves as a versatile base for Z8000 system designs. The Kernel requires only 4K bytes of either PROM or RAM memory, thus allowing configurations for a wide variety of target systems, while producing a memory-efficient, cost-effective end product.

FUNCTIONAL DESCRIPTION

The Concepts. ZRTS is both easy-to-learn and easy-to-use. Only a few simple concepts need to be understood before designing begins.

Tasks. Tasks are the components comprising a real-time application. Each task is an independent program that shares the processor with the other tasks in the system. Tasks provide a mechanism that allows a complicated application to be subdivided into several independent, understandable, and manageable units.

Semaphores. Semaphores provide a low overhead facility for allowing one task to signal another. Semaphores can be used for indicating the availability of a shared resource, timing pulses or event notification.

Exchanges and Messages. Exchanges and Messages provide the mechanism for one task to send data to another. A Message is a buffer of data, while an Exchange serves as a mailbox at which tasks can wait for Messages and to which Messages are sent and held.

The ZRTS Kernel. The Kernel is the basic building block of ZRTS and performs the management functions for tasks, semaphores, the real-time clock, memory and interrupts. The Kernel also provides for task-to-task communications via Exchanges and Messages. All requests for Kernel operations are made via system call instructions with parameters in registers, according to the standard Zilog calling conventions.

Task Management. One of the main activities of the Kernel is to arbitrate the competition that results when several tasks each want to use the processor. Each task has a unique task descriptor that is managed by the Kernel. The data contained in the descriptor include the task name, priority, state and other pertinent status information. ZRTS supports any number of tasks, limited only by the memory available to accommodate the task descriptors and stacks.

The Kernel maintains a queue of all active tasks on the system. Each task is scheduled for processor time based on its priority. The highest-priority task that's ready to run gains control of the CPU; other tasks are queued. Tasks can be prioritized up to 32767 levels, with round-robin scheduling among tasks with the same priority.

Tasks can run either segmented or non-segmented code, in either normal or system mode. The numerous operations that may be performed on tasks are listed in *Table 1*.

TABLE 1.

TASK MANAGEMENT

T__Census	Provides the status of tasks in the system.
T__Create	Creates a task dynamically
T__Destroy	Removes a dynamically created task
T__Lock	Allows a task to take exclusive control of the CPU.
T__Reschedule	Changes the priority of a task.
T__Resume	Activates a suspended task.
T__Suspend	Suspends another task.
T__Unlock	Releases exclusive control of the CPU for other tasks.
T__Wait	Suspends task execution

SEMAPHORE MANAGEMENT

Sem__Clear	Clears semaphore queue and reinitializes a semaphore.
Sem__Create	Creates a semaphore dynamically
Sem__Destroy	Removes a dynamically created semaphore.
Sem__Signal	Signals a semaphore, increments the counter.
Sem__Test	Tests a semaphore for a signal.
Sem__Wait	Causes a task to wait until a semaphore is signaled, decrements the counter.

CLOCK MANAGEMENT

Clk__Delay__Absolute	Places a task on the clock queue waiting for absolute time.
Clk__Delay__Interval	Places a task on the clock queue waiting for passage of an interval of time
Clk__Set	Sets the real-time clock.
Clk__Time	Reads the clock.

MEMORY MANAGEMENT

Mem__Census	Provides status of the memory resource.
Alloc	Dynamically allocates memory.
Release	Releases allocated memory.

INTER-TASK COMMUNICATION

M__Acquire	Gets a message from an exchange pool and assigns a destination or a reply exchange to it.
M__Assign	Assigns a new source and destination to an existing message.
M__Create	Creates a message dynamically.
M__Destroy	Removes a dynamically created message.
M__Get__Descriptor	Gets message's descriptor information
M__Read	Reads the message data.
M__Receive	Receives a message from an exchange
M__Receive__Wait	Waits to receive a message from an exchange.
M__Release	Returns a message to the exchange pool.
M__Reply	Sends a message back to destination exchange.
M__Send	Sends a message to an exchange.
M__Write	Changes message data.
X__Create	Dynamically creates an exchange with a pool of messages.
X__Destroy	Removes a dynamically created exchange.

Semaphore Management. The Kernel provides semaphore management for synchronizing interacting tasks. A typical use of semaphores is to provide mutual exclusion of a shared resource. When a resource is to be used by only one task at a time, a semaphore with a counter of 1 controls the resource. Every task requiring the resource must first wait on that semaphore. Since the counter is 1, only one task will acquire the resource. The others will be queued on the semaphore and suspended until the semaphore is signaled that the resource is once again available. At that time, the first task on the semaphore queue will be made ready to run and can use the resource. After all tasks have acquired the resource and signaled the completion of their use, the semaphore returns to its original state with a counter of 1. Counters greater than one are useful when there are a number of similar resources, (i.e., three tape drives, four I/O buffers, etc.).

In ZRTS, a semaphore can count up to 32676 signals. The commands provided by the Kernel to manage semaphores are listed in *Table 1*.

Clock Management. ZRTS operates with a real-time clock that generates interrupts at a hardware-dependent rate. It is used for timed waits, timeouts, and round-robin scheduling. All times are given in number of ticks. The clock may be manipulated by the set of commands provided by the Kernel that are listed in *Table 1*.

Memory Management. Storage for ZRTS data structures is allocated either statically at system generation time, or dynamically at run time. Dynamic allocation occurs via a system call that specifies the attributes of the structure to be created and returns a name that can be used to refer to the structure. Memory is allocated in 256-byte increments, and can be released using a system call.

The storage allocator can also be called directly to obtain blocks of memory up to 64K bytes long, which can be used by the task for any purpose.

Interrupt Management. Interrupt-handling routines are provided for system calls, non-vectored interrupts and a hardware clock. The user must provide interrupt routines for whatever other vectored interrupts are included in the target system.

ZRTS can switch control to a task waiting for an external event within 500-microseconds after the occurrence of the event. This is based on the worst case with a 4MHz Z8000. A more typical response time would be

TABLE 2.

CONSTANTS	Specifies system constants.
EXCHANGES	Defines the characteristics of application exchanges.
FILES	Indicates additional files to be included in the configuration link.
HARDWARE	Describes the target hardware configuration—Z8001, Z8002, or Development Module.
INITIALIZATION	Specifies routines that are to execute prior to beginning execution of the first task.
INTERRUPT	Associates an interrupt routine with an interrupt vector or trap and system call-handlers. Provides the facilities to specify a NVI interrupt-handler that will be called from the system NVI-handler routine.
MEMORY	Specifies the memory configuration and identifies where sections are to be placed (i.e., CODE, DATA, ...).
SECTIONS	Allows modules to be placed in a specific section, overriding the standard assignment conventions.
SEMAPHORES	Defines the characteristics of application semaphores.
SWITCHES	Allows flags that control the system generation operation to be set.
TASKS	Defines the characteristics of application tasks.

250-microseconds. Quicker service of interrupts is possible through the use of user-written routines.

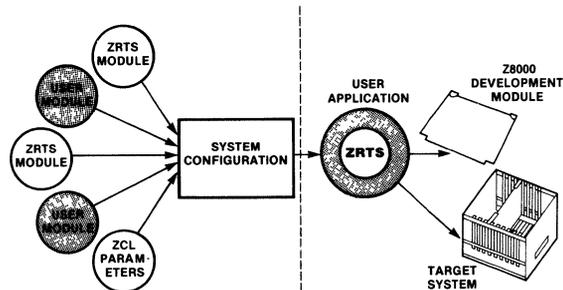
Inter-task Communication. The Kernel provides the capability for tasks to exchange information. This communication process occurs when one task sends a Message to an Exchange and another task receives the Message.

A Message contains a length indicator, a buffer with a variable amount of data, and a code that identifies the Message type. The Exchange is a system data structure that consists of a queue for Messages sent but not yet received, a semaphore on which a task can wait for a Message, and an optional "pool" list from which

Messages can be obtained quickly.

ZRTS provides several commands for inter-task communications. These are listed in *Table 1*.

ZRTS Configuration Language (ZCL). Since ZRTS's modular design leads to so many different configurations, a simple facility for generating the target operating system is a critical part of the ZRTS package. The ZRTS Configuration Language (ZCL) provides an easy-to-use means for generating the target system. Using ZCL, the designer can specify hardware information, software parameters, linkage information, and system data structures in high-level terms.



Development Environment

ZCL unburdens the user of the necessity to learn the details of the ZRTS internal structures. System data structures can be generated simply by specifying the appropriate parameters. The ZCL syntax is free-format with comments allowed to make the configuration commands more readable and maintainable.

ZCL input is comprised of a number of descriptive sections, each containing the details of the target operating system. The functions of these sections are described in *Table 2*. A sample system generation using ZCL is illustrated in *Figure 1*.

Development Environment. Application modules for ZRTS can be developed on any Zilog Z80 or Z8000-based development system and then down-loaded into a Zilog Development Module or a customized target system.

Subroutine libraries are provided for making ZRTS systems calls from programs written in PLZ/SYS, PLZ/ASM and C. Register usage in the system calls is compatible with the Zilog standard.

When using a Development Module, the Debugger can be used with the ZRTS modules for testing purposes. After the application is debugged, the system can be easily reconfigured for the final target hardware.

```

SWITCHES:
  APPLICATION

HARDWARE:
  Z8002

INTERRUPTS:

CONSTANTS:
  MINIMUM_SYSTEM_STACK_SIZE = 512;

FILES:
  REAL_TIME_CLOCK;

MEMORY:
  CODE = { $8000..$FFFF };
  DATA = { $9000..$9FFF };
  FREE_MEMORY = { $F000..$FFFF };

SECTIONS:

INITIALIZATION:

TASKS:
  input_handler_task = {entry = INPUT_HANDLER,      priority = 10};
  tim_display_task   = {entry = TIME_DISPLAY,       priority = 20};
  egg_timer_task     = {entry = EGG_TIMER,          priority = 20};
  alarm_task         = {entry = ALARM,              priority = 20};
  one_second_task    = {entry = ONE_SECOND_GENERATOR, priority = 30};

SEMAPHORES:
  ONE_SECOND_SEMAPHORE;
  TIME_DISPLAY_ENABLE_SEMAPHORE;

EXCHANGES:
  INPUT_HANDLER_ETE_EXCHANGE = {number_of_messages = 1,
                                message_size       = 8};
  EGG_TIMER_ENABLE_EXCHANGE  = {number_of_messages = 0};
  INPUT_HANDLER_A_EXCHANGE  = {number_of_messages = 1,
                                message_size       = 8};
  ALARM_EXCHANGE             = {number_of_messages = 0};

```

Figure 1. ZCL Sample Input.

ORDERING INFORMATION

Description

ZRTS/8001 Zilog Real Time Software for the Z8001
 ZRTS/8002 Zilog Real Time Software for the Z8002

Prerequisites

Zilog Development System
 MCZ/1, PDS, ZDS Series or Z-LAB 8000 (Requires Software License)



Peripheral controller chip ties into 8- and 16-bit systems

Based on one-chip-microcomputer architecture, universal peripheral controller comes with either multiplexed or nonmultiplexed address and data lines, provides ROM-less and prototyping packages for product development

by John Banning and Pat Lin, *Zilog Inc., Cupertino, Calif*

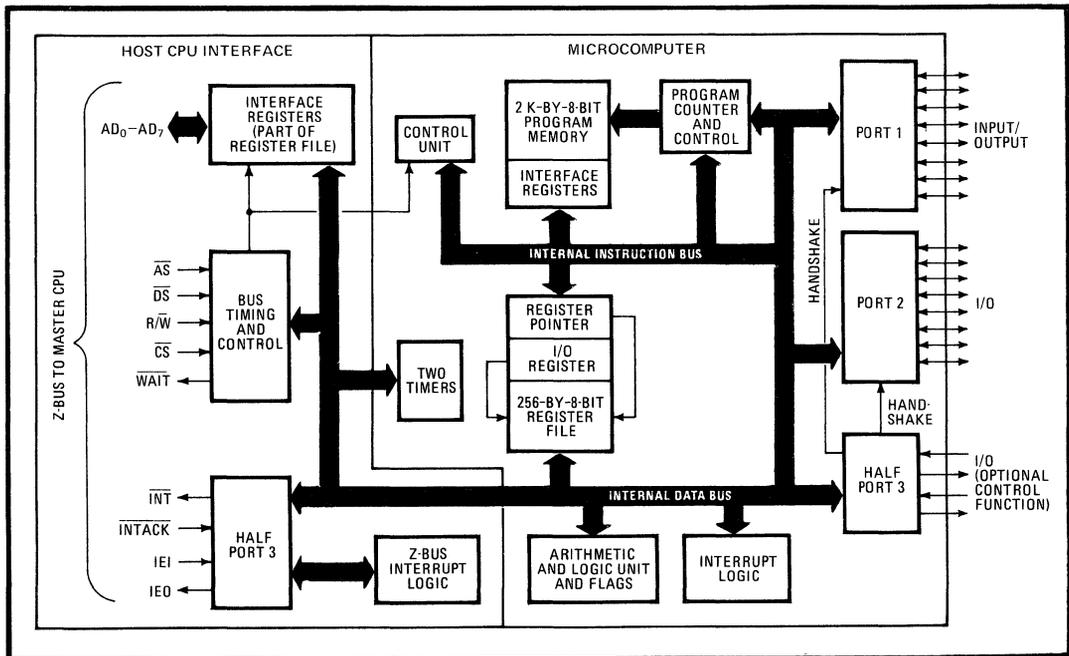
□ The growing power of high-end microprocessors and the complexity of peripheral devices attached to them has given rise to a need for general-purpose distributed processors to handle increasingly complicated input/output activities. As such, these devices must themselves have respectable processing and I/O-manipulation abilities while being able to interact efficiently with high-end microprocessors. Ideally, they would also communicate with 8-bit midrange microprocessors and be low in cost.

Just such a processor has been based on the Z8 single-chip microcomputer. The Z-UPC universal peripheral controller combines the instruction and I/O capability of the Z8 with two versions of bus interfacing: the Z-UPC offers the Z-BUS interface found on the Z8000, and the Z-UPC/U provides a Z80-compatible interface. The Z-BUS interface allows flexible connection to larger

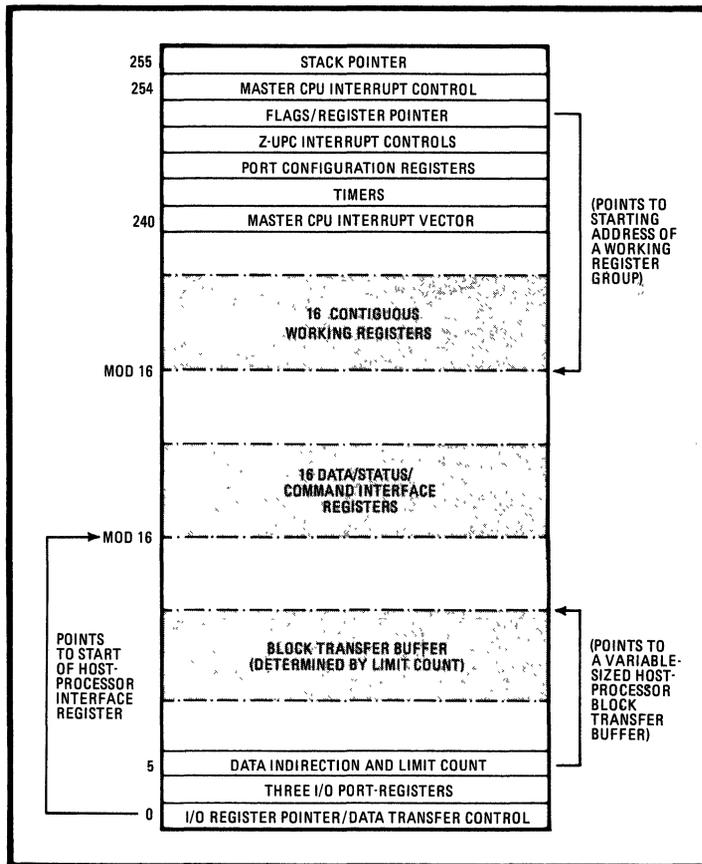
microprocessor systems and control of distributed I/O peripheral functions by means of a multiplexed address and data bus. The Z80-bus interface provides easy interfacing with 8-bit microprocessors and others that employ nonmultiplexed address and data buses.

A logical organization

The UPC is partitioned into two functional blocks: the logic for interfacing with the host-processors, and the core microcomputer (Fig. 1). In the multiplexed (Z-BUS) version, communication between the host and the UPC takes place over the Z-BUS, which provides an 8-bit bidirectional address and data port (AD₀-AD₇) and a set of control lines (\overline{AS} , \overline{DS} , $\overline{R/W}$, \overline{CS} , \overline{WAIT}). Also, under UPC program control, an optional daisy-chain interrupt structure—using request (\overline{INT}), acknowledge (\overline{INTACK}),



1. Microcomputer plus. The Z-UPC universal peripheral controller bases much of its architecture on the Z8 chip, to which it adds circuits at left for interfacing with a host processor. Shown is the Z-BUS-compatible version with multiplexed address and data lines.



2. File in. Of the UPC's 256-byte register file, 234 are general-purpose and can function as accumulators, buffers, pointers, or stack or index registers. The other 22 are specific pointers and registers, as well as status and control registers for the UPC's I/O facilities.

enable input (IEI), and enable output (IEO) lines—can be implemented. The microcomputer portion is based on the Z8 microcomputer architecture, whose central processing unit executes instructions averaging 2.2 microseconds each using a 4-megahertz clock source. The CPU's memory comprises 256 bytes of register-file random-access memory (which can be accessed directly by the host processor and the UPC), plus 2,048 bytes of read-only memory for program storage; other features include three I/O ports for device control, two timer/counters, and six vectored interrupts.

In addition to the standard 40-pin version (with 2-K bytes of ROM), there are two 64-pin versions of the Z-UPC: a ROM-less version and a RAM version, both of which have the program address, data, and control lines buffered and brought to external pins. The version with no program ROM on chip is intended as a development tool. The RAM version, which has 36 bytes of vestigial bootstrap ROM on chip, is intended as a controller whose program is downloaded from the host processor.

All three of these configurations are available with either the nonmultiplexed bus or the multiplexed Z-BUS interface to meet the needs of 8-, 16-, or even future 32-bit systems.

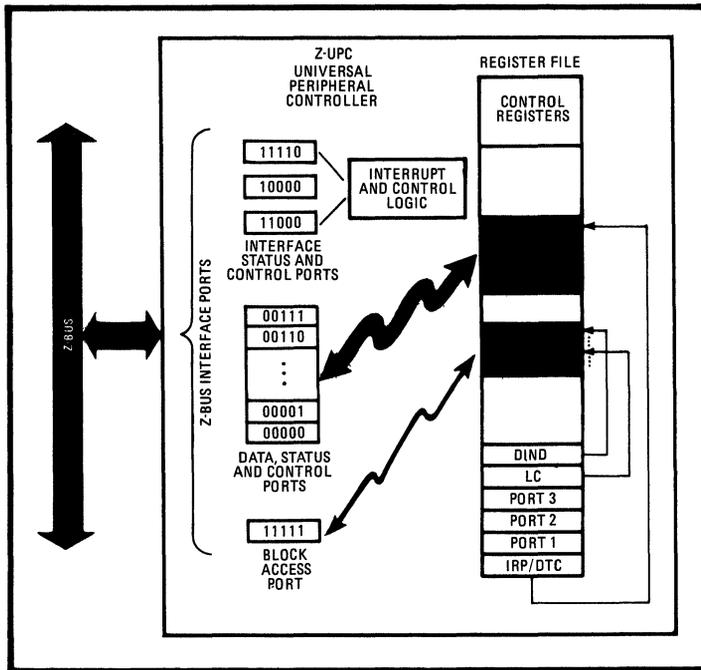
The UPC processor is organized around a 256-byte

register file (shown in Fig. 2). Besides storing data and control and I/O functions for the processor, the register file serves as buffer storage for communication between the UPC and the host CPU.

In addition to 234 general-purpose registers, the register file contains 19 control registers for configuring and controlling the Z-UPC's I/O facilities and three parallel I/O ports. The control registers both specify how the hardware is configured and should function and provide status information for it as well.

A multipurpose register file

All of the general-purpose registers can function as accumulators, data buffers, address pointers, and stack or index registers. All ports and control registers can be accessed by UPC instructions like any other register. Instructions can access the registers directly or indirectly with an 8-bit address field. However, a 4-bit addressing scheme, which makes use of a register pointer, can save memory and execution time. In this scheme the register file is divided into 16 register groups, each containing 16 contiguous locations. The register pointer determines which group is being accessed, and a 4-bit address field specifies the register within the group. This capability is especially useful to speed context switching.



3. Control. The 20 control registers in the UPC are divided into three groups: 3 for the interface status and control; 16 for data, status, and control (mapped by the I/O register pointer); and 1 block-access register for the transfer of data to or from a buffer file that is set up by the UPC.

Programs running on the UPC may communicate with those running on the master processor in a number of ways: under interrupt control (either by the UPC or master); by transfer of byte data through data, status, and command registers; and by transfer of blocks of data to and from the UPC's register file.

Three groups

The host CPU can directly access the 19 interface registers through the Z-BUS interface. As illustrated in Fig. 3, the registers are separated from the UPC's internal registers and divided into three groups:

- Those for interface status and control, which the master processor can access to control UPC-generated Z-BUS interrupts, to interrupt the UPC, and to control message transfer over the Z-BUS.
- Those for data, status, and control, which are mapped into 16 registers by the I/O register pointer, controlled by the UPC. That arrangement gives the master processor direct access to 16 registers and allows transfer of data, status information, or control commands between the UPC and master processor.
- Those for block access, used by the master to transfer blocks of data into or out of a buffer in the UPC's register file. The UPC has complete control over the placement and size of the buffer in its register file.

Each of the three types can be read or written by the master processor. Because the UPC software has complete control over how these register groups are mapped into the register space, the layout of data in the registers is independent of the host processor's software and protected from it.

The UPC and the master processor can operate com-

pletely independently of each other. The UPC can ignore the data transfer request from the master by setting a bit in its master-processor interrupt control register. Any attempt to transfer data from the master when this bit is set causes an error flag, which will cause an interrupt to the UPC (if interrupts are enabled).

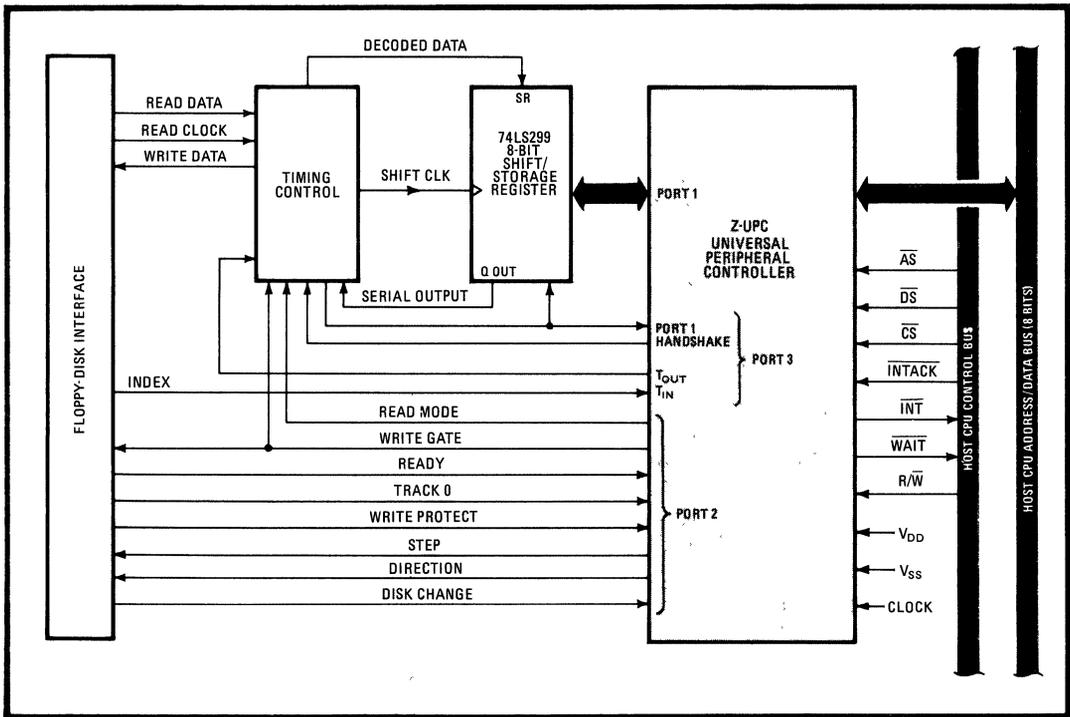
I/O lines by the dozen

The UPC has 24 lines dedicated to input and output that are grouped into three 8-bit ports. Since the ports are mapped into the register file, I/O data can be directly manipulated by any instruction. Each port has a mode-control register, which allows the port functions to be changed during program execution; for example, each line of port 1 and port 2 can be individually configured as input or output under program control. Each port can have its output lines defined as push-pull or as open-drain drivers.

Port 3 is a multifunction port. It has four input and four output lines that can be used for I/O or control functions. The control functions available through this port include interlocked handshake lines for ports 1 and 2, interrupt request inputs, timer input and output, and Z-BUS interrupt control.

Timing and counting

To support timing and counting requirements of software routines, the UPC provides two 14-bit timer/counters, T_0 and T_1 . Among the timer/counter functions that are easily implemented by the UPC are; interval delay timer, time-of-day clock, watchdog timer (as for refreshing dynamic memory), external event counting, variable pulse-train output, duration measurement for external



4. Disk jockey. The UPC makes a controller for a floppy-disk drive that actually stores the file system on chip. The host has only to specify the file name and the function to be performed. The 74LS299 shift register converts serial into parallel data, which enters port 1 on the UPC.

events, and automatic delay after an external event.

Each timer/counter is divided into a 6-bit prescaler and an 8-bit counter and is driven by the internal UPC clock, divided by four. The internal clock for T_1 may be set up for gating or triggering by an external event, or it may be replaced by an external clock input. Each timer/counter may operate in either a single-pass or a continuous mode, so that after the last count either counting stops or the counter reloads and continues counting. The counter and prescaler registers may be altered individually while the timer/counter is running; software controls whether the new values are loaded immediately or when the end of count (EOC) is reached. The two timer/counters may be cascaded using the timer-input lines on port 3.

Interrupting the controller

To serve host or I/O-device requests quickly, the UPC provides six interrupts from eight different sources: three from ports, two from timer/counters, and three from the host-processor interface. All six interrupts may be individually or globally disabled. The interrupts are prioritized, with the interrupt-priority control register providing 48 different priority schemes for handling concurrent interrupts. What's more, the masking and prioritizing of the interrupts may be dynamically modified under program control.

The UPC's interrupts are vectored. When an interrupt occurs, the program counter and flags are pushed onto

the stack, and control passes to one of six predetermined interrupt-handling routines. The routine is pointed to by an address that has been stored in the first 12 bytes of program memory. All of the interrupts are disabled after an interrupt is accepted. Interrupts can be nested by enabling them during the interrupt service routine; they are automatically enabled during the return from the routine.

The Z-UPC instruction set is compatible with the Z8 microcomputer instruction set (though the UPC's load-external-memory instruction is only available in the 64-pin RAM version of the Z8). This instruction set, comprising 129 instructions of 43 basic types and using six main addressing modes, speeds program execution and achieves byte efficiency. The types of data that it allows to be used include bits, binary-coded decimal digits, bytes, and 16-bit words.

Z-BUS support

The UPC can support the full Z-BUS interrupt structure, including daisy-chained priority resolution and vectored interrupt acknowledge. Using the interface control and status ports, the master processor has the full range of Z-BUS mechanisms for enabling or disabling Z-UPC interrupts, marking interrupts as being under service, clearing interrupts, setting interrupt vectors, and disabling interrupts from lower-priority devices.

A program running on the UPC can start the normal Z-BUS interrupt sequence (assuming interrupts are

enabled) by setting the interrupt-pending bit in its master-processor interrupt-control register. Once that is done, the UPC hardware automatically handles the Z-BUS interrupt protocol—including output of the interrupt vector, which is held in a separate control register.

The master can generate an interrupt to the UPC by setting the end-of-message flag in the master-processor interrupt-control port. In addition, UPC interrupts are generated from the master when an error condition, such as transferring a block of data that is too long, occurs.

Block transfer within limits

The UPC's block-access port is ideally matched to the block-I/O instructions of the Z8000 microprocessor. With a single block-I/O instruction the Z8000 can address the block-access port and transfer a string of bytes into or out of the UPC. Each access by the Z8000 to the block-access port causes a series of actions in the UPC involving its data-indirection register and limit-count control register. The data-indirection register points to the UPC register that is read or written when the master reads or writes the block-access port. After each such read or write, the data-indirection register is incremented and the limit-count register decremented. When the limit count reaches zero, any further transfers through the block-access port will abort and cause a length-error interrupt in the UPC.

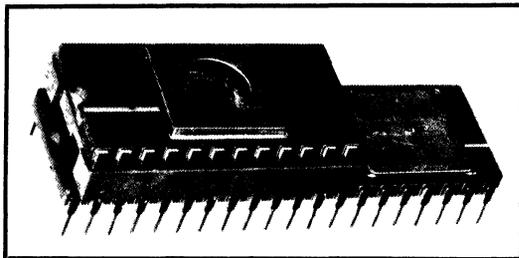
Ideas for application

The intelligence and flexibility of the UPC make it suitable for a wide variety of applications and allow it to offload the host computer in several ways. The UPC is capable of doing intensive off-line calculations, for example, such as those for data encryption. Also, it can perform the various code conversions and data formatting that are usually required in processor-to-device and device-to-device communications. Furthermore, it can buffer the data and generate controls for I/O devices such as printers and keyboards.

Figure 4 illustrates the UPC's application as a disk controller. Here, a file system can actually be implemented in the controller itself. The host processor has only to specify the file name and the function to be performed on the file. Depending on the sector size, either the register file or the external RAM in the 64-pin RAM version can be used as for data buffering. The serial-to-parallel conversion is done by a 74LS299 shift register, and the data is transferred using handshake logic in and out of port 1. Also, cyclic-redundancy error checking can be done by the UPC.

A UPC software package is available for Zilog's PDS8000 development system that includes an assembler (PLZ/ASM), a linker, and an imager. PLZ/ASM is a free-format assembler that generates relocatable and absolute object-code modules. It makes provision for external symbolic references and global symbol definitions. Data declarations, control structures, and DO loops between them supply a structured approach to the task of assembly language programming.

A development board, similar to the one that is now available for the Z8, will also be available. It uses the 64-pin version of the UPC to prototype a UPC-based



5. Prototyping. The ROM-less version of the Z-UPC, called a Prototack, is available in a special 40-pin package with a 24-pin socket on its back. Suitable for prototyping and preproduction use, it accepts a 2716 E-PROM for its first 2-K bytes of program memory.

system. The code thus developed can later be transferred to the ROM in the mask-programmable 40-pin version of the UPC, or it can be made available in image form for downloading to the RAM version.

Two serial RS-232-C interfaces will allow the 11-by-14-inch board to be used alone with a cathode-ray-tube terminal or to be connected to one of Zilog's PDS or ZDS-1 series development systems. Cable connection to such a host system will permit the transfer of software from the host—where it is developed—to the board for testing. Included on the board is a 64-pin Z8, which serves as a program monitor for the UPC.

The board also contains 4-K bytes of 2716 erasable programmable ROM (for the monitor/debugger program) and 4-K bytes of 2114 static RAM. For the user who wishes to test a ROM-based version of his code, it also offers a socket for 4-K bytes of 2716 E-PROM that may be used in place of the RAM. The monitor/debugger software, comprising a terminal handler, a debugger, command interpreter, and an upload/download handler, provides the various commands necessary for control, I/O, and debugging.

A wrapped-wire area of 40 square inches accommodates additional customer interfaces or special application circuits. This arrangement allows for wide range of user applications.

Aid in prototyping

The Z-UPC Prototack—the ROM-less version of the standard Z-UPC, housed in a pin-compatible 40-pin package (Fig. 5) that carries a 24-pin socket to accommodate a 2716 E-PROM—is used for prototype development and preproduction of mask-programmed UPC-based applications. The 24-pin socket is equipped with 12 ROM address lines, 8 ROM data lines, and the necessary control lines for interfacing with the E-PROM for the first 2-K bytes of program memory.

Pin compatibility allows the user to design the printed-circuit board for a final 40-pin mask-programmed UPC and, at the same time, allows the use of UPC Prototack to build prototype and pilot-production units. When the final program is established, the user can then switch over to the 40-pin mask-programmed UPC for large-volume production. The Prototack is also useful in applications where masked ROM setup time and mask charges are prohibitive and program flexibility is desired. □

Adapting Unix to a 16-bit microcomputer

Z-Lab software development system with text-processing utilities supports 16 users in C language, has 32-bit bus for future expansion

by Bruce Weiner and Douglas Swartz
Zilog Inc., Cupertino, Calif

□ In systems based on 16- and 32-bit microprocessors, software will account for the bulk of the development cost. As more and more logic is squeezed onto a single chip, the hardware development process is being simplified and its costs reduced. Simultaneously, however, more complex and, hence, more expensive software is going to be required.

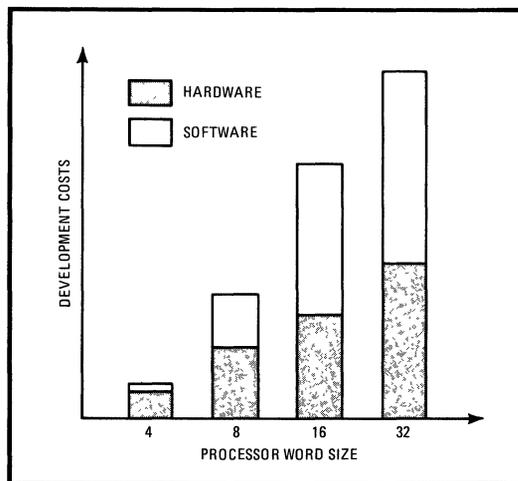
Zilog's recognition of this trend in computer technology (shown in Fig. 1) led to Zeus, the adaptation of the Unix operating system for the Z-Lab 8000 microcomputer [*Electronics*, Feb. 10, p. 33].

Both software and hardware played crucial roles in the creation of the Z-Lab development system. Components such as the Z8001 microprocessor, the memory/management unit (MMU), and the Z-bus backplane-interface (ZBI) bus structure were as critical to the potential of the system as was the software itself. Together, the Z-Lab and the Zeus operating system foster a software development environment that is a major step toward controlling the rapidly escalating software costs of microprocessor products.

Transporting a system

In selecting an operating system, there were two options: writing a new one from scratch or transporting an existing one to the Z8001. The decision was made to transport one—provided it was possible to find an existing operating system that could be adapted quickly and was well-suited to software development.

The search for such an ideal software environment ultimately led to the Unix operating system. This system was selected for four reasons: it was designed specifically for software development and text processing; it had already been transported successfully to 16- and 32-bit computers; it had a large existing software base with applications pertinent to a development environment; and it had a large user base.



1. Skyrocketing software. As products use more sophisticated microprocessors, there is an increase in the amount of engineering effort required to write software. As hardware costs drop, software costs are becoming the major product development expense.

What is Unix anyway?

As a general-purpose, multiuser, interactive operating system, Unix offers facilities that are seldom found even on larger mainframes. Through its hierarchical file structure, any file can be traced back to a single root directory, thereby facilitating the management of mass memory. In addition, peripheral device handlers, files, and interprocess communications are all compatible with each other, simplifying program development. Since one major goal of Unix is to increase programmer productivity by providing a responsive working environment, Unix includes a vast library of utilities ranging from spelling correction routines to various compiler compilers and supports over a dozen languages, including the language C in which it is written.

However, the most important role of any operating system is still managing the mass-storage files in which all the programs and data reside. Unix imposes no particular structure upon the content of these ordinary files. Instead, it distinguishes between two kinds of special files even though they are treated identically by the programmer.

The first of these is a directory file that simply lists the names and vital statistics of other files. These other files may, in turn, be programs, data, or even other directories. This hierarchical structure results in an unusually well-organized system in which a file can be specified by its path name, which is a sequence of directory names separated by slashes that terminates with the desired file name. Thus, the same name can be used for files of similar function as long as they have different path names. For example, `/Jones/Statistics` and `/Smith/Statistics` both refer to a file named `Statistics`, but they are not the same file because they have different path names indicating they are listed in the unique directories `Jones` and `Smith`.

Just as directories are treated in the same manner as ordinary files by programmers, so are the second kind of special files—input/output calls. This distinction is the

most unusual feature of Unix and one of its greatest advantages over other operating systems. These special files are read and written just like ordinary ones, except that the selected device is activated and the data is passed to it using whatever protocol is appropriate. Thus, programs can send data to, for example, a printer in exactly the same way they do to a disk file—except the name of the selected output unit is different.

Unix programs may communicate with each other in the same manner as I/O calls. The output of one program is directed to the input of another while each program thinks it is reading or writing a disk file. The communication link itself is called a pipe and can be created either by the program itself or interactively by the programmer. In this way, a group of related programs may pass data to each other in an extremely efficient manner.

Another feature of Unix is its ability to safeguard original programs. Before a program is executed, a fork, or replicate, operation copies the program, including the code, register values, open files, current directory and the like, into memory. The replicated process is executed, ensuring that the original is never lost or scrambled, in case execution does not take place properly.

Perhaps the most visible portion of Unix is the shell, or fundamental control program, which functions as the primary interface with the system user. As a command language it offers the programmer a productive working environment. Multitasking permits programs to be started without loss of control of the console. Special command files may be set up so that any sequence of shell commands can be executed by a single user command. Commands can even be strung together at the console so that the results of one are fed directly to the input parameters of the next, in the same manner as pipes interfacing programs.

-R. Colin Johnson

When transported, the Unix operating system was enhanced in several ways so that the Z8001 implementation might run more reliably. For example, in the standard Unix operating system, nothing prevents two users from simultaneously modifying a file so that one user can accidentally invalidate the other's changes. The Zeus operating system qualifies the three standard Unix file-opening modes (read, write, and read and write) with three access-control modes specifying what other users can do with the file.

The Zeus access control modes are shared, read-only, and exclusive. The shared mode, the standard Unix control mode, allows other users access to any file they desire. In the read-only mode, other users may access the file only for read operations. In the exclusive mode, other users may not access the file at all; the first user opening the file has exclusive access to it until the file is closed. Any attempted access to a file that violates these parameters results in a failure of that open operation.

Other enhancements

A full-screen text editor, called the visual editor, has been implemented in Zeus for cathode-ray-tube terminals. Its data base contains terminal-control information that permits full-screen editing for almost any combina-

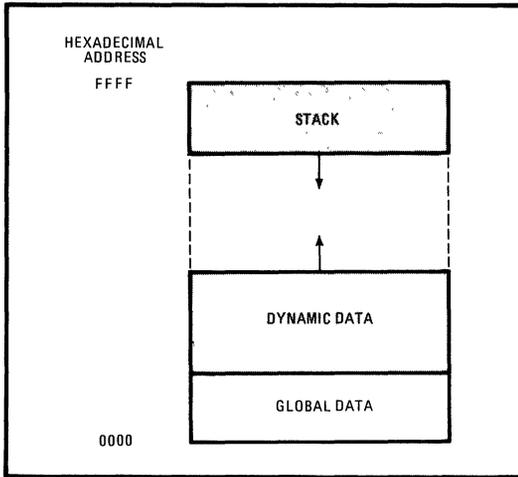
tion of CRT terminals. The terminal data base can be updated by the user when adding new terminals to the system.

The editor lets the user display text files one page at a time and rapidly move the cursor on that page, inserting or deleting characters, words, lines, or groups of lines with a minimum number of keystrokes. Several additional features are available, such as cut-and-paste and word-wrap facilities.

Rebuilding the system

Another enhancement is the Sysgen program, which automatically rebuilds the Zeus system, letting the user tailor it to specific requirements. The user can add disk and tape drives or other input/output devices using the Sysgen program as well.

Several other utility programs are supplied with Zeus. Learn, an interactive program, teaches new users how to fully exploit Zeus's facilities; Mail lets users send messages to each other in postal format; Calendar automatically reminds users of events scheduled during the day when they sign on and begin using their terminal; Spell is a spelling-error detection program that uses a 25,000 word dictionary; and Man prints selected portions of the Zeus reference manual on the user's terminal. Over



2. Subdivisions. Zeus separates the memory space for programs and data, the latter being subdivided into areas for the stack, dynamically allocated variables, and global variables. Hardware ensures that the stack and dynamic areas do not overlap.

60 other utilities are furnished with the Z-Lab.

Almost the entire Unix operating system and its application programs are written in C, a system implementation language. The key to transporting Unix software is a C compiler that generates code for the target system, in this case the Z8001-based Z-Lab 8000. Although C carries a certain level of machine independence, this does not mean that the entire Unix operating system can be transported by merely recompiling it. Most application programs, however, can be transported in this manner.

Seventh edition

Specifically, the Zeus operating system is Zilog's enhanced version of the seventh edition of the Unix operating system, which was modified by Bell Laboratories to eliminate explicit machine dependencies and ease its transportation to other computers. Some implicit machine dependencies, however, must of necessity remain in the Unix kernel. For this reason, transportation to Z-Lab is greatly simplified by creating hardware very similar to those architectural features implicit in this kernel.

The two major machine dependencies in the Unix kernel are the size of integers and pointers and the memory management capability required by Unix software. Both the Unix kernel and C assume that integers and pointers are the same size and that integer arithmetic can thus be performed on pointers. Examining the evolution of the Unix system sheds light on how this machine dependency was handled.

C originally was developed to write Unix, and Unix originally was written for Digital Equipment Corp.'s PDP-11 family of 16-bit minicomputers. Further, the seventh edition of the Unix system was written specifically for PDP-11 systems with separate code and data address spaces. Thus, microcomputer hardware that provides facilities similar to those of a minicomputer such as

DEC's PDP-11/70 should minimize the transportation effort.

On the basis of this background information, the design team decided to run the Z8001 microprocessor in the nonsegmented mode for user processes and for almost all of the kernel. In a nonsegmented mode, programs use 16-bit addresses and are limited to a single 64-K-byte segment. This means that both integers and pointers are considered 16-bit quantities and therefore integer arithmetic can be performed on them.

Because the Z8000 family can support separate code and data address spaces, user and system programs may have as much memory as a PDP-11/70—128-K bytes, of which 64-K bytes are code and 64-K bytes are data. Furthermore, the Z8001's 24-bit addressing scheme can handle a total system memory as large as 16 megabytes. Because the Z-Lab 8000 can handle up to 1.5 megabytes of memory the need for swapping programs in and out of main memory is reduced, thereby minimizing response time when a large number of users are on the system.

Much of the existing Unix software base takes advantage of the operating system's dynamic allocation of memory. This system characteristic has had a major impact on the hardware design of Z-Lab.

Memory management

Figure 2 shows how a C program's data is laid out in memory. This stack starts at the highest 16-bit data address and grows toward lower addresses. Global data that is statically allocated starts at address 0 and of course does not grow.

The Unix kernel provides system calls that allow a process to dynamically request more data memory. This dynamic data area starts just above the global data and fills in the unused addresses up to stack.

Memory space located between the stack and the dynamic data area is not necessarily allocated to one or the other. The hardware must therefore detect a memory reference in the constantly changing gap between the two memory areas and make sure they do not overlap. When an invalid access is detected, the kernel can either allocate more memory or terminate the process, as appropriate.

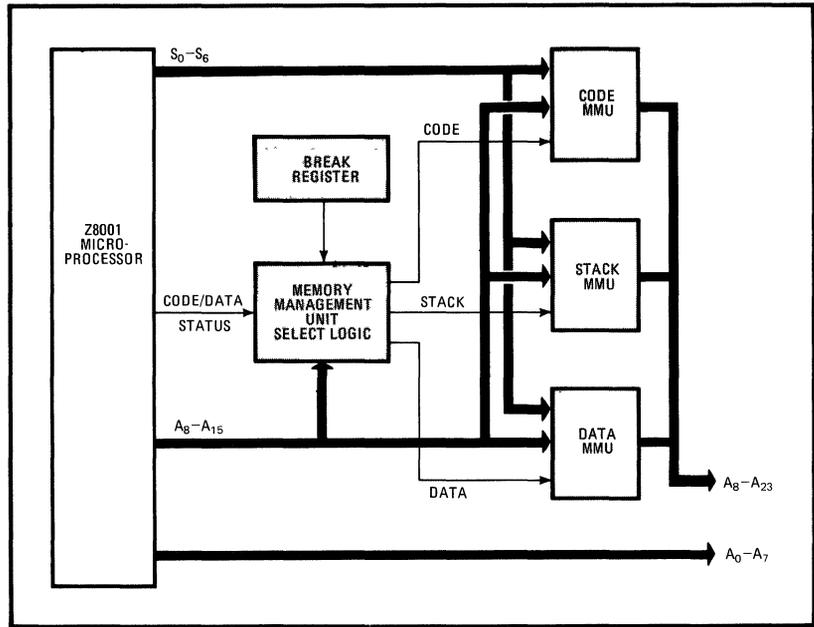
To protect the memory areas from invalid access, Zilog's Z8010 memory management unit was selected for the Z-Lab processor board. The MMU relocates addresses so that programs can be placed anywhere in physical memory and keeps the system from being corrupted if a user's program runs amok.

Nonsegmented solution

If Z-Lab were running in segmented mode, the two data areas would be placed in separate data segments, and the MMU could detect address violations as well as the need for more memory. In a nonsegmented mode, however, memory references to both bear the same segment number, so detecting a memory reference in the gap must be accomplished in another way in order to prevent the dynamic data area and the stack from overlapping.

Although the segmented-mode solution could not be used, it did provide the foundation for a nonsegmented

3. Multiple MMUs. In the Zeus operating system, the Z8001 processor runs in its nonsegmented mode, and memory management units divide the memory space into separate code, stack, and data areas. The break register stops the stack and data areas from overrunning one another.



solution, in which the references to the two dynamic data areas are made through two different MMUs. In Fig. 3, a simplified block diagram of Z-Lab's memory management architecture shows that there are separate MMUs for the code, as well as for the stack and data address spaces. The MMU select logic determines which one should be activated and guarantees that only one will be active at any given time.

The operating system sets the break register, the key element in determining whether the stack or the data MMU will be activated, pointing to the highest address in the dynamic data area. On every data reference to memory, address bits 8 through 15 from the Z8001 are compared to the value in the break register. Data addresses greater than or equal to the break value activate the stack MMU; data addresses less than the break value activate the data MMU. The MMU selection occurs quickly enough for no wait states to be required, even with a 6-megahertz Z8001.

Integrating hardware and software

The memory management design discussed above handles Unix software and nonsegmented Z8000 programs. In addition, the memory management architecture of the Z-Lab processor board can be modified under program control to support segmented user and system programs. Future software releases can thus take full advantage of the 16 megabytes of address space provided by the segmented Z8001.

The Z-Lab development project was approached as an integrated product-design effort. A broad-based project team was selected to facilitate close cooperation among the hardware, software, and mechanical engineers, and the memory management architecture thus developed by the team solved problems that could not have been

solved independently by any of the individual groups.

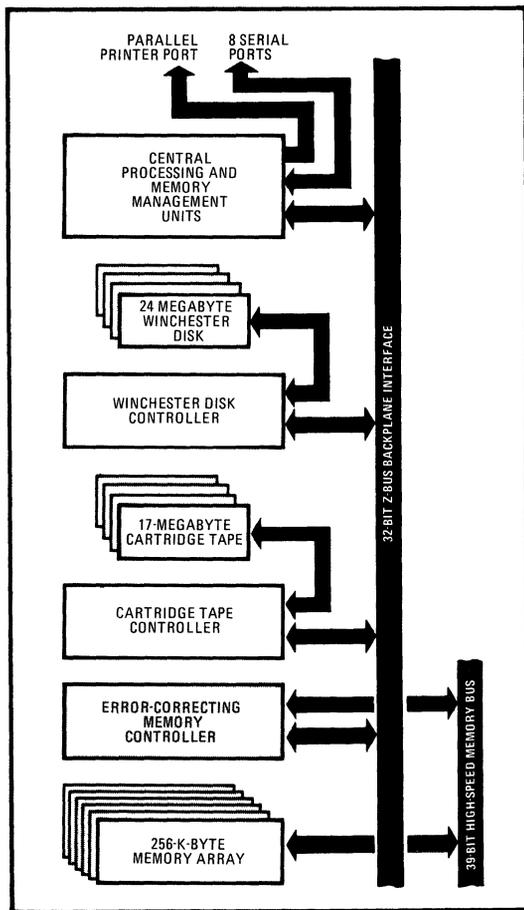
Likewise, the various goals of the Z-Lab system could be attained only with an integrated approach to the hardware, software, and mechanical engineering aspects of the project. Of these goals, the first was to design a Unix-based system with enough flexibility and file-system integrity that users could configure and maintain it themselves. A second goal was a performance level that could comfortably support up to 16 users. The final one involved packaging the system for the office environment.

To best achieve these goals, the project team sought a system design with minimum power consumption and noise levels. Thus, the Z-Lab offers high-performance minicomputer power in a quiet and easily portable package that consumes only 325 watts. It has no special power requirements and no cooling requirements, if ambient temperature stays below 40°C.

Z-Lab system hardware was also designed for expandability. Using a moderate-sized printed-circuit board (approximately 9 by 11 inches) kept the hardware configuration compact while allowing enough board area for future Z-Lab products. A highly reliable two-piece connector, although slightly more expensive than the conventional one-piece card-edge connector, improved connection reliability and permitted more connections per inch of pc-board edge.

Bus with a future

A semisynchronous bus, the ZBI, was chosen for its high level of system performance and input/output interface. All Z8000 peripheral circuits interface with the bus simply, needing buffering only to attain the TTL drive levels required on the backplane. Z80 peripherals can also be attached to the bus by generating the required



4. Architectural planning. The Z-Lab development system uses the proprietary ZBI 32-bit bus, an error-correcting memory controller that communicates with the main memory over a separate high-speed bus, and both Winchester disk and cartridge tape controllers

Z80 timing with simple interface logic.

The ZBI is a true 32-bit bus with the address and the data multiplexed on the same lines (Fig. 4). The bandwidth of the bus (8 megabytes per second) is sufficient for future high-speed 32-bit processors and for peripheral controllers as well.

The Z-Lab error-correcting memory controller (ECC) supports 8-, 16-, and 32-bit data transfers, performing 32-bit error correction with the aid of seven extra syndrome random-access memories that hold the correction bits for every 32 data RAMs. The ECC communicates with its memory array cards over a very high-speed dedicated memory bus.

Maximizing memory capacity

All timing and refresh circuitry on the controller is centralized, maximizing memory capacity in the system. In addition to a maximum of 1.5 megabytes of ECC memory in the processor module enclosure, the Z-Lab

unit has slots for the processor, cartridge tape controller, and Winchester disk controller cards as well.

Two of the Z-Lab's three peripheral controllers are intelligent, using Z80B 6-MHz microprocessors. This offers three distinct advantages.

First, device control chores are offloaded from the main processor. The operating system thus can communicate with the peripheral controllers using high-level commands that let the peripheral controllers work in parallel with the main processor. For example, Z-Lab can issue simultaneous reads or writes to more than one disk drive; the disk controller keeps track of head position, sector position, and data transfer.

Secondly, the intelligent peripheral controllers can perform self-diagnostics on power-up or on command, thus certifying to the host processor with a high degree of certainty that they are functioning correctly before processing begins.

Finally, product maintenance and upgrading is simplified by using firmware. As information is gathered on how the operating system interacts with the disk under different program mixes, the Winchester disk controller can be easily "tuned" for higher performance by altering the firmware.

Initial board set

The Z-Lab board set consists of:

- A processor board containing eight serial channels with programmable bit-rate, a parallel printer interface for either Centronics or Data Products-type printers, a memory management subsystem that supports either segmented or nonsegmented user processes, and read-only memory containing the bootstrap software and power-up diagnostics.
- An ECC memory controller that supports 32-bit error correction for up to 16 256-K-byte memory array cards. This board contains detection and reporting logic for uncorrectable errors and error-logging logic for correctable errors.
- One or more 256-K-byte memory array cards using high-speed 16-K dynamic RAMs.
- An intelligent cartridge tape controller that handles up to four tape drives for file archiving or for backup of the entire system.
- An intelligent Winchester disk controller that supports up to four 24-megabyte 8-in. Winchester disk drives.
- An optional serial I/O controller board that supports an additional eight serial lines and an additional printer port.

Several other subsystems will be offered with Z-Lab in the near future. An expansion chassis will increase the number of card slots in the unit from 10 to 20, the maximum number a ZBI bus can support, for constructing very large systems.

Another offering will be a compatible 40-megabyte Winchester drive (40- and 24-megabyte drives can be mixed on the system's Winchester controller). Zilog also will offer an intelligent serial controller that can perform direct-memory-access transfers to and from main memory, which will help improve system performance by reducing the amount of time that must be spent by the processor in servicing terminal interrupts. □

Software

Major firms join Unix parade

Transparent versions of operating system make it available for computers ranging from mainframes down to microsystems

by R. Colin Johnson, *Microsystems & Software Editor*

Devotees of Unix, the operating system whose responsiveness has been compared to that of a well-tuned sports car, are adding to their number almost daily. This rapid expansion of the user base of Unix, developed at Bell Laboratories and licensed by Western Electric Co., has been spurred by the emergence of user-transparent versions made for computers ranging in size from the likes of IBM System 370 mainframes down to Z80-based 8-bit microcomputer systems.

Item: Texas Instruments Inc., Dallas, long known for its comprehensive software development system, is planning to implement Unix through a subcontract with a third-party software house.

Item: Lifeboat Associates, a leading 8-bit software publisher in New York, has just signed an exclusive marketing contract with Microsoft for end-user sales of its 16-bit Xenix-11 adaptation for PDP-11s.

Item: Intel Corp.'s Ada compiler

for the iAPX 432 [*Electronics*, Feb. 24, p. 119] is written in Pascal on a VAX-11/780 under Unix. (When asked why Unix was used when the final compiler release will be under VMS, Nicole Allegre, Ada program manager for the Santa Clara, Calif., company, responds, "The programmers just really wanted to use it.")

Obeys orders. Those programmers at Intel are not alone. Their counterparts across the country have been taken by Unix's responsive software-development environment. Also, the language in which the original Unix is written, C, is one of the most respected of the structured languages extant [*Electronics*, May 8, 1980, p. 129].

Since Unix was developed on Digital Equipment Corp. machines, it has been widely used on PDP-11 minicomputers for some time. However, now that Western Electric allows systems with only a few users to pay a special per-user royalty fee, it has become economical for com-

mercial software houses to configure Unix for even inexpensive systems. An increasing number of original-equipment manufacturers and commercial software houses should start offering Unix for various other computer systems.

Unix is in fact making a strong bid to become a standard among operating systems for the new wave of 16-bit microsystems, though it faces stiff competition from the entrenched operating system family from Digital Research, Pacific Grove, Calif. When that company's 16-bit implementation of its MP/M becomes available, it will include many of the facilities that make Unix so desirable—plus CP/NET, which allows both 16- and 8-bit microsystems to share expensive peripherals. OEMs can look forward to a rich selection of system-level software packages from which to choose. Even the 8-bit microsystems are acquiring Unix-like capabilities without having to sacrifice CP/M capability.

Drawbacks. Unix is not without its critics. They say that the system cannot be used easily by clerical personnel and cite difficult operations, like rebuilding the linked list that describes the hierarchical file structure after a system crash. Some say that Unix does not provide adequate file-protection systems to make it completely trustworthy in commercial uses.

Such criticism stems from Unix's initial target: cooperative multiprogrammer software projects in which most of the users were professional computer specialists. That is why many of the facilities provided by it are specifically aimed at efficient

UNIX AND UNIX-LIKE OPERATING SYSTEMS

Processor or computer	Company	Name	Bell Laboratories' version	Original implementation
Z8000	Zilog Microsoft	Zeus Xenix	✓ ✓	
Z80	Cromemco Morrow Designs	Cromix μNIX		✓ ✓
LSI-11 and PDP-11	Whitesmiths Microsoft Mark Williams Co	Idris Xenix-11 Coherent	✓	✓ ✓
6809 68000	Tech System Consultants	Uniflex		✓
C/70	BBN Computer	Unix	✓	
470	Amdahl	UTS	✓	
All Perkin-Elmer 32-bit Machines	Wollongong Group	Unix	✓	

Source: *Electronics*

program development. On the other hand, Unix is probably best known for its document-preparation and -management functions, which are often used by nonprogrammers. And with the addition of a good screen-oriented editor, like Zilog's visual editor, Unix offers a wide avenue of capability for professionals and non-programmers alike.

New version. One of the latest Unix versions is the Zeus adaptation by Zilog Inc. Cupertino, Calif., for its Z-Lab software development system using the Z8000 [*Electronics*, March 24, p. 120]. And to be released next month to selected OEMs is the Z8000 version called Xenix from Microsoft in Bellevue, Wash. [*Electronics*, March 24, p. 34]. Among the first of the OEMs is Codata of Sunnyvale, which is working on a floppy- and hard-disk-based microsystem that makes use of a Multibus-compatible central processing unit. Later this year, the 8086 version of Xenix is to be delivered to Altos Computer Systems of Santa Clara for its single-board 8086-based microsystem.

After that, Microsoft plans to release a 68000 version (as does Whitesmiths Ltd. of New York in an original implementation), with an eye to the iAPX-432 and the 16000 in an attempt to establish Xenix as the standard version of Unix for 16-bit microsystems. Not only is Microsoft dedicated to marketing Unix, but it is also dedicated to using it: all

product development programming in its Consumer Products division is done in C on a PDP-11/70 under Unix and then transported to the target microsystem.

The first computer to which the operating system was transferred from the one on which it was developed was the Interdata 8/32. The Wollongong Group of Palo Alto, Calif., now offers Unix for the 8/32, as well as for the rest of Perkin-Elmer's 32-bit minicomputers (Perkin-Elmer having bought Interdata).

The same. In the Wollongong offering, a supreme attempt has been made to make this implementation virtually identical to the original as it appears to the user, in the interest of program portability and of preserving a common command language across Unix systems.

Unix is also available from AMDahl Corp. for its IBM 370 look-alike, the 470 mainframe, and even for a computer that is specially optimized for the C language—the C/70—from BBN Computer Corp. [*Electronics*, Nov. 6, 1980, p. 46]. These, like the others, are licensed by Western Electric.

However, before the licensing procedures were changed to accommodate small systems, several software developers began work on Unix look-alikes. These user-transparent, yet original, implementation projects are now coming to fruition.

One that has been around for more than a year is Whitesmiths'

Idris [*Electronics*, March 24, 1981, p. 125]. Some of the newer ones are aiming at the 8-bit market to maintain compatibility with current software bases. Two, for Z80-based microsystems using the S-100 bus, come from Morrow Designs of Richmond, Calif., and Cromemco Inc. of Mountain View, Calif., respectively.

Subtasks. Morrow Designs' version, called μ NIX, runs CP/M as one task within its multiuser environment, thereby maintaining compatibility with CP/M software while gaining the conveniences of a user-transparent Unix. The emphasis throughout has been on compatibility and portability; μ NIX is written entirely in Whitesmiths' C, which is not supplied with the package. Cromemco's version runs the CDOS operating system as a subtask and maintains compatibility with that already extensive software base, including its new C compiler.

There is even a version, from Technical System Consultants Inc., for Southwest Technical Products Corp.'s 6809-based 128-K-byte microsystem. Called Uniflex, it is written entirely in assembly language and includes most of Unix's features; it supports both floppies and a 20-megabyte hard disk. The West Lafayette, Ind., firm will add a 68000 version soon and is looking to Ada, Pascal, and C for future high-level language projects. □

Reprinted with permission of Electronics

**Zilog
Sales
Offices**

West

Sales & Technical Center
Zilog, Incorporated
10340 Bubb Road
Cupertino, CA 95014
Phone: (408) 446-9848
TWX: 910-338-7621

Sales & Technical Center
Zilog, Incorporated
18023 Sky Park Circle
Suite J
Irvine, CA 92714
Phone: (714) 549-2891
TWX: 910-595-2803

Sales & Technical Center
Zilog, Incorporated
15643 Sherman Way
Suite 430
Van Nuys, CA 91406
Phone: (213) 989-7485
TWX: 910-495-1765

Sales & Technical Center
Zilog, Inc.
2918 N. 67th Place #2
Scottsdale, AZ 85251
Phone: (602) 990-1977

Midwest

Sales & Technical Center
Zilog, Incorporated
890 East Higgins Road
Suite 147
Schaumburg, IL 60195
Phone: (312) 885-8080
TWX: 910-291-1064

Sales & Technical Center
Zilog, Inc.
28349 Chagrin Blvd.
Suite 109
Woodmere, OH 44122
Phone: (216) 831-7040
FAX: 216-831-2957

South

Sales & Technical Center
Zilog, Incorporated
2711 Valley View, Suite 103
Dallas, TX 75234
Phone: (214) 243-6550
TWX: 910-860-5850

Zilog, Incorporated
7115 Burnet Rd.
Suite 207
Austin, TX 78757
Phone: (512) 453-3216

Technical Center
Zilog, Incorporated
1442 U.S. Hwy 19 South
Suite 135
Clearwater, FL 33516
Phone: (813) 535-5571

East

Sales & Technical Center
Zilog, Incorporated
Corporate Place
99 South Bedford St.
Burlington, MA 01803
Phone: (617) 273-4222
TWX: 710-332-1726

Sales & Technical Center
Zilog, Incorporated
110 Gibraltar Road
Horsham, PA 19044
Phone: (215) 441-8282
TWX: 510-665-7077

United Kingdom

Zilog (U.K.) Limited
Babbage House, King Street
Maidenhead SL6 1DU
Berkshire, United Kingdom
Phone: (628) 36131
Telex: 848609

West Germany

Zilog GmbH
Zugspitzstrasse 2a
D-8011 Vaterstetten
Munich, West Germany
Phone: 08106 4035
Telex: 529110 Zilog d.

Japan

Zilog, Japan KK
TBS Kaikan Bldg.
3-3 Akasaka 5-Chome
Minato-Ku, Tokyo 107
Japan
Telex: ESSOEAST J22846