68xxx UniFLEX® Relocating Assembler & Linking-Loader

MANUAL REVISION HISTORY

Revision	Date	Change
A	10/84	Original Release, 68000 Relocating Assembler and Linking-Loader Version 1.00
B	01/86	Manual Update, Added documentation for the 68020 assembler
C	09/86	Manual update for Version 2.0 of 68xxx UniFLEX. Added command-line parameters, macros, new instructions for assembler, new options for both assembler and linking-loader. Miscellaneous minor changes and corrections.

DISCLAIMER

The supplied software is intended for use only as described in this manual. Use of undocumented features or parameters may cause unpredictable results for which Technical Systems Consultants, Inc. cannot assume responsibility. Although every effort has been made to make the supplied software and its documentation as accurate and functional as possible, Technical Systems Consultants, Inc. will not assume responsibility for any damages incurred or generated by such material. Technical Systems Consultants, Inc. reserves the right to make changes in such material at any time without notice.

Contents

Contents

Preface

This manual describes the operation of both the UniFLEX® 68xxx Relocating Assembler and the 68xxx Linking-Loader. It is by no means intended to teach the reader assembly language programming; nor does it present the full details of the 68xxx instruction set, which are available elsewhere (Motorola, 1984 and 1985a).

®UniFLEX registered in U.S. Patent and Trademark Office.

Chapter 1

Introduction to the 68xxx Relocating Assembler and Linking-Loader

1.1 Input to the 68xxx Relocating Assembler

The relocating assembler accepts as input a text file written in 68xxx assembly language as described in the M68000 16/32-Bit Microprocessor Programmer's Reference Manual (Motorola, 1984), the MC68020 32-Bit Microprocessor User's Manual (Motorola, 1985a), and the MC68881 Floating-Point Coprocessor User's Manual (Motorola, 1985b), with the modifications described in this manual. The code need not contain any absolute addresses. Instead, the program may use relocatable addresses--addresses which cannot be evaluated until the linking-loader processes the code.

1.2 Output from the 68xxx Relocating Assembler

By default, the assembler produces as output a binary file containing object code. The assembler generates this object code so that relocatable addresses are not bound to absolute locations at assembly time. This binding, which is called relocation, is accomplished later by the linking-loader. Relocation is necessary whenever the operand field of an instruction which expects an absolute address contains a relocatable address. The assembler writes a "relocation record" for each relocatable reference in the source code. The relocation record contains the information necessary to the linking-loader, which adjusts the address at load time. Relocation records are written to a temporary file. At the end of the assembly, the contents of this file are appended to the end of the object code. An assembled file with relocation records is called a relocatable object-code module or, more simply, a relocatable module. A relocatable module must be processed by the linking-loader in order to transform it into an executable file.

It is often desirable for parts of a program (called modules) to be developed separately. In such a case each module is assembled separately prior to the final merging of all modules, which is done by the linking-loader. A module that is part of a larger program may contain references to symbols which are not defined in that module. Such a reference is called an external reference. The assembler generates an external record for each external reference in the source code. The external record contains the information necessary to the linking-loader, which resolves these references at load time. The

assembler writes external records to the same temporary file that contains the relocation records.

1.3 Segmentation of Binary Files

The UniFLEX Operating System supports the segmentation of binary files. Therefore, during assembly, the assembler may break the file up into as many as three segments--text, data, and bss--each of which contains different kinds of material.

The text segment contains both initialized data that do not change during execution and executable instructions. Some hardware systems write protect the text segment.

The data segment contains initialized data which are likely to change during execution. Regardless of the hardware, the user may access or change a value in the data segment at any time during execution.

The bss segment, also known as uninitialized data, does not actually contain any code. Rather, the binary header for the file contains a number which tells the operating system how much memory to reserve for bss when it loads the module. Only those instructions which do not generate code may be used in the bss segment. At any point during execution of the program, data in the bss segment of memory can be accessed or altered.

The assembler segments a file by maintaining three distinct program counters, one for each segment. At any point in the assembly only one of these counters is in effect. The user determines which counter is in effect by entering the appropriate segmentation directive--"text", "data", or "bss"—in the opcode field (see Section 3.3). The assembler generates code in the segment specified until it encounters another segmentation directive, an "end" directive, or an end-of-file character. The assembler does not assume a default segment. Therefore, the user must include a segmentation directive in the source code before using any instructions which generate code or reserve memory.

1.4 Input to the 68xxx Linking-Loader

The linking-loader accepts as input independently assembled, relocatable object-code modules generated by the assembler or the linking-loader itself.

1.5 Output from the 68xxx Linking-Loader

The linking-loader combines the segments of object code from one or more relocatable modules to produce a single object-code module and, optionally, a load map, a module map, and a symbol table. As it combines the modules, the loader uses the relocation records generated by the assembler to bind relocatable addresses to absolute locations. It adjusts each relocatable address by the "relocation constant", which is the address at which the module is loaded for execution. It uses the external records to resolve all external references.

Depending on the user's preference, the loader can produce either an executable or a relocatable module. A relocatable module produced by the linking-loader is indistinguishable from a relocatable module produced by the assembler. Only the loader, however, can transform one or more relocatable modules into an executable program.

Chapter 2

Invoking the Relocating Assembler

2.1 Introduction

The UniFLEX 68xxx Relocating Assemblers, "rel68k" and "rel20", accept as input one or more files containing the source code. These files must be standard UniFLEX text files--that is, they must consist solely of lines of text, each of which ends with a carriage return. The source code should not contain line numbers. The only control characters allowed in the source code are the horizontal tab character (hexadecimal 09) and the carriage return (hexadecimal 0D). An assembler processes these files and, depending on the options specified, produces a single relocatable object-code module (also called a relocatable module), a listing of the assembled source code, or both.

2.2 The Command Line

Syntax statements for invoking the relocating assemblers follow:

```
rel20 <file_name_list> [<param_list>] [+abefFiIlLnosSu]
rel68k <file_name_list> [<param_list>] [+befFiIlLnosStu]
```

where <file_name_list> is a list of the names of the files to assemble and <param_list> is a list of parameters for the shell program to pass to the assembler source code. Files are assembled in the order in which they appear on the command line. Assembly ends when the assembler processes the last line of the last file.

2.2.1 Command-line Parameters

With each invocation of the assembler the user may pass it up to three parameters for use in the source code. Thus, the code generated by the assembler may differ from one invocation to the next.

2.2.1.1 Specifying parameters

The parameters passed to the assembler are ASCII strings which the user places in a particular format anywhere on the command line. The syntax for passing a parameter is

+<char>=[<str>]

where valid values for <char> are 'a', 'b', and 'c'. Normally, the shell program interprets a space character as a character separating two elements on the command line. A user may, however, include a space character in a command-line parameter by enclosing either the string or the entire parameter in a pair of delimiter characters (either single or double quotation marks) as shown in the following examples:

```
+a="test string"
'+b=Good-bye, Larry!'
```

The user may specify the null string by leaving the string out altogether:

```
+a=
'+a='
+a=""
```

2.2.1.2 Substitutable parameters

If the user specifies one or more command-line parameters, the assembler searches the source code for the sequence

&<char>

where valid values of <char> are once again 'a', 'b', and 'c'. Such a sequence is called a substitutable parameter. The assembler replaces the substitutable parameter "&a" with the string passed in as an argument to the command-line parameter 'a'. Similarly, it replaces the sequences "&b" and "&c" with the corresponding command-line parameter.

2.2.1.3 Ignoring a substitutable parameter

A user who uses command-line parameters but wishes to avoid substitution in the source code in a particular instance must precede the ampersand, '&', with a backslash character, '\'. Thus, if it is performing substitution with command-line parameters, the assembler sees the following line of source code

```
value  equ  mask\&count
```

as

```
value  equ  mask&count
```

Parameter substitution may occur in any field in the source code.

Unless the user specifies one or more elements in the parameter list, the assembler does not search the source code for substitutable parameters. If the user does specify one or more command-line parameters and the assembler finds a substitutable parameter for which the user did not specify the corresponding command-line parameter, the assembler replaces that sequence with a null string.

2.2.2 Options Available

Brief descriptions of the options which are available follow:

a	Produce an abbreviated listing of the assembled source ("rel20" only).
b	Suppress binary output.
e	Suppress summary information.
f	Disable formatting of the listing of the assembled source code.
F	Enable "fix" mode. In fix mode, comments which begin with a semicolon, ';', are assembled.
i	Ignore the suffix ":w", which forces an address to the size of a word.
I	Ignore the suffix ":w", which forces an address to the size of a word, unless it is part of a "jmp" or a "jsr" instruction.
J	Ignore the suffix ":w", which forces an address to the size of a word, when it is part of a "jmp" or "jsr" instruction.
l	Produce a listing of the assembled source.
L	Produce a listing of the input file or files during the first pass.

n	Produce line numbers with the listing.
o=<file_name>	Specifies the name of the file containing the relocatable module produced by the assembler.
s	Produce a listing of the symbol table.
S	Limit symbols internally to 8 characters.
t	Assemble for 68000 rather than 68010 ("rel68k" only). This option only affects the code generation of the "move from CCR/SR" instruction.
u	Classify all unresolved symbols as external.

Detailed descriptions of the options follow.

2.2.2.1 The 'a' option ("rel20" only)

The 'a' option tells the assembler to send a condensed listing of the entire assembled source to standard output. Some complicated addressing modes supported by the 68020 generate large amounts of code, which can make a listing difficult to read and can force the truncation of comments on printers which support only eighty columns. The 'a' option attempts to remedy this situation by producing an abbreviated listing which contains only one line of output for each instruction.

As the assembler produces the listing, it may insert a single character before the first character of a line of code to indicate something special about that line. A plus sign, '+', indicates that the line contains a relocatable address; an 'X', that it contains an external reference. If a line of code contains both an external and a relocatable reference, the assembler inserts the character corresponding to the last reference on the line. The assembler also marks the beginning of each line in which the user can replace a long branch with a short branch. These excessive branches are flagged so that the user can optimize the final code. The indicator character is a greater-than sign, '>'. All three indicator characters are part of the listing only; they do not exist in the binary file.

Each line of the listing generated by the 'a' option includes any indicator character, the program counter, the first two bytes of the code generated by the instruction, the label, the instruction, the operands, and as much of any comment as possible.

2.2.2.2 The 'b' option

The 'b' option suppresses the creation of a binary file (even if the user invokes the 'o' option to specify a name for the file). This option is useful when the user wants either to check for errors in an incomplete program or to obtain only a listing of the assembled source (see Section 2.2.2.9).

2.2.2.3 The 'e' option

By default, at the end of an assembly the assembler reports the size of each segment in the relocatable module as well as the number of error messages and excessive branches (see Section 2.2.2.1) generated by the source code. The 'e' option suppresses this summary information if the source code is free of errors. If the code is not error-free, the report is not suppressed (excessive branching is not considered an error).

2.2.2.4 The 'f' option

By default, the assembler formats the fields in the file containing the assembled source code (see Section 3.6). The 'f' option suppresses the field-formatting feature. Therefore, when the 'f' option is in effect, the text of the assembled source code is identical to the input file.

2.2.2.5 The 'F' option

The 'F' option enables "fix" mode. By default, the assembler processes as a comment (see Section 3.1) any line of code that contains a semicolon, ';', in the first column. However, when the assembler is in fix mode, it ignores a semicolon in the first column of a line and treats the remainder of the line as normal source code. If the second column contains a semicolon, an asterisk, or a carriage return, the line of code is still considered a comment.

2.2.2.6 The 'i' option

The 'i' option tells the assembler to ignore the suffix ":w", which forces an address to be the size of a word.

2.2.2.7 The 'I' option

The 'I' option tells the assembler to ignore the suffix ":w", which forces an address to be the size of a word, unless it is part of a "jmp" or a "jsr" instruction.

2.2.2.8 The 'J' option

The 'J' option tells the assembler to ignore the suffix ":w", which forces an address to be the size of a word, when it is part of a "jmp" or a "jsr" instruction.

2.2.2.9 The 'l' option

By default, the assembler writes each line of the assembled file containing an error to standard output. The 'l' option, like the 'a' option, tells the assembler to send a listing of the entire assembled source to standard output. As the assembler produces the listing, it may insert a single character before the first character of a line of code to indicate something special about that line. A plus sign, '+', indicates that the line contains a relocatable address; an 'X', that it contains an external reference. If a line of code contains both an external and a relocatable reference, the assembler inserts the character corresponding to the last reference on the line. The assembler also marks the beginning of each line in which the user can replace a long branch with a short branch. These excessive branches are flagged so that the user can optimize the final code. The indicator character is a greater-than sign, '>'. All three indicator characters are part of the listing only; they do not exist in the binary file.

The assembler produces the listing during the second pass over the file. It honors the "lis" and "nol" options to the "opt" directive (see Section 5.2.42).

2.2.2.10 The 'L' option

If the user specifies the 'L' option, the assembler sends to standard output a listing of the file containing the source code. The assembler numbers the lines as it writes them.

2.2.2.11 The 'n' option

This option, which must be used with the 'l' option in order to be effective (see Section 2.2.2.9), tells the assembler to assign line numbers to each line of the assembled source code. Line numbers begin with 1 and are incremented by 1.

2.2.2.12 The ´o´ option

By default, the assembler names the file containing the relocatable module it creates "<file_name_1>.r", where <file_name_1> is the name of the first file specified on the command line. The user may, however, override the selection of this name by using the ´o´ option. The syntax for this option is

o=<file_name>

where <file_name> is the name to give to the file containing the relocatable module.

2.2.2.13 The ´s´ option

The ´s´ option tells the assembler to write the symbol table to standard output at the end of the assembly. If the name of the symbol is preceded by an asterisk, ´*´, the symbol is global; otherwise, it is local.

2.2.2.14 The ´S´ option

By default, the assembler allows the user to define and use symbols of any length but recognizes only the first 255 characters. If the user invokes the ´S´ option, the assembler limits the internal representation of each symbol to eight characters.

2.2.2.15 The ´t´ option ("rel68k" only)

By default, "rel68k" produces code for the 68010 rather than the 68000. Assemblies for the 68000 and the 68010 differ only in the code generation for the instruction "Move from CCR", which is not supported by the 68000 microprocessor. A user assembling code which uses this instruction and is intended for use on a 68000-based machine, should invoke the ´t´ option. When this option is in effect, the assembler generates a "Move from SR" instruction for "Move from CCR". In addition, it disallows the use of any instructions which are not supported by the 68000.

2.2.2.16 The 'u' option

By default, the assembler reports as undefined any symbol which is neither defined nor declared as an external symbol in the source code. The 'u' option tells the assembler to treat all references to such symbols as external references.

2.3 Examples

1. rel68k asmfile
2. rel68k test.a +euo=test.r +a=D0
3. rel20 test.a test2.a test3.a +blns

The first example uses the 68000/68010 assembler to assemble the source file "asmfile" and produces the relocatable binary file "asmfile.r". The assembler sends summary information to standard output but produces no source listing. Any nonfatal errors detected are sent to standard output. The assembler does not search the source for substitutable parameters because the user did not specify any command-line parameters.

The second example uses the 68000/68010 assembler to assemble the file "test.a" and produces the relocatable file "test.r". No summary information is produced, and all unresolved references are classified as external. If the assembler detects no errors during the assembly, the user sees no output from this command. The assembler searches the source code for all substitutable parameters and replaces the sequence "&a" with the string "D0". It replaces any occurrences of "&b" and "&c" with the null string.

The third example uses the 68020 assembler to assemble the three files specified but produces no binary output. A listing with a symbol table is sent to standard output. The listing includes line numbers. The assembler does not search the source for substitutable parameters because the user did not specify any command-line parameters.

Chapter 3

Components of the Source Code

3.1 Introduction

The 68xxx Relocating Assembler is a two-pass assembler. During the first pass the assembler constructs a symbolic reference table; during the second pass, it assembles the source code. Each line of code must consist of a series of ASCII characters terminated by a carriage return (hexadecimal 0D). Certain ASCII characters have special meanings to the assembler (see both the discussion of comments in this section and Section 3.8.1). Control characters (hexadecimal 00 to 1F) other than the carriage return and the horizontal tab character (hexadecimal 09) may not be used in the code (their inclusion produces unpredictable results).

A line of code may consist of up to four fields as shown here:

```
     <label>
or
     [<label>] <opcode> [<operand_list>] [<comment>]
or
     <label> <opcode> [<operand_list>] [<comment>]
```

Each field is separated from the following field by one or more spaces or tab characters. If the opcode does not take an operand, the assembler treats the third field as the comment field. No field except the comment field may contain a space character.

The user may designate entire lines of the source as comments. These comments are ignored by the assembler when it is generating object code. They do, however, appear in the listing of the assembled source code (if the user requests one) exactly as they appear in the source code itself. The assembler recognizes as a comment any line which (1) contains an asterisk, '*', in the first column, (2) contains a semicolon, ';', in the first column (see Section 2.2.2.5), or (3) contains only a carriage return.

3.2 The Label Field

The label field contains a symbolic label which is assigned the instruction's address. The user may reference such a label anywhere in the source code. Each label must begin in the first column of the line on which it appears. If a line does not contain a label, it must begin with either a space or a tab character. The assembler supports two types of label: ordinary and local.

3.2.1 Ordinary Labels

An ordinary label must be unique to the program unless it is a label associated with the "log", "set", or "struct" directive. The label may consist of uppercase letters ('A' through 'Z'), lowercase letters ('a' through 'z'), the underscore character, '_' (hexadecimal 5F), the question mark, '?' (hexadecimal 3F), and the digits ('0' through '9'). However, the first character of an ordinary label may not be a digit. The assembler does distinguish between upper- and lowercase letters in a label. Thus, the label "ABC" is different from the label "Abc". Ordinary labels may be of any length, but the assembler recognizes only the first two hundred fifty-five characters.

3.2.2 Local Labels

Local labels may be used just like ordinary labels with the following exceptions: they may not be global; they may not be external; they may not be used in the label field of a "bfequ", "common", "equ", "log", "macro", or "set" directive. Local labels are used primarily for branching or jumping around small sections of code. They free the programmer from the necessity of thinking of a symbolic name for every label. They require less internal storage than ordinary labels, and the assembler processes them faster.

A local label consists of a string of one or two digits ('0' through '9'). Both digits are significant. Thus, the label "00" is different from the label "0".

The user references a local label by adding one of two letters to the label: an 'f', for forwards, indicates the first occurrence of the label following the reference; a 'b', for backwards, indicates the first occurrence preceding the reference. Such a reference never refers to the same line it is on. For example, in the following example both

references point to the same line of code:

```
2   beq 2f   "2f" => following occurrence of local label 2
2   jsr xx   both branches point to this line of code
2   bra 2b   "2b" => preceding occurrence of local label 2
```

3.3 The Opcode Field

The opcode field contains the instruction--an opcode (mnemonic), a directive (pseudo-op), or the name of a macro--which specifies the operation to be performed. Valid opcodes are detailed in Chapter 4; valid directives, in Chapter 5. Upper- and lowercase letters may be used interchangeably in the opcode field.

The user may append a suffix, which indicates the size of the operand on which the operation is to be performed, to certain instructions. The choices are shown in the following tables:

Table 3-1. Suffixes Common to the 68000/68010 and 68020 Assemblers

Suffix	Meaning
.b or .B	Byte (8 bits)
.l or .L	Long word (32 bits)
.w or .W	Word (16 bits)

Table 3-2. Suffixes Supported Only by the 68020 Assembler

Suffix	Meaning
.d or .D	Double-precision floating-point (64 bits)
.p or .P	Packed-decimal floating-point (96 bits)
.s or .S	Single-precision floating-point (32 bits)
.x or .X	Extended-precision floating-point (96 bits)

If the instruction imposes no restraints on the size of the operand and the user does not specify a size, the assembler performs the operation on a word.

The user can force the generation of a short branch (8-bit offset) by appending the suffix ".s" or ".S" to a branch instruction. By default, the assembler generates a word-length branch (16-bit offset) for each forward reference and the shortest possible branch for a backward reference. The user of "rel20" can force the generation of a long branch for a forward or backward reference by appending the suffix 'l' or 'L' to the branch instruction.

3.4 The Operand Field

The operand field contains information on how to locate the operands necessary for the instruction in the opcode field. An operand consists of a combination of register specifications and mathematical expressions (see Sections 3.7-8). The assembler determines whether or not the instruction in the opcode field requires an operand. If it does, it interprets the following field as the operand field; otherwise, it interprets it as the comment field.

3.5 The Comment Field

The programmer may use the comment field to insert a comment on any line of the source code. Comments are only for the programmer's convenience; the assembler ignores them. Comments may contain any characters ranging from the space character (hexadecimal 20) to the delete character (hexadecimal 7F), as well as the tab character (hexadecimal 09).

3.6 Automatic Formatting

The "rel68k" assembler automatically formats the listing of the assembled source code as follows: the label field begins in column 25; the opcode field, in 34; the operand field, in 42; and the comment field, in 56--assuming that the operand field does not extend into the comment field. By default, the "rel20" assembler automatically formats the listing of the assembled source code as follows: the label field begins in column 29; the opcode field, in 38; the operand field, in 42; and the comment field, in 62--assuming that the operand field does not extend into the comment field. If the user invokes the 'a' option, "rel20" automatically formats the listing as follows: the label field begins in column 17; the opcode field, in 26; the operand field, in 36;

and the comment field, in 50--assuming that the operand field does not extend into the comment field.

Thus, the programmer can edit the source file without impairing the readability of the listing of the assembled code. The assembler prints any label that contains more than eight characters on a line by itself, just above any code generated by the same line of source code.

In a few cases, such as lines containing errors, the automatic formatting may break down. Such cases, however, are rare, and in general they should cause no problems.

3.7 Specification of Registers by Operands

Many opcodes require that the corresponding operand specify one or more registers. Upper- and lowercase letters may be used interchangeably in the specification of registers. The following tables describe the available registers.

Table 3-3. Registers Common to the 68000/68010 and 68020 Assemblers

Name	Application
D0-D7	Data registers
A0-A6	Address registers
A7 or SP	Current stack-pointer
CCR	Condition-code register (part of SR)
PC	Program counter
SR	Status register
SSP	Supervisor stack-pointer
USP	User stack-pointer

Table 3-4. Registers Common to the 68010 and 68020 Assemblers

Name	Application
DFC	Destination function-code register
SFC	Source function-code register
VBR	Vector-base register

Table 3-5. Registers Available Only for the 68020 Assembler

Name	Application
CAAR	Cache-address register
CACR	Cache control-register
FPCR	Floating-point control-register
FPIAR	Floating-point instruction address register
FPSR	Floating-point status-register
ISP	Interrupt stack-pointer
MSP	Master stack-pointer

3.8 Expressions

Many instructions require that the corresponding operand supply further information in the form of an expression. An expression consists of one or more items (see Section 3.8.1) combined by any of the four kinds of operator: arithmetic, logical, relational, and shift (see Section 3.8.2). It may contain neither space nor tab characters. An expression is evaluated during the assembly, and the result becomes a permanent part of the program.

The 68020 assembler evaluates floating-point expressions using double precision (64 bits). Otherwise, expressions are always evaluated to 32 bits. The assembler ignores overflow. If the result of the operation is to be a byte or a word, the assembler uses the low-order portion of the expression.

3.8.1 Items

An expression consists of up to four kinds of item--numerical constants, ASCII constants, labels, and addresses. An item may stand alone as an operand or may be combined with other items by the use of operators.

3.8.1.1 Numerical constants

The programmer can supply a numerical constant to the assembler in any of four bases: binary, decimal, hexadecimal, or octal. The desired base is specified by a character preceding the number, as illustrated in the following table. By default, the assembler assumes that a number is decimal.

Base	Prefix	Characters Allowed
Binary	%	0 and 1
Decimal	None	0 - 9
Hexadecimal	$	0 - 9, A - F, a - f
Octal	@	0 - 7

A floating-point constant consists of a mantissa optionally followed by an exponent. The mantissa consists of an optionally signed string of decimal digits, which may or may not contain a decimal point in any position. The exponent consists of an 'E' or an 'e' character optionally followed by a sign and a string of decimal digits.

3.8.1.2 ASCII constants

The programmer can supply an ASCII constant to the assembler by enclosing the desired string in single or double quotation marks. The string may consist of from one to four characters, depending on the size attribute associated with the instruction. The characters in the string may not include control characters.

3.8.1.3 Labels

Any ordinary or local label (see Section 3.2) may be used in an expression.

3.8.1.4 Current address

The asterisk, '*', when used as an item in an expression, denotes the address of the current instruction (the content of the program counter). Depending on the context in which it is used, the asterisk may represent either a relocatable or an absolute value.

3.8.2 Operators

Four classes of operator are available for use in an expression: arithmetic, logical, relational, and shift. Only the addition and subtraction operators may be used with relocatable and external symbols and expressions (see Section 3.8.4). Only the arithmetic operators may be applied to floating-point expressions and constants. An operation on operands of mixed modes produces a floating-point value as the result.

3.8.2.1 Arithmetic operators

The arithmetic operators are shown in the following table:

Operator	Meaning
+	Unary or binary addition
-	Unary or binary subtraction
*	Multiplication
/	Division

3.8.2.2 Logical operators

The logical operators are shown in the following table:

Operator	Meaning
&	Logical "and" operator
|	Logical "or" operator
!	Logical "not" operator
>>	Shift right operator
<<	Shift left operator

The assembler performs logical operations bit-by-bit on 32-bit values. For example, in an "and" operation, every bit from the first operand is "anded" with the corresponding bit from the second operand. The shift operators shift the left-hand term by the number of places indicated by the right-hand term. Zeros are shifted in, and any digits shifted out are lost.

3.8.2.3 Relational operators

The following table summarizes the relational operators:

Operator	Meaning
=	Equal to
<	Less than
>	Greater than
<>	Not equal to
<=	Less than or equal to
>=	Greater than or equal to

A relational operator yields a result that is true or false. If the expression evaluates to true, each bit of the result is set to 1; if false, to 0. Relational operators are generally used in conjunction with conditional assembly.

3.8.3 Operator Precedence

The assembler evaluates certain operators in an expression before evaluating others. It evaluates operators of equal precedence from left to right. The user can overcome the default precedence by using parentheses.

The following list shows the overall operator precedence. The operator at the top of the list has the highest precedence; the one at the bottom, the lowest.

1. Expressions in parentheses
2. Unary addition and subtraction
3. Shift operators
4. Multiplication and division
5. Binary addition and subtraction
6. Relational operators
7. Logical "not"
8. Logical "and" and "or"

3.8.4 Types of Expression

The user can combine items and operators to produce three types of expression: absolute, relocatable, and external. It is essential to be able to distinguish between the different types of expression because certain instructions require the use of a certain type of expression. For example, the operand of the "rmb" instruction must be an absolute expression.

3.8.4.1 Absolute expressions

An expression is absolute if its value is unaffected by program relocation. An absolute expression may contain relocatable items as long as those items are grouped in pairs which represent the difference between two addresses in the same segment. For example, if "text_1" and "text_2" are both relocatable symbols in the text segment and "data_1" and "data_2" are relocatable symbols in the data segment, the following expressions are absolute:

```
text_1-text_2
5*(text_1-text_2)
5*(text_1-text_2)+(data_1-data_2)
```

The following expressions are not legal absolute expressions:

```
text_1 - text_2  (Spaces not allowed in an expression.)
text_1+text_2    (Sum of two relocatables is not absolute.)
(text_1-data_1)  (Paired symbols must be from the same segment.)
```

3.8.4.2 Relocatable expressions

An expression is relocatable if its value is affected by program relocation. A relocatable expression must contain an odd number of relocatable symbols. All relocatable symbols except one must be grouped in pairs that represent the difference between two addresses in the same segment. The single unpaired symbol may have a positive or negative sign. The following expressions are all relocatable:

```
-bss_2+3*5+(data_1-data_2)
text1+(data_1-data_2)+(bss_1-bss_2)
data_1-(bss_2-bss_1)
*+5
```

The first example represents negative relocation from the bss segment; the second, positive relocation from the text segment; the third, positive relocation from the data segment; and the fourth, positive relocation from the current segment, whatever that may be.

3.8.4.3 External expressions

An expression is external if its value depends on the value of a symbol defined outside the source module. An external expression may contain only one external symbol. It may also contain relocatable and absolute symbols as long as the relocatable symbols are grouped in pairs which represent the difference between two addresses in the same segment. The following are valid external expressions:

```
ext_1+(bss_1-bss_2)+(data_1-data_2)
ext_1-3
```

Chapter 4

68xxx Opcodes

4.1 Deviations from Motorola Standard

This section describes the differences in the mnemonics accepted by the 68xxx Relocating Assembler and the Motorola standard as defined in the M68000 16/32-Bit Microprocessor Programmer's Reference Manual (Motorola, 1984) and the MC68020 32-Bit Microprocessor User's Manual (Motorola, 1985a). It is assumed that the reader is familiar with the contents of Chapter 2 ("Data Organization and Addressing Capabilities") and Chapter 3 ("Instruction Set Summary") of the relevant manual. In particular, the user should be familiar with the description of the assembler syntax that accompanies the discussion of the individual instructions.

The 68xxx Relocating Assembler recognizes the standard instruction set with the exception of the "address", "quick", and "immediate" variations. Rather than having a specific opcode for these variations, the assembler infers their existence from an analysis of the operands and generates the proper code. Therefore, even though the assembler does not recognize these mnemonics, it can generate code for address, quick, and immediate instructions. This process frees the programmer from the need for remembering individual variations of each opcode. The following table shows the standard instructions which are not recognized by the assembler and the opcode that should be used in place of each one. Note that the "extend" variation is supported.

Opcode Variation	Opcode to Use
adda, addq, addi	add
andi	and
cmpa, cmpi, cmpm	cmp
eori	eor
movea, moveq	move
ori	or
suba, subq, subi	sub

4.2 Available Registers

The 68xxx microprocessor has sixteen 32-bit general-purpose registers and several special registers, including a 32-bit program counter and an 8-bit condition-code register. Table 4-1 shows the registers that are available in user state.

Table 4-1. Registers Available in User State

Register	Application	Microprocessor		
		68000	68010	68020
D0-D7	Data registers	X	X	X
A0-A6	Address registers	X	X	X
A7 or SP	Current stack-pointer	X	X	X
CCR	Condition-code register	X	X	X
FP0-FP7	Floating-point registers			X
FPCR	Floating-point control register			X
FPIAR	Floating-point instruction address register			X
FPSR	Floating-point status register			X
PC	Program counter	X	X	X

Table 4-2 shows the registers available in supervisor state.

Table 4-2. Registers Available in Supervisor State

Register	Application	Microprocessor		
		68000	68010	68020
CAAR	Cache-address register			X
CACR	Cache control-register			X
DFC	Destination function-code register		X	X
ISP	Interrupt stack-pointer			X
MSP	Master stack-pointer			X
SFC	Source function-code register		X	X
SR	Status register	X	X	X
SSP	Supervisor stack-pointer	X	X	X
USP	User stack-pointer	X	X	X
VBR	Vector-base register		X	X

Data registers may be used for 8-bit, 16-bit, and 32-bit operations. Address registers, on the other hand, may only be used for 16-bit and 32-bit operations. Address registers may be used as base registers. Both address and data registers may be used as index registers.

4.3 Introduction to Addressing Modes

Most opcodes require one or more operands, which may be stored either in memory or in a register. An addressing mode is a means of supplying the machine with the information it needs to calculate the address of the operand. The user specifies the addressing mode by combining data registers, expressions, or both in a particular way in the operand field of the code.

4.3.1 Length of Assembled Instructions

An assembled instruction uses a minimum of one word of storage, but depending on the addressing modes, it may use as many as five words on the 68000 and 68010 microprocessors and as many as eleven words on the 68020. The first word of the instruction defines the operation and the length of the instruction. Different addressing modes generate instructions of different length depending on the information needed to calculate the address of the operand or operands. The words beyond the first word of an instruction are called words of extension.

4.3.2 Use of Index Registers

The 68xxx assembler allows the user to specify how much of an index register to use by appending a suffix to the specification of the index register as follows:

.<size>

where <size> may be an 'L' or an 'l' for the entire 32 bits or a 'W' or a 'w' for the low-order 16 bits. By default, the assembler uses the low-order 16 bits.

The 68020 assembler allows the user to "scale" the index register by a factor of 1, 2, 4, or 8. In such a case the assembler multiplies the contents of the index register by the specified scale factor before calculating the effective address. The user simply appends the expression

*<scale>

where <scale> is the scale factor, to the description of the index register.

4.3.3 Syntax Conventions

Various conventions are used in the syntax statements which describe the addressing modes. Registers are designated by a 'D' where a data register is required, by an 'A' where an address register is required, or by an 'R', where either type of register may be used. Lowercase letters may also be used. The letter designating the register must be followed by a digit between 0 and 7 inclusive, indicated in the syntax statements by the letter 'n'. The user can always use the SP register instead of A7. Angle brackets enclose descriptive words which the user must replace with a specific item which answers to the description. For example, the term <displ> must be replaced by an expression specifying the displacement, which the machine sign-extends to a 32-bit integer. Unless otherwise stated, an addressing mode generates a one-word instruction. In cases where two operands are specified, the first operand is always the source; the second, the destination.

Documentation for the UniFLEX Operating System usually uses square brackets, '[' and ']', to delineate optional items in a syntax statement. However, this document does not because certain addressing modes use square brackets as part of their required syntax. The text describing each addressing mode identifies optional items except for the use of a size specification, a scaling factor, or both with an index register. These items are always optional.

Twelve addressing modes are available for use with all 68xxx assemblers. Some of these modes may be used in some form with all assemblers but have one or more additional forms which may only be used with the 68020. The word "Syntax" is preceded by an asterisk, '*', if the syntax statement or statements which follow it apply only to the 68020 assembler. Six addressing modes are completely restricted to use with the 68020 assembler and are labeled as such in the subheading that introduces them. Their syntax statements are also marked with an asterisk.

4.4 Descriptions of Addressing Modes

This section contains descriptions, syntax statements, and examples of all addressing modes available on the 68xxx assemblers. Appendix B contains a summary of all addressing modes available for use with the 68020 assembler. Further details are available in Appendix D of the MC68020 32-Bit Microprocessor User's Manual (Motorola, 1985a).

4.4.1 Data Register Direct

The operand is in the data register specified.

Syntax: Dn
Examples:

```
ext.l D0    Sign-extend data register 0 to 32 bits.
```

4.4.2 Address Register Direct

The operand is in the address register specified.

Syntax: An
Examples:

```
add.l A1,A2    Add the content of address register 1 to the
               content of address register 2, using all
               32-bits of each operand.
```

4.4.3 Address Register Indirect

The address of the operand is in the address register specified.

Syntax: (An)
Examples:

```
sub.l D5,(A4)    Subtract the content of data register 5
                 from the long operand at the address in
                 address register 4.
```

4.4.4 Address Register Indirect with Postincrement

The address of the operand is in the address register specified. After using the address, the microprocessor normally increments it by 1 if the operand is a byte, by 2 if it is a word, or by 4 if it is a long word. If, however, the register is the current stack-pointer (A7 or SP) and the operand is a byte, the microprocessor increments the address by 2.

Syntax: (An)+
Examples:

```
clr.w (A5)+   Clear the word at the address in address
              register 5. Then increment the content of
              the register by 2.
```

4.4.5 Address Register Indirect with Predecrement

The address of the operand is in the address register specified. Before using the address, the microprocessor normally decrements it by 1 if the operand is a byte, by 2 if it is a word, or by 4 if it is a long word. If, however, the register is the current stack-pointer (A7 or SP) and the operand is a byte, the microprocessor decrements the address by 2.

Syntax: -(An)
Examples:

```
clr.b -(A3)   Decrement the content of address register 3
              by 1. Then clear the byte at the address in
              register 3.
```

4.4.6 Address Register Indirect with Displacement

The address of the operand is the sum of the displacement plus the address in the address register. The displacement must be an absolute, 16-bit expression. This addressing mode requires one word of extension.

Syntax: <displ>(An)
*Syntax: (<displ>,An)

Examples:

```
move.l 6(A0),D1    Move the four bytes at the address that
                   is the sum of 6 and the address in
                   address register 0 into data register 1.
```

4.4.7 Address Register Indirect with Index

The address of the operand is the sum of the displacement, the address in the address register, and the content (possibly scaled) of the index register. The displacement is optional. If specified, it must be an absolute, 8-bit expression. If the user does not specify a value for the displacement, it defaults to 0. This addressing mode requires one word of extension.

```
 Syntax:  <displ>(An,Rn.<size>)
*Syntax:  (<displ>,An,Rn.<size>*<scale>)
 Examples:

    clr.l $A(A1,D1.W)     Clear the four bytes at the address
                          specified by the sum of the
                          displacement (hexadecimal A), the
                          address in address register 1, and the
                          low-order 16 bits of the content of
                          data register 1.

    tst.l (A2,A3.L)       Test the four bytes at the address
                          specified by the sum of the contents of
                          address registers 2 and 3. The value
                          of the displacement defaults to 0.

    68020 only:
    clr.w (4,A2,D2.L*2)   Clear the two bytes at the address
                          specified by the sum of the
                          displacement (4), the content of
                          address register 2, and 2 times the
                          32-bit content of data register 2.
```

4.4.8 Address Register Indirect with Index (Base-displacement) (68020 only)

The address of the operand is the sum of the displacement, the address in the address register, and the contents (possibly scaled) of the index register. The displacement is either 16 or 32 bits. This addressing

mode requires one, two, or three words of extension. All three components are optional. The default for any component which the user does not specify is 0.

Syntax: <displ>(An,Rn.<size><scale>)
(<displ>,An,Rn.<size>*<scale>)
Examples:

clr.L ($9A,A0,D0.L*2)	Clear the four bytes at the address specified by the sum of the displacement (hexadecimal 9A), the address in address register 0, and two times the 32-bit value stored in data register 0.
clr.l (5,D2)	Clear the four bytes at the address specified by the sum of the displacement (5) and the low-order 16 bits of the value stored in data register 2. The value of the address register defaults to 0.

4.4.9 Memory Indirect Postindexed (68020 only)

The processor first calculates an intermediate address, which is the sum of the content of the address register and the sign-extended displacement. It then calculates the effective address by adding the long word at the intermediate address to the outer displacement and the (possibly scaled) content of the index register. All four components are optional. The default for any component which the user does not specify is 0. This addressing mode requires from one to five words of extension.

Syntax: ([<displ>,An],Rn.<size><scale>,<outer_displ>)
Examples:

tst.l ([$10,A3],D1.W*4,$1A)	Calculate an intermediate address by adding the content of address register 3 to the displacement (hexadecimal 10). Multiply the low-order sixteen bits in data register 1 by 4, and add the result to the outer displacement (hexadecimal 1A) and the long word stored at the

	intermediate address to determine the effective address. Test the four bytes at the effective address.
clr.w ([A2],D2.L)	Calculate an intermediate address by adding the content of address register 2 to the default displacement of 0. Add the long word stored in data register 2 to the default outer displacement of 0 and the long word stored at the intermediate address to determine the effective address. Clear the low-order sixteen bits stored at the effective address.

4.4.10 Memory Indirect Preindexed (68020 only)

The processor first calculates an intermediate address, which is the sum of the address in the address register, the sign-extended displacement, and the (possibly scaled) content of the index register. It then calculates the effective address by adding the long word at the intermediate address to the outer displacement. All four components are optional. The default for any component which the user does not specify is 0. This addressing mode requires from one to five words of extension.

Syntax: ([<displ>,An,Rn.<size><scale>],<outer_displ>)
Examples:

tst.l ([$10,A3,D1.W*4],$1A)	Calculate the intermediate address by multiplying the low-order sixteen bits in data register 1 by 4 and adding the result to the content of address register 3 and to the displacement (hexadecimal 10). Add the long word stored at the intermediate address to the outer displacement to determine the effective address. Test the four bytes at the effective address.

clr.w ([A2,D2.L])	Calculate an intermediate address by adding the content of address register 2 to the default displacement of 0 and to the long word stored in data register 2. Add the long word stored at the intermediate address to the default outer displacement of 0 to determine the effective address. Clear the low-order sixteen bits stored at the effective address.

4.4.11 Absolute Short Address

The address of the operand is the value of the absolute or relocatable label specified or, for the 68020 only, the value of the 16-bit expression specified. The value of a label must be less than 7FFF because the machine sign-extends it before use. This addressing mode requires one word of extension.

Syntax: <label>:W
*Syntax: <expr>:W
Examples:

jsr sqrt:w	Jump to the subroutine "sqrt" using a 16-bit address.
jsr table+8:w	Jump to the subroutine at the 16-bit address specified by the sum of the value of the label "table" and 8.

4.4.12 Absolute Long Address

The address of the operand is the value of the expression or of the absolute or relocatable label specified. This addressing mode requires two words of extension.

Syntax: <label_or_expr>
Examples:

jsr sqrt	Jump to the subroutine "sqrt" using a 32-bit address.

```
jsr $400300    Jump to the subroutine at hexadecimal
               location 400300.
```

4.4.13 Program-Counter Relative

This addressing mode, which is sometimes called program counter with displacement, requires one word of extension. That word contains the value of the difference between the address of the label and the program counter. The label must be a relocatable label in the same segment as the program counter. The difference between the two must be a 16-bit value. The address of the operand is the sum of the address in the program counter and the word of extension. Thus, the displacement calculated is the address of the label relative to the program counter.

The operand field for a branch instruction requires only a label. All other instructions wishing to use this addressing mode must include the program counter in the operand field to distinguish this addressing mode from the absolute-long addressing-mode.

```
 Syntax:  <label>(PC)
*Syntax:  (<label>,PC)
 Examples:

     jmp table(PC)  Jump to the address of the label "table".
```

4.4.14 Program Counter with Index

This addressing mode requires one word of extension. That word contains the value of the difference between the address of the label and the program counter. The label must be a relocatable label in the same segment as the program counter. The difference between the two must be an 8-bit value. The address of the operand is the sum of the address in the program counter, the low-order 8 bits of the word of extension, and the content of the index register. Thus, the displacement calculated is the address of the label relative to the program counter, offset by the value in the index register.

```
 Syntax:  <label>(PC,Rn.<size>)
*Syntax:  (<label>,PC,Rn.<size>*<scale>)
 Examples
```

`jmp table(PC,D1.L)`	Jump to the address that is the sum of the relative address of the label "table" with respect to the program counter and the content of data register 1.

4.4.15 Program Counter Indirect with Index (Base Displacement) (68020 only)

The address of the operand is the sum of the displacement, the program counter, and the content (possibly scaled) of the index register. The displacement is either 16 or 32 bits. This addressing mode requires one, two, or three words of extension. The index register is optional, and its default value is 0. The program counter and displacement, however, are required. A user who wants to use a value of 0 for the program counter yet wants to generate a program-space reference must use the string "ZPC" (zero program counter) in the instruction.

```
*Syntax:  <displ>(PC,Rn.<size>*<scale>)
          (<displ>,PC,Rn.<size>*<scale>)
 Examples:
```

`clr.L $A(PC,D0.L*2)`	Clear the four bytes at the address specified by the sum of the displacement (hexadecimal A), the program counter, and two times the 32-bit value stored in data register 0.
`clr.l (5,ZPC,D2)`	Clear the four bytes at the address specified by the sum of the displacement (5) and the value stored in data register 2.

4.4.16 Program Counter Memory Indirect Postindexed (68020 only)

The processor first calculates an intermediate address, which is the sum of the program counter and the sign-extended displacement. It then calculates the effective address by adding the long word at the intermediate address to the outer displacement and the (possibly scaled)

content of the index register. This addressing mode requires from one to five words of extension. The index register and the outer displacement are optional, and the default value for both is 0. The program counter and the displacement, however, are required. A user who wants to use a value of 0 for the program counter yet wants to generate a program-space reference must use the string "ZPC" (zero program counter) in the instruction.

Syntax: ([<displ>,PC],Rn.<size><scale>,<outer_displ>)
Examples:

tst.l ([$10,PC],D1.W*4,$1A)	Calculate an intermediate address by adding the contents of the program counter to the displacement (hexadecimal 10). Multiply the low-order sixteen bits in data register 1 by 4, and add the result to the outer displacement (hexadecimal 1A) and the long word stored at the intermediate address to determine the effective address. Test the four bytes at the effective address.
clr.w ([0,PC],D2.L)	Calculate an intermediate address by adding the content of the program counter to the default displacement of 0. Add the long word stored in data register 2 to the default outer displacement of 0 and the long word stored at the intermediate address to determine the effective address. Clear the low-order sixteen bits stored at the effective address.
clr.l ([5,ZPC],D2,$1A)	Calculate an intermediate address by adding the displacement (5) to the contents of the program counter (0). Add the long word stored in data register 2 to the outer displacement (hexadecimal 1A) and the long word stored at the intermediate address to

	determine the effective address. Clear the four bytes stored at the effective address.

4.4.17 Program Counter Memory Indirect Preindexed (68020 only)

The processor first calculates an intermediate address, which is the sum of the program counter, the sign-extended displacement, and the (possibly scaled) content of the index register. It then calculates the effective address by adding the long word at the intermediate address to the outer displacement. This addressing mode requires from one to five words of extension. The outer displacement and the index register are optional, and the default value in both cases is 0. The program counter and the displacement, however, are required. A user who wants to use a value of 0 for the program counter yet wants to generate a program-space reference must use the string "ZPC" (zero program counter) in the instruction.

Syntax: ([<displ>,An,Rn.<size><scale>],<outer_displ>)
Examples:

tst.l ([$10,PC,D1.W*4],$1A)	Calculate an intermediate address by multiplying the low-order sixteen bits in data register 1 by 4 and adding the result to the program counter and to the displacement (hexadecimal 10). Add the long word stored at the intermediate address to the outer displacement to determine the effective address. Test the four bytes at the effective address.

clr.w ([0,PC,D2.L])	Calculate an intermediate address by adding the program counter to the displacement (0) and to the long word stored in data register 2. Add the long word stored at the intermediate address to the default outer displacement of 0 to determine the effective address. Clear the low-order sixteen bits stored at the effective address.
clr.l ([5,ZPC,D2],$1A)	Calculate an intermediate address by adding the displacement (5) to the program counter (0) and to the long word stored in data register 2. Add the long word stored at the intermediate address to the outer displacement (hexadecimal 1A) to determine the effective address. Clear the four bytes stored at the effective address.

4.4.18 Immediate Data

The operand is the constant in the operand field. This addressing mode requires one or two words of extension depending on the size of the operation.

Syntax: #<expr>
Examples:

move.w #4096,D2	Move the decimal value 4096 into data register 2.

4.5 Opcodes

This section contains a brief listing of all the opcodes accepted by the 68xxx assembler. The following conventions appear in the syntax statements:

An	Address register 'n' ('n' between 0 and 7 inclusive)
<bit_mask>	A 16-bit mask specifying which registers to move in a "movem" or an "fmovem" (68020 only) instruction. In predecrement addressing mode, the bit correspondence for the "movem" instruction is as follows:

Bit	Register
0	Address register 7
1	Address register 6
.	.
.	.
.	.
7	Address register 0
8	Data register 7
9	Data register 6
.	.
.	.
.	.
15	Data register 0

In predecrement addressing mode, the bit correspondence for the "fmovem" instruction is as follows:

Bit	Register
0	Floating-point register 0
1	Floating-point register 1
.	.
.	.
.	.
7	Floating-point register 7
8-15	Unused

In all addressing modes other than predecrement, the bit correspondence for the "movem"

instruction is as follows:

Bit	Register
0	Data register 0
1	Data register 1
.	.
.	.
.	.
7	Data register 7
8	Address register 0
9	Address register 1
.	.
.	.
.	.
15	Address register 7

In all addressing modes other than predecrement, the bit correspondence for the "fmovem" instruction is as follows:

Bit	Register
0	Floating-point register 7
1	Floating-point register 6
.	.
.	.
.	.
7	Floating-point register 0
8-15	Unused

This rightmost bit is bit 0; the leftmost, bit 15.

<cc>	A character string representing a condition code.
Dc	Data register used as compare operand for the "cas" opcode.
Dh	Data register for high-order 32 bits.
Dl	Data register for low-order 32 bits.
Dn	Data register 'n' ('n' between 0 and 7 inclusive).
{Dn}	K-factor for move packed-decimal data ("fmove").
Dq	Data register for quotient.
Dr	Data register for remainder.
Du	Data register used as update operand for the "cas" opcode.
<data>	8-, 16- or 32-bit data value.
<data(8)>	8-bit data value.

<data(16)>	16-bit data value.
<data(32)>	32-bit data value.
<disp>	8-, 16- or 32-bit displacement value.
<disp(8)>	8-bit displacement value.
<disp(16)>	16-bit displacement value.
<disp(32)>	32-bit displacement value.
<ea>	Effective address.
FPcr	Floating-point control register (FPCR, FPIAR, or FPSR).
<FPcr_list>	A floating-point control-register list is used with certain opcodes supported by the 68020 assembler. Such a list may consist of any combination of floating-point control registers (FPCR, FPIAR, and FRSR)in any order. Each element in the list except the last one must be followed by a slash character, '/'.
FPm	Floating-point register 'm' ('m' between 0 and 7 inclusive).
FPn	Floating-point register 'n' ('n' between 0 and 7 inclusive).
{#k}	K-factor for move packed-decimal data ("fmove").
<label>	A label in the source code.
{o:w}	Offset (o) and width (w) for bit-field instructions where 'o' is between 0 and 31 inclusive, or is a data register, and 'w' is between 1 and 32 inclusive, or is a data register.
<quick>	A data value between 1 and 8 inclusive (quick value).
Rc	Control register (DFC, SFC, USP, or VBR; also CAAR, CACR, ISP, or MSP for 68020).
<reg_list>	A register list is used with the "movem" and "fmovem" (68020 only) opcodes. Such a list can be formed in two ways. The programmer may use the slash character, '/', to separate the names of the registers whose contents are to be moved. With this method, the user must list each register individually: D1/D3/D5/A2/A3 Alternatively, the programmer may specify a range of contiguous registers by separating two names with a hyphen, '-': D1-D5/A1-A3 This register list includes registers D1, D2, D3, D4, D5, A1, A2, and A3.

Rn	Either data or address register 'n' ('n' between 0 and 7 inclusive).
<rom_offset>	A data value between 0 and 63 inclusive.
<vector>	A vector number between 0 and 15 inclusive.

Table 4-1 shows the addressing modes available on the 68xxx assembler. The table groups these modes into the categories referred to in the descriptions accompanying the list.

Table 4-1. Categories of Addressing Modes

Mode	Category					
	Data addressing	Control addressing	Alterable	Data alterable	Memory alterable	Control alterable
Data register direct	X		X	X		
Address register direct			X			
Address register indirect	X	X	X	X	X	X
Address register indirect with postincrement	X		X	X	X	
Address register indirect with predecrement	X		X	X	X	
Address register indirect with displacement	X	X	X	X	X	X
Address register indirect with index	X	X	X	X	X	X
Absolute short address	X	X	X	X	X	X
Absolute long address	X	X	X	X	X	X
Program-counter relative	X	X				
Program counter with index	X	X				
Immediate data	X					

Table 4-2 shows the addressing modes available only on the 68020 assembler. The table groups these modes into the categories referred to in the descriptions accompanying the list.

Table 4-2. Categories of Addressing Modes Available for 68020

Mode	Category					
	Data addressing	Control addressing	Alterable	Data alterable	Memory alterable	Control alterable
Address register indirect with index (base disp)	X	X	X	X	X	X
Memory indirect postindexed	X	X	X	X	X	X
Memory indirect preindexed	X	X	X	X	X	X
Program-counter with index (base displacement)	X	X				
Program counter memory indirect postindexed	X	X				
Program counter memory indirect preindexed	X	X				

An alphabetic list of the available opcodes follows. The names of opcodes which may only be used on the 68020 assembler are preceded by an asterisk, '*'. The names of opcodes which may be used on the 68010 and 68020 assemblers but not on the 68000 assembler are preceded by a plus sign, '+'. Similarly, the headings "Syntax", "Source", and "Dest." are preceded by an asterisk if the following text applies only to the 68020 assembler; with a plus sign, if only to the 68010 and 68020 assemblers. Opcodes and headings which appear without either of these indicators apply to all 68xxx assemblers.

```
abcd        Function:    Add decimal with extend
            Syntax:      abcd Dy,Dx
                         abcd -(Ay),-(Ax)
```

add	Function:	Add binary
	Syntax:	add <ea>,Dn
		add Dn,<ea>
		add <ea>,An
		add #<data>,<ea>
	Source <ea>:	All addressing modes
	Dest. <ea>:	Data alterable addressing modes
addx	Function:	Add extended
	Syntax:	addx Dy,Dx
		addx -(Ay),-(Ax)
and	Function:	and logical
	Syntax:	and <ea>,Dn
		and Dn,<ea>
		and #<data>,<ea>
		and #<data(8)>,CCR
		and #<data(16)>,SR
	Source <ea>:	Data addressing modes
	Dest. <ea>:	Data alterable addressing modes
asl	Function:	Arithmetic shift left
	Syntax:	asl Dx,Dy
		asl #<quick>,Dn
		asl <ea>
	Source <ea>:	Memory alterable addressing modes (word only)
asr	Function:	Arithmetic shift right
	Syntax:	asr Dx,Dy
		asr #<quick>,Dn
		asr <ea>
	Source <ea>:	Memory alterable addressing modes (word only)
b<cc>	Function:	Branch conditionally
	Syntax:	b<cc> <label>
	Choices:	bcc Branch on carry clear
		bcs Branch on carry set
		beq Branch on equal
		bge Branch on greater or equal
		bgt Branch on greater
		bhi Branch on high
		bhs Branch on high or same (bcc)
		ble Branch on less or equal
		blo Branch on low (bcs)
		bls Branch on low or same
		blt Branch on less than
		bmi Branch on minus
		bne Branch on not equal
		bpl Branch on plus

		bra Branch always (unconditionally)
		bvc Branch on overflow clear
		bvs Branch on overflow set
bchg	Function:	Test a bit and change
	Syntax:	bchg Dn,<ea>
		bchg #<data(8)>,<ea>
	Dest. <ea>:	Data alterable addressing modes
bclr	Function:	Test a bit and clear
	Syntax:	bclr Dn,<ea>
		bclr #<data(8)>,<ea>
	Dest. <ea>:	Data alterable addressing modes
*bfchg	Function:	Test a bit field and change
	Syntax:	bfchg <ea>{o:w}
	Source <ea>:	Control alterable or data direct addressing modes
*bfclr	Function:	Test a bit field and clear
	Syntax:	bfclr <ea>{o:w}
	Source <ea>:	Control alterable or data direct addressing modes
*bfexts	Function:	Extract bit field signed
	Syntax:	bfexts <ea>{o:w},Dn
	Source <ea>:	Control alterable or data direct addressing modes
*bfextu	Function:	Extract bit field unsigned
	Syntax:	bfextu <ea>{o:w},Dn
	Source <ea>:	Control alterable or data direct addressing modes
*bfffo	Function:	Find first 1 in bit field
	Syntax:	bfffo <ea>{o:w},Dn
	Source <ea>:	Control alterable or data direct addressing modes
*bfins	Function:	Insert bit field
	Syntax:	bfins Dn,<ea>{o:w}
	Dest. <ea>:	Control alterable or data direct addressing modes
*bfset	Function:	Set bit field
	Syntax:	bfset <ea>{o:w}
	Source <ea>:	Control alterable or data direct addressing modes

*bftst	Function:	Test bit field
	Syntax:	bftst <ea>{o:w}
	Source <ea>:	Control alterable or data direct addressing modes
*bkpt	Function:	Break point
	Syntax:	bkpt #<vector>
bset	Function:	Test a bit and set
	Syntax:	bset Dn,<ea>
		bset #<data(8)>,<ea>
	Dest. <ea>:	Data alterable addressing modes
bsr	Function:	Branch to subroutine
	Syntax:	bsr <label>
btst	Function:	Test a bit
	Syntax:	btst Dn,<ea>
		btst #<data(8)>,<ea>
	Dest. <ea>:	Data addressing modes
*callm	Function:	Call module
	Syntax:	callm #<data(8)>,<ea>
	Dest. <ea>:	Control addressing modes
*cas	Function:	Compare and swap with operand
	Syntax:	cas Dc,Dn,<ea>
	Dest. <ea>:	Memory alterable addressing modes
*cas2	Function:	Compare and swap with operand
	Syntax:	cas2 Dc1:Dc2,Dn1:Dn2,(Rn1):(Rn2)
chk	Function:	Check register against bounds
	Syntax:	chk <ea>,Dn
	Source <ea>:	Data addressing modes
*chk2	Function:	Check register against bounds
	Syntax:	chk2 <ea>,Rn
	Source <ea>:	Control addressing modes
clr	Function:	Clear an operand
	Syntax:	clr <ea>
	Source <ea>:	Data alterable addressing modes

```
cmp         Function:     Compare
            Syntax:       cmp <ea>,Dn
                          cmp <ea>,An
                          cmp #<data>,<ea>
                          cmp (Ay)+,(Ax)+
            Source <ea>:  All addressing modes
            Dest. <ea>:   Data alterable addressing modes
           *Dest. <ea>:   Data addressing modes

*cmp2       Function:     Compare register against bounds
            Syntax:       cmp2 <ea>,Rn
            Source <ea>:  Control addressing modes

db<cc>      Function:     Test condition, decrement, and branch
            Syntax:       db<cc> Dn,<label>
            Choices:      dbcc  Decrement and branch on carry clear
                          dbcs  Decrement and branch on carry set
                          dbeq  Decrement and branch on equal
                          dbf   Decrement and branch on false (never
                                branches)
                          dbge  Decrement and branch on  greater  or
                                equal
                          dbgt  Decrement and branch on greater
                          dbhi  Decrement and branch on high
                          dble  Decrement and branch on less or equal
                          dbls  Decrement and branch on low or same
                          dblt  Decrement and branch on less than
                          dbmi  Decrement and branch on minus
                          dbne  Decrement and branch on not equal
                          dbpl  Decrement and branch on plus
                          dbra  Decrement and branch always (dbf)
                          dbt   Decrement  and branch on true (never
                                branches)
                          dbvc  Decrement  and  branch  on  overflow
                                clear
                          dbvs  Decrement and branch on overflow set

divs        Function:     Signed divide
            Syntax:       divs <ea>,Dn          (32/16 = 16 r, 16 q)
           *Syntax:       divs.L <ea>,Dq        (32/32 = 32 q)
                          divs.L <ea>,Dr:Dq     (64/32 = 32 r, 32 q)
                          divsl.L <ea>,Dr:Dq    (32/32 = 32 r, 32 l)
            Source <ea>:  Data addressing modes

divu        Function:     Unsigned divide
            Syntax:       divu <ea>,Dn          (32/16 = 16 r, 16 q)
           *Syntax:       divu.L <ea>,Dq        (32/32 = 32 q)
                          divu.L <ea>,Dr:Dq     (64/32 = 32 r, 32 q)
                          divul.L <ea>,Dr:Dq    (32/32 = 32 r, 32 l)
            Source <ea>:  Data addressing modes
```

eor	Function:	Exclusive or logical
	Syntax:	eor Dn,<ea>
		eor #<data>,<ea>
		eor #<data(8)>,CCR
		eor #<data(16)>,SR
	Dest. <ea>:	Data alterable addressing modes
exg	Function:	Exchange registers
	Syntax:	exg Rx,Ry
ext	Function:	Sign extend
	Syntax:	ext Dn
*extb	Function:	Sign extend byte to a long
	Syntax:	extb Dn
*fabs	Function:	Floating-point absolute value
	Syntax:	fabs <ea>,FPn
		fabs FPm,FPn
		fabs FPn
	Source <ea>:	Data addressing modes
*facos	Function:	Floating-point arc cosine
	Syntax:	facos <ea>,FPn
		facos FPm,FPn
		facos FPn
	Source <ea>:	Data addressing modes
*fadd	Function:	Floating-point add
	Syntax:	fadd <ea>,FPn
		fadd FPm,FPn
		fadd FPn
	Source <ea>:	Data addressing modes
*fasin	Function:	Floating-point sine
	Syntax:	fasin <ea>,FPn
		fasin FPm,FPn
		fasin FPn
	Source <ea>:	Data addressing modes
*fatan	Function:	Floating-point arc tangent
	Syntax:	fatan <ea>,FPn
		fatan FPm,FPn
		fatan FPn
	Source <ea>:	Data addressing modes
*fatanh	Function:	Floating-point hyperbolic arc tangent
	Syntax:	fatanh <ea>,FPn
		fatanh FPm,FPn
		fatanh FPn
	Source <ea>:	Data addressing modes

*fb<cc> Function: Branch conditionally
Syntax: fb<cc> <label>
Choices:
fbeq Branch on equal
fbf Branch on false (never branches)
fbge Branch on greater or equal
fbgl Branch on greater or less
fbgle Branch on greater, less, or equal
fbgt Branch on greater
fble Branch on less or equal
fblt Branch on less
fbne Branch on not equal
fbnge Branch on not (greater or equal)
fbngl Branch on not (greater or less)
fbngle Branch on not (greater, less, or equal)
fbngt Branch on not greater
fbnle Branch on not (less or equal)
fbnlt Branch on not less
fboge Branch on ordered greater or equal
fbogl Branch on ordered greater or less
fbogt Branch on ordered greater
fbole Branch on ordered less or equal
fbolt Branch on ordered less
fbor Branch on ordered
fbseq Branch on signaling equal
fbsf Branch on signaling false (never branches)
fbsne Branch on signaling not equal
fbst Branch on signaling true (always branches)
fbt Branch on true (always branches)
fbueq Branch on unordered equal
fbuge Branch on unordered greater or equal
fbugt Branch on unordered greater
fbule Branch on unordered greater or equal
fbult Branch on unordered less
fbun Branch on unordered

*fcmp Function: Floating-point compare
Syntax: fcmp <ea>,FPn
fcmp FPm,FPn
fcmp FPn
Source <ea>: Data addressing modes

*fcos Function: Floating-point cosine
Syntax: fcos <ea>,FPn
fcos FPm,FPn
fcos FPn
Source <ea>: Data addressing modes

*fcosh
Function: Floating-point hyperbolic cosine
Syntax: fcosh <ea>,FPn
fcosh FPm,FPn
fcosh FPn
Source <ea>: Data addressing modes

*fdb<cc>
Function: Decrement and branch on condition
Syntax: fdb<cc> Dn,<label>
Choices:
fdbeq Decrement and branch on equal
fdbf Decrement and branch on false (never branches)
fdbge Decrement and branch on greater or equal
fdbgl Decrement and branch on greater or less
fdbgle Decrement and branch on greater, less, or equal
fdbgt Decrement and branch on greater
fdble Decrement and branch on less or equal
fdblt Decrement and branch on less
fdbne Decrement and branch on not equal
fdbnge Decrement and branch on not (greater or equal)
fdbngl Decrement and branch on not (greater or less)
fdbngle Decrement and branch on not (greater, less, or equal)
fdbngt Decrement and branch on not greater
fdbnle Decrement and branch on not (less or equal)
fdbnlt Decrement and branch on not less
fdboge Decrement and branch on ordered greater or equal
fdbogl Decrement and branch on ordered greater or less
fdbogt Decrement and branch on ordered greater
fdbole Decrement and branch on ordered less or equal
fdbolt Decrement and branch on ordered less
fdbor Decrement and branch on ordered
fdbseq Decrement and branch on signaling equal
fdbsf Decrement and branch on signaling false (never branches)
fdbsne Decrement and branch on signaling not equal
fdbst Decrement and branch on signaling true (always branches)

		fdbt Decrement and branch on true (always branches)
		fdbueq Decrement and branch on unordered equal
		fdbuge Decrement and branch on unordered greater or equal
		fdbugt Decrement and branch on unordered greater
		fdbule Decrement and branch on unordered greater or equal
		fdbult Decrement and branch on unordered less
		fdbun Decrement and branch on unordered
*fdiv	Function:	Floating-point divide
	Syntax:	fdiv <ea>,FPn fdiv FPm,FPn fdiv FPn
	Source <ea>:	Data addressing modes
*fetox	Function:	Floating-point e^x
	Syntax:	fetox <ea>,FPn fetox FPm,FPn fetox FPn
	Source <ea>:	Data addressing modes
*fetoxml	Function:	Floating-point e^x - 1
	Syntax:	fetoxml <ea>,FPn fetoxml FPm,FPn fetoxml FPn
	Source <ea>:	Data addressing modes
*fgetexp	Function:	Floating-point get exponent
	Syntax:	fgetexp <ea>,FPn fgetexp FPm,FPn fgetexp FPn
	Source <ea>:	Data addressing modes
*fgetman	Function:	Floating-point get mantissa
	Syntax:	fgetman <ea>,FPn fgetman FPm,FPn fgetman FPn
	Source <ea>:	Data addressing modes
*fint	Function:	Floating-point integer part
	Syntax:	fint <ea>,FPn fint FPm,FPn fint FPn
	Source <ea>:	Data addressing modes

*fintrz	Function:	Floating-point integer part, round to 0
	Syntax:	fintrz <ea>,FPn
		fintrz FPm,FPn
		fintrz FPn
	Source <ea>:	Data addressing modes
*flog10	Function:	Floating-point log base 10
	Syntax:	flog10 <ea>,FPn
		flog10 FPm,FPn
		flog10 FPn
	Source <ea>:	Data addressing modes
*flog2	Function:	Floating-point log base 2
	Syntax:	flog2 <ea>,FPn
		flog2 FPm,FPn
		flog2 FPn
	Source <ea>:	Data addressing modes
*flogn	Function:	Floating-point natural log (log base 'e')
	Syntax:	flogn <ea>,FPn
		flogn FPm,FPn
		flogn FPn
	Source <ea>:	Data addressing modes
*flognp1	Function:	Floating-point natural log plus 1
	Syntax:	flognp1 <ea>,FPn
		flognp1 FPm,FPn
		flognp1 FPn
	Source <ea>:	Data addressing modes
*fmod	Function:	Floating-point modulo remainder
	Syntax:	fmod <ea>,FPn
		fmod FPm,FPn
		fmod FPn
	Source <ea>:	Data addressing modes
*fmove	Function:	Move floating-point data register
	Syntax:	fmove <ea>,FPn
		fmove FPn,<ea>
		fmove.P FPn,<ea>{Dn}
		fmove.P FPn,<ea>{#k}
	Source <ea>:	Data addressable addressing modes
	Dest. <ea>:	Data alterable addressing modes
*fmove	Function:	Move system control-registers
	Syntax:	fmove <ea>,FPcr
		fmove FPcr,<ea>
	Source <ea>:	All addressing modes
	Dest. <ea>:	Alterable addressing modes
		Also address register direct if moving FPIAR

*fmovecr Function: Move from constant ROM

Syntax: fmovecr #<rom_offset>,FPn

Defined constants:

Offset	Value
$00	pi
$0B	Log base 10 of 2
$0C	e
$0D	Log base 2 of e
$0E	Log base 10 of e
$0F	0.0
$30	Natural log 2
$31	Natural log 10
$32	10^0
$33	10^1
$34	10^2
$35	10^4
$36	10^8
$37	10^16
$38	10^32
$39	10^64
$3A	10^128
$3B	10^256
$3C	10^512
$3D	10^1024
$3E	10^2048
$3F	10^4096

*fmovem Function: Move multiple floating-point registers

Syntax:
fmovem <reg_list>,<ea>
fmovem #<bit_mask>,<ea>
fmovem Dn,<ea>
fmovem <ea>,<reg_list>
fmovem <ea>,#<bit_mask>
fmovem <ea>, Dn

Source <ea>: Postincrement and control addressing modes

Dest. <ea>: Predecrement and control alterable addressing modes

*fmovem Function: Move multiple control registers

Syntax:
fmovem <FPcr_list>,<ea>
fmovem <ea>,<FPcr_list>

Source <ea>: All addressing modes

Dest. <ea>: Alterable addressing modes
Also address register direct if moving FPIAR
Also data register direct if moving a single FPcr

*fmul	Function:	Floating-point multiply
	Syntax:	fmul <ea>,FPn
		fmul FPm,FPn
		fmul FPn
	Source <ea>:	Data addressing modes
*fneg	Function:	Floating-point negate
	Syntax:	fneg <ea>,FPn
		fneg FPm,FPn
		fneg FPn
	Source <ea>:	Data addressing modes
*fnop	Function:	Floating-point no operation
	Syntax:	fnop
*frem	Function:	Floating-point IEEE remainder
	Syntax:	frem <ea>,FPn
		frem FPm,FPn
		frem FPn
	Source <ea>:	Data addressing modes
*frestore	Function:	Restore internal state
	Syntax:	frestore <ea>
	Source <ea>:	Postincrement or control addressing modes
*fsave	Function:	Save internal state
	Syntax:	fsave <ea>
	Source <ea>:	Preincrement or control alterable addressing modes
*fscale	Function:	Floating-point scale exponent
	Syntax:	fscale <ea>,FPn
		fscale FPm,FPn
		fscale FPn
	Source <ea>:	Data addressing modes
*fs<cc>	Function:	Set according to condition
	Syntax:	fs<cc> <ea>
	Source <ea>:	Data alterable addressing modes
	Choices:	fseq Set on equal
		fsf Set on false (never branches)
		fsge Set on greater or equal
		fsgl Set on greater or less
		fsgle Set on greater, less, or equal
		fsgt Set on greater
		fsle Set on less or equal
		fslt Set on less
		fsne Set on not equal
		fsnge Set on not (greater or equal)
		fsngl Set on not (greater or less)

```
                        fsngle Set on not (greater, less, or equal)
                        fsngt  Set on not greater
                        fsnle  Set on not (less or equal)
                        fsnlt  Set on not less
                        fsoge  Set on ordered greater or equal
                        fsogl  Set on ordered greater or less
                        fsogt  Set on ordered greater
                        fsole  Set on ordered less or equal
                        fsolt  Set on ordered less
                        fsor   Set on ordered
                        fsseq  Set on signaling equal
                        fssf   Set on signaling false (never
                               branches)
                        fssne  Set on signaling not equal
                        fsst   Set on signaling true (always
                               branches)
                        fst    Set on true (always branches)
                        fsueq  Set on unordered equal
                        fsuge  Set on unordered greater or equal
                        fsugt  Set on unordered greater
                        fsule  Set on unordered greater or equal
                        fsult  Set on unordered less
                        fsun   Set on unordered

*fsgldiv    Function:   Floating-point single-precision divide
            Syntax:     fsgldiv <ea>,FPn
                        fsgldiv FPm,FPn
                        fsgldiv FPn
            Source <ea>: Data addressing modes

*fsglmul    Function:   Floating-point single-precision multiply
            Syntax:     fsglmul <ea>,FPn
                        fsglmul FPm,FPn
                        fsglmul FPn
            Source <ea>: Data addressing modes

*fsin       Function:   Floating-point sine
            Syntax:     fsin <ea>,FPn
                        fsin FPm,FPn
                        fsin FPn
            Source <ea>: Data addressing modes

*fsincos    Function:   Simultaneous sine and cosine
            Syntax:     fsincos <ea>,FPc:FPs
                        fsincos FPm,FPc:FPs

*fsinh      Function:   Floating-point hyperbolic sine
            Syntax:     fsinh <ea>,FPn
                        fsinh FPm,FPn
                        fsinh FPn
            Source <ea>: Data addressing modes
```

*fsqrt	Function:	Floating-point square root
	Syntax:	fsqrt <ea>,FPn
		fsqrt FPm,FPn
		fsqrt FPn
	Source <ea>:	Data addressing modes
*fsub	Function:	Floating-point subtract
	Syntax:	fsub <ea>,FPn
		fsub FPm,FPn
		fsub FPn
	Source <ea>:	Data addressing modes
*ftan	Function:	Floating-point tangent
	Syntax:	ftan <ea>,FPn
		ftan FPm,FPn
		ftan FPn
	Source <ea>:	Data addressing modes
*ftanh	Function:	Floating-point hyperbolic tangent
	Syntax:	ftanh <ea>,FPn
		ftanh FPm,FPn
		ftanh FPn
	Source <ea>:	Data addressing modes
*ftentox	Function:	Floating-point 10 to the 'x' power (10^x)
	Syntax:	ftentox <ea>,FPn
		ftentox FPm,FPn
		ftentox FPn
	Source <ea>:	Data addressing modes
*ftrap<cc>	Function:	Trap conditionally
	Syntax:	ftrap<cc> [#<data>]
	Choices:	ftrapeq Trap on equal
		ftrapf Branch on false (never branches)
		ftrapge Branch on greater or equal
		ftrapgl Branch on greater or less
		ftrapgle Branch on greater, less, or equal
		ftrapgt Branch on greater
		ftraple Branch on less or equal
		ftraplt Branch on less
		ftrapne Branch on not equal
		ftrapnge Branch on not (greater or equal)
		ftrapngl Branch on not (greater or less)
		ftrapngle Branch on not (greater, less, or equal)
		ftrapngt Branch on not greater
		ftrapnle Branch on not (less or equal)
		ftrapnlt Branch on not less
		ftrapoge Branch on ordered greater or equal
		ftrapogl Branch on ordered greater or less

		ftrapogt Branch on ordered greater
		ftrapole Branch on ordered less or equal
		ftrapolt Branch on ordered less
		ftrapor Branch on ordered
		ftrapseq Branch on signaling equal
		ftrapsf Branch on signaling false (never branches)
		ftrapsne Branch on signaling not equal
		ftrapst Branch on signaling true (always branches)
		ftrapt Branch on true (always branches)
		ftrapueq Branch on unordered equal
		ftrapuge Branch on unordered greater or equal
		ftrapugt Branch on unordered greater
		ftrapule Branch on unordered greater or equal
		ftrapult Branch on unordered less
		ftrapun Branch on unordered
*ftst	Function:	Floating-point test operand
	Syntax:	ftst <ea>
		ftst FPn
	Source <ea>:	Data addressing modes
*ftwotox	Function:	Floating-point 2 to the 'x' power (2^x)
	Syntax:	ftwotox <ea>,FPn
		ftwotox FPm,FPn
		ftwotox FPn
	Source <ea>:	Data addressing modes
illegal	Function:	Illegal instruction
	Syntax:	illegal
jmp	Function:	Jump
	Syntax:	jmp <ea>
	Source <ea>:	Control addressing modes
jsr	Function:	Jump to subroutine
	Syntax:	jsr <ea>
	Source <ea>:	Control addressing modes
lea	Function:	Load effective address
	Syntax:	lea <ea>,An
	Source <ea>:	Control addressing modes
link	Function:	Link and allocate
	Syntax:	link An,#<disp(16)>
	*Syntax:	link An,#<disp(32)>

lsl	Function:	Logical shift left
	Syntax:	lsl Dx,Dy lsl #<quick>,Dn lsl <ea>
	Source <ea>:	Memory alterable addressing modes (word only)
lsr	Function:	Logical shift right
	Syntax:	lsr Dx,Dy lsr #<quick>,Dn lsr <ea>
	Source <ea>:	Memory alterable addressing modes (word only)
move	Function:	Move data from source to destination

	Syntax:	move <ea>,<ea>
		move CCR,<ea>
		move <ea>,CCR
		move <ea>,SR
		move SR,<ea>
		move USP,An
		move An,USP

	Source <ea>:	All addressing modes except for "move to CCR" and "move to SR", which require data addressing
	Dest. <ea>:	Data alterable addressing modes
+movec	Function:	Move to or from control register
	Syntax:	movec Rc,Rn movec Rn,Rc
movem	Function:	Move multiple registers

	Syntax:	movem <reg_list>,<ea>
		movem <ea>,<reg_list>
		movem #<bit_mask>,<ea>
		movem <ea>,#<bit_mask>

	Source <ea>:	Control addressing and postincrement addressing modes
	Dest. <ea>:	Control alterable and predecrement addressing modes
movep	Function:	Move peripheral data
	Syntax:	movep Dx,<disp(16)>(Ay) movep <disp(16)>(Ax),Dy
+moves	Function:	Move to or from address space
	Syntax:	moves Rn,<ea> moves <ea>,Rn
	Source <ea>:	Memory alterable addressing modes
	Dest. <ea>:	Memory alterable addressing modes

```
muls        Function:       Signed multiply
            Syntax:         muls.W <ea>,Dn      (16*16 = 32)
           *Syntax:         muls.L <ea>,Dn      (32*32 = 32)
                            muls.L <ea>,Dh:Dl   (32*32 = 64)
            Source <ea>:    Data addressing modes

mulu        Function:       Unsigned multiply
            Syntax:         mulu.W <ea>,Dn      (16*16 = 32)
           *Syntax:         mulu.L <ea>,Dn      (32*32 = 32)
                            mulu.L <ea>,Dh:Dl   (32*32 = 64)
            Source <ea>:    Data addressing modes

nbcd        Function:       Negate decimal with extend
            Syntax:         nbcd <ea>
            Source <ea>:    Data alterable addressing modes

neg         Function:       Negate
            Syntax:         neg <ea>
            Source <ea>:    Data alterable addressing modes

negx        Function:       Negate with extend
            Syntax:         negx <ea>
            Source <ea>:    Data alterable addressing modes

nop         Function:       No operation
            Syntax:         nop

not         Function:       Logical complement
            Syntax:         not <ea>
            Source <ea>:    Data alterable addressing modes

or          Function:       Inclusive or logical
            Syntax:         or <ea>,Dn
                            or Dn,<ea>
                            or #<data>,<ea>
                            or #<data(8)>,CCR
                            or #<data(16)>,SR
            Source <ea>:    Data addressing modes
            Dest. <ea>:     Data alterable addressing modes

*pack       Function:       pack binary data
            Syntax:         pack -(Ax),-(Ay),#<data(16)>
                            pack Dx,Dy,#<data(16)>

pea         Function:       Push effective address
            Syntax:         pea <ea>
            Source <ea>:    Control addressing modes

reset       Function:       Reset external devices
            Syntax:         reset
```

rol	Function:	Rotate left
	Syntax:	rol Dx,Dy
		rol #<quick>,Dn
		rol <ea>
	Source <ea>:	Memory alterable addressing modes (word only)
ror	Function:	Rotate right
	Syntax:	ror Dx,Dy
		ror #<quick>,Dn
		ror <ea>
	Source <ea>:	Memory alterable addressing modes (word only)
roxl	Function:	Rotate left with extend
	Syntax:	roxl Dx,Dy
		roxl #<quick>,Dn
		roxl <ea>
	Source <ea>:	Memory alterable addressing modes (word only)
roxr	Function:	Rotate right with extend
	Syntax:	roxr Dx,Dy
		roxr #<quick>,Dn
		roxr <ea>
	Source <ea>:	Memory alterable addressing modes (word only)
*rtd	Function:	Return and deallocate parameters
	Syntax:	rtd #<disp(16)>
rte	Function:	Return from exception
	Syntax:	rte
*rtm	Function:	Return from module
	Syntax:	rtm Rn
rtr	Function:	Return and restore condition codes
	Syntax:	rtr
rts	Function:	Return from subroutine
	Syntax:	rts
sbcd	Function:	Subtract decimal with extend
	Syntax:	sbcd Dy,Dx
		sbcd -(Ay),-(Ax)
s<cc>	Function:	Set according to condition
	Syntax:	s<cc> <ea>
	Choices:	scc Set on carry clear
		scs Set on carry set

```
                                seq  Set on equal
                                sf   Set on false (clear)
                                sge  Set on greater or equal
                                sgt  Set on greater
                                shi  Set on high
                                sle  Set on less or equal
                                sls  Set on low or same
                                slt  Set on less than
                                smi  Set on minus
                                sne  Set on not equal
                                spl  Set on plus
                                st   Set on true (always set)
                                svc  Set on overflow clear
                                svs  Set on overflow set
                    Source <ea>: Data alterable addressing modes

stop                Function:    Load status register and stop
                    Syntax:      stop #<data(16)>

sub                 Function:    Subtract binary
                    Syntax:      sub <ea>,Dn
                                 sub Dn,<ea>
                                 sub <ea>,An
                                 sub #<data>,<ea>
                    Source <ea>: All addressing modes
                    Dest. <ea>:  Data alterable addressing modes

subx                Function:    Subtract with extend
                    Syntax:      subx Dy,Dx
                                 subx -(Ay),-(Ax)

swap                Function:    Swap register halves
                    Syntax:      swap Dn

tas                 Function:    Test and set an operand
                    Syntax:      tas <ea>
                    Source <ea>: Data alterable addressing modes

trap                Function:    Trap
                    Syntax:      trap #<vector>

*trap<cc>           Function:    Trap on condition
                    Syntax:      trap<cc> [#<data>]
                    Choices:     trapcc  Trap on carry clear
                                 trapcs  Trap on carry set
                                 trapeq  Trap on equal
                                 trapf   Trap on false (never traps)
                                 trapge  Trap on greater or equal
                                 trapgt  Trap on greater than
                                 traphi  Trap on high
                                 traple  Trap on less than or equal
```

```
                        trapls  Trap on low or same
                        traplt  Trap on less than
                        trapmi  Trap on minus
                        trapne  Trap on not equal
                        trappl  Trap on plus
                        trapt   Trap on true (always traps)
                        trapvc  Trap on overflow clear
                        trapvs  Trap on overflow set

trapv       Function:   Trap on overflow
            Syntax:     trapv

tst         Function:   Test an operand
            Syntax:     tst <ea>
            Source <ea>: Data alterable addressing modes
           *Source <ea>: Data addressing modes

unlk        Function:   Unlink
            Syntax:     unlk An

*unpk       Function:   Unpack binary-coded decimal
            Syntax:     unpk -(Ax),-(Ay),#<data(16)>
                        unpk Dx,Dy,#<data(16)>
```

4.6 Convenience Mnemonics

The assembler also supports the mnemonics shown in Table 4-3. These mnemonics provide convenient ways of clearing and setting the bits in the condition-code register.

Table 4-3. Convenience Mnemonics

Mnemonic	Function	Target Bit in CCR	Code Generated	
clc	Clear	Carry	and	#$FE,CCR
cln	Clear	Negative	and	#$F7,CCR
clv	Clear	Overflow	and	#$FD,CCR
clx	Clear	Extend	and	#$EF,CCR
clz	Clear	Zero	and	#$FB,CCR
sec	Set	Carry	or	#$01,CCR
sen	Set	Negative	or	#$08,CCR
sev	Set	Overflow	or	#$02,CCR
sex	Set	Extend	or	#$10,CCR
sez	Set	Zero	or	#$04,CCR

Chapter 5

Directives

5.1 Introduction

In addition to the standard mnemonics discussed in Chapter 4, the assembler supports numerous directives. These directives, which are also known as pseudo-ops, instruct the assembler to perform certain operations. They may or may not generate code.

A directive is used in the opcode field of a line of code (see Section 3.3). It may or may not be followed by an operand. If the operand is composed of a list of elements, the items in the list must be separated by commas. No spaces may appear in the operand field unless the entire field must be enclosed by quotation marks or unless specifically noted in the documentation of a particular directive. Unless otherwise noted the label field and the comment field are optional.

A brief description of each directive follows (an asterisk, '*', indicates that the directive may be used only on the 68020). More detailed descriptions appear later in this chapter.

base	Set a program counter outside the text, data, and bss segments.
*bfequ	Equate the offset/width pair of a bit field to a symbol.
bss	Relocate the following instructions to the end of the bss segment.
*cnop	Align the next instruction on a quad-word boundary.
common	Begin a common block.
*cpid	Set the current coprocessor identification number.
data	Relocate the following instructions to the end of the data segment.
dc	Define a constant in memory.
define	Begin defining a set of labels as global variables.
ds	Reserve uninitialized memory.
else	Switch the sense of the preceding "if" or "ifn" directive.
end	Stop assembly.
endcom	End a common block.
enddef	End the definition of a set of global symbols.
endif	End a block of conditional assembly.
endm	End the definition of a macro.

equ	Equate a symbol to the expression in the operand field.
err	Insert a user-defined error message in the listing of the assembled source and increment the error count by 1.
even	Align the following data on an even boundary.
exitm	Terminate expansion of the macro and jump to the next "endm" instruction.
extern	Define the symbols in the operand field as external.
fcb	Set memory bytes to the values of the 8-bit expressions in the operand field.
fcc	Set memory bytes to the values specified by the delimited ASCII string.
fdb	Set memory bytes to the values of the 16-bit expressions in the operand field.
fqb	Set memory bytes to the values of the 32-bit expressions in the operand field.
global	Define the symbols in the operand field as global symbols.
if	Execute the first block of conditional code if the condition in the operand field is true.
ifc	Execute the first block of conditional code if the two strings specified in the operand field are the same.
ifnc	Execute the first block of conditional code if the two strings specified in the operand field are not the same.
ifeq	Execute the first block of conditional code if the condition in the operand field is equal to 0.
ifge	Execute the first block of conditional code if the condition in the operand field is greater than or equal to 0.
ifgt	Execute the first block of conditional code if the condition in the operand field is greater than 0.
ifle	Execute the first block of conditional code if the condition in the operand field is less than or equal to 0.
iflt	Execute the first block of conditional code if the condition in the operand field is less than 0.
ifn	Execute the first block of conditional code if the condition in the operand field is false.
ifne	Execute the first block of conditional code if the condition in the operand field is not equal to 0.
info	Store text in the information field of the file containing the assembled source code.
lib	Incorporate the source code from the specified file.
log	Calculate the base-2 logarithm of an absolute expression.
macro	Begin the definition of a macro.
name	Assign the specified name to the binary module generated by the assembly.

opt	Alter the format of the listing of the assembled source code.
pag	Perform a page eject in the listing of the assembled source code and write a header at the top of the new page.
*quad	Align the following data on a quad-word boundary.
rab	Reserve memory for storage of data, forcing the first byte to an even boundary.
rmb	Reserve memory for storage of data.
rzb	Reserve and clear memory for storage of data.
set	Set a symbol to the value of the expression in the operand field. May be used more than once for the same symbol.
spc	Write the specified number of blank lines to the listing of the assembled source code.
struct	Set a program counter outside the text, data, and bss segments.
sttl	Set the subtitle for the header of the listing of the assembled source code to the string in the operand field.
sys	Perform a system call.
text	Relocate the following instructions to the end of the text segment.
tstmp	Create in the information field of the assembled file a time stamp with the current date and time.
ttl	Set the title for the header of the listing of the assembled source code to the string in the operand field.

5.2 The Directives

A more detailed description of each directive follows.

5.2.1 base

The "base" directive sets a program counter outside the text, data, and bss segments. The syntax for this directive is simply

```
base [<expr>]
```

where <expr> is an absolute or relocatable expression. If the user does not specify an operand, it defaults to 0.

This type of program counter is used to establish a label for a particular offset from a fixed address. Generally, it is used in conjunction with storing information on a stack. The following example illustrates the use of the "base" directive:

```
            base
A6_link     ds.l    1
ret_addr    ds.l    1
arg_1       ds.l    1
arg_2       ds.l    1
arg_3       ds.l    1
            text
             .
             .
             .
            move.l  arg_1(A6),D0
            add.l   D0,D0
            sub.l   arg_3(A6),D0
            add.l   arg_2(A6),D0
             .
             .
             .
```

Symbols declared with a "base" statement may be absolute or relocatable, depending on the attributes of the operand. The user may not reuse a symbol defined in a base segment.

5.2.2 bfequ (68020 only)

The "bfequ" instruction equates the offset/width pair of a bit field to a symbol. It requires an ordinary label in the label field. The syntax for this directive is

<label> bfequ <offset>:<width>

where <offset> is between 0 and 31 inclusive, and <width> is between 1 and 32 inclusive. As for all bit-field instructions, the bits in the register or effective address are numbered from left to right with the leftmost bit being bit 0. The following example illustrates the use of the "bfequ" directive:

```
AMODE       bfequ    10:3
              .
              .
              .
            bfclr    d0{AMODE}
```

This code first equates the label AMODE to a bit field whose offset is 10 and whose width is 3. It then clears from data register 0 the twenty-first, twentieth, and nineteenth bits.

5.2.3 bss

The "bss" directive relocates the instructions which follow it to the end of the bss segment. The syntax for this directive is simply

```
bss
```

A "bss" directive remains in effect until the assembler encounters a "base", "data", "struct", or "text" directive.

5.2.4 cnop (68020 only)

The 68020 microprocessor executes an instruction more efficiently when it is aligned on a quad-word boundary (a boundary whose address is a multiple of 4). The "cnop" directive aligns the next instruction on a quad-word boundary if it is not already so aligned. It does so by generating the following two-byte instruction which does not alter the condition codes:

```
move.l A0,A0
```

(The "nop" opcode cannot be used because it causes the 68020 to flush its instruction pipeline.) The syntax for the "cnop" directive is simply

```
cnop
```

5.2.5 common

The "common" directive marks the beginning of a block of memory that can be shared by more than one module. Common blocks allow various modules to share information without having to pass it back and forth. The syntax for this directive is

```
<label> common
```

The user must place an ordinary label in the label field. The user must terminate a declaration of a common block with the directive "endcom". The only instructions which may appear between the "common" and "endcom" directives are those which define the size of the common block--"rmb", "rab", and "ds". Only one common block of a particular name should appear in any given module. A module may not consist solely of blocks of common code. The assembler treats common blocks as external references.

The following example illustrates the use of the "common" directive:

```
test    common
var_1   ds.w      5
var_2   ds.l      10
        endcom
```

This code defines a common block named "test" which contains two variables, "var_1" and "var_2". The block is 50 bytes long.

5.2.6 cpid (68020 only)

The "cpid" directive tells the assembler which coprocessor to use. The syntax for this directive is

```
cpid [<ID_num>]
```

where <ID_num> is a number between 0 and 7 inclusive. The default value is 1. The following table shows the correlation between each number and the coprocessor it designates:

Number	Coprocessor
0	MC68851 paged-memory management unit (PMMU)
1	MC68881 floating-point coprocessor
2-5	Reserved for future Motorola coprocessors
6-7	User-defined

5.2.7 data

The "data" directive relocates the instructions which follow it to the end of the data segment. The syntax for this directive is simply

```
data
```

A "data" directive remains in effect until the assembler encounters a "base", "bss", "struct", or "text" directive.

5.2.8 dc

The "dc" directive defines a constant in memory. The user may append one of the standard suffixes to the directive to indicate the size of the constant or constants defined by that line of code. In addition, the 68020 assembler supports the 'q' suffix with this directive. The 'q' suffix tells the assembler to align long words on a boundary that is a multiple of 4. The assembler pads any memory that it passes over in the process with null bytes. The syntax for this directive is

```
dc <expr_list>
```

where the elements of <expr_list> may be either actual values (constants or ASCII strings) or expressions. The user must enclose ASCII strings in single or double quotation marks. An ASCII string used with the "dc.b" directive may be of any length. A string used with the "dc.w" directive may not contain more than two characters. One used with the "dc.l" directive may not contain more than four characters. If more than one element appears in the list, the assembler treats each successive element as if it were the lone operand of another "dc" directive.

The assembler always aligns the constant on the proper boundary, as specified by the suffix used with the directive. If an ASCII string is not large enough to fill the space allocated to it, the assembler pads

on the left with null bytes. The following lines of code illustrate valid uses of the "dc" directive:

```
label_1 dc.b 3,7,'String'
label_2 dc.w 12,'a',98      Pads 'a' on the left with 1 null byte
        dc.l 'ab',131072    Pads 'ab' on the left with 3 null bytes
```

The directive "dc.b" is functionally equivalent to the directive "fcb"; "dc.w", to "fdb"; and "dc.l", to "fqb".

5.2.9 define

The "define" directive instructs the assembler to treat the labels in the label fields of the statements which follow it as global symbols. It provides a convenient way of simultaneously defining symbols and declaring them to be global. The assembler puts all global variables into the symbol table, which is used by the linking-loader. The syntax of this directive is simply

```
define
```

The statements following a "define" directive must be terminated with an "enddef" directive. The following lines of code illustrate the use of the "define" directive:

```
          data
          define
temp_1    fdb      0,$FFFF
start     move.l   #1,D1
          enddef
```

This section of code defines the labels "temp_1" and "start" as global.

5.2.10 ds

The "ds" directive reserves an uninitialized area of memory. The user may append one of the standard suffixes to the directive to indicate the size of each unit of memory. In addition, the 68020 assembler supports the 'q' suffix with this directive. The 'q' suffix tells the assembler to align long words on a boundary that is a multiple of 4. The assembler pads any memory that it passes over in the process with null bytes. The syntax for this directive is

```
ds <expr>
```

where <expr> is an absolute expression specifying the number of units of memory to reserve. The operand field may not contain a forward reference. If the value of the operand is 0, the assembler reserves no memory unless it is necessary to force the specified alignment. If a label is present, its value is the address of the lowest memory location reserved by that directive. The following lines of code illustrate valid uses of the "ds" directive:

```
          ds.b 20     Reserve 20 bytes
          ds   10     Reserve 10 words
label_1   ds.l  5     Reserve 5 long words
          ds.l  0     Force alignment to a even boundary
          ds.x  2     Reserve space for 2 extended-precision
                      floating-point numbers
```

5.2.11 else

The "else" directive switches the sense of the preceding "if" or "ifn" directive. It must appear somewhere between an "if" or an "ifn" directive and an "endif" directive. The syntax for this directive is simply

```
else
```

If the assembler processed the code preceding the "else" directive in that conditional block, it skips the code between the "else" directive and the next "endif". Otherwise, it processes it.

5.2.12 end

The "end" directive, which must be used without a label, signifies the end of a module of source code. It generates no code. The syntax for this directive is

```
end [<expr>]
```

where <expr> is an absolute or relocatable expression which defines a transfer address.

An "end" directive is not required, but it is the only means of appending a transfer address to an object-code module and of separating modules within a file. The assembler ignores "end" directives in any modules brought into the program with the "lib" directive (see Section 5.2.38).

5.2.13 endcom

The "endcom" directive marks the end of the definition of a block of common memory. It is always used in conjunction with the "common" directive. The syntax for this directive is simply

```
endcom
```

5.2.14 enddef

The "enddef" directive marks the end of a section of code which defines one or more symbols as global. The syntax for this directive is simply

```
enddef
```

It is always used in conjunction with the "define" directive.

5.2.15 endif

The "endif" directive marks the end of a block of conditional code. The syntax for this directive is simply

```
endif
```

It is always used in conjunction with an "if" or an "ifn" directive.

5.2.16 endm

The "endm" directive marks the end of the definition of a macro. The syntax for this directive is simply

```
endm
```

It is always used in conjunction with the "macro" instruction.

5.2.17 equ

The "equ" directive equates a symbol to the expression in the operand field. It requires an ordinary label in the label field. This directive generates no code. The syntax for the "equ" directive is

```
<label> equ <expr>
```

where <expr> is an absolute, relocatable, or, on the 68020 only, a floating-point expression. The assembler assigns the attribute of the expression to the label.

5.2.18 err

The "err" directive inserts a user-defined error message into the listing of the assembled source code and increments the error count by 1. The syntax for this directive is

```
err <message_to_print>
```

where the message specified is a string of ASCII characters. The user need not enclose the message in quotation marks. The message may contain space characters. When executing the "err" directive, the assembler prints the remainder of the line except leading spaces, preceding the message with three asterisks. The "err" directive is commonly used in conjunction with conditional code, so that a user-defined illegal condition is reported as an error.

5.2.19 even

The 68000 microprocessor requires that the data for all operations on a word or a long begin on an even boundary. The MC68020 does not require that such data begin on an even boundary, but it does perform more efficiently if they do. The "even" directive forces the following piece of data to an even boundary by filling in with null bytes if necessary. Its syntax is simply

```
even
```

5.2.20 exitm

The "exitm" directive tells the assembler to terminate macro expansion and skip to the next "endm" instruction. The syntax for this directive is simply

```
exitm
```

The only reasonable way to use the "exitm" instruction is in conjunction with conditional assembly and parameter substitution (see Section 5.3). The following example illustrates the use of the "exitm" instruction:

```
example   macro
            .
            .           Code that is always generated
            .
          ifnc &2,yes   Use of parameter substitution
          exitm
          endif
            .
            .           Code that is sometimes generated
            .
          endm
```

If the parameter passed in to the macro in this example is not equal to "yes", the assembler executes the "exitm" instruction, jumping to the next "endm" instruction and terminating macro expansion. Otherwise, the assembler executes the code between the "endif" and "endm" instructions.

5.2.21 extern

The "extern" directive tells the assembler to treat the specified symbols as external references. The syntax for this directive is

```
extern <label_list>
```

where each element in the list of labels is an ordinary, not a local, label (see Section 3.2). When the assembler encounters one of the labels specified in <label_list>, it writes an external record for that label. The "extern" directive usually appears at the beginning of a module. It should appear before any external references.

5.2.22 fcb

The "fcb" directive forms a constant byte by associating a byte in memory with the value represented by the corresponding operand. The syntax of the directive is

```
fcb <expr_list>
```

where each item in the list of expressions is an absolute, relocatable, or external expression. The assembler evaluates each expression to 8 bits and stores the resulting quantities in successive memory locations. The directive "fcb" is functionally equivalent to the directive "ds.b" (see Section 5.2.10).

5.2.23 fcc

The "fcc" directive forms a constant character by associating a byte in memory with the corresponding character in the operand field. The syntax of the directive is

```
fcc <delimiter><str><delimiter>
```

where a delimiter is any nonalphanumeric character except the dollar sign, '$'. The assembler converts each character in the string to its ASCII value and stores those values in successive memory locations. The string specified may include space characters. The following lines of code illustrate valid uses of the "fcc" directive:

```
label_1   fcc   'This is an "fcc" string.'
label_2   fcc   ,So is this.,
          fcc   /The label is not necessary./
```

The assembler supports a second kind of use of the "fcc" directive which is a deviation from the standard Motorola definition of the directive. The user may include in the operand field certain expressions as well as delimited strings. The expressions used must be external or absolute; they must start with a letter, digit, or dollar sign (indicating a hexadecimal value); and they must represent a value that fits into one byte. The following lines of code illustrate this kind of use of the "fcc" directive:

```
intro     fcc   'This string has CR & LF.',$0D,$0A
          fcc   ;string_1;,0,:string_2:
          fcc   $04,ext_label,/string/
```

Note that more than one delimited string may be placed on a line as in the second example.

5.2.24 fdb

The "fdb" directive forms a two-byte quantity in memory by associating two bytes with the value represented by the corresponding operand. The syntax of the directive is

```
fdb <expr_list>
```

where each item in the list of expressions is an absolute, relocatable, or external expression. The assembler evaluates each expression to 16 bits and stores the resulting quantities in successive memory locations. The directive "fdb" is functionally equivalent to the directive "ds.w" (see Section 5.2.10).

5.2.25 fqb

The "fqb" directive forms a four-byte quantity in memory by associating four bytes with the value represented by the corresponding operand. The syntax of the directive is

```
fqb <expr_list>
```

where each item in the list of expressions is an absolute, relocatable, or external expression. The assembler evaluates each expression to 32 bits and stores the resulting quantities in successive memory locations. The directive "fqb" is functionally equivalent to the directive "ds.l" (see Section 5.2.10).

5.2.26 global

The "global" directive instructs the assembler to put the labels which follow the directive into the symbol table of the assembled source code, which is used by the linking-loader. The syntax for this directive is

```
global <label_list>
```

where each element in the list of labels is an ordinary, not a local, label (see Section 3.2).

5.2.27 if

The "if" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
if <expr>
```

where <expr> is an absolute expression that the assembler can evaluate during its first pass over the source code (the expression may not contain a forward reference). When processing an "if" directive, the assembler evaluates the expression associated with it. If the expression is true (not equal to 0), the assembler processes all the code between the "if" directive and the next conditional directive (an "else" or an "endif"). If it is false (equal to 0), the assembler skips the code between the "if" directive and the next conditional directive. Every "if" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.28 ifc

The "ifc" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
ifc [<str_1>],[<str_2>]
```

where <str_1> and <str_2> are ASCII strings. Unless the user encloses a string in a pair of delimiter characters (either single or double quotation marks), the assembler interprets the string as a group of characters beginning with a nonspace character and ending with a space character or a comma. A string enclosed in quotation marks may include space characters and commas; in such a case the assembler interprets the string as all the characters between the quotation marks. The user may specify the null string either by leaving the string out completely or by placing a pair of delimiter characters side-by-side.

When processing an "ifc" directive, the assembler compares the two strings associated with it. If the strings are identical, the assembler processes all the code between the "ifc" directive and the next conditional directive (an "else" or an "endif"). If the strings are not identical, the assembler skips the code between the "ifc" directive and the next conditional directive. Every "ifc" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.29 ifeq

The "ifeq" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
ifeq <expr>
```

where <expr> is an absolute expression that the assembler can evaluate during its first pass over the source code (the expression may not contain a forward reference). When processing an "ifeq" directive, the assembler evaluates the expression associated with it. If the expression is equal to 0, the assembler processes all the code between the "ifeq" directive and the next conditional directive (an "else" or an "endif"). If it is not equal to 0, the assembler skips the code between the "ifeq" directive and the next conditional directive. Every "ifeq" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user

can nest blocks of conditional code up to twenty levels deep.

5.2.30 ifge

The "ifge" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
ifge <expr>
```

where <expr> is an absolute expression that the assembler can evaluate during its first pass over the source code (the expression may not contain a forward reference). When processing an "ifge" directive, the assembler evaluates the expression associated with it. If the expression is greater than or equal to 0, the assembler processes all the code between the "ifge" directive and the next conditional directive (an "else" or an "endif"). If it is less than 0, the assembler skips the code between the "ifge" directive and the next conditional directive. Every "ifge" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.31 ifgt

The "ifgt" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
ifgt <expr>
```

where <expr> is an absolute expression that the assembler can evaluate during its first pass over the source code (the expression may not contain a forward reference). When processing an "ifgt" directive, the assembler evaluates the expression associated with it. If the expression is greater than 0, the assembler processes all the code between the "ifgt" directive and the next conditional directive (an "else" or an "endif"). If it is less than or equal to 0, the assembler skips the code between the "ifgt" directive and the next conditional directive. Every "ifgt" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.32 ifle

The "ifle" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
ifle <expr>
```

where <expr> is an absolute expression that the assembler can evaluate during its first pass over the source code (the expression may not contain a forward reference). When processing an "ifle" directive, the assembler evaluates the expression associated with it. If the expression is less than or equal to 0, the assembler processes all the code between the "ifle" directive and the next conditional directive (an "else" or an "endif"). If it is false greater than 0, the assembler skips the code between the "ifle" directive and the next conditional directive. Every "ifle" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.33 iflt

The "iflt" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
iflt <expr>
```

where <expr> is an absolute expression that the assembler can evaluate during its first pass over the source code (the expression may not contain a forward reference). When processing an "iflt" directive, the assembler evaluates the expression associated with it. If the expression is less than 0, the assembler processes all the code between the "iflt" directive and the next conditional directive (an "else" or an "endif"). If it is greater than or equal to 0, the assembler skips the code between the "iflt" directive and the next conditional directive. Every "iflt" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.34 ifn

The "ifn" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
ifn <expr>
```

where <expr> is an absolute expression that the assembler can evaluate during its first pass over the source code (the expression may not contain a forward reference). When processing an "ifn" directive, the assembler evaluates the expression associated with it. If the expression is false (equal to 0), the assembler processes all the code between the "ifn" directive and the next conditional directive (an "else" or an "endif"). If it is true (not equal to 0), the assembler skips the code between the "ifn" directive and the next conditional directive. Every "ifn" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.35 ifnc

The "ifnc" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
ifnc [<str_1>],[<str_2>]
```

where <str_1> and <str_2> are ASCII strings. Unless the user encloses a string in a pair of delimiter characters (either single or double quotation marks), the assembler interprets the string as a group of characters beginning with a nonspace character and ending with a space character or a comma. A string enclosed in quotation marks may include space characters and commas; in such a case the assembler interprets the string as all the characters between the quotation marks. The user may specify the null string either by leaving the string out completely or by placing a pair of delimiter characters side-by-side.

When processing an "ifnc" directive, the assembler compares the two strings associated with it. If the strings are not identical, the assembler processes all the code between the "ifnc" directive and the next conditional directive (an "else" or an "endif"). If the strings are identical, the assembler skips the code between the "ifnc" directive and the next conditional directive. Every "ifnc" directive must eventually be followed by an "endif" directive. One "else" directive

may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.36 ifne

The "ifne" directive marks the beginning of a block of conditional code. The syntax for this directive is

```
ifne <expr>
```

where <expr> is an absolute expression that the assembler can evaluate during its first pass over the source code (the expression may not contain a forward reference). When processing an "ifne" directive, the assembler evaluates the expression associated with it. If the expression is not equal to 0, the assembler processes all the code between the "ifne" directive and the next conditional directive (an "else" or an "endif"). If it is equal to 0, the assembler skips the code between the "ifne" directive and the next conditional directive. Every "ifne" directive must eventually be followed by an "endif" directive. One "else" directive may also be part of the conditional block. The user can nest blocks of conditional code up to twenty levels deep.

5.2.37 info

The "info" directive stores textual material in the information field of a binary file. The user may retrieve this information by using the UniFLEX command "info". The syntax for this directive is

```
info <str>
```

where <str> may be any sequence of ASCII characters, including the space character. The user need not enclose the string in quotation marks. No comment field may be used with this directive. The directive does not generate any binary code. Any number of "info" directives may appear at any point in the source code.

When the assembler processes the "info" directive, it places the text of the operand field (except leading spaces) at the end of a temporary file named "/tmp/asmbinfo<task_ID>". At the end of the assembly, all text from this temporary file is appended to the binary file, and the temporary file is deleted.

5.2.38 lib

The "lib" directive specifies an external, assembly-language source file for the assembler to include in the assembled source code. When the assembler processes a "lib" directive, it reads and assembles code from the specified file, adding it to the assembled source code. The line of code containing the "lib" directive does not appear in the assembled code. When the assembler has read and assembled all lines from that file, it resumes reading the original source code. The syntax for this directive is

```
lib <file_name>
```

If the file name specified begins with a slash character, '/', the assembler first looks for the file as specified. If it is not found or if the name does not begin with a slash, the assembler looks for the specified file in the working directory. If it does not find the library there, it looks for the directory "lib" in the working directory. If found, the assembler attempts to find the library in that "lib" directory. If it is not found there, the assembler makes a final attempt to find the library by looking in the directory "/lib". If the library is not in any of these places, the assembler issues an error message and aborts.

The assembler ignores any "end" directive that appears in a file called with the "lib" directive. The user may place any number of "lib" directives in the source code. A file called with the "lib" directive may contain another "lib" directive. The assembler supports this nesting of "lib" directives up to six levels deep.

The user may not include the "lib" directive in the definition of a macro (see Section 5.2.40).

5.2.39 log

The "log" directive calculates the base-2 logarithm of the value represented in the operand field and assigns the integer part of that logarithm to the specified label. The syntax for this directive is

```
<label> log <expr>
```

where <expr> is an absolute expression and <label> is an ordinary label. The user may redefine the label with other "log" directives or with the

"set" directive. The "log" directive generates no code.

5.2.40 macro

The "macro" instruction marks the beginning of the definition of a macro. A macro is a section of code that the assembler can access by name. It differs from a subroutine in that when the assembler sees a call to a macro, it actually inserts the designated code into the program rather than executing it at run-time. Thus, five calls to a subroutine require only one copy of the subroutine whereas five calls to a macro result in five copies of that code. On the other hand, a macro involves less overhead than does a subroutine. Consequently, using a macro may be more efficient than using a subroutine.

The definition of a macro must always precede the first call to that macro. It is good practice to define all macros near the beginning of a program.

The syntax for the "macro" directive is

```
<macro_name>  macro
```

where <macro_name> is an ordinary label (see Section 3.2.1). Only the first seven characters of the name are significant. A "macro" instruction is followed by the appropriate instructions, which may include any instruction except the "lib" directive. The definition of a macro must always end with the "endm" instruction (see Section 5.2.16). The length of the definition of a single macro must be less than or equal to 6K. This limit applies to the unexpanded code (see Section 5.3) between the "macro" and "endm" instructions exclusive.

When the assembler encounters a "macro" instruction during its first pass over the source code, it enters the name in a table of macro names and copies the instructions which comprise the macro into a buffer for future reference. The user can then call the macro simply by using <macro_name> in the opcode field. Whenever the user calls the macro, the assembler replaces the call with the appropriate code. This process is called macro expansion. The assembler searches the table containing the names of all macros in a program before it searches the mnemonic table. Thus, the user may redefine a standard instruction by replacing it with a macro of the same name. However, a macro, once defined, can be neither purged nor redefined.

The following example defines a macro which calculates the absolute value of the value in data register 0:

```
abs   macro
      tst.l   D0
      bge.s   00f
      neg.l   D0
00
      endm
```

The user may pass up to ten parameters to a macro: nine as operands on the line calling the macro; one as a label (see Section 5.3). A macro can both call and define other macros (see Section 5.4).

If any line within a macro contains an ordinary label and the user calls that macro more than once, the assembler returns the following nonfatal error:

```
Duplicate label found.
```

The user can avoid this situation by either using local labels within the macro or by making the label a substitutable parameter and passing in a different string each time the macro is called.

5.2.41 name

The "name" directive assigns a name to the module containing the assembled source code. The syntax for this directive is

```
name <module_name>
```

where <module_name> may contain a maximum of fourteen characters.

If a module has a name, the linking-loader uses that name when it reports errors and information about addresses. Otherwise, it uses the name of the file containing the module. Because a file may contain more than one module, it is wise to name every module. A module that does not have a name cannot be included in a library (see Section 8.2).

5.2.42 opt

The "opt" directive selects various options related to the format of the listing of the assembled source code. It is effective only if the user invokes the '1' option on the command line. The syntax for this directive is

```
opt [<option_list>]
```

The options available include the following:

con	Include skipped conditional code in the listing of the assembled source code. (Such code is not assembled.)
exp	Expand macros in the listing.
lis	Begin listing the assembled source code.
noc	Do not include skipped conditional code in the listing of the assembled source code.
noe	Do not expand macros in the listing.
nol	Suppress the listing of the assembled source code.

At the beginning of an assembly the default options "noc" and "lis" are in effect. The options are set during the second pass of the assembler over the source code. If the user specifies contradictory options, the last one takes precedence.

.2.43 pag

By default, the assembler inserts a page eject into the listing of the assembled source code after fifty-five lines of code and prints a header at the top of the next page. The header consists of four lines: the first line contains the title, date, time, and page number; the second contains the subtitle; and the third and fourth are blank. If the user does not specify a title or subtitle, that field remains blank. The "pag" directive performs a page eject in the listing of the assembled source code and writes a header at the top of the new page. The syntax for this directive is simply

```
pag
```

This directive produces no code, and the line containing it does not appear in the listing of the assembled source code unless it contains an error.

The first page of a listing is page 0. It has no header. Thus, the user can set all options, the title, and the subtitle before using the "pag" directive for the first time. The listing of the assembled source code then begins on page 1, after the assembler has processed the directives affecting the formatting.

5.2.44 quad (68020 only)

The 68020 microprocessor functions more efficiently when the data are aligned on a quad-word boundary (a boundary whose address is a multiple of 4). The "quad" directive forces the following piece of data to a quad-word boundary by filling in with null characters if necessary. Its syntax is simply

```
quad
```

5.2.45 rab

The "rab" directive reserves memory for the storage of data. It ensures that the first reserved byte is on an even boundary. The syntax for this directive is

```
rab <expr>
```

where <expr> is an absolute expression which is evaluated to 32 bits. The operand field may not contain a forward reference. An "rab" directive in the text or data segment puts out a null byte if necessary to position the program counter at an even boundary, then sets the reserved memory to 0. In the bss segment, it simply increases the size of the segment.

5.2.46 rmb

The "rmb" directive reserves memory for the storage of data. The syntax for this directive is

```
rmb <expr>
```

where <expr> is an absolute expression which is evaluated to 32 bits.

he operand field may not contain a forward reference. An "rmb" irective in the text or data segment sets the reserved memory to 0. In he bss segment, it simply increases the size of the segment. It is unctionally equivalent to the "rzb" directive.

.2.47 rzb

he "rzb" directive initializes an area of memory with null (zero) ytes. The syntax for this directive is

```
rzb <expr>
```

here <expr> is an absolute expression evaluated to 32 bits. The perand field may not contain a forward reference. An "rzb" directive n the text or data segment sets the reserved memory to 0. In the bss egment, it simply increases the size of the segment. It is unctionally equivalent to the "rmb" directive.

.2.48 set

he "set" directive sets a symbol to the value specified by the xpression in the operand field, just as the "equ" directive does. The ifference between the two directives is that the user may set a symbol everal times within the source code but may equate a symbol only once. f the user does set a symbol more than once, its current value is the ne most recently set. The syntax for this directive is

```
<label> set <expr>
```

here <expr> is an absolute, relocatable, or, on the 68020 only, a loating-point expression. The label must be an ordinary label. This irective generates no code.

.2.49 spc

he "spc" directive inserts blank lines into the listing of the ssembled source code. The syntax for this directive is

```
spc <blank_lines>[,<count>]
```

where <blank_lines> and <count> are both numbers between 0 and 255 inclusive. The first operand specifies the number of blank lines to insert. The optional operand <count> offers the user more control. If less than <count> lines remain on the page, the assembler performs a page eject rather than inserting any blank lines. In any case, when a page eject occurs, processing of the "spc" directive terminates. This feature prevents a block of lines from being split across a page.

5.2.50 struct

The "struct" directive sets a program counter outside the text, data, and bss segments. The syntax for this directive is simply

```
struct [<expr>]
```

where <expr> is an absolute or relocatable expression. If the user does not specify an operand, it defaults to 0. The "struct" segment continues until the assembler encounters a "base", "bss", "data", or "text" directive.

This type of program counter is used to establish a label for a particular offset from a fixed address. Generally, it is used in conjunction with storing information on a stack. The following example illustrates the use of the "struct" directive:

```
            struct
A6_link     ds.l    1
ret_addr    ds.l    1
arg_1       ds.l    1
arg_2       ds.l    1
arg_3       ds.l    1
            text
             .
             .
             .
            move.l  arg_1(A6),D0
            add.l   D0,D0
            sub.l   arg_3(A6),D0
            add.l   arg_2(A6),D0
             .
             .
             .
```

Symbols declared within a "struct" statement may be absolute or relocatable, depending on the attributes of the operand. The user may

reuse a symbol defined in a struct segment.

5.2.51 sttl

The "sttl" directive specifies the subtitle to print in the header at the top of the page. The syntax for this directive is

```
sttl [<text_of_subtitle>]
```

where the text of the subtitle is an ASCII string which may include between 0 and 52 characters inclusive. The assembler ignores excess characters. The user need not enclose the string in quotation marks. The string may contain space characters. The line of code containing the "sttl" directive may not contain a comment field. Any number of "sttl" directives may appear in one program. The subtitle that appears in the header of a given page is the argument to the most recently processed "sttl" directive.

5.2.52 sys

The "sys" directive sets up a call to the operating system. The syntax for this directive is

```
sys <function_code>[,<param_list>]
```

where both operands may be any legal expression. The assembler stores the function code in 16 bits; each parameter in 32 bits.

The function codes for the system calls are defined in the file "/lib/sysdef". The user may place either the function code or the name of the system call in the "sys" directive. If the name is used, the user must either define the name to match the information in "/lib/sysdef" or include the entire file in the source code by using the "lib" directive.

5.2.53 text

The "text" directive relocates the instructions which follow it to the end of the text segment. The syntax for this directive is simply

```
text
```

A "text" directive remains in effect until the assembler encounters a "base", "bss", "data", or "struct" directive.

5.2.54 tstmp

The "tstmp" instruction uses the current date and time to write a time stamp to the information field of the assembled file. The syntax for this directive is

```
tstmp [<str>]
```

By default, the assembler creates a time stamp of the following form:

```
-- Created:  <time_stamp>
```

The optional argument <str> allows the user to choose a string to replace "-- Created". The string specified by the user may be any sequence of ASCII characters, including spaces. The user need not enclose the string in quotation marks.

No comment field may be used with this directive. It generates no binary code. Any number of "tstmp" directives may appear at any point in the source code.

When the assembler processes the "tstmp" directive, it places the text of the operand field (except leading spaces) at the end of a temporary file named "/tmp/asmbinfo<task_ID>". At the end of the assembly, all text from this temporary file is appended to the binary file, and the temporary file is deleted.

5.2.55 ttl

The "ttl" directive specifies the title to print in the header at the top of the page. The syntax for this directive is

```
ttl [<text_of_title>]
```

where the text of the title is an ASCII string which may include between 0 and 32 characters inclusive. The assembler ignores excess characters. The user need not enclose the string in quotation marks. The string may contain space characters. The line of code containing the "ttl" directive may not contain a comment field. Any number of "ttl" directives may appear in one program. The title that appears in the header of a given page is the argument to the most recently processed "ttl" directive.

5.3 Parameter Substitution in Macros

With each call to a macro the user may pass up to ten parameters to the assembler for use during macro expansion: nine as operands on the line calling the macro; one as a label. Thus, the code actually generated by the call to the macro may differ from one call to the next. It is this feature, known as parameter substitution, that makes the use of macros a powerful tool for the assembly language programmer.

5.3.1 Specifying Parameters

The parameters passed to a macro are ASCII strings which the user places in the operand field of the line of code that calls the macro. The size of a string is limited only by the fact that the code making the call must fit on one line. Every parameter except the last one must immediately be followed by a comma.

Normally, the assembler interprets a space character that is not enclosed in quotation marks as the end of the list of parameters; a comma, as the character that separates parameters. A user may include either of these characters in a parameter by enclosing the string in a pair of matching delimiter characters (either single or double quotation marks). Placing a pair of delimiter characters side-by-side or leaving a string out altogether (resulting in two commas side-by-side) specifies the null string.

5.3.2 Substitutable Parameters

If the user specifies one or more strings in the operand field of a call to a macro, the assembler searches the source code for the sequence

&<digit>

where <digit> may be any digit from '1' to '9' inclusive. Such a sequence is called a substitutable parameter. The assembler replaces the substitutable parameter "&1" with the first string in the operand field; the parameter "&2", with the second, and so forth.

The user may also specify a tenth substitutable parameter, "&0". The assembler replaces this parameter with the string in the label field of the line calling the macro. The assembler treats this string like an ordinary label, placing it in the symbol table so that a user can reference it like any other label. It must, of course, conform to the rules for ordinary labels (see Section 3.2.1).

5.3.3 Ignoring a Substitutable Parameter

A user who is passing parameters to a macro but wishes to avoid substitution in the source code in a particular instance must precede the ampersand, '&', with a backslash character, '\'. Thus, if it is substituting parameters in a macro, the assembler sees the following line of source code

mask\&4

as

mask&4

If the user specifies one or more parameters in the operand field of a call to a macro and the assembler finds a substitutable parameter for which the user did not specify a corresponding string (e.g., the assembler finds the sequence "&4" but the user specifies only three parameters in the call to the macro), the assembler replaces the sequence with the null string.

5.3.4 Examples

To illustrate the principles of parameter substitution, consider the following macro, which adds three 32-bit numbers found in memory and stores the result in a fourth location:

```
add3  macro
      move.l  loc_1,D0    Move first value to D0
      add.l   loc_2,D0    Add in second value
      add.l   loc_3,D0    Add in third value
      move.l  D0,sum      Store sum
      endm
```

The user calls this macro by placing its name, "add3", in the opcode field of a line of source code. Identical code results each time the user calls "add3".

This macro is specific to certain locations in memory. The user can make the macro more versatile by rewriting it to take advantage of parameter substitution:

```
add3  macro
      move.l  &1,D0  Move first value to D0
      add.l   &2,D0  Add in second value
      add.l   &3,D0  Add in third value
      move.l  D0,&4  Store sum
      endm
```

To call this version of the macro, the user not only places the name of the macro in the opcode field but also specifies four parameters in the operand field:

```
add3  loc_1,loc_2,loc_3,sum
```

When the assembler expands this call to the macro, it produces the same code as it does for the first example. However, numbers at any three locations can easily be added and stored in a fourth location by changing the parameters in the operand field, as the following call illustrates:

```
add3  loc_4,loc_5,loc_6,sum_2
```

5.4 Nesting of Macro Definitions

Macros may be nested--that is, one macro may both call and define other macros. When a user defines one macro within another, the assembler defines the inner macro as it expands the outer macro, at which time it also performs parameter substitution. The user must take care to avoid duplicating the definition of the inner macro and to ensure that parameter substitution occurs as intended.

5.4.1 Avoiding a Duplicate Definition

A user who defines one macro within another must be certain that the the assembler does not define the inner macro more than once. Such duplication may be avoided by calling the outer macro only once, by using a substitutable parameter for the name of the inner macro, or by placing the definition of the inner macro within a block of conditional code that is assembled only once. Consider, for example, the following definition:

```
mac_1   macro
        nop
mac_2   macro
        move.l    &1,&2
        endm
        endm
```

The first call to "mac_1" defines the inner macro, "mac_2". A second call to "mac_1" results in a second definition of "mac_2", which is illegal. The following example avoids this problem by using a substitutable parameter as the name of the second macro:

```
mac_1   macro
        nop
&1      macro
        move.l    &2,&3
        endm
        endm
```

In order to successfully define the inner macro, the user must specify the string to use as its name as the first element of the operand field of the line making the call to "mac_1". Each successive call to "mac_1" must pass a different string to use as the name of the inner macro in order to avoid a duplicate definition.

he following example uses a block of conditional code to ensure that he assembler defines "mac_2" only once:

```
flag    set     0
          .
          .
          .
mac_1   macro
        nop
        if      flag=0
mac_2   macro
        move.l  &1,&2
        endm
flag    set     1
        endif
        endm
```

.4.2 Parameter Substitution within Nested Definitions

arameter substitution takes place during the expansion of a macro. hus, if one macro is defined within another, the assembler replaces any ubstitutable parameters used in the definition of the inner macro as it xpands the outer macro. Parameter substitution can, as usual, be voided by preceding the ampersand, '&', with a backslash, '\'. Doing o delays the parameter substitution until the user calls the inner acro. For example, consider the following definition:

```
makstat  macro
         clr.l   &1
setstat  macro
         bset    #\&1,&1
         endm
         endm
```

he first call to the outer macro, "makstat", defines the inner macro, setstat". As the assembler defines "setstat", it replaces the second perand of the "bset" instruction with the first parameter in the perand field of the call to "makstat". The first operand of the "bset" nstruction, however, enters the definition as "&1", awaiting parameter ubstitution by a call to "setstat". Therefore, the assembler expands he following code:

```
makstat  my_word
setstat  3
setstat  4
```

to

```
clr.l    my_word
bset     #3,my_word
bset     #4,my_word
```

Chapter 6

Error Messages from the Relocating Assembler

6.1 Introduction

The assembler produces two types of error messages: nonfatal and fatal. All messages are English phrases rather than error numbers.

If the assembler encounters a nonfatal error, it inserts an error message into the listing of the assembled source code and, if the error is in a line that generates code, assembles it into some default code. Even if the user suppresses the listing of the assembled code, the assembler sends those lines containing errors to standard output. All messages produced by nonfatal errors are preceded by three asterisks so that the user can easily locate them.

A fatal error is one which causes immediate termination of the assembly. The assembler sends fatal error messages to standard error in the following format:

```
Last Line = <last_line_read>
Line Number <num>
Fatal Error - <error_message>
```

6.2 Nonfatal Errors

16-bit expression expected.
The assembler expected a 16-bit expression in the operand field, but the expression found is too large to represent in 16 bits.

8-bit expression expected.
The assembler expected an 8-bit expression in the operand field, but the expression found is too large to represent in 8 bits.

A label declared "global" was not found in the program.
All labels declared with the "global" directive must be defined in the same module.

Absolute expression required.
The assembler requires an absolute expression in this context.

Address register direct allowed only with FPIAR.
If the source or destination of an "fmovem" instruction is an address register, the register being moved must be FPIAR.

Assembling for 68000 but 68010 instruction specified.
When the 't' option is in effect, "rel68k" does not accept those instructions which are supported only by the 68010--"movec", "moves", and "rtd".

Branches not allowed across segment boundaries.
Branches cannot be made to labels in other segments or to external references.

Cannot evaluate conditional expression in pass 1.
The assembler must be able to evaluate during pass 1 the expression used as an operand with a directive that introduces a block of conditional code. Such an expression, therefore, cannot contain a forward reference to any label.

Cannot evaluate expression.
The assembler could not evaluate the expression.

Cannot evaluate expression in pass 1.
Assembler directives which reserve memory, such as "ds" and "rmb", must be evaluated during both passes of the assembler. Therefore, with such directives, only constant operands are legal, and forward references are not allowed.

Cannot find that local label.
The local label specified in the expression is not defined in the source code. Note that the two local labels "0" and "00" are distinct.

Data register required.
A data register is required as one of the operands for the instruction specified.

Data register required as source/destination of bit-field instruction.
Depending on the instruction, either the source or the destination of a bit-field instruction with two operands must be a data register.

Duplicate label found.
The label on this line is defined more than once.

Evaluator : attempt to divide by zero.
The divisor of an expression is equal to 0.

Evaluator : invalid operation for relocatable or external expression.
Only absolute expressions can be added to a relocatable or external expression.

Evaluator : more than one external found in an expression.
Only one external reference may appear in an expression.

Evaluator : must shift by positive, nonzero quantity.
The second operand in the operand field of a shift instruction must be greater than 0.

Evaluator : operator only valid for absolute expressions which are not floating-point expressions.
The following operations may be performed only on absolute expressions: and, or, exclusive or, not, multiply, divide, shift, and logical.

Evaluator : unbalanced expression (with respect to segments).
A pair of relocatable items used in an expression must both be from the same segment of the program (see 3.8.4.2).

Evaluator : unbalanced parentheses.
The parentheses in the expression are not properly balanced.

External expression not allowed.
An external expression is not allowed in this context.

Extra arguments found.
The assembler found more arguments than it expected.

Floating-point number not allowed.
A floating-point number is not allowed in the context in which it was used.

Floating-point register-pair required.
A pair of floating-point registers must be specified for the "fsincos" instruction.

Floating-point values cannot be used with relocatables or externals.
An expression involving floating-point values was mixed with externals or relocatables.

Forced short but long expression found.
The programmer used absolute-short addressing mode, but the corresponding expression is too large to fit into 16 bits.

Found zero branch length on short branch.
The destination of a short branch cannot be the next instruction.

Illegal addressing mode for instruction.
An addressing mode used in the operand field is not legal for this instruction.

Illegal character in label.
Labels must consist solely of alphabetic characters, digits, the question mark, and the underscore character. The first character may not be a digit.

Illegal expression or missing operand.
The expression evaluator cannot parse the expression.

Illegal instruction for this segment.
Instructions which generate code cannot be used in a base, struct, or bss segment.

Illegal nesting of conditionals.
Conditional code is incorrectly nested. An "endif" directive must follow any directive that introduces a block of conditional code. Only one "else" statement may apply to such a directive.

Illegal register list for "movem".
The specification for the register list in a "movem" instruction is invalid (see Section 4.5).

Illegal size for instruction.
The suffix specified (".b", ".l", or ".w" for "rel68k"; also ".d", ".p", ".s", or ".x" for "rel20") may not be used with this instruction.

Illegal special register for instruction.
The special register specified as an operand may not be used with this instruction.

Illegal use of a "bfequ" symbol.
A symbol used in a "bfequ" instruction is only valid with a bit-field instruction.

Illegal use of a macro name.
When a user invokes a macro, the name of the macro must appear in the opcode field.

Immediate size does not match instruction size.
The immediate operand is larger than the size specified by or implicit in the instruction.

Invalid argument to "bfequ".
The "bfequ" directive requires as arguments two absolute constants separated by a colon, ':'.

Invalid bit-field offset specified.
The bit-field offset must be either a number between 0 and 31 inclusive, or a data register.

Invalid bit-field width specified.
The bit-field width must be either a number between 1 and 32 inclusive, or a data register.

Invalid breakpoint vector specified.
A breakpoint vector must be between 0 and 7 inclusive.

Invalid coprocessor ID.
A coprocessor ID must be between 0 and 7 inclusive.

Invalid floating-point constant.
The floating-point constant does not have the correct format.

Invalid instruction size for use with a floating-point constant.
The instruction size is not valid for use with a floating-point constant.

Invalid local label.
A local label must consist of a string of one or two digits ('0' through '9').

Invalid number of operands.
The number of operands specified is incorrect for the instruction.

Invalid offset/width pair.
The assembler could not interpret the specified offset and width of a bit-field instruction.

Invalid option to the "opt" directive.
The option specified is not a valid option to the "opt" directive. Valid options are "con", "noc", "lis", and "nol".

Invalid or missing K-factor.
The user must specify the K-factor for packed-decimal floating-point move operations. The value of the K-factor must be between -64 and +17 inclusive.

Invalid register in control-register list.
The only valid control registers for the "fmovem" instruction are FPCR, FPIAR, and FPSR.

Invalid ROM constant number.
A ROM constant must be between 0 and 63 inclusive.

Invalid scale factor specified.
The scale factor for an index register must be 1, 2, 4, or 8.

Invalid transfer address found (external).
The assembler does not support external transfer addresses.

Invalid trap vector specified.
The trap vector must be between 0 and 15 inclusive.

Invalid use of "endm" directive.
The only valid use of the "endm" directive is at the end of the definition of a macro.

Invalid use of "exitm" directive.
The only valid use of the "exitm" directive is within the definition of a macro.

Label declared as both external and global.
A label cannot be both external and global.

Label required.
The directives "common", "equ", "log", "macro", and "set" require a label in the label field.

Long branch required.
The assembler found a branch that requires a 32-bit displacement. By default, the assembler generates branches with 16-bit displacements.

Missing argument.
The assembler expected more arguments than the user specified.

More than nine macro parameters specified.
The user may use a maximum of nine parameters in the operand field of a line calling a macro.

Negative value not allowed.
The operand for this instruction must represent a nonnegative number.

Nested "common" directives not supported.
Common blocks cannot be nested.

No closing delimiter found.
The assembler encountered an end-of-line character before it encountered the closing delimiter of a string.

No "endcom" directive found.
A common block declaration must be bracketed by the two directives "common" and "endcom" and must all be within one segment.

No label allowed on "endm" directive.
The label field of the "endm" directive must be blank.

Odd branch address found.
The assembler detected a branch to a label on an odd address. The assembler forces each opcode and some directives to start on an even boundary. However, this error can occur if a label is on a line by itself after some odd number of bytes of data or if a label is on the same line as a directive whose operand need not be aligned.

Offset and width must be absolute expressions.
The offset and width of a bit-field instruction cannot be relocatable, external, or floating-point expressions.

Offset and width must be specified in a bit-field instruction.
The user must specify the offset and width in a bit-field instruction; the assembler does not provide default values.

Only one control register allowed.
If the source or destination of an "fmovem" instruction is a data register, the user may specify only one control register.

Operand out of range.
The instruction specified requires a "quick" number (a number between 1 and 8 inclusive) for the immediate operand in the operand field.

Overlapping register list specified.
One or more registers appear more than once in the register list of a "movem" instruction.

Phasing error.
The two passes of the assembler do not agree on the address of the label on the current line. This error can only be caused by other errors in the assembly and should not appear as the only error in a given module. Only the first phasing error is reported. The assembler checks only those lines that contain labels.

Register-pair specifies the same register.
The pair of registers used in a divide instruction specifies the same register for both the remainder and the quotient.

Relocatable displacement from the same segment required.
The label used in program-counter-relative addressing mode must be a relocatable label from the same segment as the program counter.

Relocatable displacement not allowed.
A relocatable displacement is not allowed in this context.

Relocatable expression required.
A relocatable expression is required in this context.

Short branch not allowed.
  Floating-point branches must be long branches.

Symbol found in "extern" also found as program label.
  A symbol declared external with the "extern" directive is also defined elsewhere in the module.

The operand was too large for the size specified.
  The size specified in the instruction is smaller than the size of the immediate operand specified as the first operand.

Too far for a branch instruction.
  A branch cannot be made over a distance that cannot be represented by a 16-bit expression. A jump must replace the branch instruction.

Too far for a short branch.
  A short branch cannot be made over a distance that cannot be represented by an 8-bit expression. A long branch must replace the short branch.

Undefined symbol found.
  A symbol in an expression is not defined anywhere in the module.

Unknown addressing mode.
  The assembler cannot interpret the addressing mode specified.

Unknown instruction.
  The string in the opcode field is not a known instruction (see Chapters 4 and 5).

Unknown length modifier specified.
  The only legal size specifications for an index register are 'L', 'l', 'W', and 'w'.

Unknown size specified.
  The only legal size extensions for instructions are ".b", ".l", ".s", and ".w" for "rel68k"; also ".d", ".p", and ".x" for "rel20".

Word operand required for system call name.
  The system call specified is not a legal system call. The system call number must fit into a word (16 bits). See the Introduction to UniFLEX System Calls for a list of system calls.

6.3 Fatal Errors

Allocating table of local labels.
The system does not have enough memory to allow initial allocation for the table of local labels. This situation should not arise. If it does, the user should contact the vendor.

Conditional nesting too deep.
The user may not nest conditional code more than twenty levels deep.

EOF during macro definition.
The assembler reached the end of the file in the definition of a macro. Most probably, the user forgot to use the "endm" directive at the end of the definition.

Growing table of local labels.
The table of local labels, which grows dynamically, has grown to the limit imposed by the operating system.

Incompatible options: 'J' and 'I'
The 'J' option, which tells the assembler to ignore the ":w" suffix unless it is part of a "jmp" or "jsr" instruction, cannot be used with the 'I' option, which tells the assembler to ignore the ":w" suffix if it is part of a "jmp" or "jsr" instruction.

Invalid option: '<char>'
The option specified by <char> is not a valid option to the command invoked by the user.

"lib" directive in macro.
The definition of a macro may not contain a "lib" directive.

Library file "<file_name>" not found.
The assembler cannot locate the library file specified. It searches the working directory, the directory "lib" in the working directory, and the directory "/lib".

Library nesting too deep.
Libraries may not be nested more than six levels deep.

Macro buffer overflow.
The length of the definition of a single macro must be less than or equal to 6K.

No file specified.
The user did not specify a source file on the command line.

Opening "<file_name>": <reason>
The operating system returned an error when the assembler tried to open the specified file. This message is followed by an interpretation of the error returned by the operating system.

Out of space.
The assembler's symbol table, which grows dynamically, has grown to the limit imposed by the operating system. The restriction on the size of the symbol table depends on both the hardware and the configuration of the operating system. The simplest solution is to raise the size limit if possible. Otherwise, the user must break the source code into smaller modules and assemble them separately.

Reading "<file_name>": <reason>
The operating system returned an error when the assembler tried to read the specified file. This message is followed by an interpretation of the error returned by the operating system.

Seeking in "<file_name>": <reason>
The operating system returned an error when the assembler tried to seek to a particular location in the specified file. This message is followed by an interpretation of the error returned by the operating system.

Writing to "<file_name>": <reason>
The operating system returned an error when the assembler tried to write to the specified file. This message is followed by an interpretation of the error returned by the operating system.

Chapter 7

Invoking the Linking-Loader

7.1 Introduction

The "load68k" command accepts as input one or more relocatable binary modules and produces as output either a relocatable module or an executable module. The relocatable modules used as input must have been produced by the relocating assembler or the linking-loader.

7.2 The Standard Environment

A file named "/lib/std_env" is supplied with every UniFLEX Operating System. This file, which describes the standard hardware-specific environment of the particular system, contains a series of options which the linking-loader automatically processes before it processes any options from the command line. The options specify information about such things as the hardware page-size and the starting address of the text and data segments. The linking-loader uses the file to get the basic information it needs in order to load any module. If necessary, the user may override the options in this file by specifying the same options with different arguments on the command line.

7.3 The Command Line

The syntax for invoking the linking-loader is as follows:

```
load68k <file_name_list> [+aAbBcCdDefFiIlLmMnNoPqrRsStTuUwWxXyYZ]
```

where <file_name_list> is a list of the names of the files to load. Files are loaded in the order in which they appear on the command line. Brief descriptions of the options which are available follow. Those options which do not take an argument can be disabled by preceding the character with a minus sign, '-', instead of the usual plus sign, '+'. The linking-loader ignores the minus sign if it precedes an option which takes an argument.

a=<num>	Specifies the minimum number of pages to allocate to this task at all times.
A=<num>	Specifies the maximum number of pages to allocate to this task at all times.
b=<task_size>	Specifies the maximum size to which the task may grow.
B	Set a bit in the binary header of the output module which tells the operating system to zero neither the bss segment nor any memory allocated while the task is running.
c=<source_type>	Specifies the source code from which the module was created.
C=<config_num>	Specifies the configuration number of the hardware.
d	Set the "no core dump" bit in the binary header.
D[=<hex_num>]	Specifies the starting address of the data segment.
e	Print each occurrence of all unresolved external references.
f	Produce a demand-load executable module as output.
F[=<file_name>]	Specifies the name of a file of options to process.
i	Write all global symbols to the symbol table of the binary file.
I	Enable processing of floating-point interrupts (exceptions). This option is only useful on a system with an MC68881 floating-point coprocessor.
l=<library_name>	Specifies the name of a library to search.
L	Do not search any libraries for unresolved external references.
m	Produce load and module maps and write them to standard output.
M=<file_name>	Specifies the name of the file in which to put the output of the 'm' option (load and module maps) and the 's' option (a global symbol table).
n	Produce an executable module with separate instruction and data space.
N=<module_name>	Specifies the internal module name.
o=<file_name>	Specifies the name to give to the binary file.
P=<hex_num>	Specifies the page size.
q	Suppress quad-word alignment of all segments.
r	Produce a relocatable module as output. Do not search any libraries.

R	Produce a relocatable module as output. Search "/lib/syslib68k" and any libraries specified by the user for unresolved external references.
s	Write the global symbol table to standard output.
S=<hex_num>	Specifies the initial stack size.
t	Produce a shared-text executable module.
T=<hex_num>	Specifies the starting address of the text segment.
u	Do not produce any "unresolved" messages when producing a relocatable module.
U=<trap_num>	Specifies the trap number for system calls.
w	Load modules containing instructions specific to the MC68020.
W	Do not load modules containing instructions specific to the MC68020.
x=<file_name>	Specifies the name of the file whose symbol table is to form the basis of the new symbol table. The new module loads into memory immediately after the specified file.
X=<add_mask>	Specifies a 32-bit hexadecimal number, of which only the high-order 7 bits are used, which is used to mask out any of the high-order 7 bits in all addresses in the file.
y	Load modules containing instructions specific to the MC68881 coprocessor.
Y	Do not load modules containing instructions specific to the MC68881 coprocessor.
Z	Align text and data segments on 512-byte boundaries.

Detailed descriptions of the options follow.

7.3.1 The 'a' Option

The 'a' option specifies the minimum number of pages to be allocated to the task at all times. The syntax for this option is

```
a=<num>
```

where <num> is between 0 and 32767 inclusive. The default is 0. The operating system tries to honor the specified number, but if it cannot, it uses as many pages as it needs. This option is only effective on virtual-memory systems.

7.3.2 The 'A' Option

The 'A' option specifies the maximum number of pages to be allocated to the task at all times. The syntax for this option is

A=<num>

where <num> is between 0 and 32767 inclusive. The default is 0. The value specified should be greater than or equal to the value specified for the 'a' option. The operating system tries to honor the specified number, but if it cannot, it uses as many pages as it needs. This option is only effective on virtual-memory systems.

If the minimum and maximum values for page allocation provided by the user make no sense, the linking-loader automatically adjusts them according to the following rules. The value for the maximum is always greater than or equal to the value for the minimum. The value for the maximum may be 0, but if it is greater than 0, it must be at least 4.

7.3.3 The 'b' Option

The 'b' option specifies the maximum size to which the task may grow during execution. The syntax for this option is

b=<task_size>

where <task_size> may be one of the following: "128K", "256K", "512K", "1M", "2M", "8M", "16M", "32M", "64M", "128M", "256M", "512M", "1G", "2G", "4G", "S", "M", "L", "small", "medium", or "large". All letters may be in upper- or lowercase. The size of a task specified by 'S' (or "small"), 'M' (or "medium"), or 'L' (or "large") is vendor-dependent. Typically, however, 'S' specifies 128K; 'M', the size of physical memory; 'L', the maximum size allowed to a task. The default task size is 128K.

If the task size specified by the user (or the default) is not large enough to hold the code from all the modules being loaded, the linking-loader automatically adjusts the size to the smallest value that can contain all the code.

7.3.4 The 'B' Option

The 'B' option sets a bit in the binary header of the output module which tells the operating system to zero neither the bss segment nor any memory allocated while the task is running. This option may not be effective on all machines.

7.3.5 The 'c' Option

The 'c' option sets a flag in the module produced by the linking-loader which indicates the type of source code from which the module was created. This information is useful for debugging purposes. The syntax for this option is

```
c=<source_type>
```

The value for <source_type> must be one of the following: "ASSEMBLER", "C", "COBOL", "FORTRAN", or "PASCAL". The names may be specified in either upper- or lowercase letters.

7.3.6 The 'C' Option

By default, the linking-loader uses the configuration number of the current hardware. The user may, however, use the 'C' option to specify a configuration number which overrides the default. This option is useful when loading a module for a machine other than the one on which it is running. The syntax for this option is

```
C=<config_num>
```

7.3.7 The 'd' Option

The 'd' option sets the "no core dump" bit in the binary header of the module produced by the linking-loader. If this bit is set, the module can never produce a core dump. This option may not be effective on all machines.

7.3.8 The 'D' Option

The 'D' option specifies the starting address of the data segment. If the user does not specify the option, the starting address defaults to the address specified in the file "/lib/std_env". It may, however, be necessary to override this address if the user wishes to load a module for execution on another machine with different hardware requirements. The syntax for this option is

D[=<hex_num>]

where <hex_num> is a hardware-dependent hexadecimal number. If the user does not specify an argument, the data segment starts immediately after the text segment.

7.3.9 The 'e' Option

The 'e' option tells the linking-loader to print each occurrence of all unresolved external references. By default, the linking-loader prints only the first occurrence.

7.3.10 The 'f' Option

The 'f' option tells the linking-loader to produce as output a demand-load executable module. In such a module the text and data segments each begin on a 512-byte boundary. By default, when the operating system executes a module, it loads the entire data and text segments. When executing a demand-load module, it loads the entire data segment but loads the text segment only as it is needed. The operating system locks a demand-load module during execution.

7.3.11 The 'F' Option

The 'F' option specifies the name of a file of options for the linking-loader to process. Thus, the user may place the desired options in a file rather than listing them on the command line each time the linking-loader is invoked. (The file "/lib/std_env", which is supplied with the operating system, is a file of options which the linking-loader always reads before processing any options specified by the user.) The last occurrence of an option always overrides previous occurrences.

Therefore, if a file of options is specified at the beginning of the command line, any options which follow override the specifications of the same options in that file of options. The 'F' option may be used repeatedly on the command line, but it may not be used inside a file of options. The syntax of this option is

```
F[=<file_name>]
```

If the user does not specify an argument, the linking-loader reads the file "ldr_opts" in the working directory.

Each option string specified in a file must be separated from the preceding one by one or more spaces and must begin with a plus sign, '+'. The linking-loader discards all characters up the first plus sign on each line of the file. Thus, the user may insert comments before the first option string on any given line. For example, the following is a valid file of options:

```
System XYZ File of Options for Loader
Starting Addresses of Data and Text +D=400000 +T=0
Produce Maps +msM=mapout
Produce a Shared-Text Module +t
```

7.3.12 The 'i' Option

If the user specifies the 'i' option, the linking-loader includes a symbol table in the module it produces. By default, the linking-loader includes a symbol table in a relocatable module but not in an executable one.

7.3.13 The 'I' Option

The 'I' option sets a bit in the binary header of the file, which enables the processing of floating-point exceptions. In order to process such exceptions, the user must specify in the source code which exceptions to process and how to process them. If the bit in the header is not set, the operating system ignores floating-point exceptions.

7.3.14 The 'l' Option

The 'l' option, which may be used up to twelve times in a single invocation of the linking-loader, specifies the name of a library for the linking-loader to search when it is trying to resolve unresolved external references. The syntax for this option is

l=<library_name>

If the library name specified begins with a slash character, '/', the linking-loader first looks for the library as specified. If it is not found or if the name does not begin with a slash, the linking-loader searches the working directory, then the directory "lib" in the working directory, and finally the directory "/lib" for the specified library. If the user specifies less than twelve libraries, the linking-loader automatically searches the library "Syslib68k" in the working directory after it has finished searching the libraries specified by the user. The linking-loader always processes the libraries in the order in which the user specifies them on the command line.

7.3.15 The 'L' Option

The 'L' option tells the linking-loader not to search any libraries for unresolved external references.

7.3.16 The 'm' Option

When the user specifies the 'm' option, the linking-loader produces a load map and a module map. A load map contains information about the type of module being produced as well as the starting addresses of the text and data segments, the initial stack size, the page size (granularity), and the binary transfer address. It also tells how many modules the linking-loader combined to produce the new module. A module map contains the name of each module loaded, the name of the file containing each module, and the starting addresses of each segment (text, data, and bss) of each module. It also contains the final address of each segment of the new module.

7.3.17 The 'M' Option

The 'M' option specifies the name of the file in which the linking-loader is to put the output from the 'm' option (load and module maps) and the output from the 's' option (a global symbol table). The syntax of this option is

M=<file_name>

The information in this file is purely textual. The user may edit or list the file like any other text file. If the 'm' or 's' option is used without the 'M' option, the linking-loader sends the information to standard output.

7.3.18 The 'n' Option

This option sets a bit in the binary header of the module produced by the linking-loader which tells the operating system that the hardware can support separate data and instruction spaces. The operating system handles addressing accordingly.

7.3.19 The 'N' Option

The 'N' option is used to specify the internal name of the module produced by the linking-loader. It is similar to the "name" directive of the relocating assembler. The syntax for this option is

N=<module_name>

where <module_name> is limited to fourteen characters. If the user does not specify the 'N' option, the linking-loader does not name the module.

7.3.20 The 'o' Option

The 'o' option specifies the name of the file containing the module produced by the linking-loader. The syntax for this option is

o=<file_name>

If the user does not specify the 'o' option, the name of the file defaults to <file_name>.o in the working directory where <file_name> is the first argument to the "load68k" command. If a file by this name already exists, it is deleted without warning.

7.3.21 The 'P' Option

The 'P' option specifies the page size of the hardware. The syntax for this option is

P=<hex_num>

The hexadecimal number used should always be a power of 2; otherwise, the results are unpredictable.

The "load68k" command uses the page size to determine the starting address of the data segment when it immediately follows the text segment (the data segment starts at the next page boundary). The default is 0 (i.e., the linking-loader rounds the starting address to the next even location after the end of the text segment).

7.3.22 The 'q' Option

By default, as it loads each segment of a module, the linking-loader ensures that it starts on a quad-word boundary (a boundary whose address is a multiple of 4). It does so because quad-word alignment makes access to data more efficient on the MC68020. In doing so, the linking-loader wastes at most three bytes per module, per segment. This waste can be avoided by suppressing the alignment with the 'q' option.

7.3.23 The 'r' Option

The 'r' option tells the linking-loader to produce a relocatable module without searching any libraries (even if the user has specified some with the 'l' option) for unresolved external references. By default, the linking-loader produces an executable module. If the user specifies the 'r' option, the linking-loader produces a module identical to the module that would have been produced if all the modules had been in one source file and that file had been assembled by the relocating assembler.

7.3.24 The 'R' Option

The 'R' option tells the linking-loader to produce a relocatable module and to search "/lib/syslib68k" and any libraries specified by the user for unresolved external references. By default, the linking-loader produces an executable module. If the user specifies the 'R' option, the linking-loader produces a module that is identical to the module that would have been produced if all the modules had been in one source file and that file had been assembled by the relocating assembler.

7.3.25 The 's' Option

The 's' option tells the linking-loader to write the symbol table to standard output or, if the 'M' option is in effect, to the file specified by that option.

7.3.26 The 'S' Option

The 'S' option writes the initial stack size for the task into the binary header of the module produced by the linking-loader. The syntax of this option is

S=<hex_num>

where <hex_num> is the number of bytes to reserve. The default is 0, in which case the system determines the initial stack size.

The operating system reads this information from the header and assigns a minimum of 4K to the stack of any task. Thus, the user can specify more than the usual amount of stack space but not less. The operating system always assigns stack space in multiples of 4K, using the value of the next multiple if the number in the header is not an even multiple.

7.3.27 The 't' Option

The 't' option sets a bit in the binary header of the module produced by the linking-loader which tells the operating system that it is a shared-text module (see Section 9.1).

7.3.28 The 'T' Option

The 'T' option specifies the starting address of the text segment. The syntax for this option is

T[=<hex_num>]

If the user does not specify the 'T' option, the linking-loader uses the starting address specified in the file "/lib/std_env". If no address is specified there, the starting address defaults to 0. If the user specifies the 'T' option without an argument, the starting address also defaults to 0. If, however, the 'x' option is in effect, the starting address never defaults to 0, but instead defaults to the first page boundary following the bss segment of the file specified by the 'x' option (see Section 7.3.33).

7.3.29 The 'u' Option

The 'u' option causes the linking-loader to suppress messages concerning unresolved external references when producing a relocatable module.

7.3.30 The 'U' Option

The 'U' option sets the trap number for system calls. The syntax for this option is

U=<trap_num>

Trap vectors must have one of two forms: "TRAPn", where 'n' is a number between 0 and 15 inclusive, or a 4-digit hexadecimal number which represent a bit pattern to use as the system call. The word "TRAP" may be specified in either upper- or lowercase letters.

7.3.31 The 'w' Option

The 'w' option tells the linking-loader to load modules containing instructions specific to the MC68020 microprocessor regardless of the hardware configuration. This option allows the user to produce binary files for the MC68020 on an MC68000-based machine.

7.3.32 The 'W' Option

The 'W' option tells the linking-loader not to load modules containing instructions specific to the MC68020 microprocessor regardless of the hardware configuration. This option allows the user to produce binary files for the MC68000 on an MC68020-based machine. It also allows the user to produce binary files on a 68000-based machine without accidentally loading some modules that contain instructions specific to the MC68020.

7.3.33 The 'x' Option

The 'x' option specifies the name of a file whose symbol table is to form the basis of the new symbol table. The syntax for this option is

x=<file_name>

where <file_name> must be the name of a file containing an executable module. The linking-loader uses the symbol table from <file_name>, if one exists, as the basis of the symbol table for the new module. The user can, as usual, specify the address at which to begin loading the new module with the 'T' option. However, when the 'x' option is in effect, the starting address never defaults to 0, but instead defaults to the first page boundary following the bss segment of the file specified by the 'x' option (see Section 7.3.28). To be safe, in conjunction with the 'x' option the user should always specify the 'T' option without an argument, so that it overrides any starting address in the file "/lib/std_env".

7.3.34 The 'X' Option

The 'X' option specifies a mask, which is used to mask out any of the high-order 7 bits in all addresses in the file. The syntax for this option is

X=<add_mask>

where <add_mask> is a 32-bit hexadecimal number. Not all hardware supports this option.

7.3.35 The 'y' Option

The 'y' option tells the linking-loader to load modules containing instructions specific to the MC68881 coprocessor regardless of the hardware configuration. This option allows the user to produce binary files for the a machine with the coprocessor on a machine that does not have one.

7.3.36 The 'Y' Option

The 'Y' option tells the linking-loader not to load modules containing instructions specific to the MC68881 coprocessor regardless of the hardware configuration. This option allows the user to produce binary files for the a machine that does not have an MC68881 coprocessor on a machine that does. It also allows the user to produce binary files on a machine without a coprocessor without accidentally loading some modules that contain instructions specific to the MC68881.

7.3.37 The 'Z' Option

The 'Z' option instructs the loader to align the text and data segments on 512-byte boundaries, padding with null bytes as necessary. This type of alignment uses more space, but it makes the loading of the executable module by the operating system more efficient.

7.4 Examples

The following examples illustrate some of the uses of the "load68k" command.

1. load68k *.r +F=/lib/ldr_environ +t +l=Clib +o=tester
2. load68k t1.r t2.r +T=20000 +iN=mod +P=2000 +c=C +o=test
3. load68k sqrt +msM=loadmap +l=mathlib +i
4. load68k temp?.r +reo=combined.r
5. load68k t1.r t2.r +a=10 +A=100 +b=2M +l=testlib +do=test

The first example loads all files in the working directory whose names end with ".r". The linking-loader reads the file "/lib/ldr_environ" and processes the options therein. It uses the library "Clib" to resolve

external references. The executable output module, which is a shared-text module, is named "tester".

The second example loads the files specified and produces a binary file named "test". The internal module-name is "mod". The text segment begins at 20000 hexadecimal, and the data segment follows it at the next page boundary (page size 2000 hexadecimal). The source code is C. All global symbols are inserted in the symbol table of the binary file.

The third example loads the file "sqrt" and, by default, produces a binary file named "sqrt.o". The linking-loader searches the library "mathlib" for unresolved external references. It produces load and module maps, as well as a symbol table, and writes them to the file "loadmap". All global symbols are added to the symbol table of the binary file.

The fourth example loads the files in the working directory whose names match the pattern "temp?.r" and produces a relocatable module named "combined.r". The linking-loader prints each occurrence of all unresolved external references rather than only the first occurrence of each. Because the 'r' option is specified, the linking-loader does not search any libraries.

The fifth example loads the files "t1.r" and "t2.r" and produces the binary file named "test". The minimum page allocation is set to 10; the maximum, to 100. The task size of the module is set to 2 Megabytes. The executable module cannot produce a core dump.

Chapter 8

Libraries

8.1 Introduction

A library is a specially organized collection of relocatable modules used by the linking-loader to resolve external references. The user can, with the 'l' option, specify the names of up to twelve libraries for the loader to search. The loader searches the libraries in the order in which the user specifies them on the command line. If the user specifies less than twelve libraries, the loader automatically searches the system library, "Syslib68k", after it finishes searching the libraries specified by the user.

When searching for a user-specified library whose name begins with a slash, '/', the linking-loader first looks for the library as specified. If it does not find it or if it is searching for either a user-specified library whose name does not begin with a slash or the system library, the linking-loader looks in the working directory. If it does not find the library there, it looks for the directory "lib" in the working directory. If found, the linking-loader attempts to find the library in that "lib" directory. If it is not found there, the loader makes a final attempt to find the library by looking in the directory "/lib". If the library is not in any of these directories, the loader issues an error message and aborts.

The loader first makes one pass through the newly created module, trying to resolve all unresolved external references (primary references). When an external reference is resolved from a module contained in a library, that module is loaded and is then considered a "user" module. Library modules can, therefore, reference other library modules. However, the addition of a library module may introduce new unresolved references. Such references are called secondary references. The loader tries to resolve secondary references in the same way it resolves primary references--that is, it processes, in order, the libraries specified by the user on the command line. It makes as many passes as are necessary to resolve all secondary references, beginning the search for each unresolved external reference in the first library. Thus, even if two modules with the same name exist in different libraries, the loader uses only the first occurrence of the module. The version of the module to use can be changed by changing the order in which the libraries are specified.

The search for an external reference can be summarized as follows:

1. Search the user's modules.
2. Search the libraries specified by the user.
3. If the user specifies less than twelve libraries to search, search the system library.

8.2 Library Generation

Library generation is accomplished by the "lib-gen68k" command. This command either creates a new library of relocatable or executable modules or updates an existing library. Each module in a library must have a name. The name is assigned to a module by either the "name" directive of the relocating assembler or the 'N' option of the linking-loader. The "lib-gen68k" command does not accept a module without a name. As it runs, "lib-gen68k" produces a report describing the action that it takes for each module in the library. The report includes the name of the module and the file from which it was read (the old library or one of the update files).

8.2.1 The Command Line

The syntax for the "lib-gen68k" command is as follows:

```
lib-gen68k o=<old_lib> n=<new_lib> [u=<update>] [<del_list>] [+al]
```

8.2.2 The Arguments

8.2.2.1 The 'n' argument

The 'n' argument specifies the name of a new library. If a file with this name already exists, "lib-gen68k" deletes it without warning before writing the new library. If the user does not specify a name for the new library, it defaults to the name of the old library. In such a case "lib-gen68k" puts the new library in a scratch file, deletes the old library, and renames the scratch file with the name of the old library. The syntax for this argument is

n=<new_lib>

The 'n' argument, the 'o' argument, or both must appear on the command line.

8.2.2.2 The 'o' argument

The 'o' argument specifies the name of an existing library file which was previously created by the "lib-gen68k" command. If "lib-gen68k" is being called to create a new library, rather than to update an existing one, this argument is inappropriate. The syntax for this argument is

o=<old_lib>

The 'o' argument, the 'n' argument, or both must appear on the command line.

8.2.2.3 The 'u' argument

The 'u' argument specifies the name of a file containing modules to add to the library. If a module in the library has the same name as a module in an update file, the loader replaces the existing module with the one in the update file. The user may specify up to 255 update files by repeating the "u=<update>" argument for each one. However, when updating a large number of modules, it may be simpler to concatenate the modules by listing all of them with the "list" command and redirecting the output to a file. The user can then update all the modules with only one invocation of the 'u' option rather than having to invoke it for each module.

8.2.2.4 The deletion list

The argument <del_list> is a list of the names of modules to delete from the old library. By default, if "lib-gen68k" cannot find one of the files specified in the old library, it issues a warning message and continues operation.

8.2.3 Options Available

By default, "lib-gen68k" produces a report which includes the name of each module and the file from which it was read (the old library or one of the update files). The options are used to shorten or to eliminate this report.

8.2.3.1 The 'a' option

If the user specifies the 'a' option, the report contains information only about modules that were replaced, added, or deleted.

8.2.3.2 The '1' option

The '1' option suppresses the production of any report.

8.2.4 Examples

The following examples illustrate some uses of the "lib-gen68k" command.

1. lib-gen68k n=binlib u=one u=two u=three
2. lib-gen68k o=binlib u=new +a
3. lib-gen68k o=binlib u=newmods n=newlib transpose add +1

The first example creates a new library named "binlib" which contains all the modules from the files "one", "two", and "three".

The second example updates the library "binlib" by adding or replacing modules from the file "new". The command produces an abbreviated report.

The third example updates the library "binlib" by adding or replacing modules from the file "newmods" and by deleting the modules named "transpose" and "add". The updated library is written to the file "newlib". The old library is deleted.

Chapter 9

Segmentation and Address Assignment

9.1 Introduction

If the user specifies the 'r' option to the "load68k" command, the linking-loader produces a relocatable module; otherwise, it produces an executable module. An executable module can be one of two types: shared-text or non-shared-text. The only difference between the two types of executable modules is the presence of a bit in the binary header of a shared-text module which tells the operating system that it need load only one copy of the text portion of the program into memory, no matter how may users wish to access the program simultaneously. Some overhead expense is involved in the use of a shared-text program. Therefore, only those programs which are large and are used by more than one user at a time should be shared-text.

Basically, the linking-loader produces all three kinds of modules in the same way.

9.2 Segmentation

Any module consists of a combination of the following parts: a binary header, a text segment, a data segment, relocation information, symbol table, an information field, and a name. All of these components except the binary header are optional. The relocation information, of course, only appears in a relocatable file. Note that the module itself does not contain the bss segment; rather the binary header contains a number which tells the operating system how much memory to reserve for bss when it loads the module.

9.2.1 Combination of Segments for a New Module

When the linking-loader combines modules to form a new module, it uses the information in all the modules it combines to write a binary header for the new module. It then concatenates the information from each similar segment in the input modules and places it in the new module. For instance, it concatenates the text segments of all component modules to form the text segment of the new module, concatenates the data

segments to form the new data segment, and so forth. Concatenation occurs in the order in which the user specifies the modules on the command line. Material culled from any libraries is appended to the material from the modules specified by the user. For instance, the text segment of a library is appended to the text segment produced by combining the information from the user-specified modules.

The linking-loader treats common blocks within the modules it is combining slightly differently depending or whether the output is a relocatable or an executable file. In either case, because common blocks are a part of the bss, the linking-loader adjusts the number in the binary header which defines the size of the bss segment so that it includes the common blocks. If the module being created is an executable module, the linking-loader combines all common blocks of the same name and allocates enough space for the largest one. If the module is relocatable, the linking-loader does not combine common blocks.

9.2.2 End-of-segment Addresses

The linking-loader defines three global symbols--ETEXT, EDATA, and END--which mark the ends of the text, data, and bss segments. These constants, which behave like user-defined global symbols, always appear in the listing of the global symbol table. They may be used like any user-defined global symbol. Because these symbols are predefined, the user should not define any global symbols with the same names.

9.2.3 Load and Module Maps

If the user specifies the 'm' option to the "load68k" command, the linking-loader writes both a module map and a load map to standard output.

A load map contains information about the type of module being produced as well as the starting addresses of the text and data segments, the initial stack size, the page size (granularity), and the binary transfer address. (The binary transfer address is the address at which the operating system is to start execution of the program. A transfer address is created with the "end" directive of the relocating assembler. Only one of the modules being included in a program may contain such an address. If more than one module contains a transfer address, the linking-loader reports the error and aborts.) The load map also tells how many modules the linking-loader combined to produce the new module.

A module map contains the name of each module loaded, the name of the file containing each module, and the starting addresses of each segment (text, data, and bss) of each module. It also contains the final address of each segment of the new module.

For example, the following maps were produced by the linking-loader from a small "C" program.

```
* LOAD MAP *

Produced - executable, not overlapped TEXT and DATA.
Module is not shared text.
Starting TEXT address = 000000
Starting DATA address = 400000
Initial stack size = 000000
Granularity = 000000
Binary transfer address = 000B38
Number of input modules = 5

* MODULE MAP *

TEXT    DATA    BSS       MODULE NAME       FILE NAME
000000  400000  400204    test              test.r
000050  40000C  400204    Long Mul/Div      /lib/Clib
00032A  40000C  400204    C System Calls    /lib/Clib
000B04  4000A0  400204    strlen            /lib/Clib
000B38  4000A0  400204    C Wrapper         /lib/Clib

000B52  400204  400604    * Final Segment Addresses *
```

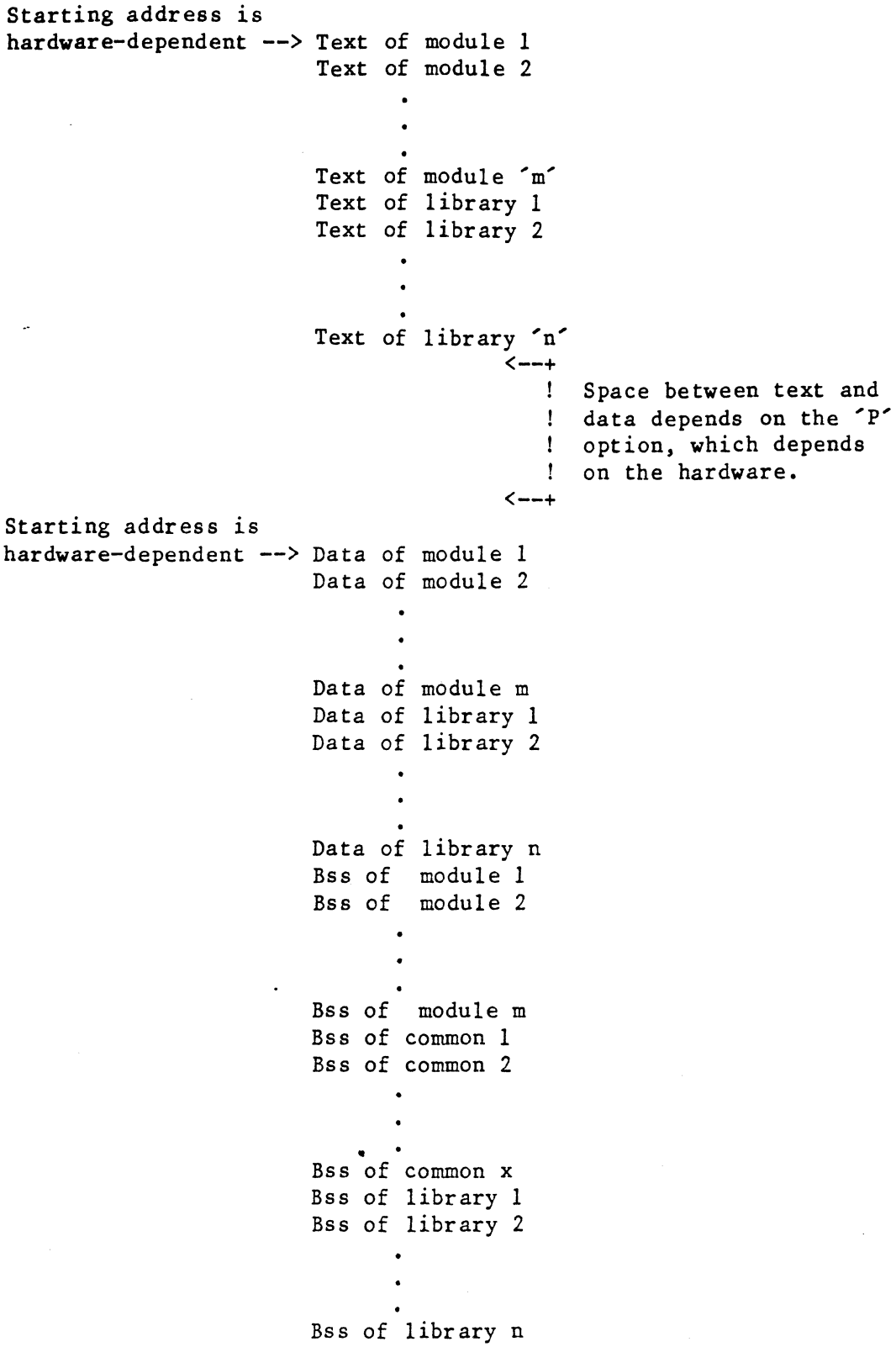
As explained earlier, the text segments from each of the modules are combined and relocated to form the text segment of the final executable module. The starting address of the text segment is 0; the starting address of the data segment is 400000. The data in the module map show that all modules have text segments, that the "Long Mul/Div" and the "strlen" modules have no data segments, and that only the "C Wrapper" module has a bss segment. Note that the bss segment immediately follows the data segment. The linking-loader found the routines called by the module "test" in the library "/lib/Clib". The binary transfer address is located in the module "C Wrapper" (address $000B38).

The following map was produced using the same file as the previous map. However, because the user did not specify a starting address for the data segment, the data segment follows the text as closely as possible. The distance between the text and data segments is determined by the page size (granularity), which was specified with the 'P' option as hexadecimal 1000. Therefore, the data segment starts at the first page boundary following the text segment. The output module is a shared-text module.

```
* LOAD MAP *

Produced - executable, not overlapped TEXT and DATA.
Module is shared text.
Starting TEXT address = 000000
Starting DATA address = 1000
Initial stack size = 000000
Granularity = 001000
Binary transfer address = 000B38
Number of input modules = 5

* MODULE MAP *

TEXT    DATA    BSS       MODULE NAME       FILE NAME
000000  001000  001204    test              test.r
000050  00100C  001204    Long Mul/Div R    /lib/Clib
00032A  00100C  001204    C System Calls    /lib/Clib
000B04  0010A0  001204    strlen            /lib/Clib
000B38  0010A0  001204    C Wrapper         /lib/Clib

000B52  001204  001604    * Final Segment Addresses *
```

9.3 Address Assignment

When the operating system loads a module produced by the linking-loader, it puts only the text, data, and bss segments into memory. The following map illustrates how a module resulting from the loading of 'm' modules, 'n' libraries, and 'x' common blocks would be loaded into memory by the operating system.


Starting address is
hardware-dependent --> Text of module 1
Text of module 2
Text of module 'm'
Text of library 1
Text of library 2
Text of library 'n'
Space between text and data depends on the 'P' option, which depends on the hardware.
Starting address is
hardware-dependent --> Data of module 1
Data of module 2
Data of module m
Data of library 1
Data of library 2
Data of library n
Bss of module 1
Bss of module 2
Bss of module m
Bss of common 1
Bss of common 2
Bss of common x
Bss of library 1
Bss of library 2
Bss of library n

Chapter 10

Error Messages from the Linking-Loader

10.1 Introduction

The linking-loader produces two types of error messages: nonfatal and fatal. It sends all error messages to standard error. If the linking-loader encounters a nonfatal error, it prints the appropriate message and continues execution. Nonfatal errors include warning messages. If it encounters a fatal error, it aborts after printing a message of the following form:

```
Fatal Error: <description_of_error>
Loader aborted!
```

10.2 Nonfatal Errors

Address overflow at $<addr> in <type> segment of module "<mod_name>".
The relocation or linking of the two-byte address in the field at the specified address in <mod_name> resulted in a number which does not fit in the field. A two-byte address must be a positive, 16-bit expression. The assembler generates such an address if and only if the programmer forces it into absolute word-addressing mode. This message indicates that the programmer forced an address into a word when the address needed more than 16 bits.

Maximum page allocation set to <num>.
The minimum and maximum page allocations specified by the user conflict. The linking-loader has adjusted the maximum as indicated.

Maximum task size set to <num>.
The maximum task size specified by the user (or by the default) is not large enough for the code being loaded. The linking-loader has changed the maximum task size as indicated.

Overflow at $<addr> in <type> segment of module "<mod_name>".
The relocation or linking of the field at the specified address in <mod_name> resulted in a number which does not fit in the field. This situation is not always an error. If the relocation or linking results in a change of the sign of the number, this message may be produced even though the result of the procedure is correct and does fit in the field provided. The user must carefully inspect the code being loaded to determine whether or not the message should be ignored.

Symbol name clash: "<symbol_name>" in module "<mod_name>".
The specified symbol has been globally declared in more than one module. The module specified is the module containing the second (or later) declaration of the symbol. The name of the symbol must be changed in one of the modules, and that module must be reassembled.

"<symbol_name>" unresolved in module "<mod_name>".
The specified symbol was referenced in <mod_name>, but the linking-loader could not locate it in any of the modules supplied by the user or in any of the libraries. This message may be expected if the linking-loader is producing a relocatable file. If an executable file is being produced, it is an error.

Warning: "/lib/std_env" not found.
The file "/lib/std_env" is supplied with every UniFLEX system. It is a file of options which is specific to the hardware. If the file has not been deliberately deleted or renamed, contact the vendor.

10.3 Fatal Errors

Bad library format for "<library_name>"!
The library specified does not have the correct format for a library created by the "lib-gen68k" utility.

BSS instruction segment!
This message, which is a check for internal consistency, should not appear. If it does, contact the vendor.

BSS transfer address!
This message, which is a check for internal consistency, should not appear. If it does, contact the vendor.

"<file_name>" contains MC68020- and MC68881-specific instructions.
Incompatible module types--no output file produced.
The specified relocatable module contains instructions specific to the MC68020 microprocessor and the MC68881 coprocessor, but the linking-loader will not load this type of module either because the machine in use does not contain the MC68020 and the MC68881 or because the user specified the 'W' and the 'Y' options. If the machine in use contains neither an MC68020 microprocessor nor an MC68881 coprocessor, the user may load such modules by specifying the 'w' and the 'y' options.

"<file_name>" contains MC68020-specific instructions.
Incompatible module types--no output file produced.
The specified relocatable module contains instructions specific to the MC68020 microprocessor, but the linking-loader will not load

this type of module either because the machine in use does not contain an MC68020 or because the user specified the 'W' option. If the machine in use does not contain an MC68020 microprocessor, the user may load such modules by specifying the 'w' option.

"<file_name>" contains MC68881-specific instructions.
Incompatible module types--no output file produced.
The specified relocatable module contains instructions specific to the MC68881 coprocessor, but the linking-loader will not load this type of module either because the machine in use does not contain an MC68881 or because the user specified the 'Y' option. If the machine in use does not contain an MC68881 coprocessor, the user may load such modules by specifying the 'y' option.

Illegal configuration specified!
The argument to the 'C' option is not a valid UniFLEX configuration.

Illegal input file "<file_name>"!
The specified file is not a relocatable file.

Illegal maximum page allocation!
The number used as an argument to the 'A' option must be between 0 an 32767 inclusive.

Illegal minimum page allocation!
The number used as an argument to the 'a' option must be between 0 an 32767 inclusive.

Illegal relocation!
This message, which is an check for internal consistency, should not appear. If it does, contact the vendor.

Illegal task size!
The argument to the 'b' option is not a valid task size. Valid arguments are "128K", "256K", "512K", "1M", "2M", "4M", "8M", "16M", "32M", "64M", "128M", "256M", "512M", "1G", "2G", "4G", "S", "M", and "L". All letters may be either upper- or lowercase.

Illegal trap vector!
The trap vector specified on the command line is illegal. Trap vectors must have one of two forms: "TRAPn", where 'n' is a number between 0 and 15 inclusive, or a 4-digit hexadecimal number representing an instruction.

Incompatible options: 'D' and 'r' or 'R'
The 'D' option, which specifies the starting address of the data segment, may not be used with either the 'r' or 'R' option, both of which tell the linking-loader to produce a relocatable file.

Incompatible options: 'f' and 'Z'
The 'f' and 'Z' options cannot be specified simultaneously.

Incompatible options: 'r' and 'R'
The 'r' option, which tells the linking-loader to produce a relocatable file without searching any libraries for external references, may not be used with the 'R' option which tells the linking-loader to produce a relocatable file and to search "/lib/syslib.68k" and any libraries specified by the user for external references.

Incompatible options: 'T' and 'r' or 'R'
The 'T' option, which specifies the starting address of the text segment, may not be used with the 'r' or 'R' option, both of which tell the linking-loader to produce a relocatable file.

Incompatible options: 'w' and 'W'
The 'w' option, which tells the linking-loader to load modules containing MC68020-specific instructions, may not be used with the 'W' option, which tells the linking-loader not to load such modules.

Incompatible options: 'x' and 'r' or 'R'
The 'x' option, which forces the linking-loader to produce an executable file, may not be used with the 'r' or 'R' option, both of which tell the linking-loader to produce a relocatable file.

Incompatible options: 'y' and 'Y'
The 'y' option, which tells the linking-loader to load modules containing MC68881-specific instructions, may not be used with the 'Y' option, which tells the linking-loader not to load such modules.

Incremental load file must be executable.
The file specified as an argument to the 'x' option must be an executable file.

Invalid option: '<char>'!
The character specified is not a valid option to the "load68k" command.

Library "<library_name>" not found!
The linking-loader could not locate the specified library in the working directory, in the directory called "lib" in the working directory, or in the directory "/lib".

Multiple transfer addresses!
Only one module may contain a binary transfer address. The user specified two or more modules which contain a transfer address.

Multiple 'x' options specified!
The 'x' option may be used only once each time the linking-loader is invoked.

Nested ´F´ options!
The ´F´ option may not be used inside a file of options. It may, however, be used repeatedly on the command line.

No files given!
The user specified no files on the command line.

Opening "<file_name>": <reason>
The operating system returned an error when the linking-loader tried to open the specified file. This message is followed by an interpretation of the error returned by the operating system.

Reading "<file_name>": <reason>
The operating system returned an error when the linking-loader tried to read the specified file. This message is followed by an interpretation of the error returned by the operating system.

Seeking to <location> in "<file_name>": <reason>
The operating system returned an error when the linking-loader tried to seek to the specified location in <file_name>. This message is followed by an interpretation of the error returned by the operating system.

Too many libraries!
A maximum of twelve libraries may be specified on the command line.

Unknown source type!
The argument to the ´c´ option is not valid. The linking-loader only recognizes "FORTRAN", "C", "PASCAL", "COBOL", and "ASSEMBLER".

Writing to "<file_name>": <reason>
The operating system returned an error when the linking-loader tried to write to the specified file. This message is followed by an interpretation of the error returned by the operating system.

Appendix A

Syntax Conventions

The following conventions are used in syntax statements throughout this manual.

1. Items that are not enclosed in angle brackets, '<' and '>', or square brackets, '[' and ']', are "keywords" and should be typed as shown.

2. Angle brackets, '<' and '>', enclose descriptions which the user must replace with a specific argument. For example, in the line

   o=<file_name>

   the name of a file must be specified in the place indicated by <file_name>.

3. Square brackets, '[' and ']', indicate optional items. These items may be omitted if their effect is not desired. In the section on addressing modes, square brackets are not used to delineate optional items because certain addressing modes use square brackets as part of their required syntax (see Section 4.3.3).

4. The underscore character, '_', is used to link separate words that describe one term, such as "file" and "name".

5. Characters other than spaces that are not enclosed in angle brackets or square brackets must appear as they appear in the syntax statement.

6. If the word "list" appears as part of a term in a syntax statement, that term consists of one or more of the elements described by the rest of the term, separated by commas or spaces. For example, the term

   param_list

   represents a list of command-line parameters separated by spaces.

7. The "rel68k", "rel20", and "load68k" commands support several optional features, known as options, which alter the effect of the command.

Options consist of either a single character or a single character followed by an equals sign, '=', followed by an argument. An "option string" is a plus sign followed by one or more options. An option string may contain any number of single-character options but only one option which takes an argument. An option requiring an argument must be the last option in an option string. Thus, the command line must contain a separate option string for each option requiring an argument. It may or may not contain a separate option string for each single-character option.

The following commands contain valid option strings:

```
rel68k <file_name> +blns
rel68k <file_name> +b +l +n +s
load68k <file_name_1> +msM=<file_name_2>
load68k <file_name_1> +msM=<file_name_2> +l=mathlib +i
```

The following commands contain invalid option strings:

```
load68k <file_name_1> +mM=<file_name_2>s
load68k <file_name_1> +msM=<file_name_2>l=<file_name_3>
```

Option strings may appear anywhere on the command line after the "rel68k" or "load68k" command.

Many common terms appear (often as abbreviations) in more than one syntax statement. The manual does not explain these terms each time they appear; however, the following table describes each one.

Table A-1. Common Terms Used in Syntax Statements

Term	Meaning
char	Character
file_name	A valid file name
hex_num	Hexadecimal number
list	Term is a list of elements
num	Number

Appendix B

Syntax for 68020 Addressing Modes

B.1 Introduction

This appendix lists the syntaxes recommended by Motorola (Motorola, 1985a) for the addressing modes available for the 68020 microprocessor and other acceptable syntaxes. The string ZPC may be used anywhere in place of the string PC; SP may replace A7 anywhere.

B.2 Syntaxes Recommended by Motorola

The following syntaxes are accepted by the "rel20" command and follow the guidelines for recommended syntax in Appendix D of Motorola's manual for the 68020 (Motorola, 1985a).

```
#data
Rn
disp
(Rn)
(An)+
-(An)
(disp,An)
(disp,An,Rn)
([disp])
([Rn])
([An,Rn])
([disp,Rn])
([disp,An,Rn])
([disp],Rn)
([disp,An],Rn)
([An],Rn)
([disp],disp)
([Rn],disp)
([An,Rn],disp)
([disp,Rn],disp)
([disp,An,Rn],disp)
([disp],Rn,disp)
([disp,An],Rn,disp)
([An],Rn,disp)
(disp,PC)
(disp,PC,Rn)
```

```
([disp,PC])
([disp,PC,Rn])
([disp,PC],Rn)
([disp,PC],disp)
([disp,PC,Rn],disp)
([disp,PC],Rn,disp)
```

B.3 Other Acceptable Syntaxes

Also accepted are the following forms:

```
[Rn]
[disp]
disp(An)
disp(An,Rn)
[disp,Rn]
[disp,PC]
disp(PC)
disp(PC,Rn)
[disp,An,Rn]
[disp,PC,Rn]
[An,Rn]
[PC,Rn]
```

B.4 Elliptical Syntax Statements

In addition to the syntaxes described in Sections B-1 and B-2, the assembler accepts the full form of the addressing mode with any or all parts missing. For example, the following are valid addressing modes:

```
([,,],)
([,],)
(,,)
```

All of these forms generate effective addresses of 0.

References

Motorola. 1984. M68000 16/32-Bit Microprocessor Programmer's Reference Manual. 4th ed. Englewood Cliffs: Prentice Hall.

-----. 1985a. MC68020 32-Bit Microprocessor User's Manual. 2nd ed. Englewood Cliffs: Prentice Hall.

-----. 1985b. MC68881 Floating-Point Coprocessor User's Manual. Austin: Motorola.