EY-0061E-SG-0101

Programming RSX-11M in FORTRAN

Volume I

digital

EY-0061E-SG-0101

Programming RSX-11M in FORTRAN

Student Workbook
Volume I

Prepared by Educational Services
of
Digital Equipment Corporation

CONTENTS

Volume I

SG STUDENT GUIDE

1 USING SYSTEM SERVICES

4 USING DIRECTIVES FOR INTERTASK COMMUNICATION

5 MEMORY MANAGEMENT CONCEPTS

Volume II

6 OVERLAYING TECHNIQUES

7 STATIC REGIONS

FIGURES

TABLES

EXAMPLES

STUDENT GUIDE

INTRODUCTION

Programming RSX-11M in FORTRAN is intended for FORTRAN programmers who use services of the RSX-11M operating system beyond those provided by the FORTRAN programming language itself. This course describes the various services and how to use them from a task which you write.

This course is self-paced, which means that you learn at whatever rate is comfortable for you.

Instead of a teacher, you have a course administrator and a subject matter expert. In some cases, the same person can perform both functions. The course administrator manages the mechanics of the course and makes sure you have easy access to the system and the on-line course materials. As you finish modules, s/he records your progress. The subject matter expert helps you if you have a technical question. Before you consult the expert, however, read the course materials and references in an effort to answer the question yourself.

This Student Guide covers the following topics:

- Course prerequisites
- Course goals (and nongoals)
- Course organization
- Course map description
- Course resources
- How to take the course
- Personal Progress Plotter

PREREQUISITES

To be prepared for this course, you must have taken the following DIGITAL courses, or you must have equivalent experience.

1. RSX-11M Utilities and Commands. Specifically, you must be able to logon/logoff, edit files, and develop/run/debug programs under RSX-11M.

2. Programming in FORTRAN.

COURSE GOALS AND NONGOALS

On completion of this course, you should be able to write tasks which:

1. Use executive directives

2. Perform intertask communication and coordination

3. Perform synchronous and asynchronous I/O operations

4. Use overlays

5. Use memory management facilities to communicate between tasks and make more effective use of available memory

This course does not teach the following:

1. The FORTRAN programming language

2. The Digital Command Language (DCL) or Monitor Console Routine (MCR)

3. The program development cycle.

COURSE ORGANIZATION

This course is self-paced for independent study. The course material is structured in modules. Each module is a lesson on one or more skills required to fulfill the course goals. A module consists of:

- An introduction to the subject matter of the module
- A list of objectives, which describe what you should achieve by studying the module
- A list of resources that provide reference materials and additional reading for the module
- The module text, including explanatory text, figures, tables, examples, and references to readings in the manuals
- Learning activities (for some modules), consisting of reading assignments or written exercises which are essential to your learning the material
- Written and/or lab tests/exercises (bound separately) which you can use to measure your achievement. Solutions are provided for all exercises.

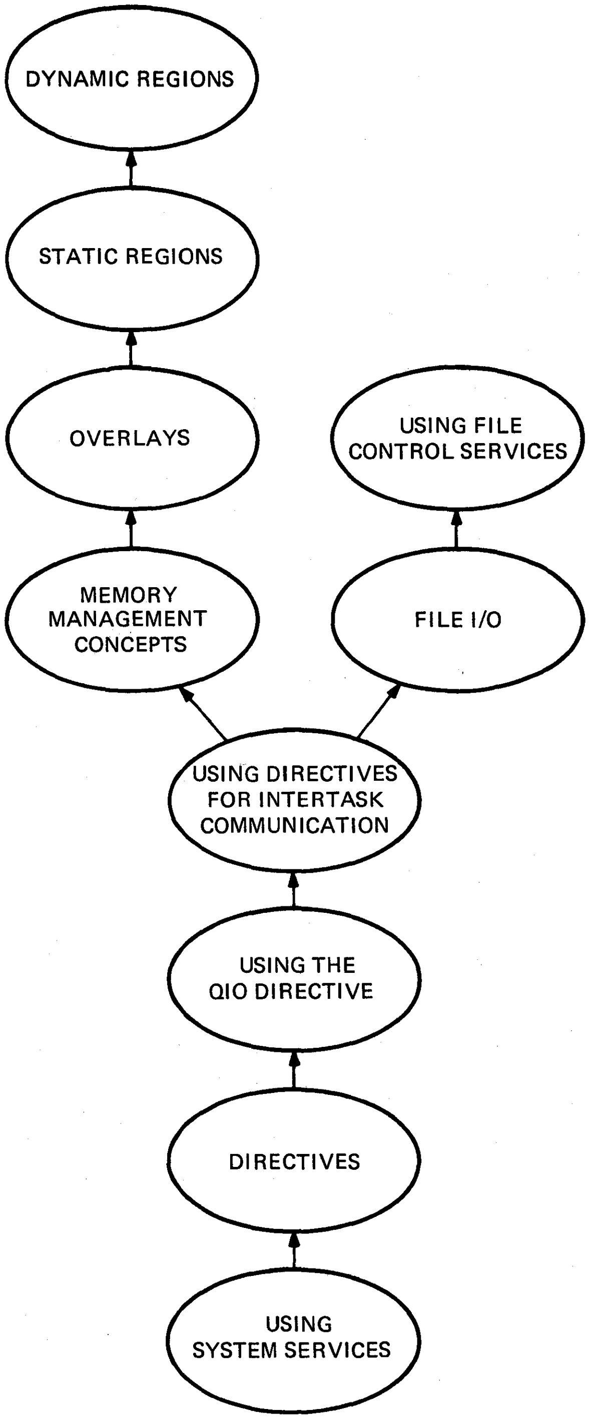
The course is bound in three volumes. The first two volumes contain this student guide, the 1Ø modules (except for their tests/exercises), and the appendices. The third volume contains the tests/exercises for each module.

COURSE MAP DESCRIPTION

The course map shows how each module relates to the other modules and to the course as a whole. Before beginning a specific module, it is recommended that you first complete all modules with arrows leading into that module. These prerequisite modules present material necessary to understanding the module you are about to study.

If you have no preference, study the modules in numerical order, 1 through 1Ø.

COURSE MAP



TK-7749

COURSE RESOURCES

Required References

1. RSX-11M/M-PLUS Executive Reference Manual (AA-L675A-TC)
2. RSX-11M/M-PLUS I/O Drivers Reference Manual (AA-L677A-TC)
3. RSX-11M/M-PLUS Task Builder Manual (AA-L680A-TC)

Optional References

1. PDP-11 Processor Handbook (EB-19402-20/81)
2. FORTRAN IV User's Guide
3. FORTRAN IV-PLUS User's Guide
4. FORTRAN 77 User's Guide

HOW TO TAKE THE COURSE

Because this is a self-paced course, you determine how much time to devote to each subject. You can pass quickly over familiar topics. You can spend more time on topics which are of interest to you, or which you can use often in your job, and less time on topics which have little use in your job.

Each time you are ready to begin a new module, first read the introduction and the objectives. If you feel that you already understand the material in the module, you can go immediately to the tests/exercises for that module. If you don't understand much of the material, read the module. If you understand some of the concepts but not others, just look over the program examples for the concepts you understand. Read the text and study the examples for concepts you don't understand. The text explains new concepts and refers you to related readings in the manuals. The program examples provide working examples which show you how to apply the concepts.

Some of the readings in the manuals are required and others are optional. Required readings are contained in learning activities and are indented to set them apart from the module text. These readings are required because they cover material not otherwise covered in this course. The optional readings are mentioned within the module text and are designed to help you in two ways. First, they teach you more about a given topic. Second, they offer another explanation in case you have trouble understanding the explanation in this course.

In addition, you will need the manuals to look up the specifics involved in invoking the various services. This is especially true for the executive directives.

Keep the module objectives in mind. If a skill is listed as an objective, be sure to master it. Later modules may depend on this skill.

The module text contains many example programs to show you how to use the skills you are learning. All of the example programs in this book should be available on-line. The standard location for these files is UFD [2Ø2,1] on your system disk. Check your system and if the files are not located there, check with your course administrator to find out where they are located.

Do not modify the files in UFD [202,1] or in their original location. Instead, copy the files you plan to use to your own UFD and use them there. In that way, the original files in UFD [202,1] will remain intact for other students.

Each example program contains the following:

- Source code
- A sample run session
- Bulleted items which are described in the text.

The source code contains the name of the file which contains the code on-line. Following this is a brief description, telling what the example does. Any special compile and task-build instructions, and any special install and run instructions follow this. Only special, nonstandard instructions are included. The code itself includes line comments plus some additional comments.

The sample run session shows what happens during a typical run of the task. Any special install and run instructions are shown in the run session.

The bulleted items match the example notes in the text, which describe the code in more detail. Study the examples and the notes that describe them carefully.

In the module on Using File Control Services, many of the examples create output files. A dump of any created file follows the run session. The file dumps were created using the DMP utility.

If the examples are available on-line, compile and task-build them, and then run them. This will help you to understand the examples better. Many of the tests/exercises ask you to make minor changes to existing examples, and then run them again. Do the tests/exercises for a module in the Tests/Exercises book only after you have done all of the reading and have run the example programs. If you prefer, you can do them as soon as you cover the necessary material in the module.

The same Tests/Exercises book is used in this course and the Programming RSX-11M in MACRO course. Do all tests/exercises except those which specifically say "in MACRO". All exercises have solutions in the Tests/Exercises book. In addition, any solutions involving programs should be available on-line, in UFD [202,2]. Compare these solutions to your own.

If you have mastered the module objectives, ask your course administrator to record your progress on your Personal Progress Plotter. You will then be ready to begin a new module. If you haven't yet mastered the module objectives, return to the module text for further study.

With a self-paced course, it is impossible to give a schedule that applies to all students. The amount of time that students spend on a module depends on both their experience and their interest in the topics in that module. Use Table 1 as a guide when you set your schedule.

In addition to the 10 modules, the Student Workbook contains several appendices. These are:

Appendix A - Glossary

Appendix B - Conversion Tables. This appendix contains a table for converting between decimal and octal, and among words, bytes, and memory blocks. It also contains a table for converting from active page registers (APRs) to virtual addresses.

Appendix C - FORTRAN/MACRO-11 Interface. This appendix contains an explanation of the techniques which you should use to write a FORTRAN callable subroutine in MACRO. It also explains how to call such a subroutine from FORTRAN.

Appendix D - Privileged Tasks. This appendix contains a description of the various types of privileged tasks supported under RSX-11M, and how to create them.

Appendix E - Task Builder Use of Psect Attributes. This appendix contains a description of the effect of Psect attributes on how the Task Builder collects together scattered occurrences of program sections.

Appendix F - Additional Shared Region Topics. This appendix contains several additional shared region topics. They are: overlaid shared regions, referencing multiple regions from a single task, interlibrary calls, and cluster libraries.

Appendix G - Additional Example. This appendix contains the source code for any program examples which are required for the Tests/Exercises but are not included elsewhere in the Student Workbook. These examples should also be available on-line, under UFD [202,1]. They are included here in case they are not available on-line on your system.

Appendix H - Learning Activity Answer Sheet. This appendix contains the solutions to any Learning Activity questions in this course. After you do a Learning Activity, check your answers against those provided.

Table SG-1 Typical Course Schedules

Module	More Experienced Student	Less Experienced Student
1. Using System Services	2.Ø hours	3.Ø hours
2. Directives	5.Ø hours	7.5 hours
3. Using the QIO Directive	4.Ø hours	6.Ø hours
4. Using Directives for Intertask Communication	5.Ø hours	7.5 hours
5. Memory Management Concepts	2.Ø hours	3.Ø hours
6. Overlays	5.Ø hours	7.5 hours
7. Static Regions	4.5 hours	7.Ø hours
8. Dynamic Regions	4.5 hours	7.Ø hours
9. File I/O	2.Ø hours	3.Ø hours
1Ø. File Control Services	6.Ø hours	9.Ø hours
Totals	4Ø.Ø hours of study and lab	6Ø.5 hours of study and lab

PERSONAL PROGRESS PLOTTER

MODULE	DATE STARTED	DATE COMPLETED	TIME SPENT	SIGN-OFF INITIAL
1. USING SYSTEM SERVICES				
2. DIRECTIVES				
3. USING THE QIO DIRECTIVE				
4. USING DIRECTIVES FOR INTERTASK COMMUNICATION				
5. MEMORY MANAGEMENT CONCEPTS				
6. OVERLAYS				
7. STATIC REGIONS				
8. DYNAMIC REGIONS				
9. FILE I/O				
10. FILE CONTROL SERVICES				

USING SYSTEM SERVICES

INTRODUCTION

RSX-11M provides system services which perform many operations commonly needed by user-written application programs. Use of these services can:

- Improve the efficiency of your tasks by reducing the size and execution time
- Decrease the time it takes to code and debug your tasks
- Increase the reliability of your task
- Provide you with controlled access to system features
- Improve the overall performance of your system

This module discusses what services exist and how they are called from a task.

OBJECTIVES

1. Identify the facilities provided through system services.
2. List the various system libraries and the facilities they provide.

RESOURCES

1. RSX-11M/M-PLUS Executive Reference Manual, Chapter 1
2. FORTRAN IV User's Guide, Appendix B
3. FORTRAN IV-PLUS User's Guide, Appendix D
4. FORTRAN-77 User's Guide, Appendix D

WHAT IS A SYSTEM SERVICE?

An RSX-11M system service is a function or service performed for a running task during the task's execution. The software which provides the service is either in the Executive or in other system supplied code.

WHY SHOULD YOU USE SYSTEM SERVICES?

To Extend the Features of Your Programming Language

System services offer you additional features not inherently part of your programming language. Examples of this are:

- Accessing shared resources in a properly synchronized way
- Coordinating multiple tasks
- Controlling memory allocation and mapping
- Interacting with the Executive

To Ease Programming and Maintenance

DIGITAL provides the code to perform these services, hence less time is needed for the user to develop working programs. The supplied code has a well defined modular structure which eases user design for his programs.

The code for system services is well debugged. This makes it easier to debug and maintain programs, since there are fewer potential points of failure and only the user written code needs to be debugged. When maintenance is required in the code for the supplied system services, patches are released by DIGITAL with clear-cut installation procedures.

To Increase Performance

The supplied code to perform system services is generally written in MACRO-11 which assures minimum execution time. It is often possible to share the code among several different tasks, with minimal additional overhead. This can result in any or all of the following performance gains:

- Increase in your task's throughput
- Increase in your system's throughput
- Increase in memory usage efficiency on your system
- Decrease in your task's size
- Increase in available space on mass storage volumes

WHAT SERVICES ARE PROVIDED?

The system services can be divided into a number of classes. For each, a few examples are mentioned to give you a feeling for the kinds of services available.

Note that a number of the services provided to tasks parallel those provided to operators through DCL commands.

System and Task Information

You can obtain information from the system. For example, you can:

- Obtain information about your task
  - its priority
  - its logical unit (LUN) assignments
- Obtain information about a partition on the system
  - its base address
  - its length
- Obtain the current time and date

Task Control

You can start up and stop tasks, and alter task states. For example, you can:

- Request another task to run
- Abort a task
- Suspend or resume a task
- Alter the running priority of an active task

Task Communication and Coordination

You can create a set of tasks that communicate with one another and coordinate the interaction of the tasks. For example, you can:

- Send data from one task to another

- Have one task notify other tasks that an event has occurred (e.g., that a job has been completed)

- Have one task pass a command to another task and have it obtain an indication from the other task about the status of the execution of the command.

I/O to Peripheral Devices

You can interact with peripheral devices on your system. For example, you can:

- Perform special I/O functions which cannot be accomplished by FORTRAN READ or WRITE statements such as reading from a terminal with the NOECHO feature invoked.

- Attach a device for exclusive use by a task

- Read or set variable characteristics of a device (e.g., for a terminal - baudrate or hold screen mode)

Memory Use

You can use system services to control the amount of memory your task uses or to permit several tasks to share an area of memory. For example, you can:

- Run a task in less memory than its total size, by using overlays to load pieces of the program at any one time
- Allocate space in memory for a temporary work buffer, and then return that space to the system when the task is finished using it
- Share a data area in memory among several tasks
- Share a single copy, in memory, of a commonly used subroutine, among several tasks

OTHER SERVICES AVAILABLE

You can use system services to perform often needed functions. For example, you can:

- Convert between Radix-5Ø and ASCII format
- Get the date in dd-mon-yr format or as three integers

These services are generally supplied as subroutines located in the system object library (LB:[1,1]SYSLIB.OLB).

HOW SERVICES ARE PROVIDED

When a system service is needed in a task, it is called via the CALL statement just as for any other subroutine. Services are provided using two different methods:

1. The Executive is invoked by the task to perform the service (an executive directive)

2. The code to perform the service is placed directly into the task

Executive Directives

Figure 1-1 shows how the first method works. The following steps are involved:

❶ The user task makes a service request and invokes the Executive

❷ The Executive takes control and performs the service

❸ The Executive returns control to the user task, at the statement following the service request.

Figure 1-2 shows a more complex version of the first method. In this case task A and task B use a system service to interact through the Executive.

Task A starts up and at some point needs task B to do some work; possibly a calculation. Task A sends the data to task B, requests task B to run, and then waits until task B sends back the answer. Task B starts running, performs the calculation, and then sends the answer back to task A. Task B also notifies task A that the job is finished. Task A then starts up again and uses the answer. The steps outlined above for method one would actually be used a number of times in this example.

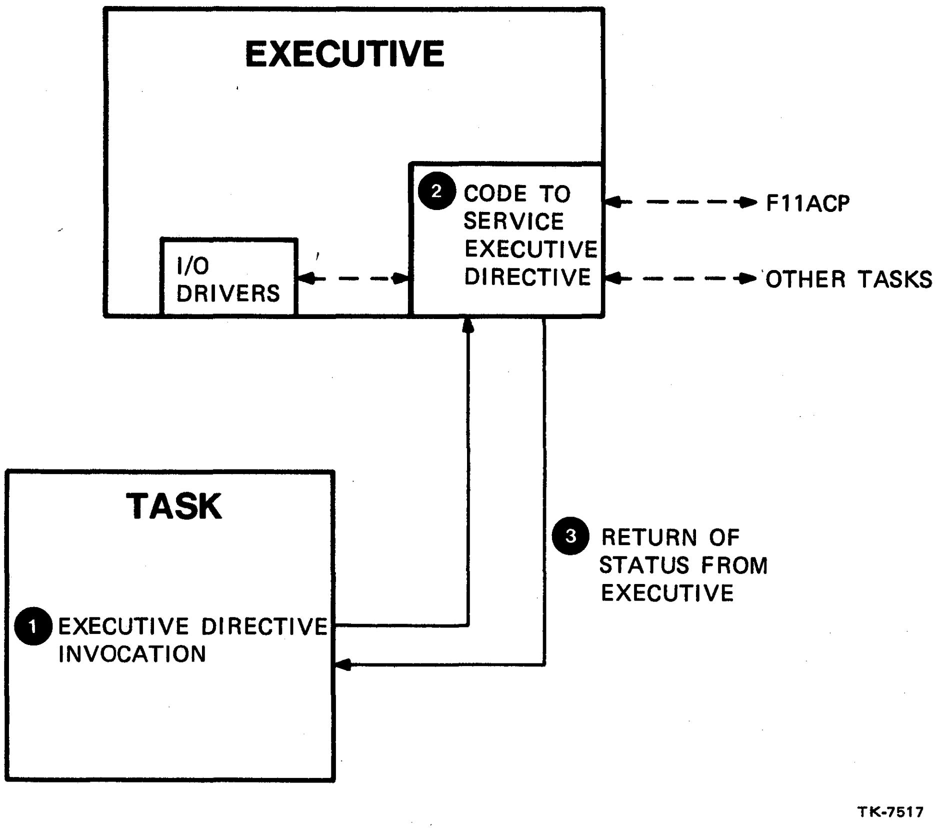

EXECUTIVE
I/O DRIVERS
2 CODE TO SERVICE EXECUTIVE DIRECTIVE
F11ACP
OTHER TASKS
TASK
1 EXECUTIVE DIRECTIVE INVOCATION
3 RETURN OF STATUS FROM EXECUTIVE
TK-7517

Figure 1-1 Using Executive Directives to Service a Task

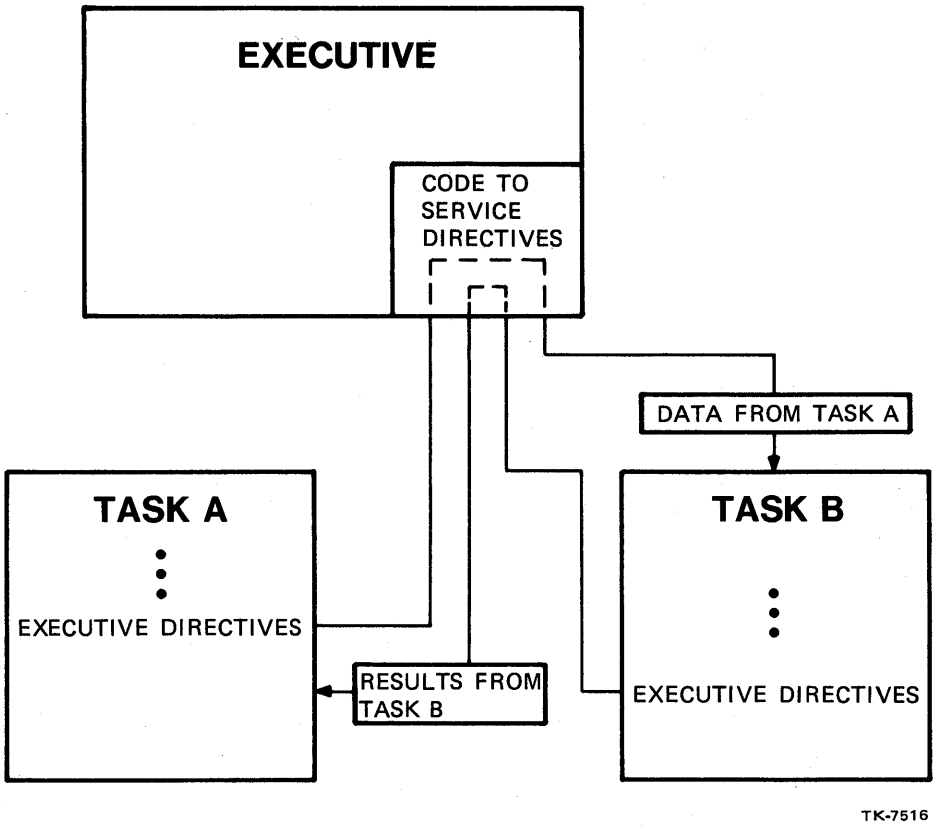


Figure 1-2 Using Executive Directives to Receive Services From Other Tasks

Code Inserted into Your Task Image

The second method of providing system services is illustrated in Figure 1-3. The code to perform the service is inserted directly into the user task. For system subroutines, the subroutine call results in a transfer of control to the subroutine code, located in another part of the user task.

Certain services must be provided by invoking the Executive. Any service which involves synchronization or access to shared resources must be coordinated by the Executive. For example, if a request activates another task, the Executive must enter the task in the active task list, which sets the task up to compete for memory space and then CPU time. It is much easier to have the Executive coordinate all the tasks, rather than require that each task check with every other task before using a shared resource. Also, any activity that involves communication or coordination among multiple tasks usually must be performed by the Executive.

Placing the code in the user task is appropriate for a service which is performed independently by a task. For example, if a task converts an ASCII decimal value which is input at a terminal to a Radix-5Ø value for internal use, there is no need for the Executive to coordinate that activity. It does not affect shared resources or other tasks.

If a service can be provided without need for the Executive, and that service is needed often by a number of different tasks, it is possible to share one copy of the code among several tasks. Using special techniques, often used subroutines can be collected and a single copy of each subroutine can be shared in memory among several tasks. The procedure for producing and using a shared collection of subroutines, called a resident library, is discussed in the Static Regions module of this course.

Some of the services covered in this course are provided by making special requests when you task-build your task. In some cases, the Task Builder transparently places code directly in your user task. In other cases, it sets your task up in a special way to provide the service. We will discuss the techniques for accessing services with the Task Builder in later modules.

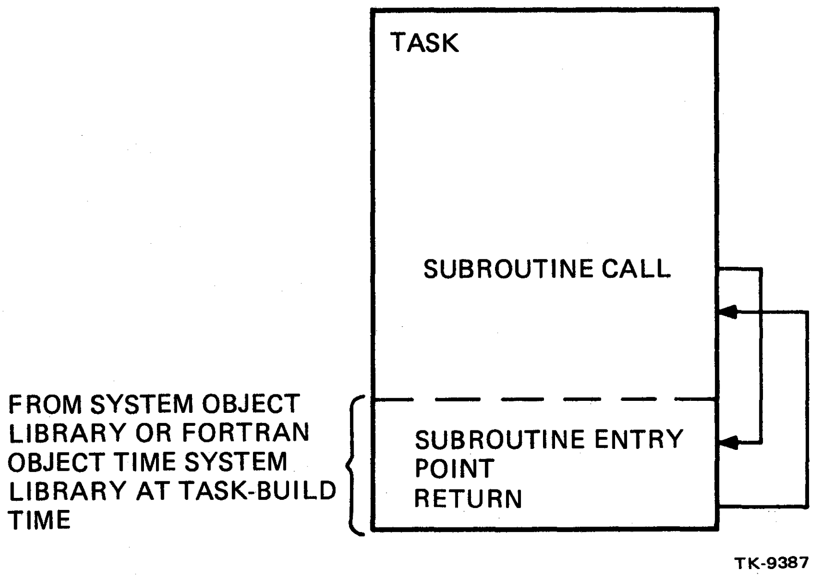

TASK
SUBROUTINE CALL
FROM SYSTEM OBJECT LIBRARY OR FORTRAN OBJECT TIME SYSTEM LIBRARY AT TASK-BUILD TIME
SUBROUTINE ENTRY POINT
RETURN
TK-9387

Figure 1-3 Code Inserted into Your Task Image

AVAILABLE FILE AND RECORD ACCESS SYSTEMS

There are two file and record access systems available under RSX-11M, File Control services (FCS) and Record Management Services (RMS). Both offer an interface between tasks and the Files-11 structure used to maintain disk directories and files.

FCS is the standard access system supplied with RSX-11M. Many of the utilities (e.g., PIP, EDT, and the Task Builder) use FCS for their file interface. RMS offers all of the FCS functionality plus additional capabilities not available with FCS, such as indexed files and more sophisticated file sharing.

While it is transparent to the FORTRAN user, all READ or WRITE statements ultimately result in calls to various FCS or RMS subroutines.

SYSTEM LIBRARIES

Table 1-1 contains a list of the libraries which are used during program development of a task using system services. They are usually located in LB:[1,1]. SYSLIB.OLB is the system object library searched by default by the Task Builder.

Table 1-1 Standard Libraries

Langages Using Library	Version of FORTRAN Using Library	Contents	Notes
SYSLIB.OLB	FORTRAN	Executive directive calls for FORTRAN	Default object library for Task Builder
		FCS subroutines	
		Other file access routines	
		Command retrieval and parsing routines	
		Assorted conversion routines, arithmetic routines, memory management routines	
RMSLIB.OLB	FORTRAN indirectly used	RMS subroutines	
FOROTS.OLB	FORTRAN IV	FORTRAN IV Object Time System (OTS)	Optional software may be included in SYSLIB.OLB
F4POTS.OLB	FORTRAN IV-PLUS FORTRAN-77	FORTRAN IV-PLUS OTS FORTRAN-77 OTS	Optional software may be included in SYSLIB.OLB

One or the other of the last two libraries must be included when task-building a FORTRAN task unless, as the note states, the libraries are included in SYSLIB.OLB.

Check with your system manager to determine what additional software may be included in SYSLIB.OLB at your site.

Table 1-2 contains a list of the shareable resident libraries which may also be on your system depending upon your installation. You will learn how to use these resident libraries in Module 7, the Static Regions module. Check with your system manager to find out whether the preferred method of including these routines is through linking the code into your task image or through using the resident libraries.

Table 1-2 Resident Libraries

Resident Library	Routines Extracted From	Notes
FCSRES.TSK	SYSLIB.OLB	Generally contains most FCS routines
FORRES.TSK F4PRES.TSK	FOROTS.OLB F4POTS.OLB	May contain all or some FORTRAN OTS routines
RMSRES.TSK	RMSLIB.OLB	Full-functionality RMS resident library
RMSSEQ.TSK	RMSLIB.OLB	RMS resident library for sequential access only

Now do the Tests/Exercises for this module in the Tests and Exercises Book. They are all written problems. Check your answers against those provided in that book.

If you think that you have mastered the material, ask your course administrator to record your progress in your Personal Progress Plotter. You will then be ready to begin a new module.

If you think that you have not yet mastered the material, return to this module for further study.

DIRECTIVES

INTRODUCTION

As stated in the previous module, system services can be placed into two groups:

- Those which are handled entirely by the user task (via the subroutine representing the service)
- Those which require the intervention of the Executive

The services in the second group are known as executive directives (directives). This module discusses the services available as directives and how to make various directive calls.

OBJECTIVES

1. To write programs in FORTRAN which use directives
2. To use information returned by the Executive to perform error checking
3. To use event flags and ASTs with directives

RESOURCES

1. RSX-11M/M-PLUS Executive Reference Manual, Chapters 1 and 2 plus specific directives in Chapter 5
2. FORTRAN IV User's Guide, Appendix B
3. FORTRAN IV-Plus User's Guide, Appendix D
4. FORTRAN 77 User's Guide, Appendix D

INVOKING EXECUTIVE DIRECTIVES FROM A USER TASK

Directive Processing

When an executive directive is called from a FORTRAN task, a standard CALL is generated with an argument list containing each argument in the CALL. When the Task Builder builds the task, the code for the subroutine which invokes the directive is placed in the task. (The subroutines are found in LB:[1,1]SYSLIB.)

At execution time this code generates a Directive Parameter Block (DPB) and then pushes the DPB onto the stack. The DPB contains all of the information needed by the Executive to perform the requested service. This includes a Directive Identification Code (DIC) which identifies which directive is being requested and the length of the DPB. The length is included because the length can vary depending on what directive is being called.

At execution time, the following steps occur:

- The DPB is pushed onto the stack and a trap is made to the Executive.
- A dispatcher routine (part of the Executive) receives the DPB and determines which directive has been requested.
- The dispatcher routine enters the Executive at the appropriate point, depending on the DIC, and the Executive executes the code for the directive (note that the code for the directive actually resides in the Executive, not in the task).
- The Executive sends a Directive Status Word to the task and then returns control to the user task.

Most directives pass control back to the user task at completion of the directive. Certain directives by their nature do not return to the user task. For instance, the Exit Task directive causes the task to EXIT. For the Exit Task directive and other directives of this type, control passes back to the user task only if an error occurs in issuing the directive.

Functions Available Through Executive Directives

Table 2-1 lists many of the Executive directives which are available on your system. For a complete list of the directives under each group, see section 5.1 (on Directive Categories) in the RSX-11M/M-PLUS Executive Reference Manual.

This module, along with later modules on Using the QIO Directive, Using Directives for Intertask Communication, and Dynamic Regions introduce many of the functions which are available. No attempt is made to go over every executive directive. However, at the end of this course, you should know how look up any directive in the manual and invoke it. Each directive is documented individually in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual. The directives appear there in alphabetical order by MACRO-11 name; the FORTRAN CALL name for directives is similar to the MACRO-11 name and is also included in the list. A condensed list of the MACRO and FORTRAN directive names also exists in Table 1-1 in section 1.5.2. To find the page reference for a particular directive, look under "CALL" in the index.

Table 2-1 Types of Directives

Type	CALL Name	Description
Task Execution Control	ABORT	Abort task
	EXIT	Exit task
	REQUES	Request task
	RESUME	Resume task
	RUN	Run task
	START	Run task
	SUSPND	Suspend task
	STOP	Stop task
	USTP	Unstop task
Task Status Control	ALTPRI	Alter priority
	DISCKP	Disable checkpointing
	ENACKP	Enable checkpointing
Informational	GETPAR	Get partition parameters
	Several	Get time parameters
Event-Associated	CLREF	Clear event flag
	CRGF	Create group global flags
	ELGF	Eliminate group global flags
	MARK	Mark time
	WAIT	Mark time
	READEF	Read all event flags
	READEF	Read extended event flags
	SETEF	Set event flag
	WAITFR	Wait for single event flag
Trap-Associated	SREA	Specify requested exit AST
I/O and Intertask Communications	ASNLUN	Assign LUN
	QIO	Queue I/O request
	WTQIO	Queue I/O request and wait
	RECEIV	Receive data
	SEND	Send data
Memory Management	CRRG	Create region
	MAP	Map address window
Parent/ Offspring Tasking	EXST	Exit with status
	SPAWN	Spawn task

The Directive Status Word (DSW)

Upon completion of directive processing, the Executive returns a code in the Directive Status Word (DSW) which gives the status of the request. In order to examine the contents of the DSW and hence determine success or failure, a specific argument must be included in the CALL for the directive. This argument is always the last argument in the list. While this argument is optional, it should always be included since examining the DSW is the only way to determine the success or failure of a directive. The system does not look on a directive failure as an error; hence it is up to the user to check the DSW after a directive CALL. The variable name "IDSW" is frequently used for the DSW; it must be an integer variable.

Successful completion is usually indicated by a DSW value of +1. A negative value indicates an error. Different negative values correspond to different reasons for errors. These values and their general meanings appear in Appendix B of the RSX-11M/M-PLUS Executive Reference Manual and in the RSX-11M/M-PLUS Executive Reference Manual. Specific error values and any special meanings are documented with each executive directive call in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual.

See Example 2-1 for an illustration of how to use the DSW.

Sample Program

Example 2-1 illustrates the use of the Request Task and the Exit Task directives. The directives are given below, along with a description of their functionality:

The Exit Task Directive

- format: CALL EXIT - this CALL has no arguments
- used to make a task inactive and to free up the system resources it uses

The Request Task Directive

- format: CALL REQUES(TASKNM,,IDSW) where TASKNM is the name of the task to be requested
- used to request the specified installed task
- this directive offers the same functionality as the DCL RUN command for an installed task

Each program example in the course contains the following:

- Source code
- A sample run session
- Bulleted items which are described in the text

See the Student Guide for additional information on how to use the examples.

The following comments are keyed to Example 2-1.

❶ If the appropriate OTS library has been included in SYSLIB, the reference to the OTS library is dropped.

❷ Invoke the Request Task directive. Note that the six character (or less) task name must be provided in Radix-5Ø format. This is accomplished by using the R (Radix-5Ø) data type in the DATA statement. (Radix-5Ø is a method of representing a limited set of ASCII characters, such that three characters can be packed into a single PDP-11 word.) The task name must be the installed name (...PIP), not just PIP.

The task is always assumed to be six Radix-5Ø characters; hence you should always pad a name of less than six characters with trailing blanks. For instance, TASKNM 6R/ABC / should be used rather than TASKNM 3R/ABC/.

See Appendix A of the Language Reference Manual for additional information on Radix-5Ø.

❸ The only case in which control will return to the user task after a CALL EXIT is when an error occurred in issuing the directive.

❹ In case of an error, display a message and the DSW value.

❺ A run session is provided for each example program. Note that the PROGRAM name is REKWST, not REQUES. REQUES cannot be used because it is the name of a directive.

```
        PROGRAM REKWST
C
C FILE REQUES.FTN
C
C This task displays a message, requests PIP, and
C then exits
C
C Task-build instructions:
C
C       If your LB:[1,1]SYSLIB.OLB does not contain the
C       FORTRAN Object Time System, then you must
C       specify the appropriate object library
C
C           With FORTRAN IV:
C
C               LINK/MAP REQUES,LB:[1,1]FOROTS/LIBRARY
1 C
C           With FORTRAN IV-PLUS and FORTRAN-77
C
C               >LINK/MAP/CODE:FPP REQUES,LB:[1,1]F4POTS-
C               ->/LIBRARY
C               ! /CODE:FPP includes space in the task
C               ! header for saving the state of the
C               ! floating point processor.
C
C Data statement for RAD50 task name
        DATA TASKNM /6R...PIP/
C Display startup text
        WRITE (5,50)
50      FORMAT (' REQUES HAS STARTED AND WILL REQUEST PIP')
C Request PIP
2       CALL REQUES (TASKNM,,IDSW)
C Check for Directive error
        IF (IDSW .NE. 1) GOTO 1000
C No error, so exit
3       CALL EXIT
C Error code. Display error message and then exit.
4 1000  WRITE (5,1010) IDSW
1010    FORMAT (' ERROR REQUESTING TASK. IDSW = ',I5)
        CALL EXIT
        END
```

Run Session

5

```
>RUN REQUES
REQUES HAS STARTED AND WILL REQUEST PIP
PIP>^Z
>
```

Example 2-1 Requesting a Task From Another Task

Example Using Other Directives

The following directives are used in Example 2-2.

Suspend Task (CALL SUSPND)

- Used to suspend the issuing task
- The task can be resumed by another task issuing a resume task directive or by an operator using the DCL CONTINUE command.

Alter Priority (CALL ALTPRI)

- Alters the running priority of an active task.

Disable Checkpointing (CALL DISCKP)

- Disables checkpointing for a checkpointable task.

Enable Checkpointing (CALL ENACKP)

- Enables checkpointing again after a DISCKP directive.

Extend Task (CALL EXTTSK)

- Modifies the size of the task by an increment or decrement of 32-word blocks.

Example 2-2 shows the use of a variety of directives. See the run demonstration below the source code. The following comments are keyed to the example. Items 2,3,5 and 7 refer to the run session following the program listing.

1. Task suspends itself. This allows the operator to use the DCL SHOW TASKS/ACTIVE command to examine the task parameters.

2. Note that the task is loaded at addresses Ø1123600(8) to Ø117Ø1ØØ(8). SPN means the task is suspended.

3. The operator must use the DCL CONTINUE command to resume the task.

4. Suspend again after disabling checkpointing and altering the running priority.

5. Note the change in PRI (running priority). CKD in the output from the DCL command SHOW TASKS/ACTIVE indicates that checkpointing has been disabled.

6. Suspend again after enabling checkpointing, altering the priority back to 5Ø(1Ø), and extending the task.

7. Note the change in priority. Note also that the task was checkpointed and is now loaded at addresses Ø11236ØØ(8) to Ø121Ø1ØØ(8). This is a task size of 643ØØ(8) bytes, compared to 443ØØ(8) bytes before. The extend is for 2ØØ(8) blocks, where each block is 1ØØ(8) bytes long, meaning 2ØØØØ(8) bytes extra. See Appendix B for a conversion table for bytes to blocks and of octal to decimal.

```
 Ø117Ø1ØØ(8)           Ø121Ø1ØØ(8)
-Ø11236ØØ(8)          -Ø11236ØØ(8)
 -----------           -----------
    443ØØ(8)              643ØØ(8)
```

```
          PROGRAM MISC
   C+
   C FILE MISC.FTN
   C
   C This task uses some miscellaneous Executive directives
   C to suspend itself, alter its running priority, disable
   C and enable checkpointing, and extend its task size.
   C
   C Task-build instructions:
   C
   C       LINK/CHECKPOINT/MAP MISC,LB:[1,1]FOROTS/LIBRARY
   C       since the task must be checkpointable to disable
   C       checkpointing and to extend its size.
   C
   C Install and Run instructions:
   C
   C       Install the task. Then Run it to start it up.
   C       The task will suspend itself several different
   C       times. Each time, use the command
   C       SHOW TASKS:MISC/ACTIVE/FULL (MCR ATL MISC)
   C       to examine the changes. Use the command
   C       CONTINUE MISC (MCR RESUME MISC)
   C       to resume the task.
   C-
          INTEGER DSW,DRCTV
❶         CALL SUSPND(DSW)           ! Suspend to allow check
   C                                 !  of status
          IF (DSW.LT.0) GOTO 1010 ! Branch on directive
   C                                 !  error
   C Make some changes and then suspend again
          CALL DISCKP(DSW)           ! Disable checkpointing
          IF (DSW.LT.0) GOTO 1020
          CALL ALTPRI(,10,DSW)       ! Alter running priority
          IF (DSW.LT.0) GOTO 1030
❹         CALL SUSPND(DSW)           ! Suspend to allow check
   C                                 !  of status
          IF (DSW.LT.0) GOTO 1040
   C Make some other changes and then suspend again
          CALL ENACKP(DSW)           ! Reenable checkpointing
          IF (DSW.LT.0) GOTO 1050
          CALL ALTPRI(,,DSW)         ! Return priority to
   C                                 !  original
          IF (DSW.LT.0) GOTO 1060
          CALL EXTTSK("200,DSW)      ! Extend task size by
   C                                 !  200(8) blocks
          IF (DSW.LT.0) GOTO 1070
❻         CALL SUSPND(DSW)           ! Suspend again
          IF (DSW.LT.0) GOTO 1080
          CALL EXIT                  ! Exit
   C Error handling
   1010   WRITE (5,1015) DSW
   1015   FORMAT (' ERROR ON 1ST SUSPEND. DSW = ',I5)
          GOTO 1100
   1020   WRITE (5,1025) DSW
```

Example 2-2 Using Some Miscellaneous Directives (Sheet 1 of 2)

```
1025    FORMAT (' ERROR ON DISABLE CHECKPOINTING. DSW = '
        1,I5)
        GOTO 1100
1030    WRITE (5,1035) DSW
1035    FORMAT (' ERROR ON 1ST ALTER PRIORITY. DSW = ',
        1I5)
        GOTO 1100
1040    WRITE (5,1045) DSW
1045    FORMAT (' ERROR ON 2ND SUSPEND. DSW = ',I5)
        GOTO 1100
1050    WRITE (5,1055) DSW
1055    FORMAT (' ERROR ON ENABLE CHECKPOINTING. DSW = '
        1,I5)
        GOTO 1100
1060    WRITE (5,1065) DSW
1065    FORMAT (' ERROR ON 2ND ALTER PRIORITY. DSW = ',
        1I5)
        GOTO 1100
1070    WRITE (5,1075) DSW
1075    FORMAT (' ERROR ON EXTEND TASK. DSW = 'I5)
        GOTO 1100
1080    WRITE (5,1085) DSW
1085    FORMAT (' ERROR ON 3RD SUSPEND. DSW = ',I5)
1100    CALL EXIT
        END
```

Run Session

```
    >INS MISC
    >RUN MISC
    >SHOW TASKS/ACTIVE FULL MISC
2  [MISC   067250  GEN    070310 01123600-01170100  PRI - 50.  DPRI - 50.
   [   STATUS:  SPN -PMD
       TI - TT11:  IOC - 0.  BIO - 0.  EFLG - 000000 000000  PS - 170000
       PC - 001640  REGS 0-6  000001 001242 001242 000000 001432 003572 001242
3   >CONTINUE MISC
    >SHOW TASKS/ACTIVE FULL MISC
5  [MISC   067250  GEN    070310 01123600-01170100  PRI - 10.  DPRI - 50.
   [   STATUS:  CKD SPN -PMD
       TI - TT11:  IOC - 0.  BIO - 0.  EFLG - 000000 000000  PS - 170000
       PC - 001640  REGS 0-6  003626 001242 001242 000000 001432 003572 001242
    >CONTINUE MISC
    >SHOW TASKS/ACTIVE FULL MISC
7  [MISC   067250  GEN    070310 01123600-01210100  PRI - 50.  DPRI - 50.
   [   STATUS:  SPN -PMD
       TI - TT11:  IOC - 0.  BIO - 0.  EFLG - 000000 000000  PS - 170000
       PC - 001640  REGS 0-6  003626 001242 001242 000000 001432 003572 001242
    >CONTINUE MISC
    >SHOW TASKS/ACTIVE FULL MISC
    ATL -- Task not active
```

Example 2-2 Using Some Miscellaneous Directives (Sheet 2 of 2)

Run Time Conversion Routines

As mentioned earlier, the system maintains task names, partition names, and certain other data in Radix-5Ø format in order to save space. There are times when conversions between ASCII and Radix-5Ø format need to be performed at run time. For example, you can modify Example 2-1 (REQUES.FTN), so an operator can type in the task name at run time. This ASCII name would have to be converted at run time to Radix-5Ø format. The function RAD5Ø or the subroutine IRAD5Ø is used to perform the conversion. The code segment shown below illustrates the use of the function RAD5Ø:

```
        DIMENSION TASKNM(2)
        READ(5,1)TASKNM
   1    FORMAT(2A4)
        CALL REQUES(RAD5Ø(TASKNM),,IDSW)
```

If the Get Task directive (CALL GETTSK) is used to retrieve task information, the task name and partition name are returned in Radix-5Ø format. If you wish to display these, you need to convert them to ASCII format. The subroutine R5ØASC is provided for this purpose. The program shown below illustrates the use of the R5ØASC subroutine:

```
        DIMENSION IBUFF(16)
        CALL GETTSK(IBUFF)
        CALL R5ØASC(6,IBUFF(1),TASKNM)
        CALL R5ØASC(6,IBUFF(3),PARTNM)
        WRITE(5,1)TASKNM
        WRITE(5,2)PARTNM
   1    FORMAT(' TASK NAME IS ',A6)
   2    FORMAT(' PARTITION NAME IS ',A6)
        END
```

NOTIFYING A TASK WHEN AN EVENT OCCURS

Often a task needs to know when an event has occurred. The event may have occurred within another task; for example, when the task has completed a requested function. The event may instead have occurred within the system; for example, when a requested I/O operation is completed. There are two methods for implementing synchronization, event flags and asynchronous system traps.

Event Flags

There are three types of event flags: local, global (or common), and group global. Ninety-six event flags are made available to tasks, each with a unique number (1(1Ø)-96(1Ø)).

Local event flags are provided for each task. There are 32(1Ø) local event flags, numbered 1(1Ø)-32(1Ø). These flags are used to synchronize a task with an Executive service, such as an I/O transfer. One task cannot reference another task's local event flags, so they cannot be used to synchronize tasks with one another. Local event flags 25(1Ø)-32(1Ø) are reserved for system use and hence should not be used by a user task.

Global or Common event flags are provided for synchronization among different tasks. There is one set of 32(1Ø) global event flags for the system numbered 33(1Ø)-64(1Ø). These flags can be referenced by any task. Global event flags 57(1Ø)-64(1Ø) are reserved for system use and should not be used by user tasks.

NOTE

There is no way to protect against other tasks using global event flags. Great care must be taken to ensure that global event flags aren't used at the same time by several different users. Check with your system manager before using any global event flag to insure that it is not used for some other purpose.

There are only 32(10) global event flags available system-wide. If additional event flags are needed, another set of event flags can be created for synchronization among different tasks. 32(10) group global event flags, numbered 65(10)-96(10), can be created for any UIC group number. These event flags can be referenced by any task running under the correct group number. Hence, they can be used to synchronize tasks running under that group number, and offer an additional advantage in that they cannot be referenced by tasks running under other group numbers.

Group global event flags are created using the DCL SET GROUPFLAGS CREATE (FLA /CRE in MCR) command or the Create Group Global Event Flags (CRGF$) directive. When users in a group don't need them anymore, the group global event flags can be marked for deletion using the DCL SET GROUPFLAGS DELETE (FLA /ELIM in MCR) command or the Eliminate Group Global Event Flags (ELGF$) directive. After that, when all active tasks in the group have finished using them, the group global event flags are eliminated.

Using Event Flags for Synchronization

LEARNING ACTIVITY 2-1

Read section 2.2 (on Event Flags) in the RSX-11M/M-PLUS Executive Reference Manual. Pay particular attention to the examples. This section covers how event flags can be used for synchronizing tasks. When you have finished reading the material, answer the following questions. The answers are provided in Appendix G.

Questions:

1. **In Example 1 in the reading, how can Task B do some work while waiting for event flag 35 to be set by Task A?**

2. **What would happen in Example 2 if a local event flag (e.g., 1) were used instead of a common event flag?**

3. **Why is a local flag satisfactory in Examples 3 and 4?**

Examples of the Use of Event Flags for Synchronization

Examples 2-3 and 2-4 show the use of event flags to synchronize two tasks. WFLAG creates the group global event flags for the group. It then clears event flag 65(10) and waits for that flag to be set. SFLAG sets event flag 65(10), which unblocks WFLAG. Run WFLAG first, then run SFLAG.

The following notes are keyed to the examples. Note 5 is in SFLAG; all others are in WFLAG.

1. Create the group global event flags. The default group number (used here) is the group number that the task is running under.

2. An error is reported if the flags already exist. This isn't a fatal error, so we check for this condition. If the flags do exist, print a message and continue.

3. The flag is in an unknown state at startup. Therefore, we must clear the flag before waiting for it to be set.

4. Wait for the event flag to be set by SFLAG. This causes WFLAG to be blocked. Now run SFLAG.

5. Set event flag 65 in task SFLAG. This allows WFLAG to become unblocked. SFLAG then exits.

6. When WFLAG is unblocked and it continues executing, it starts up here. We check for any directive error entering the Wait For state, print a message, and exit.

In certain programming situations it may be necessary to test one or more event flags to see if they are currently clear or set. The CALL READEF directive can be used to read a single flag. After the flag has been read, the contents of the DSW will determine the condition of the flag. If DSW=2, the flag was set; if DSW=0, the flag was clear.

```
              PROGRAM WFLAG
      C
      C FILE WFLAG.FTN
      C
      C This task creates the group global event flags, and
      C then clears event flag 65. and waits for it to be set.
      C When the flag is set, it writes a message and exits
      C
      C Install and run instructions:
      C
      C       Run WFLAG, then run SFLAG. At least one of the
      C       tasks must be installed, or else the RUN command
      C       will try to install both tasks under the same
      C       name (TTnn)
      C
              WRITE (5,10)
      10      FORMAT (' WFLAG IS CREATING THE GROUP GLOBAL EVENT
              1 FLAGS')
            1 CALL CRGF (,IDSW)
            2 IF (IDSW .LT. 0) GOTO 900
      15      WRITE (5,20)
      20      FORMAT (' CLEAR AND THEN WAIT FOR EF 65. TO BE SET')
            3 CALL CLREF (65,IDSW)
              IF (IDSW .LT. 0) GOTO 1100
            4 CALL WAITFR (65,IDSW)
             [IF (IDSW .LT. 0) GOTO 1200
             |WRITE (5,30)
            6 |
      30     |FORMAT (' EF 65. HAS BEEN SET. WFLAG WILL NOW EXIT')
             [CALL EXIT
      C Error processing
      C
      C Check for code -17, meaning flags already exist
     [900     IF (IDSW .NE. -17) GOTO 1000
     |C In that case, just dislay a message and continue.
   2 |        WRITE (5,910)
     |910     FORMAT (' GROUP GLOBAL EVENT FLAGS ALREADY EXIST')
     [        GOTO 15
      C Here for fatal errors, display message and exit
      1000    WRITE (5,1010) IDSW
      1010    FORMAT (' DIRECTIVE ERROR CREATING GROUP GLOBAL
              1EF''S. DSW = ',I5)
              CALL EXIT
      1100    WRITE (5,1110) IDSW
      1110    FORMAT (' DIRECTIVE ERROR CLEARING EVENT FLAG 65.
              1 DSW = ',I5)
              CALL EXIT
```

Example 2-3 Waiting for an Event Flag (Sheet 1 of 2)

```
1200    WRITE (5,1210) IDSW
1210    FORMAT (' DIRECTIVE ERROR WAITING FOR EVENT FLAG
        1 65. DSW = ',I5)
        CALL EXIT
        END
```

Run Session

```
>INS WFLAG
>INS SFLAG
>RUN WFLAG
>
 WFLAG IS CREATING THE GROUP GLOBAL EVENT FLAGS
CLEAR AND THEN WAIT FOR EF 65. TO BE SET
RUN SFLAG
>
 EF 65. IS BEING SET. THEN SFLAG WILL EXIT.
EF 65. HAS BEEN SET. WFLAG WILL NOW EXIT
```

Example 2-3 Waiting for an Event Flag (Sheet 2 of 2)

```
      PROGRAM SFLAG
C
C FILE SFLAG.FTN
C
C This task sets event flag 65. It assumes that the
C group global event flags have already been created.
C
C Install and run instructions:
C
C       Run WFLAG, then run SFLAG. At least one of the
C       tasks must be installed, or else the RUN command
C       will try to install both tasks under the same
C       name (TTnn).
C
        WRITE (5,10)
10      FORMAT (' EF 65. IS BEING SET. THEN SFLAG WILL EXIT')
   ❺    CALL SETEF (65,IDSW)
C The DSW value returned for SETEF is 2 if it was set
C and 0 if it was clear. A 1 is NOT returned for success
        IF (IDSW .LT. 0) GOTO 1000
        CALL EXIT
C Error code
1000    WRITE (5,1010)
1010    FORMAT (' DIRECTIVE ERROR SETTING EF 65. DSW = '
        1,I4)
        CALL EXIT
        END
```

Example 2-4 Setting an Event Flag

ASYNCHRONOUS SYSTEM TRAPS(ASTs)

Asynchronous System Traps (ASTs) are used to detect events that occur asynchronously to a task's execution. We say that they occur asynchronously to a task's execution because they occur at unpredictable times, depending on conditions which the task cannot control. By doing some work and then periodically checking an event flag to check on an event, a task can do work while waiting for an event to occur. However, this means that the task must periodically stop its work to check the flag.

Using an AST gives the Executive the responsibility for monitoring the event. The Executive will "interrupt" the task and transfer control to a special user written routine when the event has occurred. Using this technique is more efficient because the task doesn't have to do any periodic checking, and it probably results in faster notification because the task is notified right after the event occurs. With periodically reading the flag, it may take quite a while to notice that the event has occurred if it occurs immediately after a check.

The only directives which allow the use of ASTs from FORTRAN are CNCT, PWRUP, SDRC, SDRP, SPAWN, SREA and SREX.

Figure 2-1 shows how an AST routine works. The following notes are keyed to the figure.

1. The user specifies an AST routine in a directive. The Executive sets up for the AST.

2. The Executive returns control to the user task.

3. When the system determines that the event has occurred which corresponds to the specified AST routine, the Executive passes control to the AST routine, executing it before any other user code in the task. This means that if the task is executing at the time of the AST, the task is "interrupted" until the AST routine is executed. The AST routine is executed even if the task is stopped or blocked. In that case, the task returns to its stopped or blocked state after the AST routine is executed, unless the AST routine or some external event unstops or unblocks the task in the meantime.

4. The AST routine is a user written routine contained within the task.

5. The AST routine uses a standard RETURN statement to return control to the main code via the Executive. However, before the actual return, the Executive checks to see if any other ASTs have occurred while the AST routine was executing. Any such additional ASTs are queued in an AST pending queue in a first-in-first-out order; these ASTs are also serviced before the Executive returns to the point at which the AST interrupt occurred.

For additional information on ASTs, see section 1.5.4 in Chapter 1 and sections 2.3.3 and 2.3.4 in Chapter 2 of the RSX-11M/M-PLUS Executive Reference Manual.

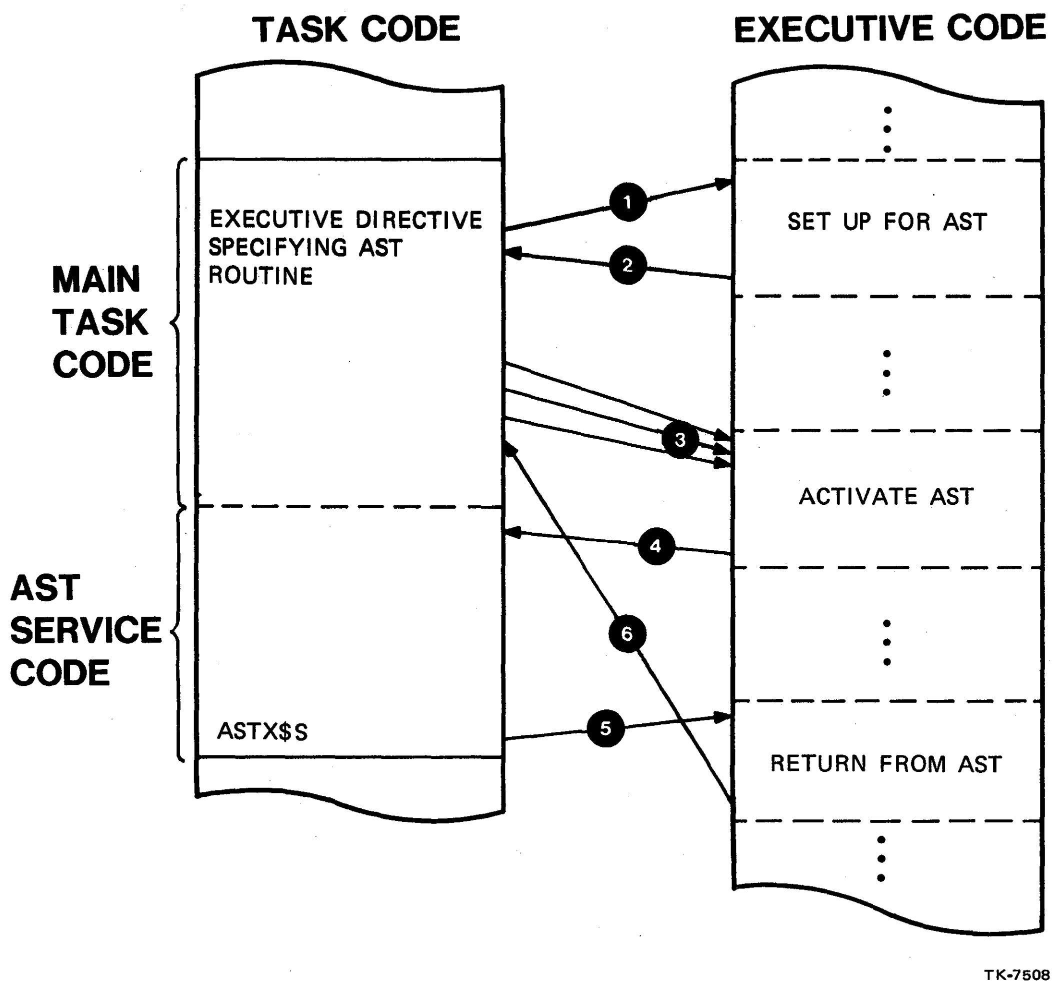


Figure 2-1 AST Sequence

Example 2-5 shows the use of ASTs. An AST routine is entered if an abort request is made by either another task or an operator.

The following notes are keyed to the example.

1. Set up for AST on abort attempt.

2. Loop until abort request comes in.

3. Service routine entered on first abort request. For this particular AST, a nonprivileged task enters this routine only once and further ASTs are cancelled. If the task is built as a privileged task, the routine is entered each time an abort attempt comes in. See Appendix D for an explanation of privileged tasks.

4. Note that FORTRAN I/O cannot be performed in an AST routine because the I/O code is not reentrant; therefore any I/O to be done in an AST routine must be done via QIO directives. The next module will discuss the QIO directive in detail.

Another directive, SREX, gives extended capabilities. An entry passed to the AST routine indicates whether the abort request came from a privileged or nonprivileged task or user and further, whether it came from an Abort Task directive or a DCL (or MCR) command. Each case can be handled differently.

```
        PROGRAM ASTEX
C+
C FILE ASTEX.FTN
C
C This task sets up a Specify Request Exit AST routine.
C It then sits in a loop until someone tries to abort
C it. At that point, it enters the AST routine and sends
C out a message. It won't abort the first time. A second
C abort attempt will succeed because for this particular
C AST, the first AST entry cancels any further AST's for
C this event
C
C Compile instructions:
C
C The AST routine must be compiled with traceback
C disabled.  Since in this case the AST routine source
C is in the same file as the mainline, compile both with
C traceback disabled.
C
C For FORTRAN IV:
C
C       FORTRAN/NOLINE_NUMBERS/LIST ASTEX
C
C For FORTRAN IV-PLUS or FORTRAN-77
C
C       FORTRAN[/F4P or /F77]/NOTRACE/LIST ASTEX
C
C Run notes: Remember to use the name the task is
C installed under when attempting to abort the task.
C-
        INTEGER DSW
        EXTERNAL REXAST
     ❶  CALL SREA(REXAST,DSW)    ! Set up Specify Exit AST
        IF (DSW.LT.0) GOTO 1001 ! Branch on error
        TYPE *,'ASTEX STARTING UP. WILL WORK UNTIL ABORTED.
C Do some work.
10      DO 20 I= -32767,32767
❷ 20    CONTINUE
        GO TO 10
C Error code
1001    TYPE *,'ERROR ON DIRECTIVE, DSW = ',DSW
        CALL EXIT
        END
```

Example 2-5 Using a Requested Exit AST (Sheet 1 of 2)

```
C
    ❸  SUBROUTINE REXAST
C
C AST service routine
C
       INTEGER PLIST(3),IOWVB
       REAL TEXT1(6),TEXT2(7)
       DATA IOWVB/"11000/
       DATA TEXT1 /'TRYI','NG T','O AB',
       1'ORT ','ME ','EH? '/
       DATA TEXT2 /'WE W','ON''T',' LET',
       1' YOU',' THI','S TI','ME! '/
C Set up for QIO directive
       CALL GETADR(PLIST(1),TEXT1(1))
       PLIST(2) = 23
       PLIST(3) = "40
C Use QIO directive to display text
    ❹  CALL WTQIO(IOWVB,5,1,,,PLIST)
C Set up for 2nd line of text
       CALL GETADR(PLIST(1),TEXT2(1))
       PLIST(2) = 27
C Use QIO directive to display text
       CALL WTQIO(IOWVB,5,1,,,PLIST)
       RETURN
       END
```

Run Session

```
>INS ASTEX
>RUN ASTEX
>
 ASTEX STARTING UP. WILL WORK UNTIL ABORTED.
ABORT/TASK ASTEX
>
 TRYING TO ABORT ME, EH?
WE WON'T LET YOU THIS TIME!
ABORT/TASK ASTEX
10:57:02  Task "ASTEX " terminated
          Aborted via directive or CLI
>
```

Example 2-5 Using a Requested Exit AST (Sheet 2 of 2)

DIRECTIVES

Now do the tests/exercises for this module in the tests/exercises book. They are all lab problems. Check your answers against the solutions provided, either in that book or in on-line files, under UFD [2Ø2,2].

You will need the program READF.FTN to do question 1. It should be available on-line probably under UFD [2Ø2,1]. In case it is not available on-line, the source code is listed in Appendix G.

If you think that you have mastered the material, ask your course administrator to record your progress in your Personal Progress Plotter. You will then be ready to begin a new module.

If you think that you have not yet mastered the material, return to this module for further study.

3 USING THE QIO DIRECTIVE

INTRODUCTION

All Input/Output under RSX-11M is performed using QIO directives. In this module, you will learn how to use the QIO directive, concentrating on its use for input/output to a terminal.

OBJECTIVES

1. To use the QIO directive to perform I/O to a non-file-structured device
2. To choose either synchronous or asynchronous I/O as the most effective method
3. To perform complete error checking upon I/O completion

RESOURCES

1. RSX-11M/M-PLUS Executive Reference Manual, Chapter 5 for specific directives
2. RSX-11M/M-PLUS I/O Driver's Reference Manual, Chapters 1 and 2

OVERVIEW OF QIO DIRECTIVES

All I/O operations under RSX-11M are performed using QIO directives. While transparent to the user, all FORTRAN READ and WRITE statements are ultimately transformed into QIO directives. The QIO directive causes an I/O request to be passed to the appropriate service routine. The service routine is either a device driver or a system task called an ancillary control processor (ACP). There is a device driver for each device type on the system. There are three ACP's provided, F11ACP for FILES-11 structured disks, MTAACP for ANSII magtape, and NETACP for DECNET.

The I/O packet is placed in an I/O queue for the service routine. The packets are queued in the order of the priority of the issuing tasks. If there are multiple requests at a given priority, those requests are queued first-in-first-out (FIFO). The QIO directive does not perform the I/O operation itself, but simply queues the request to the appropriate service routine, which performs the actual I/O transfer. After the I/O request has been queued, the Executive returns control to the issuing task, unless the task requests the Executive to place the task in a Wait For state until the I/O transfer completes.

PERFORMING I/O

QIO directives are generally used by a programmer for I/O on non-file-structured devices such as terminals. For file I/O, all READ and WRITE statments are passed off to the File Control Services (FCS) or Record Management Services (RMS), which in turn issue the appropriate QIOs for you. When using QIOs, you specify which I/O operation (e.g., Read Virtual Block or Write Virtual Block) is to be performed by means of an I/O function code. Specify the device by means of the logical unit number (LUN). You specify additional information about the I/O operation (e.g., what buffer to write and how many characters) by means of an I/O parameter list (IOPL). All of this information is passed to the Executive through parameters in the Directive Parameter Block (DPB), as it is with all directives.

USING QIO DIRECTIVES IN FORTRAN

There are two basic reasons for using QIO directives in FORTRAN.

- To acheive asynchronous I/O.

  All READ and WRITE statements are synchronous; i.e., the program is put into a Wait For state until the I/O is complete. If you need to perform asynchronous I/O, it can only be done via QIO directives.

- To perform I/O functions not possible with READ and WRITE statements.

  Certain I/O functions, particularly to a terminal, cannot be done by READ and WRITE statements. Some examples are read with no echo and cursor control on a video terminal. By using QIO directives, these functions can be done from FORTRAN programs.

I/O FUNCTIONS

Each device type has its own set of legal I/O functions. Certain functions are called standard or common, since they are available on all devices. The seven standard I/O functions are listed in Table 3-1. Logical block transfers (Read Logical Block and Write Logical Block) can usually be performed for any device. For file structured devices, virtual block transfers can be performed only if a file is open on the device.

If Virtual Block I/O is requested for a non-file-structured device, such as a terminal, it is converted to logical block I/O for you. In addition, devices may have additional device specific functions, such as Read No Echo at a terminal. Each function requires its own set of parameters, which are specified in an I/O parameter list.

Table 3-1 Common (Standard) I/O Functions

Global Symbol in MACRO	Octal Value	Function	Suggested FORTRAN Name
IO.ATT	ØØ14ØØ	Attach device	IOATT
IO.DET	ØØ2ØØØ	Detach device	IODET
IO.KIL	ØØØØ12	Cancel I/O requests	IOKIL
IO.RLB	ØØ1ØØØ	Read Logical Block	IORLB
IO.RVB	Ø1Ø4ØØ	Read Virtual Block	IORVB
IO.WLB	ØØØ4ØØ	Write Logical Block	IOWLB
IO.WVB	Ø11ØØØ	Write Virtual Block	IOWVB

Throughout the literature you will find I/O function codes given in the form used by MACRO programmers; for example, IO.ATT and IO.DET. In MACRO, these codes can be used directly as function codes in a QIO directive, and the proper octal values will be inserted. In FORTRAN, you must determine the octal value for the function and use that value in the CALL QIO or CALL WTQIO. For instance, to issue an ATTACH, you could use:

```
    CALL QIO("14ØØ,,,,,,,)
```

In order to make QIO calls more readable, it is recommended that you create a DATA statement for any needed QIO functions using the suggested variable names shown above. While the actual name is arbitrary, (but must be an integer variable), a certain degree of standardization will be achieved by using the MACRO symbol without the period (IOATT versus IO.ATT). Hence to perform an ATTACH:

```
    DATA IOATT/"14ØØ/
    CALL QIO(IOATT,,,,,,,)
```

The octal values for all function codes can be found in Appendix B of the RSX-11M/M-PLUS I/O Driver's Reference Manual.

LOGICAL UNIT NUMBERS (LUNs)

The device for an I/O operation is specified by means of a logical unit number. The correspondence between logical unit numbers and physical devices is made initially at task-build time.

The default LUN assignments set up by the Task Builder are as follows:

```
    LUN #1 - SY:
    LUN #2 - SY:
    LUN #3 - SY:
    LUN #4 - SY:
    LUN #5 - TI:
    LUN #6 - CL:
    LUN #7 - TI:
```

LUN 7 is typically used for error messages.

These default assignments may be overridden at task-build time by using the ASG option. Additional LUNs can be created (up to a total of 25Ø LUNs) by using the UNITS option.

Once a task is installed, an operator can check the LUN assignments for the task by using the DCL SHOW LOGICAL_UNITS command (LUN in MCR). The assignments can be changed by an operator using the DCL ASSIGN/TASK command (REA in MCR). The LUN assignments can also be checked at run time using the Get LUN directive (CALL GETLUN), and changed using the Assign LUN directive (CALL ASNLUN).

SYNCHRONOUS AND ASYNCHRONOUS I/O

There are two kinds of I/O, synchronous I/O and asynchronous I/O. With synchronous I/O, the Executive provides sychronization. When a task issues an I/O request, it doesn't get control back from the Executive until after the I/O packet is queued, and the I/O operation (the transfer performed by the service routine itself) is completed. In other words, the synchronous I/O request asks the Executive to queue the I/O packet and then place the task in a "Wait For" state, waiting for the specified event flag to be set, at which time the actual I/O is complete.

Figure 3-1 shows the flow of instructions during the processing of a synchronous I/O operation. The task does not execute the instruction following the QIO directive until after the I/O transfer itself has completed.

Figure 3-2 shows a time diagram illustrating the same I/O operation. Note that once the QIO directive is executed at step 1, the task doesn't execute again until step 8, after the transfer has completed. The system handles all synchronization with synchronous I/O. Use the CALL WTQIO directive to invoke this type of I/O. (CALL WTQIO is a combination of a QIO and a WAITFR).

Commentary to Figures 3-1 and 3-2:

❶ User task executes WTQIO directive.

❷ Executive queues the I/O request.

❸ Executive calls the driver.

❹ Driver begins the I/O transfer.

❺ Driver handles I/O transfer as necessary.

❻ I/O transfer completes.

❼ Driver finishes up and notifies task the I/O is completed.

❽ User task continues.

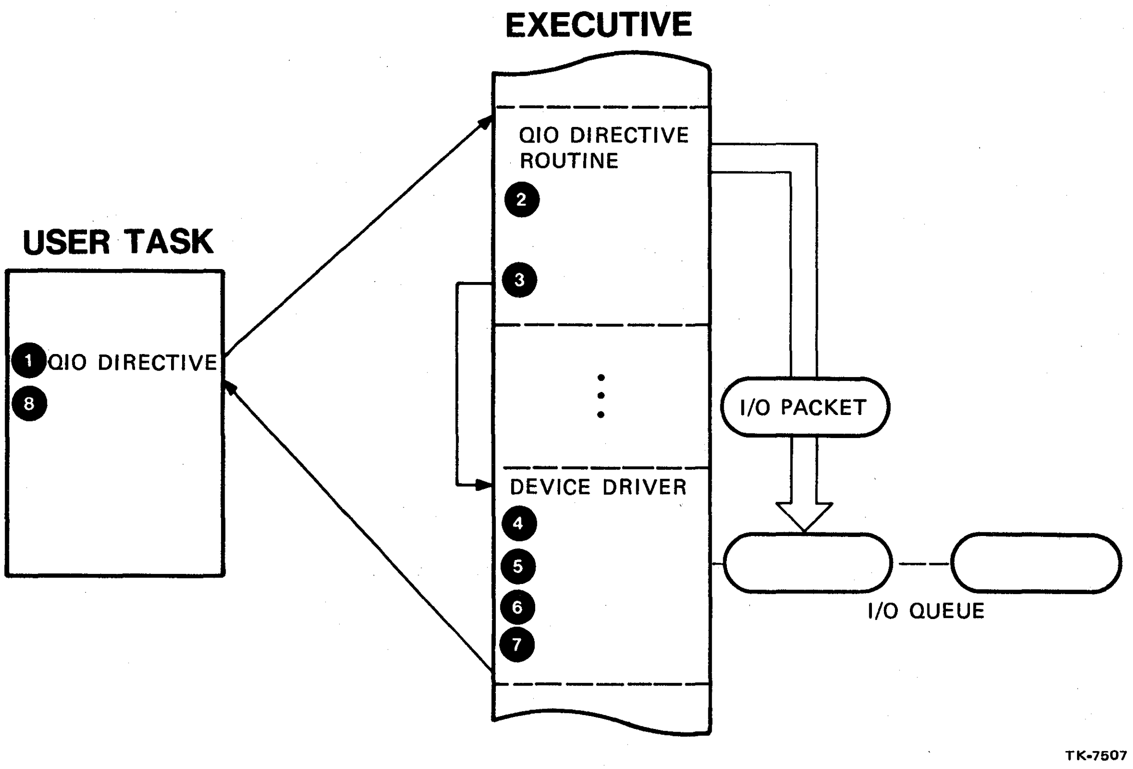


Figure 3-1 Execution of a Synchronous I/O Request

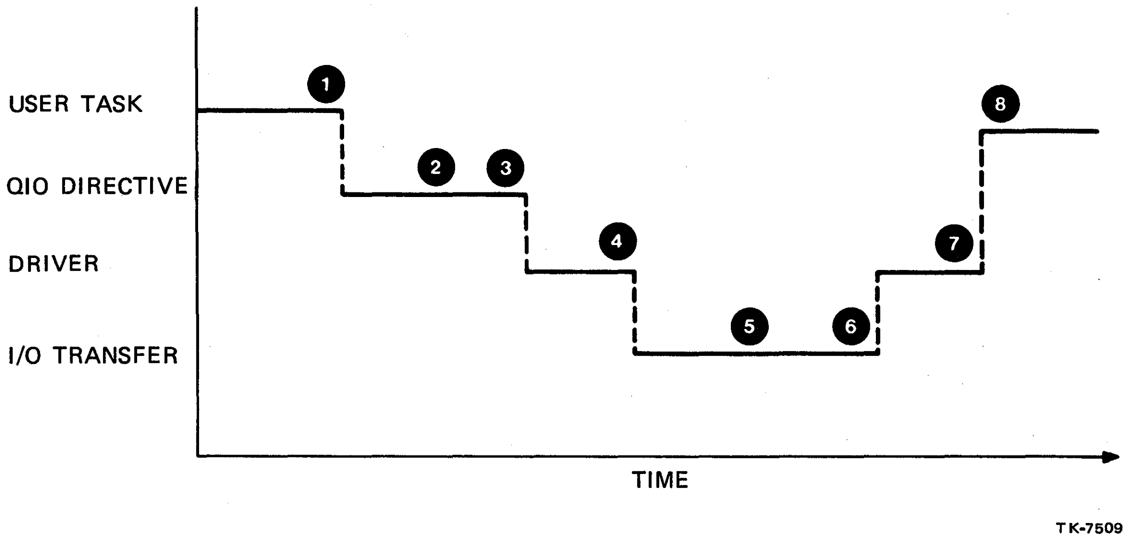


Figure 3-2 Events in Synchronous I/O

With asynchronous I/O, the Executive still queues the I/O request. However, when a task issues an asynchronous I/O request, the Executive passes control back to the task immediately after the I/O packet is queued to the driver. You must provide synchronization concerning the completion of the actual I/O transfer. This could occur at various times, depending upon such factors as how many other I/O packets are already in the driver's I/O queue, and the speed of the device itself. The task executes in parallel while the I/O transfer takes place. In Figure 3-3, the instruction after the QIO request is executed after the I/O packet is queued and the driver has started the transfer, not after the I/O transfer completes. The task continues executing unless it chooses to wait. Figure 3-4 shows a time diagram illustrating asynchronous I/O.

Note that after the QIO directive is executed at 1 , the task begins executing again at step 5 . In this example, the task waits for the I/O transfer to complete at step 5a . If you use asynchronous I/O, you must provide any synchronization yourself, using event flags or by testing the I/O status block. The task shown in Figures 3-3 and 3-4 uses a Wait For Event Flag directive at step 5a . Use the directive CALL QIO to invoke this type of I/O.

The advantage of asynchronous I/O is that your task can continue processing in parallel with the I/O transfer. For example, you can perform computations while waiting for a read or a write operation to complete. Of course, if you need the information from the read before you can do anything else, it is better to use synchronous I/O.

Commentary to Figures 3-3 and 3-4:

1. User task executes QIO directive.
2. Executive queues I/O request.
3. Executive calls the driver.
4. Driver begins the I/O transfer, and passes control back to the user task.
5. Driver handles I/O transfer as necessary. User task executes in parallel with I/O transfer.
6. User task waits for I/O operation to complete.
7. I/O transfer completes.
8. Driver finishes up and Executive notifies task that I/O is completed.
9. User task continues.

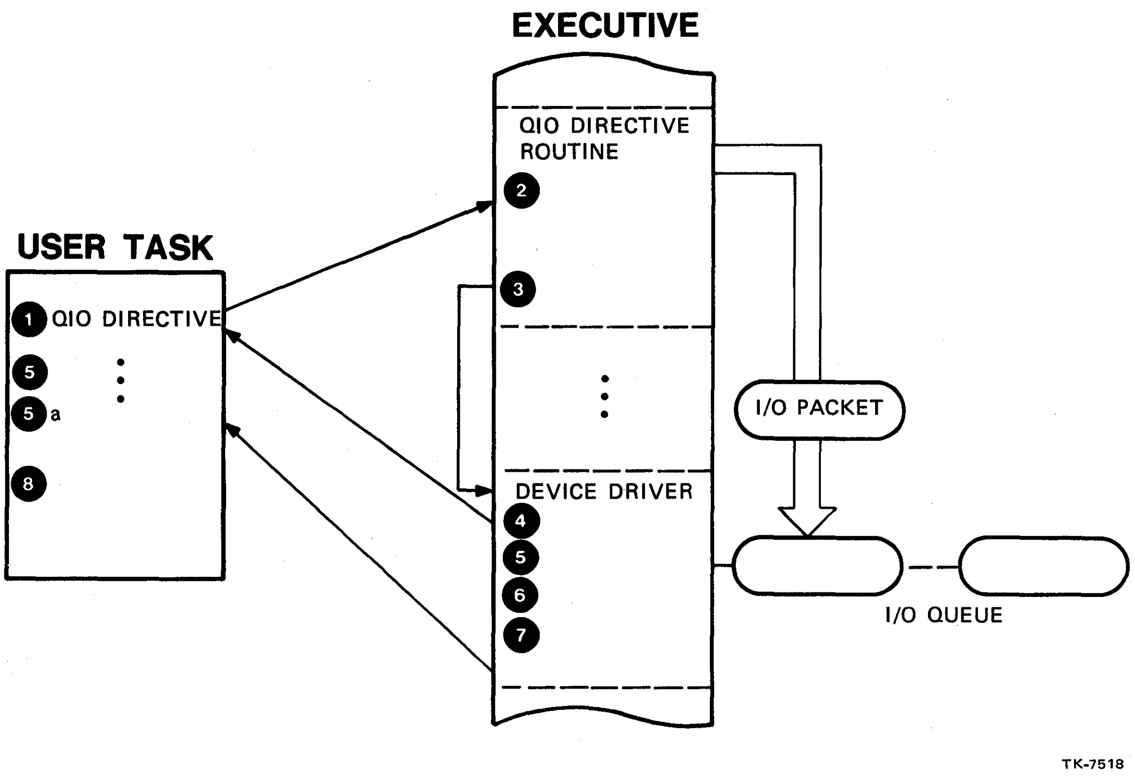


Figure 3-3 Execution of an Asynchronous I/O Request

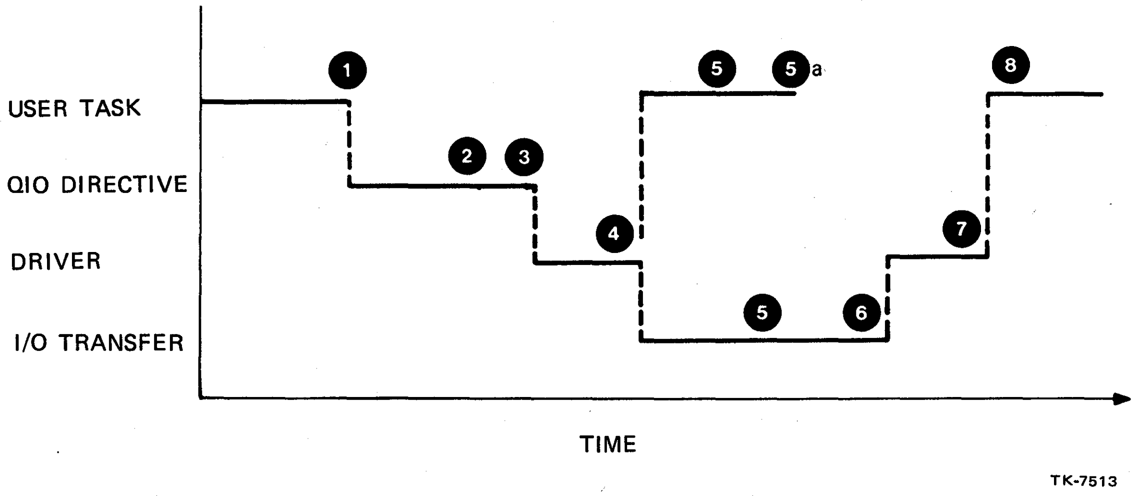


Figure 3-4 Events in Asynchronous I/O

MAKING THE I/O REQUEST

Specify the following information in the CALL QIO or CALL WTQIO when requesting I/O:

- Synchronous or asynchronous I/O, by using the appropriate directive.
- The I/O function to be performed.
- The LUN to be used for the I/O operation.
- An event flag number, if any, to be used for synchronization. This is required for synchronous I/O.
- The address of an I/O Status Block (IOSB). The IOSB is used to pass status and other information about the I/O operation back to the task.
- The I/O parameter list (up to six words) which specifies information for the particular device and I/O function requested.
- The Directive Status Word (DSW)

Table 3-2 shows the I/O parameter list arguments which are needed for each of the standard I/O functions with the full-duplex terminal driver. Note that for write logical block and write virtual block, the vertical format control characters are the standard FORTRAN carriage control characters.

Table 2-3 in section 2.3 of the RSX=11M/M-PLUS I/O Driver's Reference Manual lists these standard functions and the other device-specific functions available with the full-duplex terminal driver. The device-specific functions will be discussed later in this module. If your RSX-11M system has the half-duplex terminal driver, Table 3-3 in section 3.3 lists the functions available with that driver. For other devices, there is a corresponding table in the appropriate chapter of the manual.

Table 3-2 I/O Parameter List for Standard I/O Functions

Function	I/O Parameter List
Attach	None needed
Detach	None needed
Kill	None needed
Read Virtual Block and Read Logical Block	word 1 - buffer starting address word 2 - buffer size (in bytes) word 3 - optional timeout count (in 1Ø second intervals) NOTE: Only used if a special sub-function bit is set. See the section on Terminal I/O. words 4, 5, and 6 - unused
Write Virtual Block and Write Logical Block	word 1 - buffer starting address word 2 - buffer size (in bytes) word 3 - vertical format control, as follows (these are the standard FORTRAN carriage control characters):

Octal value	ASCII character	Meaning
Ø4Ø	blank	single space
Ø6Ø	Ø	double space
Ø61	1	form feed
Ø44	$	prompting output-stay in same location after output
Ø53	+	overprint
ØØØ	null	no implied format control - use internal control

words 4, 5, and 6 - unused

THE I/O PARAMETER LIST IN FORTRAN

The parameter list must refer to an integer array declared in a DIMENSION statement with a dimension of six. When used in a QIO directive, the parameter list array name is used without a subscript.

```
    DIMENSION IPAR(6)
    CALL QIO(,,,,,IPAR,)
```

Some entries in the parameter list must be the addresses of arrays or variables. Since FORTRAN does not provide this capability, you must use a system subroutine called GETADR to get the addresses. (See the RSX-11M/M PLUS Executive Reference Manual and the appropriate user's guide for further information. To get the addresses of two variables IBUFF and JBUFF and place the addresses in array K, use the following:

```
    DIMENSION K(2)
    CALL GETADR(K,IBUFF,JBUFF)
```

The address of IBUFF will be in K(1) and the address of JBUFF will be in K(2) at the completion of the CALL GETADR.

ERROR CHECKING AND THE I/O STATUS BLOCK

There are two kinds of errors which can be produced by QIO directives, directive errors and I/O errors. The various directive and I/O status codes and their meanings are listed in Appendix B of the RSX-11M/M-PLUS I/O Drivers Reference Manual and also in the RSX-11M Mini-Reference.

Directive errors are produced due to errors in processing the directive and getting the I/O packet queued up to the device driver. As with other directives, QIO directive errors are indicated by a negative value in the DSW upon return to the task code. Success is indicated by a positive value (typically +1) in the DSW. Thus, the directive status indicates the success or failure of the attempt to queue the I/O packet. Check for directive errors immediately upon return to the task, after the QIO directive is issued.

Upon completion of the I/O transfer itself, the Executive returns status information concerning the I/O transfer to the I/O Status Block laid out as follows:

Device Dependent	I/O Status	word
Actual Number of Bytes Transferred		word

The low-order byte of the first word of the I/O Status Block contains the I/O status code. Note that this is a byte value, not a word value. A positive I/O status code (usually +1) indicates success. Negative values indicate various error conditions. The second word of the I/O status block indicates the number of bytes actually transferred, which is significant in the case of any read or of a write which ends after only some of the data is transferred. The device dependent byte indicates, for reads, the character which was used as a terminating character (<RET>, CTRL/Z, <ESC>, etc.).

The I/O status byte should be checked only after the I/O transfer completes. For synchronous I/O, the I/O status should be checked immediately after checking the DSW, since the I/O transfer itself also completes before control is returned to you. For asynchronous I/O, on the other hand, the I/O status should be checked when the task is notified by the Executive that the transfer is complete. Synchronization is discussed in the following section, after an example of synchronous I/O.

THE QIO DIRECTIVES

Synchronous I/O

The format of the CALL WTQIO is:

```
    CALL WTQIO(ifn,lun,efn,pri,iosb,iopl,ids)
```

where

```
ifn  - I/O function code
lun  - Logical unit number
efn  - Event flag number (required for synchronous I/O)
pri  - Priority (not used but must be present)
iosb - I/O status block address
iopl - I/O parameter list, integer array up to six elements
ids  - Directive status word
```

An event flag must be specified for synchronous I/O. If one is not specified, the I/O request is handled as an asynchronous I/O request. The priority is included to allow compatibility with RSX-11D. It is not used in RSX-11M. The I/O parameter list is a single directive parameter. Hence, the entry must be for an array of up to six elements. Six words are always placed in the DPB for the I/O parameter list, whether or not all six words are specified. It is best to reserve six words. If you do not, you may end up with data from the array(s) defined immediately after the variable defined for the parameter being used in the parameter list for a QIO.

Example 3-1 shows the use of synchronous QIOs. The following notes are keyed to the example.

1. The two-word (four-byte) I/O status block for return of I/O status and the buffer into which the data will be read and from which the data will be displayed. IOSB is declared as a byte array so that the program can examine the I/O status byte in IOSB(1). The program also needs to use the byte count of the number of bytes read by the QIO. This count is found in IOSB(3) and IOSB(4). Since the program needs this as an integer value, the EQUIVALENCE(NUM,IOSB(3)) is used.

   IBUF is the buffer used to hold the characters read by the WTQIO directive.

2. Issue the read request. We are using LUN 5, event flag 1, and IOSB which is the four-byte (two-word) array to receive I/O status after the IORVB. The I/O parameter list is set up as a single parameter (IPAR) which refers to an integer array. IPAR(1) must contain the address of IBUFF which is the buffer into which the characters will be read by the IORVB. Since this is an address, use CALL GETADR to get the address into IPAR(1). IPAR(2) is the maximum buffer size for the IORVB. If input is terminated with a terminating character, such as a carriage return, before 8Ø characters are typed, the number of characters actually read will be returned in the second word of the status block (IOSB(3)). Input will be terminated automatically after the eightieth character, if 8Ø characters are typed. In that case, 8Ø will be returned in the second word of the status block.

3. Check for directive error - failure to queue the I/O packet.

4 With synchronous I/O, the I/O operation has completed when we get control, so also check the I/O status. A value less than Ø indicates an error in the I/O transfer.

5 The count of characters typed in is in NUM (IOSB(3)). Check on and convert only this many characters. Check each character to see if it is in the range ASCII A to ASCII Z. If so, convert to lowercase by adding 32(1Ø)=4Ø(8) to that value, or else continue.

6 Write the buffer BUFF, which has the converted message. This is a Write Virtual Block. The third argument in the I/O parameter list, "4Ø, is for vertical format control. "4Ø, which is an ASCII space, indicates single line feed before writing the line.

7 Check for directive error or I/O error.

NOTE

Although both virtual block and logical block operations are permitted to a terminal, it is safer to use virtual block operations. If the I/O is actually performed at a terminal, the virtual block request gets converted to a logical block request. If logical block writes are used and someone reassigns the LUN to a disk, for example, the write may overwrite a block on the disk. If, on the other hand, someone reassigns the LUN and write virtual blocks are used to a disk, the write will only be allowed if a file is open on the disk, which will fail in most cases if the program is writing to a terminal.

```
        PROGRAM SYNCHQ
C
C FILE SYNCHQ.FTN
C
C This program reads a line of text from the terminal,
C converts any upper case characters to lower case and
C prints the converted message back at the terminal.
C It uses synchronous QIO directives.
C
     ❶  BYTE IOSB(4),IBUF(80)
        DIMENSION IPAR(6)
        EQUIVALENCE (NUM,IOSB(3))
        DATA IOWVB/"11000/
        DATA IORVB/"10400/
        DATA IVFC/"40/
C Set up values for the QIO
        IUNIT=5
        IPAR(2)=80
     ❷  IPAR(3)=IVFC
C Get the address of the I/O buffer
        CALL GETADR(IPAR(1),IBUF(1))
C       Issue the QIO
        CALL WTQIO(IORVB,IUNIT,1,,IOSB,IPAR,IDS)
C Check the directive and I/O statuses
    ❸   IF (IDS .LT. 0) GO TO 800
      ❹ IF (IOSB(1) .LT. 0) GO TO 810
C Check for uppercase characters and convert them to lowercase
        DO 100 I=1,NUM
       [IF (IBUF(I) .LT. 'A') GO TO 100
     ❺ |IF (IBUF(I) .GT. 90) GO TO 100            !Z is 90(10)
       [IBUF(I)=IBUF(I)+32
100     CONTINUE
C Place the number of characters to write in the I/O parameter list
        IPAR(2)=NUM
C Write the converted line to the terminal
     ❻  CALL WTQIO(IOWVB,IUNIT,1,,IOSB,IPAR,IDS)
C Check directive and I/O status
     ❼  IF (IDS .LT. 0) GO TO 820
        IF (IOSB(1) .LT. 0) GO TO 830
        GO TO 850
800     WRITE(5,900)IDS
        GO TO 850
810     WRITE(5,910)IOSB(1)
        GO TO 850
820     WRITE(5,920)IDS
        GO TO 850
```

Example 3-1 Synchronous I/O (Sheet 1 of 2)

```
830     WRITE(5,930)IOSB(1)
850     CALL EXIT
900     FORMAT(' DIRECTIVE ERROR ON READ, CODE = ',I4)
910     FORMAT(' I/O ERROR ON READ, CODE = ',I4)
920     FORMAT(' DIRECTIVE ERROR ON WRITE, CODE = ',I4)
930     FORMAT(' I/O ERROR ON WRITE, CODE = ',I4)
        END
```

Run Session

```
>RUN SYNCHQ
ABCDEFGHIJklmnopqrstuvwxyz12345678[]\
abcdefghijklmnopqrstuvwxyz12345678[]\
>
```

Example 3-1 Synchronous I/O (Sheet 2 of 2)

Asynchronous I/O

The format of the CALL QIO is:

```
        CALL QIO(ifn,lun,efn,pri,iosb,iopl,idsw)
```

where

ifn	- I/O Function code
lun	- Logical Unit Number
efn	- Event Flag Number
pri	- Priority (not used but must be present)
iosb	- I/O Status Block Address
iopl	- I/O Parameter List (up to 6 words)
idsw	- Directive status word

Synchronization With Asynchronous I/O -- As mentioned earlier, event flags may be used for synchronization. If an event flag is specified, the Executive clears the event flag when the I/O packet is queued and sets the flag again when the I/O transfer completes. This happens with both synchronous and asynchronous I/O, if an event flag is specified. With asynchronous I/O, the task can specify a flag and use it for synchronization using one of the following techniques:

1. Do some work, then wait for the flag to be set.

2. Work the entire time, periodically checking the flag until it is set.

A third technique is to monitor the contents of the I/O status byte of the I/O status block. The entire I/O status block is cleared when the I/O request is queued to the driver. Later, it is filled in when the I/O transfer completes. Therefore, the user task can periodically check the contents of the I/O status byte for a nonzero value.

Example 3-2 demonstrates the use of asynchronous I/O to perform the same function performed in Example 3-1. This task can do some work in parallel with the I/O transfer. The following notes are keyed to the example.

1. Issue the read via CALL QIO instead of CALL WTQIO. All arguments are the same as for a CALL QIO. The Executive will clear event flag 1 when the I/O packet is queued and set it when the I/O operation completes.

2. Check for directive errors immediately. Here, we are checking for an error in queueing the I/O packet.

3. While the I/O transfer itself takes place, we can do some work. Here we fill the array at K with the values 64, 128, ...,640.

4. When we are finished with our work, we wait for the event flag specified in the CALL QIO directive. It will be set when the I/O operation completes.

5. Now that the I/O operation is finished, check for I/O errors.

6. After converting the message to lowercase, issue the write.

7. This time, we wait for the flag to be set right after we check the directive status. We could do some more work here. If in fact we are going to just wait, it is simpler and more efficient to use synchronous I/O (WTQIO). Synchronous I/O is more efficient because we perform both functions (QIO and WAIT) in one Executive directive call.

If you use an asynchronous QIO for either reading or writing, you should not use a FORTRAN READ or WRITE to the same lun until you are certain that the QIO has completed.

```
      PROGRAM ASYNCQ
C
C FILE  ASYNCQ.FTN
C
C This program reads a line of text from the terminal,
C converts any upper case characters to lower case and
C prints the converted message back at the terminal.
C It uses asynchronous QIOs and an event flag for
C synchronization.
C
      BYTE IOSB(4),IBUF(80)
      DIMENSION IPAR(6),K(10)
      EQUIVALENCE (NUM,IOSB(3))
      DATA IOWVB/"11000/
      DATA IORVB/"10400/
      DATA IVFC/"40/
C Set up values for the QIO
      IUNIT=5
      IPAR(2)=80
      IPAR(3)=IVFC
C Get the address of the I/O buffer
      CALL GETADR(IPAR(1),IBUF(1))
C Issue the QIO
   ❶  CALL QIO(IORVB,IUNIT,5,,IOSB,IPAR,IDS)
C Check the directive status
   ❷  IF (IDS .LT. 0) GO TO 800
C Do some work while I/O operation is being performed
   ❸  DO 50 I=1,10
      K(I)=64*I
50    CONTINUE
C     Wait for I/O to complete
   ❹  CALL WAITFR(5,IDS)
C     Check directive status
      IF (IDS .LT. 0) GO TO 805
C Check the I/O status
   ❺  IF (IOSB(1) .LT. 0) GO TO 810
C Convert to lowercase
      DO 100 I=1,NUM
      IF (IBUF(I) .LT. 'A') GO TO 100
      IF (IBUF(I) .GT. "132) GO TO 100
      IBUF(I)=IBUF(I)+32
100   CONTINUE
C Set up I/O Parameter List for write
      IPAR(2)=NUM
C Write the converted line to the terminal
   ❻  CALL QIO(IOWVB,IUNIT,5,,IOSB,IPAR,IDS)
C     Check directive status
      IF (IDS .LT. 0) GO TO 820
```

Example 3-2 Asynchronous I/O Using Event Flags for Synchronization (Sheet 1 of 2)

```
C Wait for the I/O to complete
     7  CALL WAITFR(5,IDS)
C Check directive status
        IF (IDS .LT. 0) GO TO 825
C Check the I/O status
        IF (IOSB(1) .LT. 0) GO TO 830
        GO TO 850
800     WRITE(5,900)IDS
        GO TO 850
805     WRITE(5,905)IDS
        GO TO 850
810     WRITE(5,910)IOSB(1)
        GO TO 850
820     WRITE(5,920)IDS
        GO TO 850
825     WRITE(5,925)IDS
        GO TO 850
830     WRITE(5,930)IOSB(1)
850     CALL EXIT
900     FORMAT(' DIRECTIVE ERROR ON READ, CODE = ',I4)
905     FORMAT(' DIRECTIVE ERROR ON 1ST WAIT, CODE = ',I4)
910     FORMAT(' I/O ERROR ON READ, CODE = ',I4)
920     FORMAT(' DIRECTIVE ERROR ON WRITE, CODE = ',I4)
925     FORMAT(' DIRECTIVE ERROR ON 2ND WAIT, CODE = ',I4)
930     FORMAT(' I/O ERROR ON WRITE, CODE = ',I4)
        END
```

Run Session

```
>RUN ASYNCQ
abcdefghKJHKJHKHFRTEWqwryuyiupoZCVcvbvcnbMBNM7(8534243":'
abcdefghkjhkjhkhfrtewqwryuyiupozcvcvbvcnbmbnm7(8534243":'
>
```

Example 3-2 Asynchoronous I/O Using Event Flags for Synchronization (Sheet 2 of 2)

TERMINAL I/O

Device Specific Functions

In the following discussion, references to function codes and subfunction codes are made via the global symbols used when programming in MACRO. This is done because all references in the literature to these codes use the MACRO symbols. Several examples of how to use these in FORTRAN programs are shown below.

Some device specific function codes are listed in Table 3-3, shown below. Table 2-3 in section 2.3 (on the QIO macros) of the RSX-11M/M-PLUS I/O Driver's Reference Manual lists all of the available special functions for the full-duplex terminal driver. As noted, some of these functions are SYSGEN options. Many of the device-specific functions are selected using subfunction codes. These codes may be ORed with standard or device-specific function codes to produce special functions. For instance, the subfunction TF.TMO (read with timeout) may be ORed with a read function such as IO.RLB to produce a function of "read logical block with timeout."

The octal values for IO.RLB and TF.TMO are 1000 and 200, respectively; hence the combination of the two functions is represented by the octal value 1200.

This can coded in FORTRAN as follows:

```
    CALL QIO("1200,,,,,,,)
```

or, to improve readability:

```
    INTEGER TFTMO
    DATA TFTMO/"200/
    DATA IORLB/"1000/
    CALL QIO(IORLB.OR.TFTMO,,,,,,)
```

Another way to produce this function which you may find simpler is:

```
    INTEGER RLBTMO
    DATA RLBTMO/"1200/
    CALL QIO(RLBTMO,,,,,,)
```

Table 2-4 in Chapter 2 of the I/O Driver's Reference Manual lists the various combinations which are possible. For example, TF.TMO (read with timeout) ORed with a read function (IO.RLB, IO.RPR, IO.RNE, etc.) terminates the read if the specified time period goes by between keystrokes. For additional information on the device-specific function codes, see section 2.3.2 (on Device-Specific Functions) in the RSX-11M/M-PLUS I/O Drivers Reference Manual. Examples of the use of Read After Prompt, Read No Echo, and Read With Timeout are included in this module.

Note that if you use subfunction codes with read or write function codes, you should use logical operations rather that virtual; i.e., use IO.RLB and IO.WLB rather than IO.RVB and IO.WVB. The reason for this is that when a virtual operation is requested on a terminal, the Executive converts the operation to a logical operation. In the process, any subfunction codes are lost.

I/O Status Block and Terminating Characters

As for other I/O functions, the low-order byte of the first word of the I/O status block contains the I/O status byte, indicating the success or failure of the I/O operation. Also, the second word contains the count of characters actually transferred. For reads from a terminal, the high-order byte of the first word of the I/O status block contains the terminating character in ASCII (<RET>, CTRL/C, etc.) for successful reads. Normally, CTRL/Z is treated as an error. The I/O status byte is set to IE.EOF (-1Ø.) and the character count contains the count of characters read before the CTRL/Z.

Example 3-4, which follows, shows how CTRL/Z can be handled specially in a program. Two special function codes, IO.RST and IO.RTT, allow reads to be successfully terminated by nonstandard terminating characters. The first allows any non-alphanumeric character to terminate input; the second allows the user to specify which character or characters should terminate the read.

Table 3-3 Some Special Terminal Function Codes

Global Symbol	Octal Value	Function	I/O Parameter List
IO.RNE	ØØ1Ø2Ø	Read With No Echo (Same as IO.RLB!TF.RNE)	<stadr,size[,tmo]>
IO.RPR	ØØ44ØØ	Read After Prompt	<stadr,size,[tmo], pradr,prsize,vfc>
IO.RST	ØØ1ØØ1	Read With Any Special Terminators (Same as IO.RLB!TF.RST)	<stadr,size[,tmo]>
IO.RTT	ØØ5ØØ1	Read With Specified Special Terminators	<stadr,size,[tmo], table>
IO.WBT	ØØØ5ØØ	Write Logical Block, through ongoing I/O (Same as IO.WLB!TF.WBT) Task must be privileged	<stadr,size,vfc>
none	ØØ12ØØ	Read With Timeout (IO.RLB!TF.TMO)	<stadr,size,tmo>

Read After Prompt

The Read After Prompt function allows the combination of a write of prompting text followed by a read in a single QIO request. The I/O parameter list contains six parameters, three for the read, and three for the write. The following notes are keyed to Example 3-3.

1. WTQIO for Read After Prompt. The function code is IO.RPR (4400(8)). The first three parameters in the I/O parameter list are for the read, the last three are for the write. The write is performed first, followed by the read. The 44(8) for the vertical format control causes the prompt text to appear on the next line, followed immediately on the same line by the prompt for the read.

2. Use a normal FORTRAN WRITE to echo the input string.

3. If the operator types a CTRL/Z, an error status is returned. In this case, we just wish to exit normally. Therefore, we must check for this condition and handle it specially.

The ability to use certain function codes, including Read After Prompt, is dependent on whether the option was included in the SYSGEN for your system. Before attempting to use these functions, check with your system manager to see if they are available.

```
       PROGRAM PROMPT
C
C File PROMPT.FTN
C
C This task issues a QIO for READ AFTER PROMPT, echo's it
C and prompts again. This continues until a CNTRL/Z is typed.
C
C
       BYTE     PROM(22)          ! Buffer for prompt string
       BYTE     BUFF(80)          ! READ buffer
       BYTE     IOSB(4)           ! I/O status block
       INTEGER  PARM(6)           ! I/O parameter block
       EQUIVALENCE (NCHAR,IOSB(3)) ! NCHAR is for I/O
C                                   !  count
       DATA     PROM              ! Fill the prompt buffer
       1        /'P','l','e','a','s','e',' ','t','y','p','e',
       2         ' ','a','n','y','t','h','i','n','g',':',' '/
       DATA     IORPR /"4400/     ! Read after prompt
C                                 !  function code
C
C START:
C  Set up params. for QIO
C
       CALL GETADR(PARM(1),BUFF(1)) ! P1 is the address
C                                   !  of BUFF
       CALL GETADR(PARM(4),PROM(1)) ! P4 is the address
C                                   ! of the prompt
       PARM(2) = 80               ! P2 is the length of
C                                 !  the buffer
       PARM(5) = 22               ! P5 is the length of
C                                 !  the prompt
       PARM(6) = 36               ! P6 is the prompt
C                                 !  format control
C
10   ❶ CALL WTQIO(IORPR,5,1,,IOSB,PARM,IDS) ! Invoke QIO
       IF( IDS .LT. 0 ) GO TO 100 ! Directive error?
       IF( IOSB(1) .LT. 0 ) GO TO 110  ! I/O error?
C
     ❷ WRITE (5,15) (BUFF(I),I=1,NCHAR) ! Echo input
C                                       !  string
15     FORMAT( 1X,'You typed: ',80A1 ) ! FORMAT for
C                                      !  echo message
       GO TO 10                        ! Start over
C
100    TYPE *,'Directive error on QIO to READ AFTER
       1PROMPT. DSW = ',IDS      ! Dir error
       CALL EXIT
C I/O error. Check for ^Z
110    IF (IOSB(1) .EQ. -10) GO TO 150 ! Branch on ^Z
     ❸ TYPE *,'I/O error on QIO to READ AFTER PROMPT.
       1 DSW = ',IOSB(1)         ! I/O error
150    CALL EXIT
       END
```

Example 3-3 Prompting for Input (Sheet 1 of 2)

```
Run Session

>RUN PROMPT
Please type anything: sJkshJHGJHGHFY134435
You typed: sJkshJHGJHGHFY134435
Please type anything: hello there
You typed: hello there
Please type anything: ^Z
>
```

Example 3-3 Prompting for Input (Sheet 2 of 2)

Read No Echo

Read No Echo is used to override the default of echoing each character as it is typed. This is used for passwords and other private information. Example 3-4 uses this function. The following notes are keyed to the example.

1. Write prompting text, then leave cursor at that position for input. This is done by having '$' as the first character in the FORMAT.

2. Read No Echo QIO. Standard read parameters except for the function code.

3. As in the previous example, we display the text typed in, preceded by our own message. Since the Read No Echo doesn't echo any characters back and hence doesn't move the cursor on the screen, we precede the text with a carriage return (15(8)) to get the cursor back to the start of the line. Else, the NO LONGER A SECRET WORD message will begin away from the left-hand margin, after the : "SECRET WORD:".

```
        PROGRAM NOECHO
C
C File NOECHO.FTN
C
C This task prompts for input, reads it without echo and
C then skips to the next line and displays the input
C text and exits.
C
C
        BYTE    BUFF(80),IOSB(4),CR(1)
        INTEGER PARM(6)
C
        DATA    IORNE /"01020/          ! QIO Read no echo code
        DATA    CR /"15/                ! Carriage return character
C
     ❶  WRITE (5,1)                     ! write prompt
1       FORMAT ('$SECRET WORD: ')       ! Prompt string
C Set up the I/O parameter list
        CALL GETADR (PARM(1),BUFF(1))   ! buffer address
        PARM(2) = 80                    ! Buffer length
C Issue read no echo
     ❷  CALL WTQIO (IORNE,5,1,,IOSB,PARM,IDS)
        IF (IDS .LT. 0) GO TO 100       ! Dir error?
        IF (IOSB(1) .LT. 0) GO TO 110   ! I/O error?
     ❸  WRITE (5,2) CR,(BUFF(I),I=1,IOSB(3)) ! Echo input
2       FORMAT (' ',A1,'NO LONGER A SECRET WORD: ',80A1)
        CALL EXIT
C
C Error conditions
C
100     TYPE *, 'DIRECTIVE ERROR ON READ. STATUS = ',IDS
        CALL EXIT
110     TYPE *, 'I/O ERROR ON READ. CODE = ',IOSB(1)
        CALL EXIT
        END
```

Run Session

```
>RUN NOECHO
SECRET WORD:
NO LONGER A SECRET WORD: ADD
>
```

Example 3-4 Read No Echo

Read With Timeout

Example 3-5 is a repeat of Example 3-1, but with a timeout on the read. The following notes are keyed to the example. Note 2 is in the run session.

1. To invoke the timeout mechanism, TFTMO is ORed with the read function (IORLB). We must use Read Logical Block here, because any subfunction bits are stripped off when a Read Virtual Block is translated to a Read Logical Block function. In addition, the third parameter in the I/O parameter list specifies the length of the timeout in 1Ø-second intervals. This timeout occurs if that amount of time passes between successive keystrokes. If a timeout occurs, input is terminated, but no error is reported. Instead, the success code +2 is returned rather than the standard +1.

2. In the first run, the QIO timed out after KJHKJjjj. In the second run, the QIO was terminated with a carriage return before it timed out.

   To handle the timeout specially, just check the I/O status byte for a value of +2 (IS.TMO). Another alternative for placing a time limit is to use a Mark Time directive (CALL MARK). The timeout with a Mark Time is for the entire input, rather than for the next keystroke.

```
      PROGRAM QIOTIM
C+
C FILE QIOTIM.FTN
C
C This task reads a line of text from the terminal,
C converts all upper case characters to lower case, and
C prints the converted message back at the terminal. It
C uses synchronous QIOs, with a timeout on the read.
C-
      INTEGER IOSB(2),PLIST(3),DSW,DRCTV
      BYTE BUFF(80),SUCCOD
      EQUIVALENCE (SUCCOD,IOSB) ! Success code is low
C                               !  byte of I/O status
C                               !  block
C MNEMONICS
      INTEGER IORLB,TFTMO,IOWVB
      DATA IORLB,TFTMO,IOWVB/"1000,"200,"11000/
      CALL GETADR(PLIST,BUFF) ! Fill in buffer address
      PLIST(2) = 80           ! Length of buffer
      PLIST(3) = 1            ! Timeout count
    ❶ CALL WTQIO(IORLB.OR.TFTMO,5,1,,IOSB,PLIST,DSW)
C                             ! Issue read
      IF (DSW.LT.0) GOTO 1001 ! Branch on dir error
      IF (SUCCOD.LT.0) GOTO 1011 ! Branch on I/O error
C
      DO 10 I=1,IOSB(2)       ! Get count of characters
C                             !  typed in
C Check for uppercase ASCII character; must be between A
C and Z
      IF (BUFF(I).LT.'A'.OR.BUFF(I).GT.'Z') GOTO 10
C It is upper case, so convert
      BUFF(I) = BUFF(I)+32
10    CONTINUE
      PLIST(2) = IOSB(2)      ! Character count and
      PLIST(3) = "40          ! Format control for
C                             !  output
      CALL WTQIO(IOWVB,5,1,,IOSB,PLIST,DSW) ! Output
C                                           !  results
      IF (DSW.LT.0) GOTO 1002 ! Branch on dir error
      IF (SUCCOD.LT.0) GOTO 1012 ! Branch on I/O error
      CALL EXIT
```

Example 3-5 Read With Timeout (Sheet 1 of 2)

```
C
C Error code
C
1001     DRCTV = 1                    ! Error on 1st QIO
         GOTO 1003                    ! Print message and exit
1002     DRCTV = 2                    ! Error on 2nd QIO
1003     TYPE 1004,DRCTV,DSW
1004     FORMAT (' DIRECTIVE ERROR ON DIRECTIVE #',I2,',
         1 DSW =',I6)
         CALL EXIT
1011     DRCTV = 1                    ! Error on 1st QIO
         GOTO 1013                    ! Print message and exit
1012     DRCTV = 2                    ! Error on 2nd QIO
1013     TYPE 1014,DRCTV,IOSB(1)
1014     FORMAT (' I/O ERROR ON DIRECTIVE #',I2,', I/O
         1CODE =',I6)
         CALL EXIT
         END
```

Run Session

```
   >RUN QIOTIM
   KJHKJJJJ
           kjhkjjjj
2  >RUN QIOTIM
   JJJafhkjfiur<RET>
   jjjafhkjfiur
   >
```

Example 3-5 Read With Timeout (Sheet 2 of 2)

Terminal-Independent Cursor Control

Terminal-independent cursor control is provided if selected at SYSGEN time. If this is done, certain I/O requests are automatically converted for you by the terminal driver for the particular device for which the I/O request is made. This is typically done with escape sequences used for positioning the cursor. This allows a task to move the cursor to any position on the screen and then write a message. This also can be done by imbedding the terminal-specific escape sequences into the write buffer. The advantage of using terminal-independent cursor control is that the same program will work at different terminals (VT-52s and VT-1ØØs, for example), without any need for modification.

To provide cursor control, place the proper value in the vertical forms control word of the I/O parameter list. If the high-order byte in the VFC word is nonzero, the word is interpreted as a cursor position. The high-order byte is the line number, and the low-order byte is the column number. Home position, the upper left corner of the screen, is defined as line 1, column 1. To start the display at line 1Ø, column 25, place a 1Ø in the high-order byte and a 25 in the low-order byte. To do this, use the expression 1Ø*256.OR.25. In general, X*256.OR.Y corresponds to position X,Y on the screen. If the high-order bit in the line number byte is set, the screen is cleared before the cursor is moved.

Example 3-6 demonstrates the use of terminal-independent cursor control. The following notes are keyed to the example.

❶ Issue a Mark Time directive. In the CALL MARK(3,1,3,DSW), the first parameter is EFN 3. The 1 is the time interval magnitude. The second 3 is the time interval unit. A 3 indicates minutes. Hence the directive will set EFN 3 in one minute.

❷ Issue the second Mark Time directive. This one will set event flag 2. It is used as the time interval for updating the time. When the one second goes by and the flag is set, we check for one minute gone by and update the time display again if it has not.

❸ Put the address of TIMMSG into PLIST(1).

❹ Put X (line) in high byte and Y (column) in low byte of PLIST(3), which is the vertical forms control for a QIO. When the high-order byte of the VFC is nonzero, the word is interpreted as a cursor position.

❺ Issue the write. The vertical format control (X*256).OR.Y places the cursor before the write at line X, column Y. The TF.RCU subfunction code (TFRCU) instructs the terminal driver to save the cursor position before moving it and then to restore it after writing the message. This allows an operator to continue typing in commands while this task runs.

❻ Wait for one second to go by.

❼ Read event flag 3. If it is set, the one minute is up and we should exit.

The display will actually appear at line X, column Y on the screen, and is updated every second.

```
          PROGRAM DATTIM
C+
C FILE DATTIM.FTN
C
C This task places the date and the time at line X,
C column Y and then updates the display every Z seconds
C for 1 minute.
C-
          INTEGER X,Y,Z
          DATA X,Y,Z/5,32,1/
          INTEGER DSW,IOSB(2),PLIST(3)
          BYTE SUCCOD,TIMMSG(18)
          EQUIVALENCE (SUCCOD,IOSB) ! Low byte of status
C                                   !  block is success code
          INTEGER IOWLB,TFRCU,ISCLR
          DATA IOWLB,TFRCU,ISCLR/"400,1,0/
C
      ❶   CALL MARK(3,1,3,DSW)     ! Set up to exit after
                                   !  1 minute
          IF (DSW.LT.0) GOTO 1001  ! Branch on dir error
10    ❷   CALL MARK(2,Z,2,DSW)     ! Set event flag 2 after
                                   !  Z seconds
          IF (DSW.LT.0) GOTO 1002
          CALL DATE(TIMMSG)        ! Get date in bytes 1-9
          TIMMSG(10) = ' '         ! Insert space between
                                   !  date and time
          CALL TIME(TIMMSG(11))    ! Get time in byte 11-18
      ❸   CALL GETADR(PLIST,TIMMSG)
          PLIST(2) = 18
      ❹   PLIST(3) = (X*256).OR.Y
C Display time
      ❺   CALL WTQIO(IOWLB.OR.TFRCU,5,1,,IOSB,PLIST,DSW)
          IF (DSW.LT.0) GOTO 1003 ! Branch on dir error
          IF (SUCCOD.LT.0) GOTO 1004   ! Branch on I/O
                                   !  error
      ❻   CALL WAITFR(2,DSW)       ! Wait for mark time to
                                   !  expire
          IF (DSW.LT.0) GOTO 1005 ! Branch on dir error
C Check for 1 minute gone by
      ❼   CALL READEF(3,DSW)       ! Check event flag
          IF (DSW.LT.0) GOTO 1006 ! Branch on dir error
          IF (DSW.EQ.ISCLR) GOTO 10   ! Check for flag
C                                  !  already clear. If
C                                  !  clear, minute
C                                  !  not up yet, update
C                                  !  display again
          GOTO 1100
```

Example 3-6 Terminal-Independent Cursor Control (Sheet 1 of 2)

```
C Error code
1001    WRITE (5,1051) DSW
        GOTO 1100
1002    WRITE (5,1052) DSW
        GOTO 1100
1003    WRITE (5,1053) DSW
        GOTO 1100
1004    WRITE (5,1054) SUCCOD
        GOTO 1100
1005    WRITE (5,1055) DSW
        GOTO 1100
1006    WRITE (5,1056) DSW
        GOTO 1100
1051    FORMAT (' ERROR ON MARK TIME FOR 1 MINUTE. DSW =
        1 ',I5)
1052    FORMAT (' ERROR ON MARK TIME FOR 1 SECOND. DSW =
        1 ',I5)
1053    FORMAT (' DIRECTIVE ERROR ON WRITE. DSW = ',I5)
1054    FORMAT (' I/O ERROR ON WRITE. CODE = ',I5)
1055    FORMAT (' ERROR ON WAIT FOR. DSW = ',I5)
1056    FORMAT (' ERROR ON CLEAR EVENT FLAG. DSW = ',I5)
1100    CALL EXIT
        END
```

```
Run Session

                          12-MAR-82 11:12:54
>RUN DATTIM       ! DISPLAY WILL START AT LINE 5, COLUMN 32
```

Example 3-6 Terminal-Independent Cursor Control (Sheet 2 of 2)

Now do the tests/exercises for this module in the Tests/Exercises book. They are all lab problems. Check your answers against the solutions provided, either in that book or in on-line files.

If you think that you have mastered the material, ask your course administrator to record your progress in your Personal Progress Plotter. You will then be ready to begin a new module.

If you think that you have not yet mastered the material, return to this module for further study.

USING DIRECTIVES FOR INTERTASK COMMUNICATION

INTRODUCTION

The RSX-11M program development features allow modular development of programs; the multitasking feature allows a modular approach to applications.

A system of multiple tasks may require one or more of the following services provided by executive directives under RSX-11M.

- First task requests that the second task be run.
- First task is notified of completion of the second task operation.
- Tasks pass data to each other.

This module explains how to use system directives for this type of coordination between tasks.

OBJECTIVES

- To use directives which control task execution to synchronize cooperating tasks
- To use the send/receive directives to pass data between tasks
- To write tasks which spawn subtasks using parent/offspring directives

RESOURCE

- RSX-11M/M-PLUS Executive Reference Manual, Chapters 2 and 4 plus specific directives in Chapter 5

USING TASK CONTROL DIRECTIVES AND EVENT FLAGS

It is generally good programming practice to divide a single complex task into a number of separate tasks, with each task performing a distinct logical function. The use of a group of tasks to perform a complex function frequently makes good sense, especially where different parts of the process may run at widely differing speeds, each more or less independent of the others.

Suppose, for instance, that one needs to simulate customer transactions at a bank. There are, say, five windows and up to 15 customers can physically stand in line at a time, given the size of the waiting area. One might design a group of tasks, one task per line, to simulate this complex system. This approach has the advantage of simulating the related, but essentially parallel, processes in a more realistic manner than would a single, serial, simulation. A further advantage of a multitasking approach to such a job is that changes in the behavior of the system that are caused by changes in a single line (e.g., by assigning different sorts of transactions to different lines) can be easily simulated merely by modifying the task that simulates that line.

An RSX-11M programmer typically uses a mix of four multitasking methods:

- Common or group global event flags, together with synchronization and task scheduling directives, are used to synchronize tasks.
- Resident commons are used to share data in memory.
- Memory management directives are used to create and/or share data areas dynamically at run time.
- File handling routines are used to open disk files for shared access.

The use of shared regions, memory management directives and files are covered in later modules.

Directives

Table 4-1 lists the various task control directives which are available for task synchronization. Most of them were discussed in earlier modules. All of the directives are documented individually in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual.

Table 4-2 shows the differences between suspending and stopping a task. The major difference is that stopping puts the task into a stopped state which effectively lowers the task priority to Ø, allowing any active task to checkpoint it if it is checkpointable. Suspending or waiting, on the other hand, keeps the task competing for memory space on the basis of its running priority. This means that if the task is checkpointable, only tasks of higher priority can checkpoint it. Waiting for an event flag affects checkpointability the same way as suspending.

Table 4-3 lists the various event flag directives which are available for synchronization.

Table 4-1 Task Control Directives and Their Use for Synchronizing Tasks

Directive	Example of Use for Synchronization
FORTRAN CALL	
REQUES RUN	Issuing task activates target task; target task then performs some operation for issuing task
ABORT	Issuing task aborts target task
SUSPND STOP	A task suspends or stops itself to wait for completion of another task operation A task suspends or stops itself until it is needed by another task
RESUME USTP	A task resumes or unstops another task which has suspended or stopped itself while waiting for it to complete some operation A task resumes or unstops another task when it needs the other task's services A task can also be resumed: - by its own AST routine - by an operator using a DCL CONTINUE command (RESUME in MCR) A task can also be unstopped: - by its own AST routine - by an operator using a DCL START command (UNS in MCR)

Table 4-2 Stopping Compared to Suspending or Waiting

Stopping	**Suspending or Waiting**
Priority is effectively dropped to Ø	Priority remains unchanged
Task can be checkpointed by any other task (if checkpointable)	Task can be checkpointed only by tasks of higher priority
Likelihood of being checkpointed increases.	Likelihood of being check-pointed remains normal
Frees memory for other tasks	Continued allocation of memory can block out lower priority tasks
Task response time increases dramatically if task is checkpointed	No change in task response time, because no change in likelihood of being checkpointed

Table 4-3 Event Flag Directives and
Their Use for Synchronizing Tasks

Directive	**Example of Use for Synchronization**
FORTRAN CALL	
CLREF	A task clears the event flag, then waits for it to be set by another task
SETEF	A task performing an operation for another task sets an event flag to signify completion of the operation
WAITFR STOPFR	A task waits for completion of an operation by another task by waiting or stopping for that task to set an event flag
STLOR WFLOR	A task waits or stops for the completion of the first of some set of operations
READEF	A task tests for completion of an operation by another task, without waiting or stopping for it

Example 4-1 shows the use of the Request Task (REQUES), Suspend (SUSPND), and Resume (RESUME) directives for synchronization. The following notes are keyed to the example. Notes 1,2 and 5 are in TASKA. Notes 3 and 4 are in TASKB.

❶ TASKA requests TASKB. This means that TASKB must be installed under the name TASKB. After this, both tasks are active and compete for memory and CPU time.

❷ TASKA suspends itself. After this it still competes for memory at its regular priority, but not for CPU time.

❸ TASKB types out a message and then resumes TASKA. More typically, TASKB would perform some service for TASKA rather than just typing a message. After TASKB resumes TASKA, they both compete for CPU time again.

❹ TASKB displays another message and then exits.

❺ TASKA, now resumed, displays a message and exits.

Depending on the relative priorities of TASKA and TASKB and on the particular task scheduling options on your system (e.g., round robin scheduling, etc.), steps 4 and 5 may be reversed on the run session.

```
        PROGRAM TASKA
C
C FILE TASKA.FTN
C
C This task requests TASKB to run, and then suspends
C itself. TASKB resumes this task and exits.
C
C Install and run instructions: TASKA and TASKB must be
C installed. Just run TASKA.
C
        INTEGER DSW
        DATA TASKB/5RTASKB/
C
        TYPE *,'TASKA BEGINS AND REQUESTS TASKB'
    1   CALL REQUES(TASKB,,DSW)
        IF (DSW.LT.0) GOTO 900
C
        TYPE *,'TASKA IS SUSPENDING ITSELF'
    2   CALL SUSPND(DSW)
        IF (DSW.LT.0) GOTO 910
C
       [TYPE *,'TASKA HAS BEEN RESUMED'
    5  [CALL EXIT
900     TYPE *,'TASKA UNABLE TO REQUEST TASKB. DSW =
        1,DSW
        GOTO 1000
910     TYPE *,'TASKA UNABLE TO SUSPEND. DSW = ',DSW
1000    CALL EXIT
        END
```

Example 4-1 Synchronizing Tasks Using Suspend and Resume (Sheet 1 of 2)

```
        PROGRAM TASKB
C
C FILE TASKB.FTN
C
C This task is activated by TASKA.  It performs its
C operation and resumes TASKA, which has suspended
C itself.
C
        INTEGER DSW
        DATA TASKA/5RTASKA/
C
C START:
C Any operation could be performed here, but in this
C case it's only a typeout.
C
     ❸ [TYPE *,'TASKB IS ALIVE AND RUNNING'
       [CALL RESUME(TASKA,DSW)
        IF (DSW.LT.0) GOTO 900
     ❹ [TYPE *,'TASKB HAS RESUMED TASKA AND IS EXITING'
       [CALL EXIT
900     TYPE *,'TASKB UNABLE TO RESUME TASKA. DSW = ',DSW
        CALL EXIT
        END
```

Run Session

```
>INS TASKA
>INS TASKB
>RUN TASKA
>
 TASKA BEGINS AND REQUESTS TASKB
TASKA IS SUSPENDING ITSELF
TASKB IS ALIVE AND RUNNING
TASKA HAS BEEN RESUMED
TASKB HAS RESUMED TASKA AND IS EXITING
```

Example 4-1 Synchronizing Tasks Using Suspend and Resume
(Sheet 2 of 2)

Example 4-2 shows the use of event flags for synchronization. In Module 2, there is a similar example. Here, TASKC requests TASKD, rather than requiring an operator to start both tasks. Also, Stop For Single Event Flag is used rather than Wait For Single Event Flag. The difference between the two is that the first causes the task to enter a stopped state and the other causes the task to enter a Wait For (like a suspended) state. The following notes are keyed to the example. Notes 1,2,3 and 6 are in TASKC. Notes 4 and 5 are in TASKD.

1. Clear the event flag to initialize it. It's initial state is unpredictable, since other tasks may have set or cleared it.

2. Request TASKD.

3. Stop until the event flag is set by TASKD.

4. TASKD displays a message and sets the event flag.

5. TASKD displays a message and exits.

6. TASKC displays a message and exits.

Depending on the relative priorities of the two tasks, significant events in the system, and other scheduling considerations, the order of the steps may vary. In particular, steps 3 and 4 above may be reversed, as well as 5 and 6 .

The event flag must be a common or group global, and not a local one. In either case, the users on the system must coordinate to avoid several users using the same event flag for different purposes. If a group global event flag is used, the flags for that group may have to be created using either the Create Group Global Event Flags directive (CRGF) or the DCL SET GROUPFLAGS/CREATE (FLA /CRE in MCR) command.

The Executive only scans the Active Task List and schedules tasks for CPU time after a significant event. Setting an event flag does <u>not</u> cause a significant event. This means that TASKC normally won't compete for CPU time until at least the next significant event in the system. If it is important that TASKC being executing sooner than that, TASKD should issue the Declare Significant Event directive (DECLAR), causing the Executive to reschedule tasks. For a discussion of significant events, see Chapter 2 of the <u>RSX-11M/M-PLUS Executive Reference Manual</u>.

```
      PROGRAM TASKC
C
C FILE TASKC.FTN
C
C This task clears an event flag and requests TASKD to
C run, and then stops until the event flag is set by
C TASKD
C
C Install and run instructions: TASKD must be installed.
C Just run TASKC.
C
      INTEGER DSW,FLAG
      DATA FLAG/33/              !MNEMONIC FOR EVENT FLAG
      DATA TASKD/5RTASKD/
C
      TYPE *,'TASKC BEGINS AND REQUESTS TASKD'
C
  1   CALL CLREF(FLAG,DSW)
      IF (DSW.LT.0) TYPE *,'TASKC UNABLE TO INITIALIZE
      1EVENT FLAG. DSW = ',DSW
C
  2   CALL REQUES(TASKD,,DSW)
      IF (DSW.LT.0) TYPE *,'TASKC UNABLE TO REQUEST
      1TASKD. DSW = ',DSW
C
      TYPE *,'TASKC IS WAITING FOR EVENT FLAG'
  3   CALL STOPFR(FLAG,DSW)
      IF (DSW.LT.0) TYPE *,'TASKC''S WAIT REQUEST
      1REJECTED. DSW = ',DSW
C
     [TYPE *,'TASKC HAS BEEN UNSTOPPED AND WILL NOW
  6  [1EXIT'
     [CALL EXIT
      END
```

Example 4-2 Synchronizing Tasks Using Event Flags (Sheet 1 of 2)

```
        PROGRAM TASKD
C
C FILE TASKD.FTN
C
C This task is activated by TASKC.  It performs its
C operation and sets the flag for which TASKC is waiting
C
        INTEGER DSW,FLAG
        DATA FLAG/33/               !MNEMONIC FOR EVENT FLAG
C
C START:
C
C Any operation could be performed here, but in this
C case it's only a typeout.
C
    4   TYPE *,'TASKD IS ALIVE AND RUNNING'
        CALL SETEF(FLAG,DSW)
        IF (DSW.LT.0) TYPE *,'TASKD UNABLE TO SET EVENT
        1FLAG. DSW = ',DSW
        TYPE *,'TASKD HAS SET THE EVENT FLAG AND IS
    5   1EXITING'
        CALL EXIT
        END
```

Run Session

```
>INS TASKC
>RUN TASKC
TASKC BEGINS AND REQUESTS TASKD
TASKC IS STOPPING FOR EVENT FLAG
TASKD IS ALIVE AND RUNNING
TASKD HAS SET THE EVENT FLAG AND IS EXITING
TASKC HAS BEEN UNSTOPPED AND WILL NOW EXIT
>
```

Example 4-2 Synchronizing Tasks Using Event Flags
(Sheet 2 of 2)

SEND/RECEIVE DIRECTIVES

General Concepts

The Send and Receive directives are used to transmit a 13 word block of data between tasks. The sequence of events is as follows:

1. A task issues a Send Data request, specifying a receiver task and a data buffer.
2. The Executive copies the data buffer into a data packet in the dynamic storage region (DSR or pool).
3. The Executive places the data packet FIFO (first-in-first-out) into the receive queue of the specified receiving task.
4. Later, the receiving task issues a Receive Data request, specifying a data buffer.
5. The Executive copies the data packet into the buffer specified by the receiving task.

Directives

Table 4-4 lists the Send Data directive and the various Receive Data directives. The difference among the Receive Data directives concerns what happens if there are no data packets in the receiver's receive queue.

All receive directives receive 15(1Ø) words, including the sender task name (in Radix-5Ø format) plus the data. If no sender task is specified in a Receive Data directive, the first packet in the receive queue is dequeued, regardless of which task sent it. If a sender task is specified, only a packet sent by that task is dequeued.

Table 4-4 The Send/Receive Data Directive

Directive Name	Directive Call	Notes
Send Data	SEND	Sends a 13(1Ø) word buffer to receiver Event flag (if used) set when packet queued to receiver
Receive Data	RECEIV	Error if no data packets queued
Receive Data or Exit	RECOEX	Exit if no data packets queued
Receive Data of Stop	RCST	Stop if no data packets queued

Synchronizing Send Requests With Receive Requests

Event flags can be used for synchronization. The event flag is specified by the sending task. This event flag is set when the data packet has been queued to the receiving task. Thus, a global or group global event flag may be used to unblock a receiving task which is active and waiting for the event flag to be set.

In addition, the task control directives can be used for synchronization. Table 4-5 summarizes the various synchronization techniques which might be used. Keep in mind that a Receive Data directive (RECEIV) causes an error condition (DSW = -8, IE.ITS; directive inconsistent with task state) if there is no data packet in the receive queue. Receive Data or Stop (RCST) and Receive Data or Exit (RECOEX), on the other hand, cause the task to stop or exit, respectively, if there is no data queued. For further information about possible synchronization problems, see the writeup on the Receive Data directive (RECEIV) in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual.

Table 4-5 Methods of Synchronizing a Receiving Task (RECEIV) with a Sending Task (SEND)

Method	Advantages	Disadvantages
RECEIV issues a Wait For or a Stop For Event Flag directive, followed by a Receive directive. SEND uses that (common or group global) event flag in its SEND directive.	Low system scheduling overhead.	Requires care in initializing and setting flag. (See Examples 4-3 and 4-4.)
RECEIV issues a Suspend or a Stop directive followed by a Receive directive. SEND issues a Send directive followed by a Resume or an Unstop directive.	Does not require an event flag.	Possible problems in sequence of Suspend or Stop, and Resume or Unstop, if the Resume Unstop is issued before the receive suspends or stops.
RECEIV issues a Receive Data or Stop directive. SEND issues a Send followed by an Unstop directive.	Low system scheduling overhead. Does not require an event flag.	Possible delay starting RECEIV again, if RECEIV was checkpointed. (Can be avoided if RECEIV is built non-checkpointable.)
RECEIV monitors an event flag periodically. When the event flag is set, RECEIV issues the Receive directive. SEND specifies that event flag in its Send directive.	RECEIV can continue processing in parallel with RECEIV.	RECEIV must periodically reissue a Read Event Flag or Clear Event Flag directive. Requires care in initializing and setting the flag.

Examples 4-3, 4-4, and 4-5 show the use of Send and Receive directives by a pair of tasks. Examples 4-3 and 4-4 use an event flag for synchronization; Example 4-5 uses Receive Data or Stop along with Unstop for synchronization. The following notes are keyed to Example 4-3. Notes 1, 5, 6 and 7 are in SEND1. Notes 2, 3, 4, 8, 9 and 1Ø are in RECV1.

1. RECV1 must be run first, or else the event flag will already be set by SEND1 to indicate that a data packet has been sent. In that case, RECV1 will clear the flag and wait for it to be set again, and won't realize that a data packet is already queued to it.

2. Use a DO loop with "I" as the message counter. We will receive and display three messages and then exit.

3. Initialize the event flag.

4. Wait for the flag to be set after SEND1 sends the data packet, placing it in RECV1's receive queue.

5. Get the data to be sent.

6. Send the data and set event flag 33 when the data packet is queued to RECV1.

7. SEND1 exits.

8. Receive data from anyone.

9. Display a header and the data sent. We skip the first two words (four-bytes) of the buffer, which contain the name of the sender task in Radix-5Ø format.

10. Go through the loop which clears the event flag and receives again if we have not yet received three messages. If we have, display a message and exit.

```
          PROGRAM SEND1
C
C FILE SEND1.FTN
C
C This task prompts at TI: for a line of text and sends
C the data to RECV1 for processing.  Synchronization is
C handled through a common event flag.
C
C Install and run instructions: RECV1 must be installed
C and run prior to running SEND1. RECV1 continues to run     1
C until it receives 3 data packets.
C
          BYTE BUFFER(26)
          DATA IEFN /33/            ! Event flag
          DATA RTASK/6RRECV1 /      ! Receiver task
C Prompt for input
          TYPE *,'TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS'        5
          READ (5,10) BUFFER        ! Read text
10        FORMAT (26A1)
          CALL SEND (RTASK,BUFFER,IEFN,IDSW) ! Send data
          IF (IDSW .LT. 0) GOTO 900 ! Branch on dir error            6 7
          CALL EXIT                 ! Exit
C Error code
900       TYPE *,'UNABLE TO QUEUE DATA TO RECV1. DSW = ',IDSW
          CALL EXIT
          END
```

Example 4-3 Synchronizing a Receiving Task Using Event Flags
(Sheet 1 of 3)

```
            PROGRAM RECV1
C
C FILE RECV1.FTN
C
C This task receives data from any sender task (e.g.,
C RECV1). It prints the data on TI:.  Then it waits for
C another data packet. It does this until it has received
C 3 messages and then exits.
C
C This task synchronizes with its sender through an
C event flag.
C
C Install and run instructions: RECV1 must be installed
C and run before running SEND1.
C
            INTEGER RBUFF(15)          ! Receive buffer
            DATA IEFN /33/             ! Event flag
C
      ❷ ❸ [DO 100 I=1,3
           [CALL CLREF (33,IDSW)      ! Clear flag
            IF (IDSW .GE. 0) GOTO 10
            TYPE *,'ERROR INITIALIZING FLAG. DSW = ',IDSW
            GOTO 1000
10        ❹ CALL WAITFR (33,IDSW)     ! Wait for a send
            IF (IDSW .EQ. 1) GOTO 20
            TYPE *,'WAIT DIRECTIVE FAILED. DSW = ',IDSW
            GOTO 1000
20        ❽ CALL RECEIV (,RBUFF,,IDSW) ! Receive from anyone
            IF (IDSW .EQ. 1) GOTO 30
            TYPE *,'RECEIVE DIRECTIVE FAILED IN "RECV1".
            1 DSW = ',IDSW
            GOTO 1000
30         [TYPE *,'DATA RECEIVED BY "RECV1":'
          ❾[WRITE (5,35) (RBUFF(K),K=3,15)
35          FORMAT (' ',13A2)
100       ❿ CONTINUE
            TYPE *,'"RECV1" HAS RECEIVED 3 MESSAGES AND WILL
            1 NOW EXIT'
1000        CALL EXIT
            END
```

Example 4-3 Synchronizing a Receiving Task Using Event Flags (Sheet 2 of 3)

```
Run Session

>INS RECV1
>RUN RECV1
>RUN SEND1
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
1111111
DATA RECEIVED BY "RECV1":
>
 1111111
>RUN SEND1
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
2222222222222222
DATA RECEIVED BY "RECV1":
>
 2222222222222222
>RUN SEND1
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
33333333333333333333333333
DATA RECEIVED BY "RECV1":
>
 33333333333333333333333333
"RECV1" HAS RECEIVED 3 MESSAGES AND WILL NOW EXIT
>
```

Example 4-3 Synchronizing a Receiving Task Using Event Flags
(Sheet 3 of 3)

If you wish to run the tasks in Example 4-3 in any order, RECV1 must be modified to receive data packets on startup if SEND1 has already sent data. It gets complicated because SEND1 may have already sent several data packets. It's also possible that event flag 33. was left set by someone else. In that case the Receive directive will fail, but we should not abort. Example 4-4 shows the modifications which must be made to Example 4-3 to allow the tasks to be run in any order. The following notes are keyed to Example 4-4.

1. We use a flag word (IBEF) to distinguish whether we are working on messages sent before or after RECV1 starts up. Note that RECV1S must be installed as RECV1, since SEND1 sends to RECV1.

2. Check for event flag set on startup. If it is set, issue a Receive. If SEND1 has been run one or more times, the Receive will succeed. If SEND1 has not been run yet, the flag was set by another task and the Receive will fail.

3. If the flag was not set, SEND1 hasn't sent any messages before we started. Clear the IBEF flag, so we know that a Receive failure after the flag is set again is a real failure.

4. In the case of a Receive failure, we check to see if we are receiving data packets sent before RECV1 started up. If we are, we know we have received all data packets already queued up before RECV1 started executing.

5. If IBEF is clear, this was a failure after receiving all data packets sent before RECV2 started up, so display an error message and exit.

6. If IBEF is set, we have already received all data packets which were queued up before RECV1 started up. Now clear IBEF and wait for the flag to be set at 40.

7. Check to see if we are still receiving data packets sent before RECV1 started up. If so, Receive again. Keep receiving until either we get all three packets or we get a Receive failure. If, on the other hand, we have received a message sent since startup, clear the flag and wait for it to be set again when a new message is sent.

If a task runs and then exits with data packets in its receive queue, those unreceived data packets are flushed from the queue on exit. Hence, if SEND1 sent four messages before RECV1 was run, the fourth message would be lost.

```
      PROGRAM RECV1S
C
C FILE RECV1S.FTN
C
C This task and receives data from any sender task
C (e.g. SEND1). It prints the data on TI:.  Then it
C waits for another data packet. It exits after
C receiving and displaying 3 messages.
C
C This task synchronizes with its sender through an
C event flag. Because of this synchronization, and the
C care we take on startup to get messages already
C sent, the tasks can be run in any order, with any
C relative priorities.
C
C Install and run instructions: RECV1S must be installed
C under the name RECV1 to work with SEND1.
C
❶C IBEF is the "before" flag, used to keep track of whether
C we are receiving messages sent before RECV1 started up.
C If the event flag is set at startup time, keep receiving
C messages until we get a failure. We then wait until the
C flag is set to receive again. 1 means receiving messages
C sent before RECV1 started, 0 means finished receiving
C messages sent before
      INTEGER IBEF,IEFN,RBUFF(15),MCNT
      DATA IEFN /33/            ! Event flag
      DATA IBEF /1/             ! Before flag, assume
C                               !  there are messages
      DATA MCNT /3/             ! Message counter
      CALL CLREF (IEFN,IDSW)
   ❷  IF (IDSW .LT. 0) GOTO 900 ! Branch on dir error
      IF (IDSW .EQ. 2) GOTO 50  ! Branch if flag set
C Here if flag not initially set
   ❸  IF (IBEF .EQ. 0) GOTO 40
      IBEF=0                    ! 0 before flag
40    CALL WAITFR (33,IDSW)     ! Wait for next message
      IF (IDSW .LT. 0) GOTO 910 ! Branch on dir error
C Get here when the flag is set
50    CALL RECEIV (,RBUFF,,IDSW) ! Receive from anyone
      IF (IDSW .EQ. 1) GOTO 80 ! Branch on directive ok
C Here on failure of Receive directive
   ❹  IF (IBEF .EQ. 0) GOTO 920 ! Check for failure
C                               ! on messages received
C                               ! before startup.
C Here if failure after receiving messages already there
C at startup
      IBEF=0                    ! Clear before flag
      GOTO 40                   ! Wait for flag to be set
```

Example 4-4 A Receiving Task Which Can be Run Before or After the Sender (Sheet 1 of 3)

```
C Successful receipt
80        TYPE *,'DATA RECEIVED BY "RECV1":'
          WRITE (5,85) (RBUFF(K),K=3,15)
85     7  FORMAT (' ',13A2)
          MCNT=MCNT-1              ! Decrement message counter
          IF (MCNT .EQ. 0) GOTO 100 ! Branch back if not done
C Set up for another receive
          IF (IBEF .NE. 0) GOTO 50 ! Check for still
C                                  !  receiving messages sent
C                                  !  before startup. If so,
C      6                           !  receive again.
          CALL CLREF (33,IDSW)     ! If not, clear flag
          IF (IDSW .LT. 0) GOTO 930 ! Branch on dir error
          GOTO 40                  ! Wait for flag set again


C Here when three messages received
100       TYPE *,'"RECV1" HAS RECEIVED 3 MESSAGES AND WILL
          1 NOW EXIT'
          CALL EXIT
C Error code
900       TYPE *,'ERROR INITIALIZING FLAG. DSW = ',IDSW
          GOTO 1000
910       TYPE *,'WAIT DIRECTIVE FAILED. DSW = ',IDSW
          GOTO 1000
920       TYPE *,'RECEIVE DIRECTIVE FAILED IN "RECV1". DSW
       5  1 = ',IDSW
          GOTO 1000
930       TYPE *,'ERROR RECLEARING FLAG. DSW = ',IDSW
1000      CALL EXIT
          END
```

Example 4-4 A Receiving Task Which Can be Run Before or After the Sender (Sheet 2 of 3)

```
Run Session

>INS/TASK_NAME:RECV1 RECV1S
>RUN SEND1
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
1111 11
>RUN SEND1
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
2222222222
>RUN RECV1
>
DATA RECEIVED BY "RECV1":
1111 11
DATA RECEIVED BY "RECV1":
2222222222
RUN SEND1
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
33333
>
DATA RECEIVED BY "RECV1":
33333
"RECV1" HAS RECEIVED 3 MESSAGES AND WILL NOW EXIT
>
```

Example 4-4 A Receiving Task Which Can be Run Before or After the Sender (Sheet 3 of 3)

Example 4-5 uses Receive Data or Stop in the Receiver and Send Data followed by Unstop in the sender. These tasks can be run in any order. The potential synchronization problems are considerably easier to deal with when using this technique of synchronization. We will go through it first for running RECV2 before running SEND2. Then we will discuss the other possibilities. The following notes are keyed to the example.

❶ We issue a Receive Data or Stop directive. If there is no data packet queued, RECV2 stops and must be unstopped by SEND1. If, on the other hand, there is a data packet queued, we want to receive it. The DSW equals IS.SET(+2) if the task was stopped and then unstopped, and equals IS.SUC(+1) if a data packet was received. If RECV2 is run first, we stop.

❷ SEND2 gets the data and sends it. We do not need to specify an event flag in the Send Data directive since we use Stop/Unstop for synchronization.

❸ Unstop RECV2. In this order, this directive will successfully unstop RECV2 because RECV2 stopped when it issued the Receive Data or Stop directive.

❹ There are two directive errors on UNSTOP which are not errors for this set of tasks. Check for these errors and if found, assume that everything worked correctly. If RECV2 is active but not stopped, it must be receiving another packet. In that case, RECV2 will receive this packet on the next Receive Data or Stop directive. If RECV2 is not active, it has not been run yet. The packet is still in RECV2's receive queue and RECV2 will receive it when it is activated. The above situation will not occur the first time through if RECV2 is run first.

If the error is not one of the two errors checked for, display an error message and exit.

❺ Check for whether we stopped and unstopped. If so, we didn't receive the data packet yet. If not, we did receive the data. In this case, if RECV2 is run first, we did stop and unstop.

❻ Since we have not yet received the data packet, issue another Receive.

❼ If there is still nothing in the Receive queue, something is wrong. Display an error message and exit.

8 After a successful Receive, whether immediately or after Stop and Unstop, display the received message. In that case, issue another Receive Data or Stop and loop through again if we have not yet received three messages. If there is another data packet queued, we will receive it. Otherwise, we stop until SEND2 sends data and unstops us again.

If SEND2 is run once before RECV2, then the Unstop directive at 3 will fail. If in fact RECV2 is not active at all, or is active but not stopped, it will dequeue the data packet when it issues a Receive. Hence, we check for these conditions at 4 and just exit if either condition caused the Unstop error. When we run RECV2, we do actually receive a data packet at 1. At 5, DSW = +1(IS.SUC) which means that we received a packet and didn't stop. Therefore, we display the data and Receive or Stop again. This time we will stop until SEND1 unstops us again.

If SEND2 is run two or three times before RECV2, any data packets already sent are received and displayed. In the case of two sent, the third RCDS will cause RECV2 to stop until SEND2 sends a third packet and unstops it. In the case of three packets already sent, RECV2 will receive all three and then exit.

As in Example 4-4, if SEND2 sends more than three packets, any additional packets will be lost because the receive queue is flushed when the task exits.

```
      PROGRAM SEND2
C
C FILE SEND2.FTN
C
C This task prompts at TI: for a line of text and sends
C the data to RECV2 for processing.  The receiver will
C continue to run until it receives 3 messages.
C Synchronization is handled through RECV2's stop bit.
C RECV2 and SEND2 may be run in any order.
C
C Install and run instructions: RECV2 must be installed.
C
      BYTE BUFFER(26)          ! Send buffer
      INTEGER DSW
      REAL RECV2
      DATA RECV2/5RRECV2/      ! Receiving task name
      INTEGER IEITS,IEACT      ! Error mnemonics
      DATA IEITS,IEACT/-8,-7/
C
      TYPE *,'TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS'
    ❷ READ (5,5) BUFFER
5     FORMAT (26A1)
      CALL SEND(RECV2,BUFFER,,DSW) ! Send data to RECV2
      IF (DSW.EQ.1) GOTO 10
      TYPE *,'UNABLE TO QUEUE DATA TO "RECV2". DSW = '
     1,DSW
10  ❸ CALL USTP(RECV2,DSW)     ! Unstop RECV2
      IF (DSW.EQ.1) GOTO 20    ! Branch on directive ok
      IF (DSW.EQ.IEITS) GOTO 20 ! Isn't he stopped?
C                              !  That's ok, he'll pick
C                              !  up data when he
C                              !  executes RCDS$
    ❹ IF (DSW.EQ.IEACT) GOTO 20 ! Is he not active? If
C                              !  not, he'll pick up
C                              !  data when activated
      TYPE *,'UNABLE TO UNSTOP "RECV2". DSW = ',DSW
                               ! Any other error is bad
20    CALL EXIT                ! Exit
      END
```

Example 4-5 Synchronizing a Receiving Task Using RCDS
(Sheet 1 of 3)

```
        PROGRAM RECV2
C
C FILE RECV2.FTN
C
C This task receives data from another task (e.g. SEND2).
C It prints the data, along with a header, on TI:.  Then
C it waits for another data packet, continuing this
C until it has received 3 messages.
C
C This task synchronizes with its sender using RCST.
C Because of this synchronization, the tasks can be run
C in any order, with any relative priorities.
C
C Install and run instructions: RECV2 must be installed.
C
C
        INTEGER RBUFF(15)        ! Receive buffer
        INTEGER DSW,ISSET
        DATA ISSET/2/            ! DSW code mnemonic
C
C
        DO 100, I=1,3
   1    CALL RCST(,RBUFF,DSW)    ! Receive from anyone
        IF (DSW.GE.0) GOTO 50
        Type *,'RECEIVE DIRECTIVE FAILED IN "RECV2".
        1 DSW = ',DSW            ! Display error message
        GOTO    1000             !  and exit
C
C Successful receipt or unstopped by another task. First
C check for unstopped after being stopped, in which case
C we have to receive the data
50      IF (DSW.NE.ISSET) GOTO 60 ! Were we stopped due
C                                 !  to no data? If not
C  5                              !  (NE), we have a
C                                 !  data packet
C Stopped due to no data:
   6    CALL RECEIV(,RBUFF,,DSW)  ! Now get the packet
        IF (DSW.EQ.1) GOTO 60
        TYPE *,'RECEIVE DIRECTIVE FAILED AFTER "RECV2"
   7    1UNSTOPPED. DSW = ',DSW   ! Display error
        GOTO 1000                 !  message and exit
C Display data
60      TYPE 75,(RBUFF(J),J=3,15)
75 8    FORMAT (' DATA RECEIVED BY "RECV2":'/1X,13A2)
100     CONTINUE
C Have received 3 messages
        TYPE *,'"RECV2" HAS RECEIVED 3 MESSAGES AND WILL
        1 NOW EXIT'
1000    CALL EXIT                ! Exit
        END
```

Example 4-5 Synchronizing a Receiving Task Using RCDS
(Sheet 2 of 3)

```
Run Session

! Run RECV2 first, then run SEND2 3 times
>INS RECV2
>RUN RECV2
>RUN SEND2
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
111111111
DATA RECEIVED BY "RECV2":
>
 111111111
>RUN SEND2
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
22222
DATA RECEIVED BY "RECV2":
>
 22222
>RUN SEND2
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
3333333333333333333333333
DATA RECEIVED BY "RECV2":
>
 3333333333333333333333333
"RECV2" HAS RECEIVED 3 MESSAGES AND WILL NOW EXIT


! Run SEND2 once first, then run RECV2, and then run SEND2 twice more
>RUN SEND2
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
44444
>RUN RECV2
>
 DATA RECEIVED BY "RECV2":
44444
>RUN SEND2
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
55555555
>
 DATA RECEIVED BY "RECV2":
55555555
>RUN SEND2
TYPE A LINE OF TEXT, 26 CHARACTERS OR LESS
66
>
 DATA RECEIVED BY "RECV2":
66
"RECV2" HAS RECEIVED 3 MESSAGES AND WILL NOW EXIT
```

Example 4-5 Synchronizing a Receiving Task Using RCDS
(Sheet 3 of 3)

Using Send/Receive Directives for Synchronization

If it is desirable to pass data as well as notify another task of the occurrence of an event, the Send/Receive directives can be used to perform this double function. The advantage of this approach is that data can be sent in addition to notifying the other task of the occurrence of the event. The receiving task can synchronize with the event using any of the techniques listed in Table 4-5.

Slaving the Receiving Task

Normally, a task runs under the UIC and the TI: of its initiator, the operator issuing the RUN command, or the task issuing the Request Task directive (REQUES). A receiver task which is run from the same terminal as the sender is assigned the same UIC and TI: as the sender. However, if the receiver is run from another terminal or by a different user, it's UIC and/or TI: may be different from that of the sender. Also, a receiver might receive data from several different tasks initiated at several different terminals.

If it is desirable to have the receiver task take on the UIC and the TI: of the sender each time data is received, the receiver task can be built as a slaved task. The advantages of this approach are that the receiver then acquires the same privileges as the sending task and can also do I/O directly to the sending task's terminal (through TI:). To build a task as a slaved task, either task-build with the /SLAVE qualifier or install with the /SLAVE qualifier.

PARENT/OFFSPRING TASKING

In multitasking situations, it is often useful to have one task activate and monitor other tasks, or monitor already active tasks. In particular, the requesting task may wish to receive periodic status reports from the other tasks during execution, or when the tasks exit.

For example, a task to secure a nuclear reactor in case of accident activates a pair of subordinate tasks, one task to issue warnings to personnel and the other to initiate the shutdown procedure. The task which activates the two subordinate tasks can receive periodic status reports from each of the other tasks, so that it can take appropriate action in case of a problem. In particular, it would want to know about any failure in either operation.

Under RSX-11M, parent/offspring tasking provides a facility for setting tasks up in the structure described above. As we shall see, this is easier to program than Send/Receive directives. A parent task is one which connects to or spawns another task, called an offspring task.

When a task spawns another task, it both activates the task and establishes a connection to it. If the task is already active, a parent task should just connect to the offspring. Figure 4-1 shows this relationship. When a task spawns another task it can also send a command line or data of up to 79 characters (or bytes) to the offspring.

Once the connection is established using Spawn or Connect, the offspring can send or emit status by using the Emit Status (EMST) directive. This allows the offspring to send a one-word status value to the parent. Upon exit, a success code (EX$SUC=+1) is returned if the standard EXIT is used, or a specified one-word status can be returned if the Exit With Status directive (EXST) is used. If the task is aborted, a standard severe warning code (EX$SEV=+4) is returned. The status is automatically returned in a status block in the parent task -- no receive directive is needed. Synchronization can be handled using event flags or an AST routine. The flag is set or the AST routine entered when the status is received. Table 4-6 shows the standard status codes.

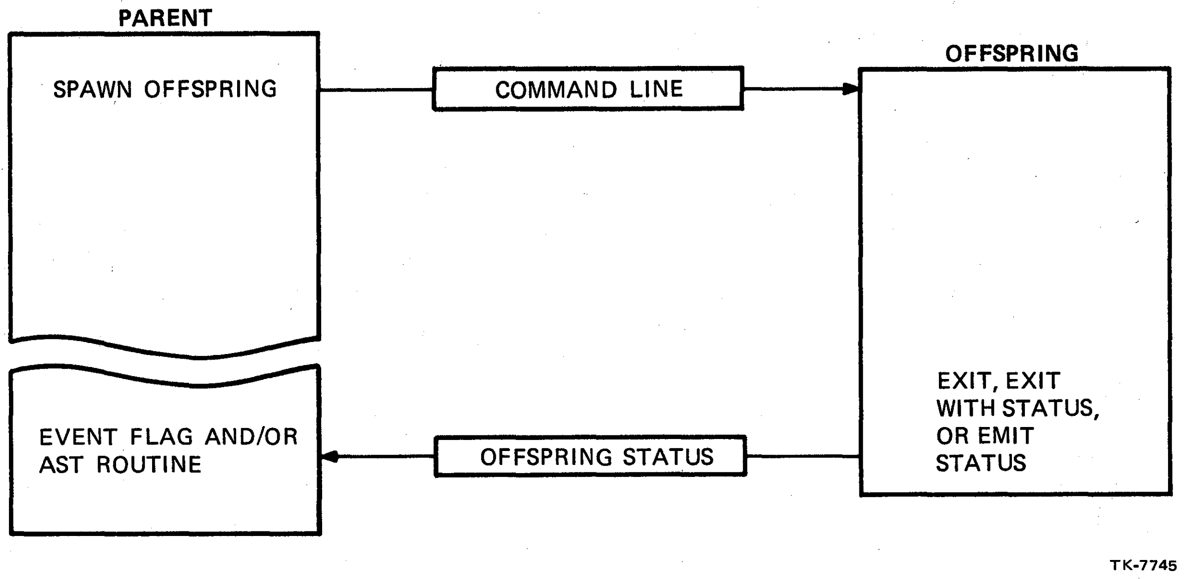

PARENT
SPAWN OFFSPRING
COMMAND LINE
OFFSPRING
EVENT FLAG AND/OR AST ROUTINE
OFFSPRING STATUS
EXIT, EXIT WITH STATUS, OR EMIT STATUS
TK-7745

Figure 4-1 Parent/Offspring Communication Facilities

Additional directives are provided for parent/offspring support. The Send Data, Request, and Connect directive combines the functions of the three separate directives (Send, Request, and Connect) into a single directive. This is similar to Spawn, but sends a 13 word data packet rather than a 79 byte command line. It also just sends data and connects if the task is already active. Spawn is rejected if the task is already active, unless the task is a CLI (Command Line Interpreter).

Two other directives are provided to allow chaining, or passing a parent/offspring connection from an offspring to another task. We will discuss chaining in more detail later in this module.

Table 4-6 Standard Exit Status Codes

Mnemonic	Value	Meaning
EX$WAR	Ø	Warning -- task succeeded, but irregularities are possible
EX$SUC	1	Success -- results as expected
EX$ERR	2	Error -- results unlikely to be as expected
EX$SEV	4	Severe Error -- one or more fatal errors were detected, or offspring aborted.

The above symbols could be used in a FORTRAN program by dropping the $ sign from the symbol and using them as a variable name with the appropriate values.

Directives Issued by a Parent Task

Table 4-7 summarizes the directives which may be issued by a parent task. Note that parent and offspring are relative terms, an offspring of one task may be the parent of another.

Table 4-7 Comparison of Parent Directives

Characteristic	Spawn	Connect	Send, Request and Connect
Can be used for offspring which is not yet active	Yes	No	Yes
Can be used with offspring which is already active	No, except if offspring is a Command Line Interpreter (CLI)	Yes	Yes
Can pass data (or command) to offspring as part of directive*	Yes (up to 79 bytes)	No	Yes (13 words)
Can be used to pass commands to a Command Line Interpreter (CLI)	Yes	No	No

* If a parent/offspring relationship is established via Connect, the tasks can of course exchange data using Send/Receive. The table above indicates whether the passing of data from parent to offspring is a capability of the directive in and of itself.

LEARNING ACTIVITY

Chapter 4 of the RSX-11M/M-PLUS Executive Reference Manual contains a good discussion of the Parent/Offspring directives and in particular it gives a number of possible uses for them. We will not discuss these various uses anywhere in this course.

Read Sections 4.1, 4.2, and 4.3 of the RSX-11M/M-PLUS Executive Reference Manual for a discussion of the Parent/Offspring directives and examples of their use in applications.

Table 4-8 summarizes the arguments for the Spawn directive, the Connect directive, and the Send Data, Request, and Connect directive. For additional information, see the writeup on each directive in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual.

Table 4-8 Directives Used by a Task to Establish a Parent/Offspring Relationship

Directive Name	Directive Call
Spawn	CALL SPAWN(tsk,grp,mem,efn,ast,esb,param, cmdlin,cmdlen,unum,dnam,dsw)
Connect	CALL CNCT(tsk,efn,ast,esb,param,dsw)
Send, Request, and Connect	CALL SDRC(tsk,buf,efn,ast,esb,param,dsw)

tsk - offspring task

grp,mem - UIC offspring will run under

efn - event flag to be set when offspring exits or emits status

ast - AST routine to be entered when offspring exits or emits status.

esb - exit status block address

param - name of a word to receive the status block address when the AST occurs

cmdlin,cmdlen - address of buffer with command line, length of command line

unum,dnam - device to be TI: for offspring

buf - 13(10) word buffer to be sent

dsw - directive status word

Example 4-6 shows a task which spawns PIP to display a directory at TI:. The following notes are keyed to the example.

1. The command line to be passed to PIP. We include the three character command name to be consistent with the way MCR passes commands if a utility command is typed to MCR.

2. Display startup message.

3. Spawn ...PIP. Event flag 1 will be set when ...PIP exits or emits status. EXSTAT is the address of the eight-word status block (only the first word is used). CMD is the starting address of the command line and LEN is its length.

4. Wait for event flag 1 to be set when ...PIP exits or emits status. Notice that this is a local event flag, local to this task, which is cleared by the Executive when the task is spawned and set by the Executive when the spawned task exits or emits status.

5. The high-order byte of the exit status code may contain unexpected data. Therefore, clear that byte by specifying the logical AND of the code and 377(8) before displaying the code.

6. On the Run Session - The first run session shows a successful exit by ...PIP, the second one shows ...PIP aborted by an operator. Note the different status codes.

NOTE

On an RSX-11M system, an attempt to spawn ...PIP will fail if ...PIP is already active. This works diffently from initiating PIP from MCR, where an attempt is made to install the task ...PIP under the name PIPTnn if ...PIP is already active. A solution to this problem is to spawn CLI... (the current CLI), ...DCL (DCL) or MCR... (MCR) and send it the command line. It will in turn start up the appropriate PIP task under ...PIP or PIPTnn, as if the command was typed in by an operator. See section 4.4 (on Spawning System Tasks) of the RSX-11M/M-PLUS Executive Reference Manual for additional information.

```
        PROGRAM SPWN
C
C File SPAWN.FTN
C
C This program spawns PIP, passes it a command line to
C display a directory at TI:, waits for it to exit, and
C then displays its exit status.
C
C Data
        INTEGER EXSTAT(8),PLIST(3),DSW
        BYTE BUFF(80)
        REAL PIP,CMD(3)
        DATA PIP/6R...PIP/
     ❶  DATA CMD/'PIP ','*.MA','C/LI'/
C Code
        WRITE (5,15)              ! Write message
15   ❷  FORMAT (' SPAWN IS STARTING AND WILL SPAWN PIP')
     ❸  CALL SPAWN(PIP,,,1,,EXSTAT,,CMD,12,,,DSW)
                                  ! Spawn PIP
        IF (DSW.LT.0) GOTO 900    ! Branch on dir error
     ❹  CALL WAITFR(1,DSW)        ! Wait for task to exit
        IF (DSW.LT.0) GOTO 910    ! Branch on dir error
     ❺  WRITE (5,25) EXSTAT(1).AND."377 ! Display low
                                  ! byte of exit status
25      FORMAT (' SPAWN REPORTING: PIP EXITED, EXIT
        1STATUS WAS ',I1,'.')
        CALL EXIT                 ! Exit
C Error handling code
900     TYPE *,'ERROR SPAWNING PIP. DSW = ',DSW
        GOTO 1000
910     TYPE *,'ERROR WAITING FOR EVENT FLAG. DSW = ',DSW
1000    CALL EXIT
        END
```

Example 4-6 A Task Which Spawns PIP (Sheet 1 of 2)

```
Run Session

>RUN SPAWN
SPAWN IS STARTING AND WILL SPAWN PIP

Directory DB1:[305,301]
8-MAR-82 12:15

W.MAC;1              1.         20-MAY-81 13:04
A1.MAC;2             1.         09-DEC-80 16:58
A.MAC;1              1.         10-JUN-81 15:21
                          .
                          .
                          .
SPAWN.MAC;22         1.         08-SEP-81 11:20

Total 127./129. blocks in 25. files
>
 SPAWN REPORTING: PIP EXITED. EXIT STATUS WAS 1.
>

6

>RUN SPAWN
SPAWN IS STARTING AND WILL SPAWN PIP

Directory DB1:[305,301]
8-MAR-82 12:15

W.MAC;1              1.         20-MAY-81 13:04
A1.MAC;2             1.         09-DEC-80 16:58
A.MAC;1              1.         10-JUN-81 15:21
DCL>ABORT/TASK ...PIP
A9.MAC;12            4.         21-MAY-81 13:50
12:15:15 Task "...PIP" terminated
         Aborted via directive or CLI
         And with pending I/O requests
>
 SPAWN REPORTING: PIP EXITED. EXIT STATUS WAS 4.
>
```

Example 4-6 A Task Which Spawns PIP (Sheet 2 of 2)

Example 4-7 is a more generalized spawning task, which prompts for the name of a task and a command line and then spawns that task, sending the input command line to it. The following notes are keyed to the example.

❶ Prompt for and get the task name. The task name must be entered in all uppercase characters. To allow lowercase characters, the code must be modified to check for any lowercase characters and convert them to uppercase.

❷ Convert ASCII task name to Radix-5Ø format.

❸ Prompt for and get command line.

❹ Spawn task specified. We are using event flag 1 for synchronization. The status will be returned in EXSTAT.

❺ Wait for event flag 1 to be set, indicating that the task has exited (or emitted status).

❻ Clear high-order byte of the status word and display it.

❼ Note that CLI... passes the command line to the current CLI (DCL) which in turn invokes task DIRT11 to display the directory. (This is task DIR spawned at terminal T11)

❽ BUFFER(1) and (2) are set to blanks in case a name of less than six characters is entered. By clearing to blanks, a short name is assured of having trailing blanks.

```
        PROGRAM GSPAWN
C
C FILE GSPAWN.FTN
C
C This task prompts at ti: for a task name and command
C line, then spawns the specified task and passes it the
C command line. After that it waits until the offspring
C task exits and displays its exit status.
C
C Run instructions: The name of the task to be spawned
C must be typed in using all upper case characters.
C
        REAL BUFFER(20),TSKNAM
        INTEGER EXSTAT(8)
C Pad task name buffer with blanks in case name is short
     ❽  DATA BUFFER(1),BUFFER(2) /'    ','    '/
C
        WRITE (5,15)
15   ❶  FORMAT (' TASK NAME?')
        READ (5,25) BUFFER(1),BUFFER(2)
25      FORMAT (2A4)
C Convert task name to Radix-50 format
     ❷  CALL IRAD50 (6,BUFFER,TSKNAM)
        WRITE (5,35)
35      FORMAT (' COMMAND LINE (79 CHARACTERS OR LESS)?')
     ❸  READ (5,45) N,BUFFER
45      FORMAT (Q,20A4)
C Spawn task with command line
     ❹  CALL SPAWN (TSKNAM,,,1,,EXSTAT,,BUFFER,N,,,IDSW)
        IF (IDSW .LT. 0) GOTO 900 ! Branch on dir error
C Wait for task to exit
     ❺  CALL WAITFR (1,IDSW)
        IF (IDSW .LT. 0) GOTO 910 ! Branch on dir error
        WRITE (5,55) EXSTAT(1) .AND. "377
55   ❻  FORMAT ('0',10X,'TASK EXITED. STATUS WAS ',I2,'.'
        1/)
        GOTO 1000                    ! Go to common exit
C Error code
900     WRITE (5,905) IDSW
905     FORMAT (' DIRECTIVE ERROR SPAWNING TASK. DSW = '
        1,I4)
        GOTO 1000
910     WRITE (5,915) IDSW
915     FORMAT (' DIRECTIVE ERROR ON WAIT FOR. DSW = ',
        1I4)
1000    CALL EXIT
        END
```

Example 4-7 A Generalized Spawning Task (Sheet 1 of 2)

```
Run Session

>RUN GSPAWN
TASK NAME?
...PIP
COMMAND LINE (79 CHARACTERS OR LESS)?
PIP *.DIS/LI

Directory DB1:[305,301]
8-SEP-81 15:09

FRIENDS.DIS;2          1.          10-AUG-81 11:13
FRIENDSNL.DIS;2        1.          31-AUG-81 11:42

Total of 2./10. blocks in 2. files


          TASK EXITED. STATUS WAS 1.


>RUN GSPAWN
TASK NAME?
...DCL
COMMAND LINE (79 CHARACTERS OR LESS)?
DIRECTORY *.MAC

Directory DB1:[305,301]
8-SEP-81 15:10

W.MAC;1                1.          20-MAY-81 13:04
A1.MAC;2               1.          09-DEC-80 16:58
A.MAC;1                1.          10-JUN-81 15:21
A9.MAC;12              4.          21-MAY-81 13:50
FORMAT.MAC;34          6.          21-AUG-81 11:53
PROGY.MAC;1            1.          30-JAN-81 14:27
PROGZ.MAC;1            1.          30-JAN-81 14:30
RAY.MAC;1              4.          30-JAN-81 14:39
PROGX.MAC;6            1.          30-JAN-81 14:42
C.MAC;5                1.          21-MAY-81 10:01
A2.MAC;2               1.          21-MAY-81 10:04
C2.MAC;1               1.          21-MAY-81 10:04
         [Task "DIRT11" terminated
       7 [Aborted via directive or CLI
         [And with pending I/O requests

          TASK EXITED. STATUS WAS 4.
```

Example 4-7 A Generalized Spawning Task (Sheet 2 of 2)

Directives Issued by an Offspring Task

Table 4-9 summarizes the directives which can be used by an offspring to return status to a parent task. Table 4-6 shows the standard exit status codes used on the system. An offspring can also spawn or connect to other tasks as well.

Table 4-9 Directives Which Return Status to a Parent Task

Directive	Effect/Use
CALL EXIT	Exits and returns "Success" status to all current parent tasks. Special case of CALL EXST
CALL EXST(status)	Exits and returns specified one-word status to all current parent tasks. Terminates parent/offspring relationship.
CALL EMST (parent-task,status,dsw)	Emit specified status to specified parent (or to all parents, if parent task name is omitted). Terminates the parent/offspring relationship. The connection can be reestablished by the parent, using the Connect Directive.

NOTE

The Executive returns "Severe Exit" status if the task is aborted or if a fatal error occurs.

Chaining of Parent/Offspring Relationships

An offspring can chain or pass its parent/offspring connection on to another task. In that case the connection between the parent and the offspring which passes the connection is broken. In its place, a connection is made between the parent and the new offspring.

Figure 4-2 shows the difference between an offspring spawning another task versus chaining its connection to another task. Note that with spawn, the connection between the parent and the first offspring still exists, plus a new connection is established between the first offspring and the new offspring.

Table 4-1Ø summarizes the directives which can be used to chain parent/offspring relationships. Request and Pass Offspring Information (RPOI) is similar to Spawn in function, in that it starts up the task and can pass a 79 byte command line. Send Data, Request, and Pass Offspring Control Block (SDRP) is similar to Send Data, Request and Connect, in that it sends a 13 word data packet and it succeeds even if the task is already active.

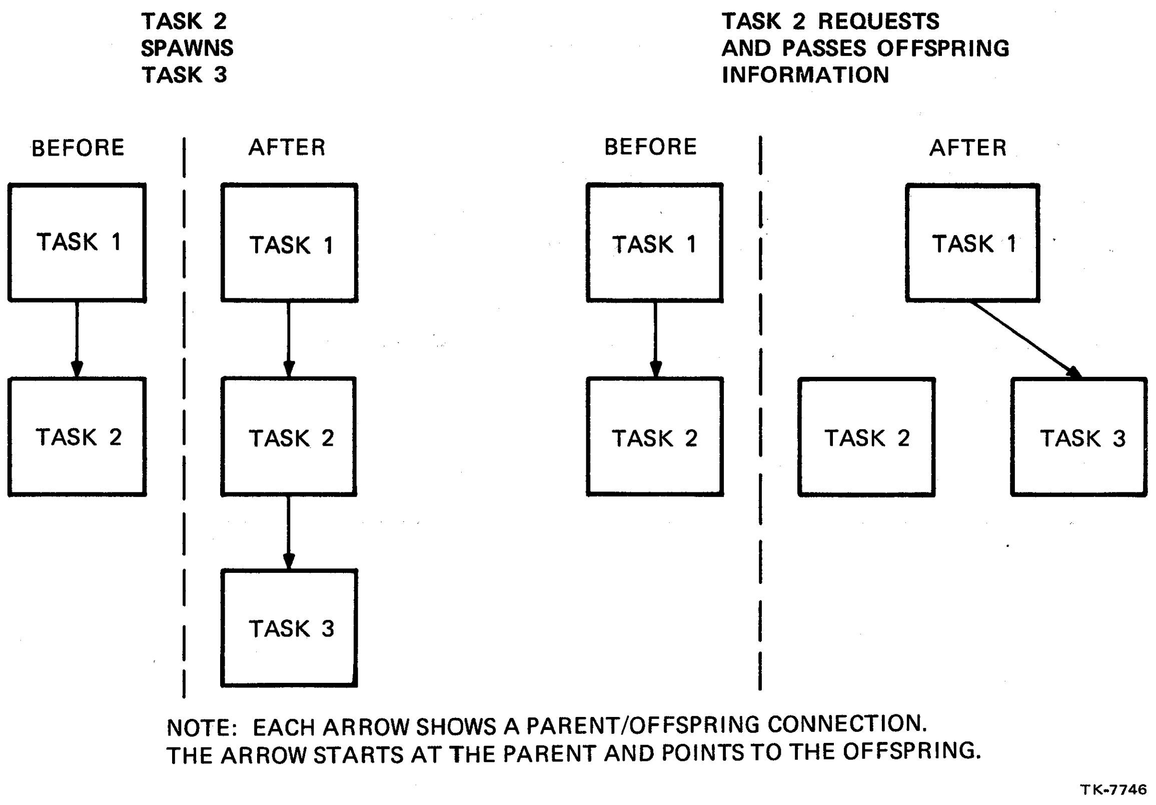


Figure 4-2 Spawning Versus Chaining
(Request and Pass Offspring Information)

Table 4-1Ø Directives Which Pass Parent/Offspring Connections to Other Tasks

Characteristic	RPOI	SDRP
Can be used for a new offspring which is not yet active	Yes	Yes
Can be used for a new offspring which is already active	No, unless the offspring is CLI	Yes
Can pass data (or a command) to a new offspring	Yes (up to 79 bytes)	Yes (13 words)
Can be used to pass commands to a CLI	Yes	No

Example 4-8 shows the use of the Request and Pass Offspring Information (RPOI) directive. This task is similar to example 4-6, but it uses RPOI instead of SPAWN. Two run sessions are provided, showing the difference between an offspring passing its parent/task connection and an offspring spawning another offspring. In the first run session, GSPAWN spawns PASSIT (Example 4-8), which starts up PIP, passing its connection (GSPAWN/PASSIT) on to PIP. In the second run session, GSPAWN spawns SPAWN (Example 4-6), which spawns PIP. Note that with PASSIT, ...PIP returns its exit status directly to GSPAWN. GSPAWN is no longer connected to PASSIT once the connection is passed on to PIP. With SPAWN, ...PIP returns its exit status to SPAWN. SPAWN displays that status and then exits, sending its own exit status to GSPAWN.

The following notes are keyed to the example.

1. Use RPOI instead of SPAWN. No event flag is needed nor is a status block set up since this task won't receive status from ...PIP. The seventh argument in the argument list (MACRO symbolic name RP.OAL, suggested FORTRAN name RPOAL) determines what parent (fixed) connections are passed, if any. If RPOAL has a value of 1, as in the example, all connections are passed. (In this example there is only one connection.) A connection is established between the parent of PASSIT (GSPAWN) and ...PIP. The connection between GSPAWN and PASSIT is broken.

2. Display a message and exit with a status of 1Ø., to make it easy to tell whether the status is from this task or from ...PIP. Note in SPAWN that the CALL EXIT is used, which results in a Success Code (+1) being sent as the exit status.

3. On the First Run Session (GSPAWN spawns PASSIT) - The exit status from ...PIP is returned directly to GSPAWN.

4. On the Second Run Session (GSPAWN spawns SPAWN) - The exit status from ...PIP is returned to SPAWN, then SPAWN returns its own exit status to GSPAWN.

If you wish to chain the connection from only one of several parents, specify a single task, and do not specify RPOAL in the RPOI directive call.

If RPOAL is not specified and no task is specified, then no connections are passed. This might be useful to request a task and send 79 bytes of data when a connection is not needed.

```
        PROGRAM PASSIT
C
C File PASSIT.FTN
C
C This program requests PIP, passes it a command line to
C display a directory at TI:, and passes it all of its
C parent connections as well
C
C Data
        INTEGER PLIST(3),DSW
        BYTE BUFF(80)
        REAL PIP,CMD(3)
        DATA PIP/6R...PIP/
        DATA CMD/'PIP ','*.MA','C/LI'/
C Code
        WRITE (5,15)               ! Write message
15      FORMAT (' PASSIT IS STARTING AND WILL REQUEST PIP')
     ❶  CALL RPOI (PIP,,,,CMD,12,1,,,,,DSW) ! Request PIP
        IF (DSW.LT.0) GOTO 900     ! Branch on dir error
       [WRITE (5,25)               ! Write message
25   ❷ |FORMAT (' PASSIT REQUESTED PIP AND WILL NOW EXIT')
       [CALL EXST (10)             ! Exit with status of 10
C Error handling code
900     TYPE *,'ERROR REQUESTING PIP. DSW = ',DSW
        CALL EXIT
        END
```

Example 4-8 An Offspring Task Which Chains Its Parent/Offspring Connection to PIP (Sheet 1 of 3)

```
Run Session

>INS PASSIT
>RUN GSPAWN
TASK NAME?
PASSIT
COMMAND LINE (79 CHARACTERS OR LESS)?

PASSIT IS STARTING AND WILL REQUEST PIP
PASSIT HAS REQUESTED PIP AND WILL NOW EXIT
>
Directory DB1:[305,301]
8-MAR-82 15:22

W.MAC;1              1.          20-MAY-81 13:04
A1.MAC;2             1.          09-DEC-80 16:58
                          .
                          .
                          .
SPAWN.MAC;1          4.          08-SEP-81 13:32

Total of 13./66. blocks in 15. files


       ❸ TASK EXITED. STATUS WAS 1.


>RUN GSPAWN
TASK NAME?
PASSIT
COMMAND LINE (79 CHARACTERS OR LESS)?

PASSIT IS STARTING AND WILL REQUEST PIP
PASSIT HAS REQUESTED PIP AND WILL NOW EXIT
>
Directory DB1:[305,301]
8-SEP-81 15:23

W.MAC;1              1.          20-MAY-81 13:04
A1.MAC;2             1.          09-DEC-80 16:58
A.MAC;1              1.          10-JUN-81 15:21
A9.MAC;12            4.          21-MAY-81 13:50
15:24:10  Task "...PIP" terminated
          Aborted via directive or CLI
          And with pending I/O requests

       ❸ TASK EXITED. STATUS WAS 4.
>
```

Example 4-8 An Offspring Task Which Chains Its Parent/Offspring Connection to PIP (Sheet 2 of 3)

```
Run Session

>INS SPAWN
>RUN GSPAWN
TASK NAME?
SPAWN
COMMAND LINE (79 CHARACTERS OR LESS)?

SPAWN IS STARTING AND WILL SPAWN PIP

Directory DB1:[305,301]
8-MAR-82 15:22

W.MAC;1              1.          20-MAY-81 13:04
A1.MAC;2             1.          09-DEC-80 16:58
                          .
                          .
                          .
SPAWN.MAC;1          4.          08-SEP-81 13:32

Total of 13./66. blocks in 15. files


SPAWN REPORTING: PIP EXITED, EXIT STATUS WAS 1.

      4  TASK EXITED. STATUS WAS 1.


>RUN GSPAWN
TASK NAME?
SPAWN
COMMAND LINE (79 CHARACTERS OR LESS)?

SPAWN IS STARTING AND WILL SPAWN PIP

Directory DB1:[305,301]
8-SEP-81 15:23

W.MAC;1              1.          20-MAY-81 13:04
A1.MAC;2             1.          09-DEC-80 16:58
A.MAC;1              1.          10-JUN-81 15:21
DCL>ABORT/TASK ...PIP
A9.MAC;12            4.          21-MAY-81 13:50
15:24:10  Task "...PIP" terminated
          Aborted via directive or CLI
          And with pending I/O requests
  [SPAWN REPORTING: PIP EXITED, EXIT STATUS WAS 4.
4 [
  [         TASK EXITED. STATUS WAS 1.
>
```

Example 4-8 An Offspring Task Which Chains Its Parent/Offspring Connection to PIP (Sheet 3 of 3)

Other Parent/Offspring Considerations

Retrieving Command Lines in Spawned Tasks - Use the Get MCR Command Line directive (GETMCR). The passed command is returned to a buffer specified in the GETMCR call.

Spawning a Utility or Other MCR Spawnable Task - Utilities are generally installed under task names of the form ...tsk. This makes them MCR spawnable tasks, which notifies MCR to spawn multiple copies of the task under names tskTnn if the task is invoked as an MCR command using the three-character task name (e.g., PIP /LI). In fact, any task is spawnable, but only tasks installed under a name of this form are spawned as multiple copy tasks by MCR. When such a task is invoked by MCR, MCR passes it the entire command line, including the three-character task name (e.g., PIP /LI). Even if you spawn a utility directly, you should pass a command line which includes the three-character task name. This maintains compatibility with the format used by MCR to pass commands to utilities, and avoids potential problems caused when the utility parses your command line.

On RSX-11M systems, there is more likelihood of getting a task already active failure if you spawn a utility directly using the name ...tsk than there is if you instead spawn MCR... and pass the command line which includes the task name. This is due to the fact that if a task is spawned directly using ...tsk, the spawn attempt fails if the task ...tsk is aready active. No attempt is made to install the task under the name tskTnn if ...tsk is already active, as is the case if you spawn MCR... (MCR) to start up the utility.

Example 4-9 shows a spawned task which retrieves a simple command line of the form SPW n, where n is a single character. If n=1, SPW performs a simple addition exercise and displays the answer. If n=2, SPW performs a simple multiplication exercise instead. Else, SPW displays the message "NO OTHER OPERATIONS ALLOWED". This task, like most system utilities, will run correctly whether spawned directly by a task (as ...SPW), started by MCR as the result of a command line sent when spawning MCR, or invoked by an operator using an MCR command.

The following notes are keyed to Example 4-9.

❶ CALL GETMCR to get the command line.

❷ Display the command line as received.

❸ Check the value of n for an ASCII 1, skipping over the characters SPW and the blank after SPW. Note that in a real application, the first part of the command line should be checked as well to see that it really is SPW and a blank. Branch if not equal.

❹ Check n for an ASCII 2. If not branch to error at 7.

❺ If n=1, perform a simple addition (2+5).

❻ If n=2, perform a simple multiplication (2x5).

❼ If n is neither a 1 nor a 2, display an error message and exit with warning status(Ø).

❽ If n was 1 or 2, display a message giving the results of the computation and then exit with success status (+1).

❾ This run session shows ...SPW being spawned three times by MCR, when an operator types an MCR command line.

❿ This run session shows ...SPW being invoked three times by running GSPAWN, which in turn spawns ...SPW.

```
      PROGRAM SPWNED
C
C File SPWNED.FTN
C
C This task uses the GETMCR directive to get a command
C line from either TI: or the parent task. It then
C echoes the command line and does an add or multiply,
C types out the answer and emits status on exit
C
C Task-build instructions:
C
C       >LINK/MAP SPWNED,LB:[1,1]PROGSUBS/LIBRARY,FOROTS-
C       /->LIBRARY
C
C Install and run instructions: To make this task MCR
C spawnable, install it under the name ...SPW. Command
C lines should be of the form SPW function - valid
C functions are 1 and 2.
C
      BYTE INBUFF(80)
      INTEGER IOSB(2),DSW
      INTEGER NUM1,NUM2,ANS
      DATA NUM1,NUM2/5,2/
      BYTE OP
C
    ❶ CALL GETMCR(INBUFF,DSW)      ! Get command line
      IF (DSW.GT.0) GOTO 10
      TYPE *,'DIDN''T GET COMMAND LINE. DSW = ',DSW
10  ❷ TYPE 15,(INBUFF(I),I=1,DSW) ! Display the
C                                 !  command line
15    FORMAT (1X,80A1)
C Check for function 1, branch if not
    ❸ IF (INBUFF(5).NE.'1') GOTO 20
     ❺ANS = NUM1 + NUM2           ! Do addition
      OP = '+'                    ! Set operation sign
      GOTO 30                     ! Display results
C                                 !  and exit
C Check for function 2, branch if not
20  ❹ IF (INBUFF(5).NE.'2') GOTO 40
     ❻ANS = NUM1 * NUM2           ! Do multiplication
      OP = '*'                    ! Set operation sign
30    TYPE 35,NUM1,OP,NUM2,ANS    ! Display results
35  ❽ FORMAT (1X,I1,1X,A1,I2,' =',I2,'.')
      CALL EXST(1)                ! Exit with success
C                                 !  status
C Display no op message
40    TYPE *,' NO OTHER OPERATIONS ALLOWED'
    ❼ CALL EXST(0)                ! Exit with warning
C                                 !  status
      END
```

Example 4-9 A Spawned Task Which Retrieves a Command Line (Sheet 1 of 2)

Run Session

9
```
>INS/TASK_NAME:...SPW SPWNED
>MCR SPW 1
SPW 1
5 + 2 = 7.
>MCR SPW 2
SPW 2
5 * 2 = 10.
>MCR SPW 3
SPW 3
  NO OTHER OPERATIONS ALLOWED
```

10
```
>RUN GSPAWN
TASK NAME?
...SPW
COMMAND LINE (79 CHARACTERS OR LESS)?
SPW 1
SPW 1
5 + 2 = 7.

          TASK EXITED. STATUS WAS 1.
>RUN GSPAWN
TASK NAME?
...SPW
COMMAND LINE (79 CHARACTERS OR LESS)?
SPW 2
SPW 2
5 * 2 = 10.

          TASK EXITED. STATUS WAS 1.
>RUN GSPAWN
TASK NAME?
...SPW
COMMAND LINE (79 CHARACTERS OR LESS)?
SPW 3
SPW 3
  NO OTHER OPERATIONS ALLOWED

          TASK EXITED. STATUS WAS 0.
>
```

Example 4-9 A Spawned Task Which Retrieves a Command Line (Sheet 2 of 2)

Task Abort Status - When establishing a parent/offspring connection, it is possible to request a second word of status when a task exits. In that case, the second word of the status block returns the task abort status. This word allows a parent task to distinguish the different reasons for return of "Severe Error" status.

Table 4-11 lists the various task abort status codes. To get the second status word, place any nonzero value in the high byte of the event flag argument word. To do this, specify the logical OR of 256 and the event flag number.

Example:

```
        CALL SPAWN(TASKS,,,,,256.OR.12,,STAT,CMD,LCMD)
```

Table 4-11 Task Abort Status Codes

Mnemonic	Value	Exit Status	Meaning
S.CEXT	-2(1Ø)	EX$SUC=1	Task exited normally
S.COAD	Ø	EX$SEV=4	Odd address and traps to 4
S.CSGF	2(1Ø)	EX$SEV	Segment fault
S.CBPT	4(1Ø)	EX$SEV	Break point or trace trap
S.CIOT	6(1Ø)	EX$SEV	IOT instruction
S.CILI	8(1Ø)	EX$SEV	Illegal or reserved instruction
S.CEMT	1Ø(1Ø)	EX$SEV	Non RSX EMT instruction
S.CTRP	12(1Ø)	EX$SEV	Trap instruction
S.CFLT	14(1Ø)	EX$SEV	11/4Ø floating-point exception
S.CSST	16(1Ø)	EX$SEV	SST abort - bad stack
S.CAST	18(1Ø)	EX$SEV	AST abort - bad stack
S.CABO	2Ø(1Ø)	EX$SEV	Abort via directive or CLI command
S.CLRF	22(1Ø)	EX$SEV	Task load request failure
S.CCRF	24(1Ø)	EX$SEV	Task checkpoint read failure
S.IOMG	26(1Ø)	EX$SEV	Task exit with outstanding I/O
S.PRTY	28(1Ø)	EX$SEV	Task memory parity error
S.CPMD	3Ø(1Ø)	EX$SEV	Task aborted with PMD request
S.CINS	32(1Ø)	EX$SEV	Task installed in two different systems

Summary of Various Methods of Data Transfer Between Tasks

Table 4-12 summarizes and compares the various methods of data transfer between tasks which we have discussed so far.

Comparison of Methods of Data Transfer

Table 4-12 Comparison of Methods of Data Transfer Between Tasks

Method	Maximum Amount	Direction/ Repetition Restrictions	Pool Requirements	Notes
Send/ Receive	13(1Ø) words	None	Data packet is buffered in pool	Both tasks must be written to use Send/Receive directives
Spawn Command Line	79(1Ø) bytes	Parent to offspring only Offspring must exit for parent to pass another command	Command line is buffered in pool Offspring Control Block (OCB) is created in pool	Used with any task which uses GETMCR directive or Get Command Line (GCML) routine
Off-spring Status	1 or 2 words	Offspring to parent only Parent must reconnect to offspring to receive status again	Only OCB is in pool	No separate directive needed in parent to receive status Any exiting task automatically returns status

Other Methods of Transferring or Sharing Data Between Tasks

If large amounts of data are to be transferred between tasks or shared between tasks, two other techniques are available. Tasks can use files on mass storage devices. This technique is advantageous if really quick transfer is not essential and/or if a permanent copy of the data is desired.

Tasks can also be written to share a data area in memory. This technique is particularly useful if transfer time is critical and a permanent copy of the data is either not needed at all or is not needed until a later time. Both of these techniques are discussed in later modules.

Now do the tests/exercises for this module in the Tests/Exercises book. They are all lab problems. Check your answers against the solutions provided, either in that book or in on-line files.

If you think that you have mastered the material, ask your course administrator to record your progress in your Personal Progress Plotter. You will then be ready to begin a new module.

If you think that you have not yet mastered the material, return to this module for further study.

MEMORY MANAGEMENT CONCEPTS

INTRODUCTION

The use of memory management hardware in mapped systems permits the use of more physical memory, task relocation, and the sharing of data and code. It also offers a memory protection feature. This module explains how the memory management hardware works and how the software interacts with the hardware. Later modules explain the use of memory management for overlays and shared regions.

OBJECTIVES

1. To list the differences between mapped and unmapped systems
2. To list the advantages of memory management
3. To use virtual and physical addresses, windows, and regions to describe the mapping of a task.

RESOURCES

1. RSX-11M/M-PLUS Task Builder Manual, Chapter 2
2. PDP-11 Processor Handbook, Chapter 6 (optional)

GOALS OF MEMORY MANAGEMENT

The KT-11 memory management unit is a device available on medium and larger PDP-11's. While the 16-bit addressing structure of the PDP-11's limits processors without a memory management unit to 32K words of addressing, processors with a memory management unit can support up to 128K words, or even as much as 2ØØØK words (2 Meg words), depending on the model of the processor.

In addition to this extension of the processor's addressing space, a memory management unit offers other features not otherwise available. With memory management, tasks can be loaded and executed at different locations in memory without being modified in any way. This means that the operating system can load a task into any available space within a system-controlled partition; therefore a task need not wait until a specific location is available. It also means that the Executive can move tasks around to make better use of available space (shuffling).

Memory management also provides a mechanism for controlling tasks' access to memory. Memory areas can be protected: unrelated tasks can reside in memory simultaneously and are normally prevented from accessing each other's memory. However, tasks which do need to share memory locations are allowed to do so, under the rules of memory access built into the Executive.

HARDWARE CONCEPTS

Mapped Versus Unmapped Systems

A system which has the KT-11 memory management unit installed and enabled is called a mapped system. Otherwise, it is called an unmapped system. Small PDP-11's, such as the PDP-11/Ø3 and PDP-11/Ø4 are always unmapped. The KT-11 unit is available as an option on some medium sized processors, including the PDP-11/35 and PDP-11/4Ø. It is a standard feature on large and newer processors such as the PDP-11/7Ø, PDP-11/24, PDP-11/23-PLUS and PDP-11/44.

Table 5-1 shows a comparison of unmapped and mapped systems on various PDP-11's.

Table 5-1 Mapped Versus Unmapped Systems

System	**Addressing**	**Memory Size**	**Other Addressing**	**Addressing Limit**
Unmapped	16-bit	28K words (56K bytes)	I/O page 4K words (8K bytes)	177777 32K words (64K bytes)
Mapped	18-bit	124K words (248K bytes)	I/O page 4K words (8K bytes)	777777 128K words (256K bytes)
Mapped	22-bit	192ØK words (384ØK bytes)	I/O page 4K words (8K bytes) UNIBUS map 124K words (248K bytes)	17777777 2Ø48K words (4Ø96K bytes)

Figures 5-1 to 5-3 show physical address space on the various systems. Appendix B contains a conversion chart between decimal and octal, and between various word and byte values, which may be helpful as you read this module.

Figure 5-1 shows the layout of an unmapped system. Sixteen-bit addresses are all that are allowed. This corresponds to an addressing limit of 32K words or 64K bytes. Of this, 28K words correspond to actual physical memory and 4K words correspond to the I/O page. The addresses in the I/O page are assigned to peripheral devices which are used in performing I/O operations. On an RSX-11M system, the Executive, including the Dynamic Storage Region (DSR or POOL), takes up something less than or equal to 2ØK words (as little as 8K words). Tasks occupy the area between the top of the Executive and the top of memory.

Figure 5-2 shows the layout of a mapped system with 18-bit addressing. Eighteen bits give an addressing limit of 128K words or 256K bytes. Again, the top 4K words correspond to the I/O page, leaving 124K words of physical memory. The Executive, including POOL, usually takes either 16K words or 2ØK words, leaving the rest, either 1Ø8K words or 1Ø4K words, for tasks.

Figure 5-3 shows the layout of a mapped system with 22-bit addressing. Twenty-two bits give an addressing limit of 2048K words or 4096K bytes. Again, the top 4K words correspond to the I/O page. 124K words are used for UNIBUS mapping, which is needed when peripheral devices access memory directly (DMA devices). UNIBUS mapping is necessary to convert 18-bit UNIBUS addresses to 22-bit physical memory addresses. This leaves 1920K words of physical memory. Again, the Executive, including POOL, usually takes 16K words or 20K words, leaving 1904K words or 1900K words for tasks.

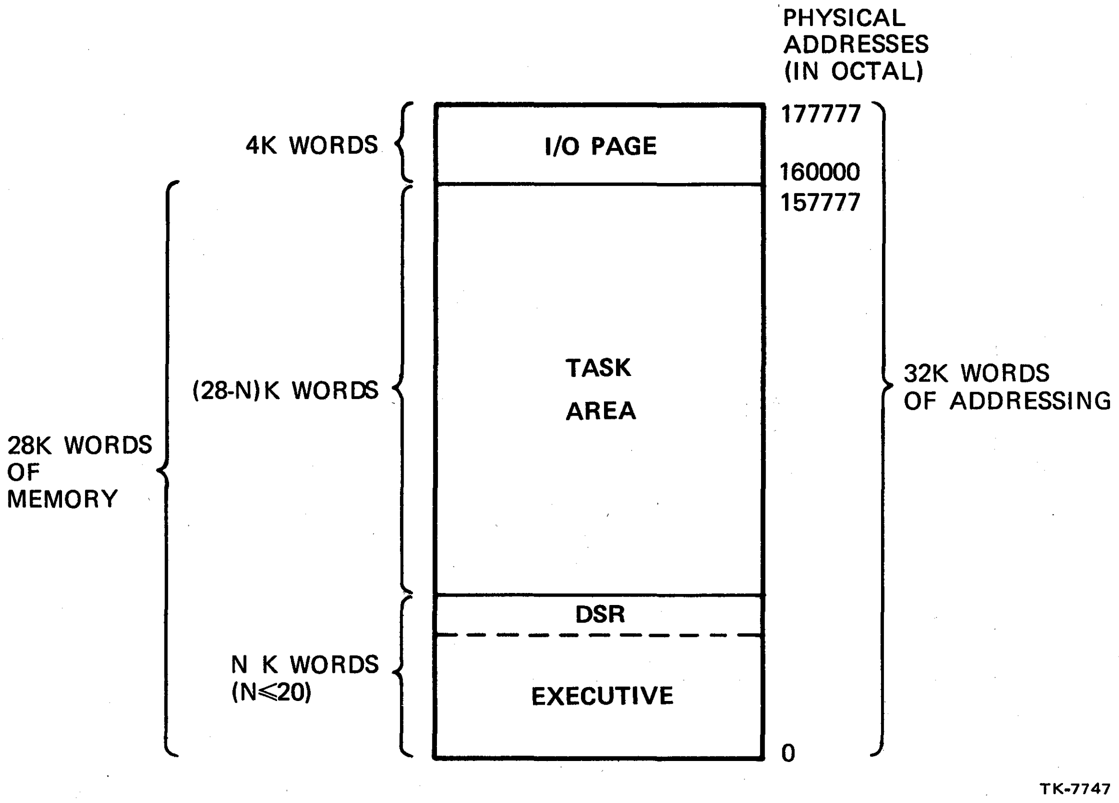


Figure 5-1 Physical Address Space in an Unmapped System

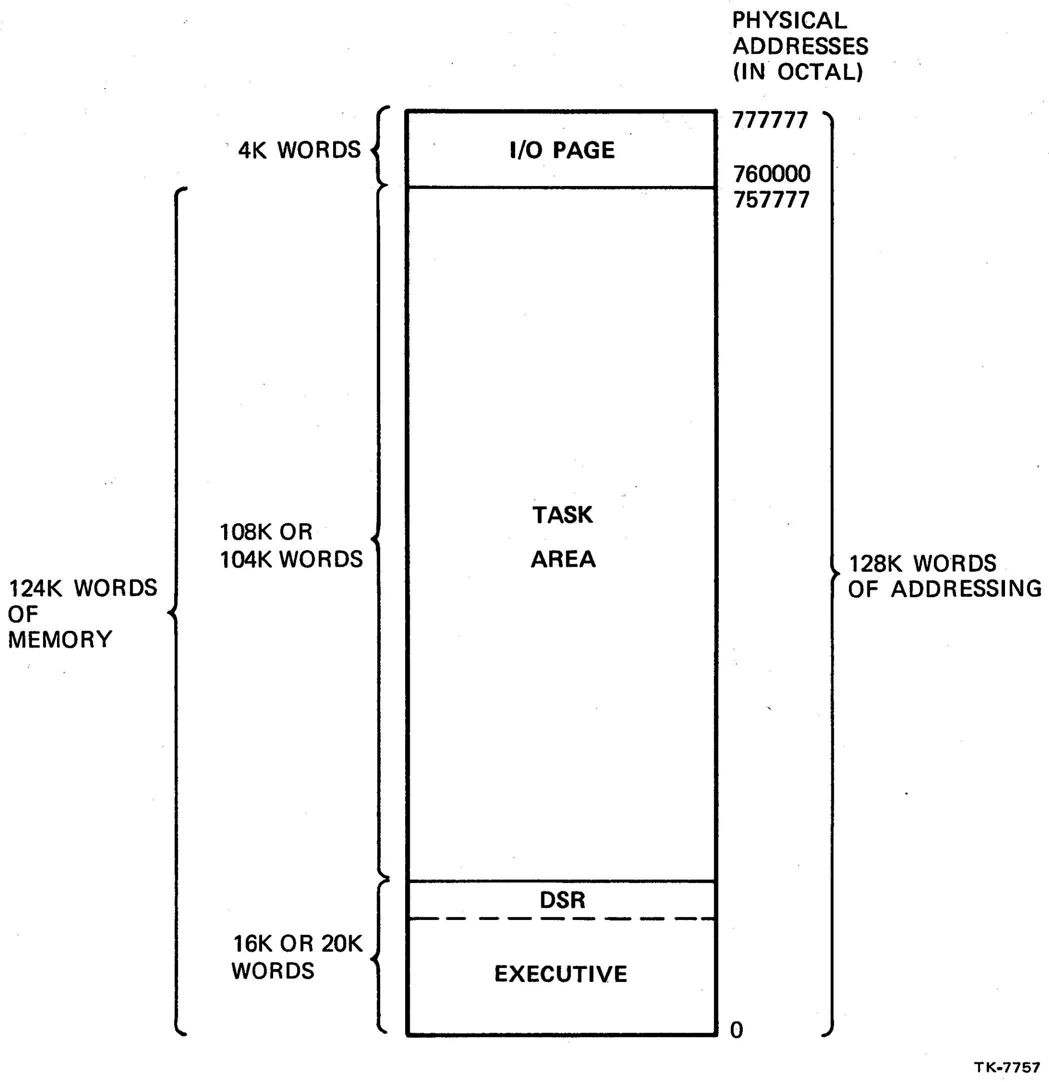


Figure 5-2 Physical Address Space in an 18-Bit Mapped System

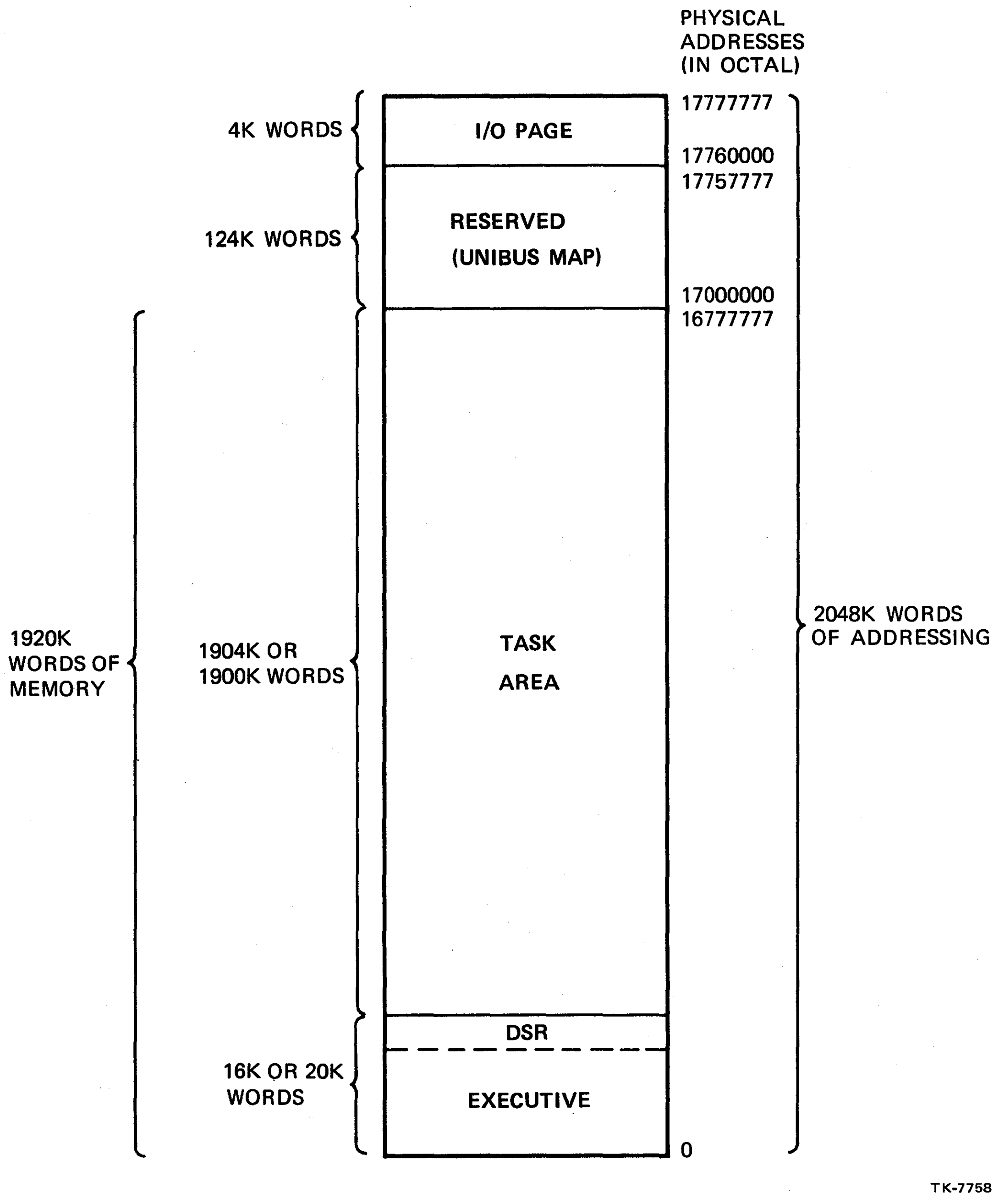


Figure 5-3 Physical Address Space in a 22-Bit Mapped System

Virtual and Physical Addresses

Virtual addresses are used within a task itself. They are always 16-bit addresses in the range Ø(8) to 177777(8), or 32K words. When a task is task-built, virtual addresses are assigned typically starting at Ø(8) at the beginning of the task.

Physical addresses are used in physical memory, the I/O page, and, with 22-bit systems only, the UNIBUS map. They begin with Ø(8) at the beginning of memory and include all of physical memory, the UNIBUS map, and the I/O page.

On an unmapped system, no address relocation is performed. Therefore, virtual addresses match physical addresses. Figure 5-4 shows a task's virtual addresses and the corresponding physical addresses in an unmapped system. The task is loaded beginning at physical address 6ØØØØ(8), and addresses referenced in the task code reference physical addresses directly.

On a mapped system, the memory management hardware translates or "maps" a task's virtual addresses to the physical addresses in physical memory where the task is actually loaded. In the simplest case, the virtual addresses are offsets from a base physical address where the task is loaded. If a task is later relocated to another location in physical memory, the virtual addresses are then offset from the new base physical address.

Figure 5-5 shows a task loaded at two different locations. As shown below, at time 1, virtual address 1534(8) in the task is at the location 425134(8) in physical memory. At time 2, virtual address 1534(8) in the task is at location 141534(8) in physical memory. Since all addresses are converted at execution time, references to location 1534(8) in the task are resolved correctly regardless of where the task is loaded in physical memory.

```
243400(8) Base physical address
  1534(8) Offset (task virtual address)
---------
425134(8) Actual physical address

14ØØØØ(8) Base physical address
  1534(8) Offset (task virtual Address)
---------
141534(8) Actual physical address
```

On a mapped system, the Task Builder fixes a task's code in virtual address space, but the actual mapping of virtual addresses to physical addresses is performed at run time by the memory management unit. Tasks may be loaded at different physical addresses and still run correctly. As you will see later, mapping also allows a task to access several separate pieces of physical memory.

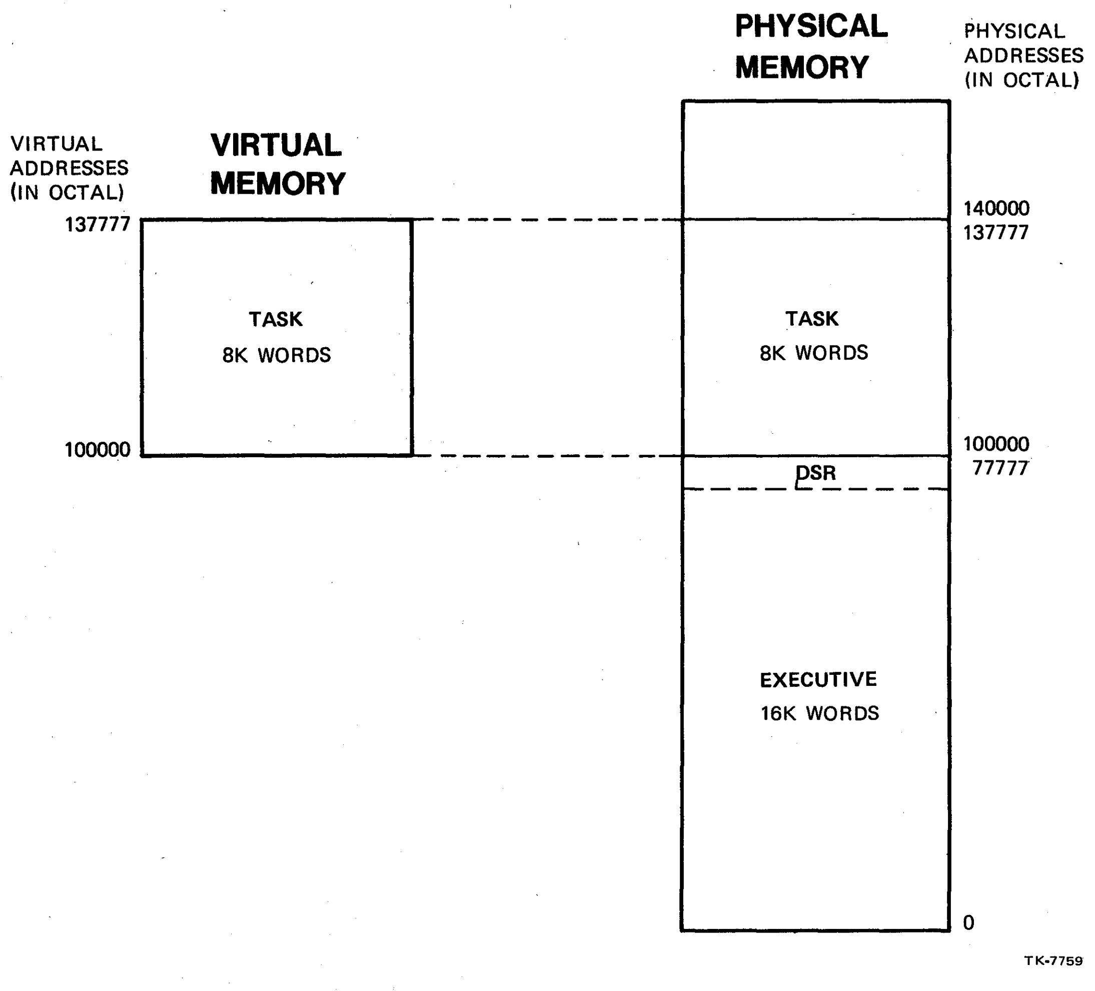


Figure 5-4 Virtual Addresses Versus Physical Addresses on an Unmapped System

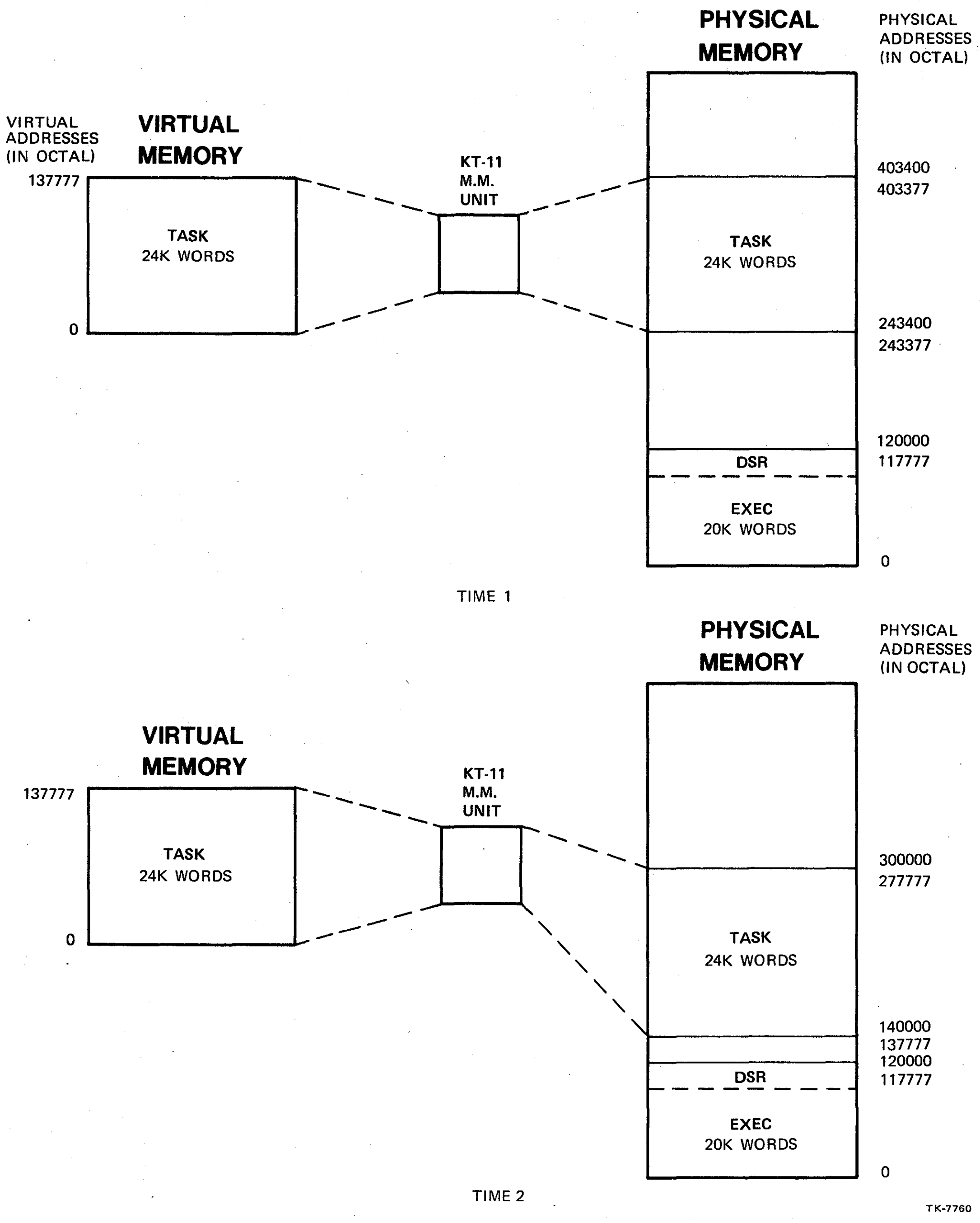


Figure 5-5 Virtual Addresses Versus Physical Addresses on a Mapped System

The KT-11 Memory Management Unit

Mode Bits - Bits 15 and 14 and bits 13 and 12 of the processor status word (PSW) indicate, respectively, the current and previous modes of processor operation. The mode may be:

- Kernel mode (ØØ)
- User mode (11)
- Supervisor mode (Ø1). (Supervisor mode is not used on RSX-11M, and is available only on 11/45, 11/55, 11/44, and 11/7Ø.)

The purpose of having different processor modes is to provide for a privileged mode (kernel) where the Executive can execute privileged instructions (e.g., HALT), and can manipulate privileged locations (e.g., PSW), and a non-privileged and protected mode (user) where tasks usually execute.

Active Page Registers (APRs) - The Active Page Registers (APRs) in the KT-11 memory management unit are used to define the mapping or correspondence between virtual and physical addresses. On an RSX-11M system, one set of eight APRs is used at a time to define this mapping. There is one set of APR's used for each processor mode; one is used in user mode and another set is used in kernel mode.

At any given time, the set of APRs in use is determined by the mode bits in the processor status word. Each APR in the set in use maps a specific range of virtual addresses, as shown in Table 5-2. The APR can map zero words, if not in use, up to the full 4K words, always in even multiples of 32 words. In actuality, the hardware may contain additional sets of APRs, but they are not used under RSX-11M.

Each APR consists of two 16-bit registers, a page address register (PAR) and a page descriptor register (PDR). The page address register contains a base address used in mapping the appropriate range of virtual addresses.

Table 5-2 APR and Virtual Address Correspondence

APR Number	Virtual Address Range	K Words
7	160000-177777(8)	28-32K
6	140000-157777(8)	24-28K
5	120000-137777(8)	20-24K
4	100000-117777(8)	16-20K
3	60000- 77777(8)	12-16K
2	40000- 57777(8)	8-12K
1	20000- 37777(8)	4- 8K
0	0- 17777(8)	0- 4K

Because the page address register contains only 16 bits, but the actual physical addresses on the larger PDP-11's contain 18 or 22 bits, the 16 bits cannot contain an actual physical address. Instead, the 16 bits contain a block number, which corresponds to the high-order 16 bits (12 bits with 18-bit addressing) of the actual physical address. A block of memory is 32(10) words (= 64(10) bytes = 100(8) bytes) long and starts on a 100(8) boundary. Therefore, the base physical address 00134200(8) is the start of block number 001342(8) and the base address 12445700(8) is the start of block number 124457(8).

To obtain the block number from a physical address which ends in at least two octal zeros, just strip off the last two zeros from the actual address. To obtain the physical address from the block number, append two zeros to the end of the block number in octal.

The page descriptor register (PDR) contains information about the page of memory in use, such as the length of the page (up to 4K words) and the access rights (read/write, read-only, etc.). The fields for length and access rights in the PDR provide the capability for hardware memory protection. If any reference in a task is beyond the actual area in use or violates the access rights, a memory protect violation is reported.

For a more complete description of the PARs and the PDRs, see Chapter 6 of the PDP-11 Processor Handbook.

Figure 5-6 shows the values in the page address registers for an example task. The main part of the task is 14K words long; therefore it needs four APRs; three APRs (APR 0,1, and 2) mapping 4K words each, and a fourth APR (APR3) mapping the last 2K words. The base physical address of the task is 00432400(8), which is obtained by converting the block number 004324(8) to a byte address.

All virtual addresses within the main task area are mapped to physical addresses beginning at location ØØ4324ØØ(8). This means in effect that each virtual address corresponds to an offset from location ØØ4324ØØ(8). The page descriptor registers, not illustrated, indicate that APRs Ø, 1, and 2 map 4K words each, but that APR 3 maps only 2K words.

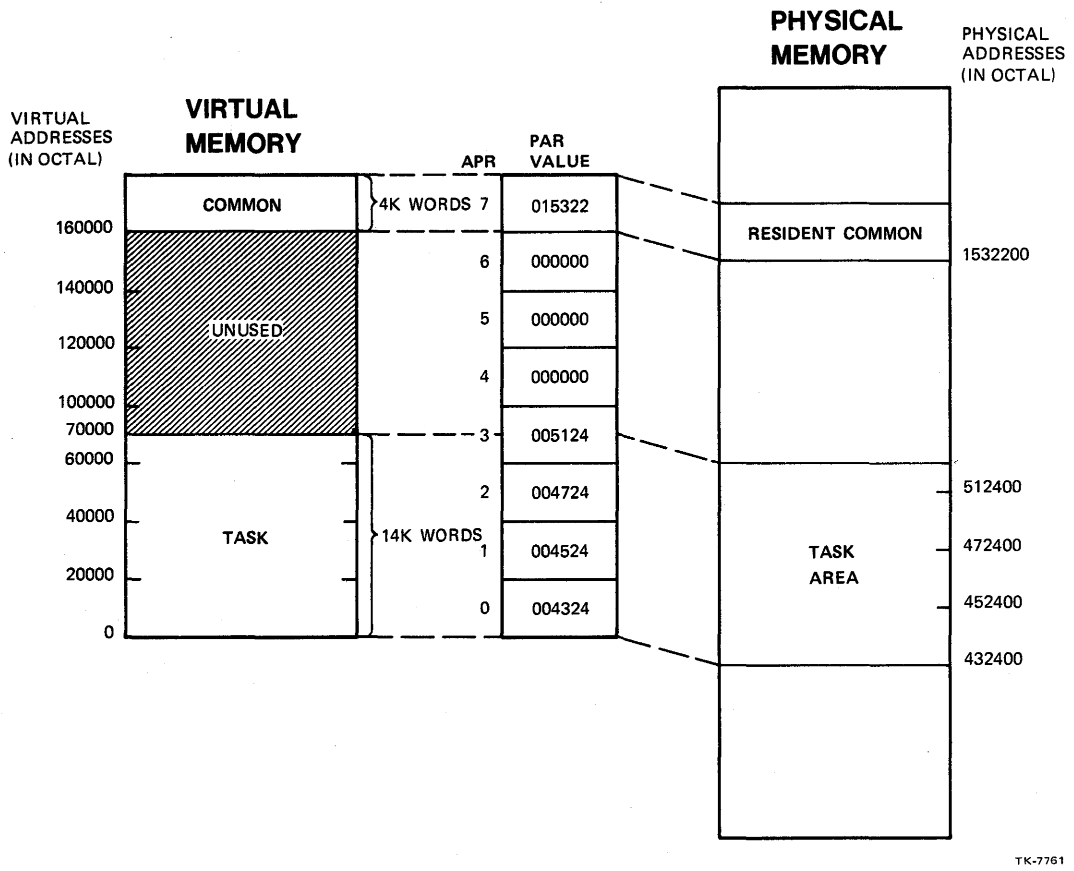


Figure 5-6 Page Address Registers Used in Mapping a Task

The task in Figure 5-6 is also mapped to a resident common. APR 7 is used to map this 4K word area, beginning at location Ø1532ØØ(8) in physical memory. Therefore, virtual addresses from 16ØØØØ(8) to 177777(8) map to physical addresses Ø15322ØØ(8) to Ø1732177(8). Virtual address 1653414(8) corresponds to physical address Ø15322ØØ(8) + [1653414(8)-16ØØØØ(8)] = Ø16Ø5614(8).

Note that a task can be loaded into a minimum of one or a maximum of eight separate contiguous areas of memory, because each APR must map to a contiguous area of memory. If a 32K word task is loaded into one large contiguous area, eight APRs are still used, but each APR maps only part of the large contiguous area.

Converting Virtual Addresses to Physical Addresses

The following two examples show how the KT-11 memory management unit converts virtual addresses to physical addresses for the task shown in Figure 5-6.

Example 1

The KT-11 unit takes a virtual address and uses the value in the appropriate APR to convert it to a physical address. The virtual address range indicates which APR to use (Table 5-2).

Since Ø53422(8) is in the range Ø4ØØØØ-Ø57777, APR 2 is used. Or, looking at the address in binary, the high-order three bits indicate which APR to use. Bits Ø through 12 indicate the displacement or offset from the base physical address of the page. This is equal to Ø53422(8) - Ø4ØØØØ(8) = 13422(8), the distance of this virtual address from the base virtual address for this APR.

```
                 Active
                 Page          Displacement
                 Field         Field
               +-------+---------------------------+
Ø53422(8) =    | Ø 1 Ø | 1 Ø 1 1 1 Ø Ø Ø 1 Ø Ø 1 Ø | (2)
               +-------+---------------------------+
                  2              13422(8)
                 APR             Offset
```

In easier terms, virtual address 40000(8) will be located at the base physical address. A virtual address 13422(8) bytes above that will be 13422(8) bytes above that physical location. The base physical address is determined by converting the block number in APR2, 004724(8), to the physical address 00472400(8). (Recall that a block of memory is 100(8) bytes.) Therefore, address 053422(8) is mapped to the location shown below.

```
  00472400(8) Base physical address
+    13422(8) Displacement
  -----------
  00506022(8) Actual physical address
```

Example 2

Convert the virtual address 165275(8)

```
            +-------+---------------------------+
165275(8) = | 1 1 1 | 0 1 0 1 0 1 0 1 1 1 1 0 1 | (2)
            +-------+---------------------------+
                7              05275(8)
               APR             Offset

APR 7 = 015322(8) blocks = 01532200(8) Base physical address
                         +    05275(8) Displacement
                           -----------
                           01537475(8) Actual physical address
```

The memory management unit performs this conversion using an adder and a number of internal registers. The conversion is performed at extremely fast speeds. Chapter 6 of the PDP-11 Processor Handbook discusses this conversion process in more detail.

SOFTWARE CONCEPTS

Virtual Address Windows

A virtual address window, or simply a window, is a contiguous range of virtual addresses within a task. A window is always mapped as a unit, to a contiguous range of physical locations. A task which resides in a single contiguous area of physical memory generally has a single window, called the task window, which is mapped to the area. An example of this was shown in Figure 5-5, which has a single window 24K words long. Notice that the window is the same at time 1 and time 2, but it maps to different locations in physical memory. On the other hand, a task which must access two separate pieces of physical memory at the same time would have two windows (as in Figure 5-6) to map those areas.

Windows are mapped using APRs. A virtual address window always corresponds to at least one, but possibly more than one, APR (up to all eight). A window always begins at a 4K word boundary, corresponding to the lowest address mapped by the first APR used for the window. Successively higher APRs are then used, until the entire window is provided for. The Task Builder assigns most virtual addresses and creates most windows, determining which APRs will be used during that task's execution.

The task window for a task begins at virtual address Ø (therefore using APR Ø) and extends upward as far as necessary to accommodate the task's header, stack, main code and data. Other windows can begin at any 4K word boundary above the high limit of the task window. Typically, additional windows are assigned from the top of virtual address space working downwards. For example, if an additional address window of 4K words or less is needed, it is assigned a base address of 16ØØØØ(8), using APR 7.

If, on the other hand, a window is needed between 4K words and 8K words in size, the window will be given a base address of 14ØØØØ(8). In this case, the window would use APRs 6 and 7. Additional windows would be assigned lower base addresses that correspond to other available APRs.

NOTE

Under no circumstance can two windows exist at the same time within a task using the same APR or the same virtual addresses.

The Task Builder allocates space in the task header for the windows it has created and records information that specifies how these windows are to be mapped. This information is used to load the APRs with appropriate values before the task executes.

Memory management directives can be used to create and initialize additional windows while a task executes. Space for these additional windows must be allocated in the task header at task-build time, using the "WNDWS" option. Memory management directives and their use are discussed in Module 8 on Dynamic Regions.

Regions

A region is a contiguous area of physical memory to which a task may get access rights. A region must be contained completely within a partition. It can be part of a partition or the entire partition.

There are three types of regions in an RSX-11M system.

1. Task region - an area in a user-controlled partition or a system-controlled partition into which a task is loaded and then executes.

2. Static Common Region - an area in a common type partition; e.g., a shared common for data or a shared library for code.

3. Dynamic Region - an area in a system-controlled partition which is created dynamically, at run time, using the memory management directives.

A task gets access rights to a region by "attaching" to the region. Before the Executive attaches a task to a region, it checks its needed access against the protection on the region. This is similar to checking file protection before allowing file access. If the task passes the check on access rights, then the Executive attaches the task to the region by establishing a connection between the two. The total amount of physical memory, made up of regions, to which a task is attached is called a task's logical address space.

After a task is attached to a region, it actually accesses or uses the region by first "mapping" one of its virtual address windows to a part or to all of the region. During this process, the Executive uses the window and region information to fill in the APRs. After this, references in the task to virtual addresses in that window map to physical addresses within the region. A region does not have to be the same size as a window. Generally it is of equal or larger size than the window.

Attaching and mapping are done automatically by the Executive for regions linked to at task-build time. Alternatively, a task can use memory management directives at run time to dynamically create regions, attach to regions, and map windows to regions.

Figure 5-7 shows a task which has three virtual address windows mapped to three different regions. Figure 5-8 shows the same task after it attaches to another region (the work area) and maps to it. Notice that virtual addresses beginning at 1ØØØØØ(8) are used to map this region. For example, this area might be used as a temporary work buffer. Figure 5-8 does not include the actual PAR values or the actual physical addresses. This simpler form of mapping diagram will be used from now on in this course to make things easier, unless the actual PAR values and physical addresses are significant to the discussion.

Now do the tests/exercises for this module in the Tests/Exercises book. They are all written problems. Check your answers against those provided in that book.

If you think that you have mastered the material, ask your course administrator to record your progress in your Personal Progress Plotter. You will then be ready to begin a new module.

If you think that you have not yet mastered the material, return to this module for further study.

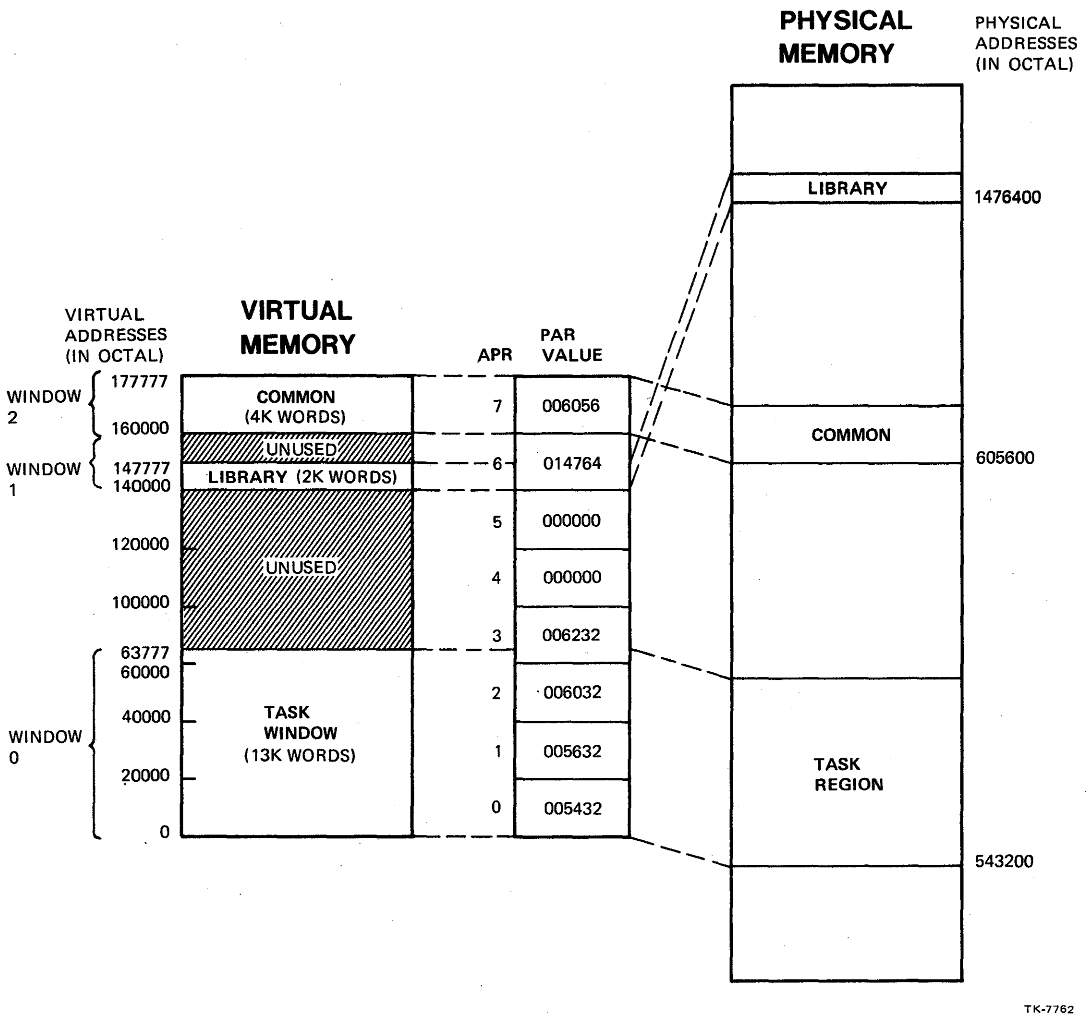


Figure 5-7 A Task with Three Windows Mapped to Three Regions

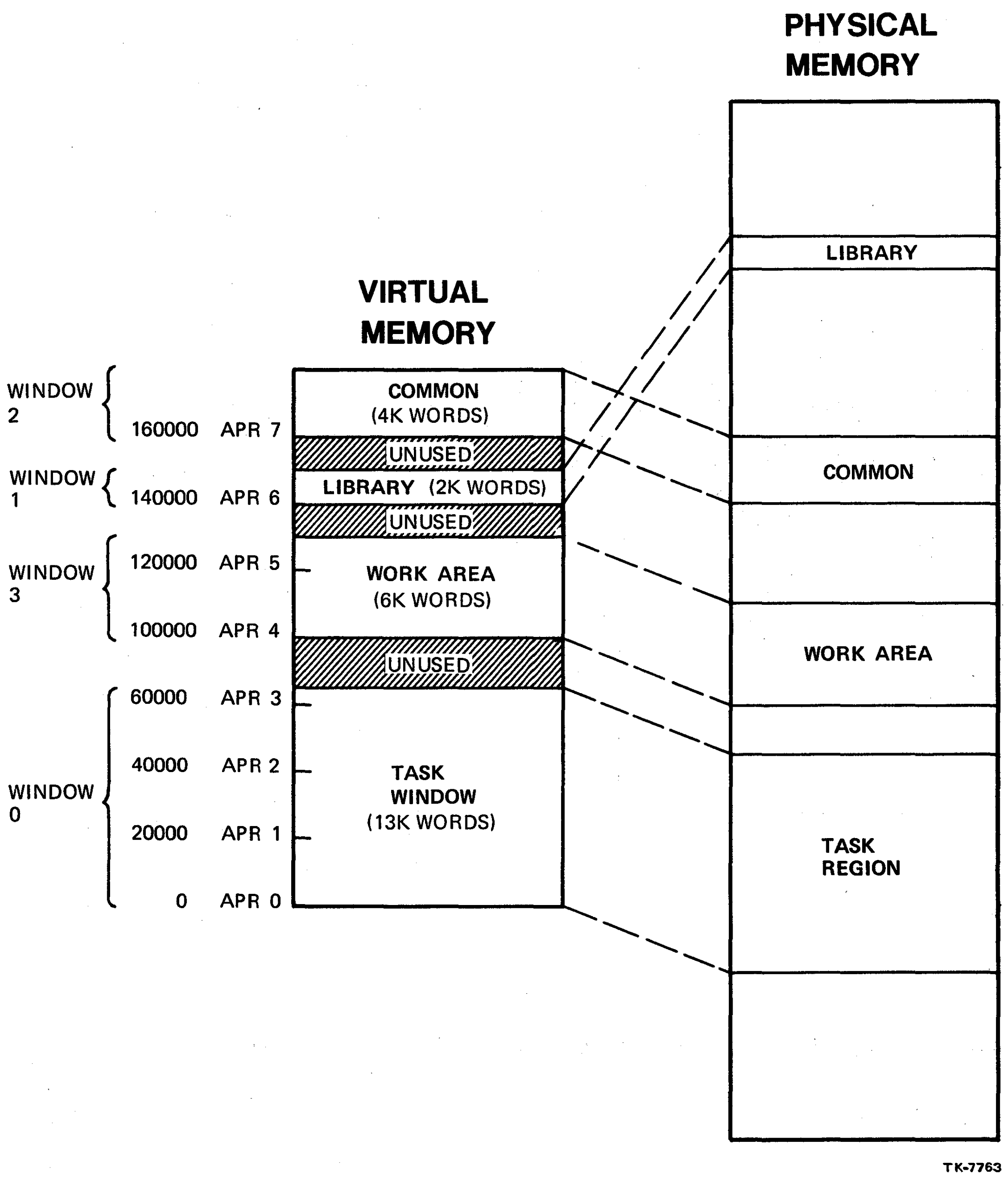


Figure 5-8 Task in Figure 5-7 after Attaching to and Mapping to a Fourth Region

Digital Equipment Corporation • Bedford, MA 01730