EY-0060E-SG-0101

Programming RSX-11M in MACRO

A Self-Paced Course

Volume I

digital

EY-0060E-SG-0101

Programming RSX-11M in MACRO

A Self-Paced Course

Student Workbook
Volume I

Prepared by Educational Services
of
Digital Equipment Corporation

Copyright © 1982, Digital Equipment Corporation.
All Rights Reserved.

The reproduction of this material, in part or whole, is strictly prohibited. For copy information, contact the Educational Services Department, Digital Equipment Corporation, Bedford, Massachusetts 01730.

Printed in U.S.A.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may not be used or copied except in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECSYSTEM-20	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	RSTS
UNIBUS	VAX	RSX
	VMS	IAS

9/82-14

CONTENTS

VOLUME I

SG STUDENT GUIDE

INTRODUCTION 3
PREREQUISITES. 4
COURSE GOALS AND NONGOALS. 4
COURSE ORGANIZATION. 5
COURSE MAP DESCRIPTION 5
COURSE MAP 6
COURSE RESOURCES 7
Required References. 7
Optional References. 7
HOW TO TAKE THE COURSE 8
PERSONAL PROGRESS PLOTTER. 13

1 USING SYSTEM SERVICES

INTRODUCTION 17
OBJECTIVES 17
RESOURCES. 17
WHAT IS A SYSTEM SERVICE?. 19
WHY SHOULD YOU USE SYSTEM SERVICES?. 19
To Extend the Features of Your Programming Language 19
To Ease Programming and Maintenance. 19
To Increase Performance. 20
WHAT SERVICES ARE PROVIDED?. 20
System and Task Information. 20
Task Control 21
Task Communication and Coordination. 21
I/O Peripheral Devices 21
File and Record Access 21
File and Record Access Systems 22
Memory Use 22
OTHER SERVICES AVAILABLE 23
HOW SERVICES ARE PROVIDED. 25
Executive Directives 25
Code Inserted into Your Task Image 28
SYSTEM LIBRARIES 30

2 DIRECTIVES

INTRODUCTION 35
OBJECTIVES 35
RESOURCES. 35
INVOKING EXECUTIVE DIRECTIVES FROM A USER TASK 37
Directive Processing 37

Functions Available Through Executive Directives 39
The Directive Parameter Block (DPB). 41
The Directive Status Word (DSW). 42
Sample Program .43
DIFFERENT FORMS OF THE DIRECTIVE CALLS 46
The $ Form 46
The $C Form. 49
The $S Form. 51
Repeated Use of a Directive with Different Arguments. 58
ADDITIONAL DIRECTIVE CONSIDERATIONS. 62
An Alternative Method for Error Checking 62
Run Time Conversion Routines 68
Notifying a Task When an Event Occurs. 69
Event Flags. 69
Using Event Flags for Synchronization. 70
Asynchronous System Traps (ASTs) 75
Synchronous System Traps (SSTs). 82

3 USING THE QIO DIRECTIVE

INTRODUCTION 91
OBJECTIVES 91
RESOURCES. 91
OVERVIEW OF QIO DIRECTIVES 93
PERFORMING I/O 93
I/O FUNCTIONS. 94
Logical Unit Numbers (LUN) 95
Synchronous and Asynchronous I/O 95
MAKING THE I/O REQUEST 101
Error Checking and the I/O Status Block. 103
THE QIO DIRECTIVES 105
Synchronous I/O. 105
Asynchronous I/O 111
Synchronization With Asynchronous I/O. 112
TERMINAL I/O 120
Device Specific Functions. 120
I/O Status Block and Terminating Characters. 120
Read After Prompt. 123
Read No Echo 126
Read with Timeout. 128
Terminal-Independent Cursor Control. 131
Formatting Output Data 135
Formatting ASCII Data. 145

4 USING DIRECTIVES FOR INTERTASK COMMUNICATION

INTRODUCTION 151
OBJECTIVES 151
RESOURCE 151
USING TASK CONTROL DIRECTIVES AND EVENT FLAGS. 153
Directives 154
SEND/RECEIVE DIRECTIVES. 163
General Concepts 163
Directives 163
Synchronizing Send Requests With Receive Requests 164
Using Send/Receive Directives for Synchronization. 181
Slaving the Receiving Task 181
PARENT/OFFSPRING TASKING 182
Directives Issued by a Parent Task 184
Directives Issued by an Offspring Task 194
Chaining of Parent/Offspring Relationships 195
Other Parent/Offspring Considerations. 201
Task Abort Status. 206
Summary of Various Methods of Data Transfer Between Tasks. 208
Other Methods of Transferring or Sharing Data Between Tasks. 209

5 MEMORY MANAGEMENT CONCEPTS

INTRODUCTION 213
OBJECTIVES 213
RESOURCES. 213
GOALS OF MEMORY MANAGEMENT 215
HARDWARE CONCEPTS. 215
Mapped Versus Unmapped Systems 215
Virtual and Physical Addresses 220
The KT-11 Memory Management Unit 223
Mode Bits. 223
Active Page Registers (APRs) 223
Converting Virtual Addresses to Physical Addresses. 226
SOFTWARE CONCEPTS. 228
Virtual Address Windows. 228
Regions. 229

6 OVERLAYS

INTRODUCTION 235
OBJECTIVES 235
RESOURCE 235
CONCEPTS 237
OVERLAY STRUCTURE. 238
STEPS IN PROGRAM DEVELOPMENT USING OVERLAYS. 241
THE OVERLAY DESCRIPTOR LANGUAGE (ODL). 241
    ODL Command Line Format. 241
TYPES OF OVERLAYS. 245
    Disk-Resident. 245
    Memory-Resident. 247
LOADING METHODS. 251
    Autoload 251
    Manual Load. 253
    Comparison of a Task With No Overlays,
    to One With Disk-Resident Overlays, and
    One With Memory-Resident Overlays. 253
    Overlaying Techniques. 254
LIBRARIES. 262
GLOBAL SYMBOLS IN OVERLAID TASKS 268
    Resolution of Global Symbols 268
    Subroutine Calls 271
    Data References. 271
    Placing Data in the Root and Referencing It. . . . 272
CO-TREES 282

VOLUME II

7 STATIC REGIONS

INTRODUCTION 289
OBJECTIVES 289
RESOURCE 289
TYPES OF STATIC REGIONS. 291
MEMORY ALLOCATION. 293
MAPPING. 293
REFERENCES TO A SHARED REGION. 299
    Techniques of Referencing. 3Ø1
    Using Overlaid Psects (Data Only). 3Ø1
    Using Global Symbols (Data or Subroutines) 3Ø2
    Using Virtual Addresses (Data Only). 3Ø3
PROCEDURE FOR CREATING SHARED REGIONS
AND REFERENCING TASKS. 3Ø7
    Creating a Resident Common or Resident Library . . 3Ø7
    Creating a Referencing Task. 315
DEVICE COMMONS 326

8 DYNAMIC REGIONS

INTRODUCTION 337
OBJECTIVES 337
RESOURCE 337
SYSTEM FACILITIES. 339
REQUIRED DATA STRUCTURES 341
    Region Definition Block (RDB). 341
    Creating an RDB in MACRO-11. 345
    Window Definition Block (WDB). 347
    Creating a WDB in MACRO-11 349
CREATING AND ACCESSING A REGION. 351
    Creating a Region. 352
    Attaching to a Region. 355
    Creating a Virtual Address Window. 356
    Mapping to a Region. 356
SEND- AND RECEIVE-BY-REFERENCE 365
    The Mapped Array Area. 373

9 FILE I/O

INTRODUCTION 383
OBJECTIVES 383
RESOURCES. 383
OVERVIEW 385
TYPES OF DEVICES 385
    Record-Oriented Devices. 385
    File-Structured Devices. 385
        Types of File-Structured Devices 386
COMMON CONCEPTS OF FILE I/O. 388
    Common Operations. 388
    Steps of File I/O. 388
FILES-11 389
    FILES-11 Structure 389
    Directories. 394
    Five Basic System Files. 397
    Functions of the ACP 398
OVERVIEW AND COMPARISON OF FCS AND RMS 401
    Common Functions 401
FCS FEATURES 403
    File Organizations 403
    Supported Record Types 403
    Record Access Modes. 407
    File Sharing 409
RMS FEATURES 410
    File Organizations 410
    Record Formats 412
    Record Access Modes. 412
    File Sharing Features. 414
    Summary. 415

10 FILE CONTROL SERVICES

INTRODUCTION 419
OBJECTIVES 419
RESOURCE 419
REVIEW OF FILE I/O 421
INTRODUCTORY EXAMPLE 422
USING FCS 427
Preparing to Open a File 427
Initialization of the FSR. 429
The File Descriptor Block (FDB). 431
Functions of the FDB 431
Allocating Space for FDBs. 432
Initializing an FDB. 432
Specifying New File Characteristics. 433
Selecting Data Access Methods. 435
Specifying Data Access Methods 437
Additional Initialization of the FDB
for Record I/O 438
Additional Initialization for Block I/O. 439
Initializing the File-Open Section of FDB. 440
Setting Up a File Specification in the FDB . . 440
Setting Up the Dataset Descriptor. 441
Setting Up the Default Filename Block. 442
Initializing the File-Open Section
Prior to Opening the File. 443
Opening a File 450
ERROR CHECKING 453
PERFORMING RECORD I/O. 456
Different Forms of PUT$ and GET$ 456
Sequential Access. 457
Random Access. 459
Closing the File 460
PERFORMING BLOCK I/O 477
READ$ and WRITE$ Calls 477
Synchronization and Error Checking 478
ADDITIONAL TOPICS. 487
Deleting a File. 487
File Control Routines. 487
Command Line Processing. 488

AP APPENDICES

APPENDIX A SUPPLIED MACROS. 491
APPENDIX B CONVERSION TABLES. 513
APPENDIX C FORTRAN/MACRO-11 INTERFACE 515
APPENDIX D PRIVILEGED TASKS 517
APPENDIX E TASK BUILDER USE OF PSECT ATTRIBUTES 519

APPENDIX F ADDITIONAL SHARED REGION TOPICS. 523
APPENDIX G ADDITIONAL EXAMPLES. 537
APPENDIX H LEARNING ACTIVITY ANSWER SHEET 541

GL GLOSSARY

FIGURES

1-1 Using Executive Directives to Service a Task. 26
1-2 Using Executive Directives to Receive Services from Other Tasks. 27
1-3 Code Inserted into Your Task Image. 29

2-1 Directive Implementation. 39
2-2 The Directive Parameter Block 41
2-3 The $ Form. 47
2-4 The $C Form 50
2-5 The $S Form 52
2-6 AST Mechanics 76
2-7 Stack as Set Up by the Executive for ASTs 78
2-8 SST Sequence. 84
3-1 Execution of a Synchronous I/O Request. 97
3-2 Events in Synchronous I/O 97
3-3 Execution of an Asynchronous I/O Request. 100
3-4 Events in Asynchronous I/O. 100

4-1 Parent/Offspring Communication Facilities 183
4-2 Spawning Versus Chaining (Request and Pass Offspring Information). 195
5-1 Physical Address Space in an Unmapped System. 217
5-2 Physical Address Space in an 18-Bit Mapped System . . . 218
5-3 Physical Address Space in a 22-Bit Mapped System. . . . 219
5-4 Virtual Addresses Versus Physical Addresses on an Unmapped System 221
5-5 Virtual Addresses Versus Physical Addresses on a Mapped System. 222
5-6 Page Address Registers Used in Mapping a Task 225
5-7 A Task with Three Windows to Three Regions. 231
5-8 Task in Figure 5-7 After Attaching to and Mapping to a Fourth Region. 232

6-1 A Memory Allocation Diagram 240
6-2 An Overlay Tree 240
6-3 An Example of Disk-Resident Overlays. 246
6-4 An Example of Memory-Resident Overlays. 249
6-5 Task With Two Overlay Segments. 263
6-6 Resolution of Global Symbols. 270

6-7 Use of Co-Trees 283
6-8 Task With Co-Trees. 284

7-1 Tasks Using a Position Independent Shared Region. . . 295
7-2 Tasks Using an Absolute Shared Region 297
7-3 Program Development for Shared Regions. 300
8-1 The Region Definition Block 342
8-2 The Window Definition Block 348
8-3 The Mapped Array Area 375

9-1 Example of Virtual Block to Logical Block, to Physical Location Mapping. 391
9-2 How the Operating System Converts Between Virtual, Logical, and Physical Blocks 392
9-3 FILES-11 Structures Used to Support Virtual-to-Logical Block Mapping 393
9-4 Directory and File Organization on a Volume 395
9-5 Locating a File on a FILES-11 Volume. 396
9-6 Flow of Control During the Processing of an I/O Request 400
9-7 Move Mode and Locate Mode 402
9-8 Sequential Files. 403
9-9 RMS File Organizations. 411

10-1 The File Storage Region 426
10-2 Move Mode Versus Locate Mode for Record I/O 428
10-3 Block I/O Operations. 429
10-4 The File Descriptor Block 431

F-1 A Shared Region With Memory-Resident Overlays 524
F-2 Referencing Two Resident Libraries. 526
F-3 Referencing Combined Libraries. 528
F-4 Building One Library, Then Building a Referencing Library 530
F-5 Revectoring 531
F-6 Using Revectoring When Referencing Library Has Overlays. 533
F-7 Cluster Libraries 535

TABLES

SG-1 Typical Course Schedules. 12

1-1 Examples of Use of Other Services 24
1-2 Standard Libraries. 30
1-3 Resident Libraries. 32

2-1 Types of Directives 40
2-2 Summary of Directive Forms. 61

3-1 Common (Standard) I/O Function Codes. 94
3-2 I/O Parameter List for Standard I/O Functions 102
3-3 Some Special Terminal Function Codes. 122
3-4 Sample Editing Directives for $EDMSG. 137

4-1 Task Control Directives and Their Use
for Synchronizing Tasks 155
4-2 Stopping Compared to Suspending or Waiting. 156
4-3 Event Flag Directives and Their Use
for Synchronizing Tasks 156
4-4 The Send/Receive Data Directive 164
4-5 Methods of Synchronizing a Receiving Task (RECEIV)
with a Sending Task (SEND). 165
4-6 Standard Exit Status Codes. 184
4-7 Comparison of Parent Directives 185
4-8 Directives Used by a Task to Establish
a Parent/Offspring Relationship 186
4-9 Directives Which Return Status to a Parent Task . . . 194
4-10 Directives Which Pass Parent/Offspring Connections
to Other Tasks. 196
4-11 Task Abort Status Codes 207
4-12 Comparison of Methods of Data Transfer
Between Tasks 208

5-1 Mapped Versus Unmapped Systems. 216
5-2 APR and Virtual Address Correspondence. 224

6-1 Comparison of Overlaying Methods. 260
6-2 How Global Symbols Are Resolved 269

7-1 Types of Static Regions Available on RSX-11M. 292
7-2 Techniques of Referencing a Shared Region 305
7-3 Effect of /CODE:PIC, /SHAREABLE:COMMON, and
/SHAREABLE:LIBRARY on a Shared Region's STB 306
7-4 Required Switches and Options for Building
a Shared Region 309

8-1 Memory Management Directives. 340
8-2 Region Status Word. 344
8-3 Window Status Word. 349

9-1 Comparison of Physical, Logical and Virtual Blocks. . 390
9-2 Examples of Use of F11ACP Functions 399
9-3 Comparison of FCS Record Types. 406
9-4 Comparison of Sequential Access I/O and
Random Access I/O 408
9-5 File Organization, Record Formats, and Access Modes . 413
9-6 Comparison of FCS and RMS 415

10-1 When the User Record Buffer Is Needed 436
10-2 Types of Access 445

B-1 Decimal/Octal, Word/Byte/Block Conversions. 513
B-2 APR/Virtual Addresses/Words Conversions 513

EXAMPLES

2-1 Requesting a Task .45
2-2 Using thec $ Form of the Directives 54
2-3 Using the $C Form of the Directives 56
2-4 Using the $S Form of the Directives 57
2-5 Using Several Directives. 66
2-6 Waiting for an Event Flag 72
2-7 Setting an Event Flag in a Task 74
2-8 Using a Requested Exit AST. 79
2-9 Using an AST in the Mark Time Directive 81
2-10 Using SSTs. 86

3-1 Synchronous I/O 109
3-2 Asynchronous I/O Using Event Flags for Synchronization 114
3-3 Asynchronous I/O Using an AST for Synchronization . . 118
3-4 Prompting for Input 124
3-5 Read No Echo. 127
3-6 Read With Timeout 129
3-7 Terminal Independent Cursor Control 133
3-8 Formatting Numeric Data 140
3-9 Formatting Directive and I/O Error Messages 143
3-10 Formatting ASCII Data 146

4-1 Synchronizing Tasks Using Suspend and Resume. 158
4-2 Synchronizing Tasks Using Event Flags 161
4-3 Synchronizing a Receiving Task Using Event Flags. . . 168
4-4 A Receiving Task Which Can be Run Before or After the Sender. 173
4-5 Synchronizing a Receiving Task Using RCDS$. 178
4-6 A Task Which Spawns PIP 188
4-7 A Generalized Spawning Task 191
4-8 An Offspring Task Which Chains its Parent/Offspring Connection to PIP 198
4-9 A Spawned Task Which Retrieves a Command Line 203

6-1 Description of An Overlaid Task 239
6-2 Map File of Example 6-1 Without Overlays. 255
6-3 Map File of Example 6-1 With Disk-Resident Overlays. 257

6-4 Map File of Example 6-1 With Memory-Resident Overlays. 259
6-5 A Task With Two Overlay Segments. 266
6-6 Complex Example Using Overlays. 276

7-1 Resident Common Referenced With Overlaid Psects . . . 313
7-2 Resident Common Referenced With Global Symbols. . . . 320
7-3 Shared Library. 324
7-4 Creating and Using a Device Common. 331

8-1 Creating a Named Region 354
8-2 Creating a Region and Placing Data in It. 359
8-3 Attaching to an Existing Region and Reading Data From It 363
8-4 Send-by-Reference 368
8-5 Receive-by-Reference. 371
8-6 Use of the Mapped Array Area. 378

10-1 Creating a File in MACRO-11 424
10-2 Creating a File of Fixed Length Records, Initializing FDB at Assembly Time. 463
10-3 Creating a File of Fixed Length Records, Initializing FDB at Run Time 467
10-4 Accessing a File in Locate Mode 470
10-5 Accessing a File in Random Mode 474
10-6 Creating a File With Block I/O. 480
10-7 Reading a File With Block I/O 484

G-1 Reading the Event Flags (for Exercise 1-1). 537
G-2 Using the Routines GCML and CSI (for Exercise 10-6) . 538

SG

STUDENT GUIDE

INTRODUCTION

Programming RSX-11M in MACRO is intended for MACRO-11 programmers who use services of the RSX-11M operating system beyond those provided by the MACRO-11 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 MACRO-11.

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

6. Use File Control Services to create and maintain files.

This course does <u>not</u> teach the following:

1. The PDP-11 instruction set and the MACRO-11 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 and exercises (bound separately) which you can use to measure your achievement. Solutions are provided for all exercises.

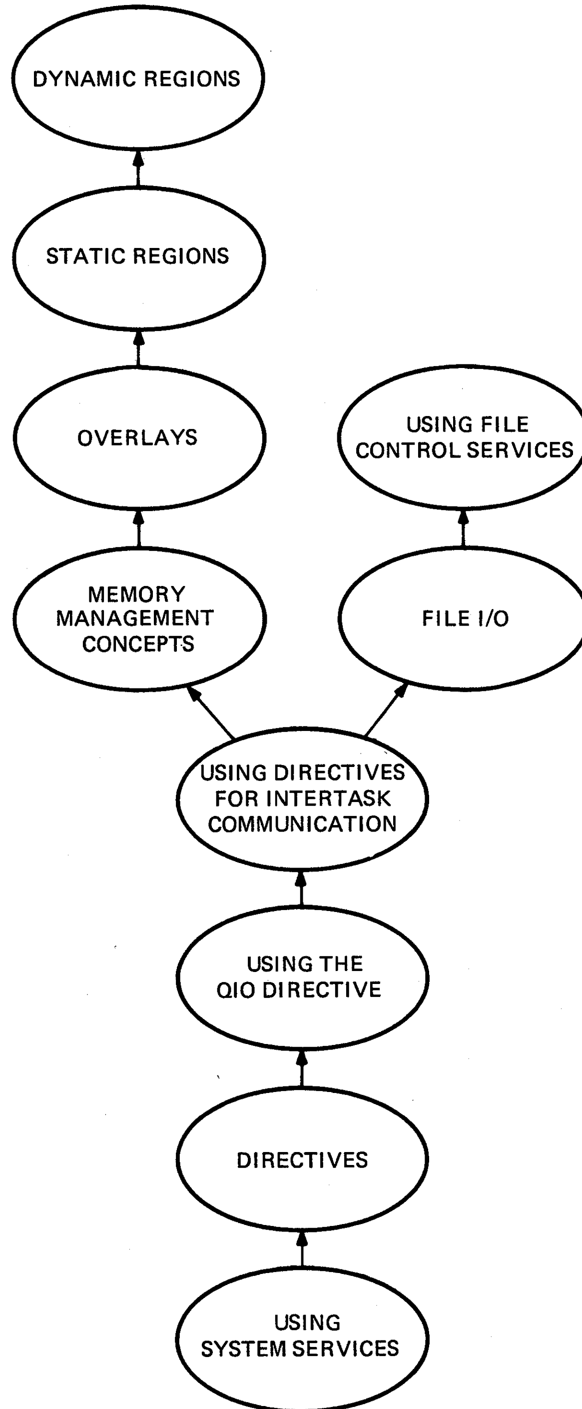
The course is bound in two volumes. The first volume contains this student guide, the 10 modules (except for their tests/exercises), the appendices, and a glossary. The second volume contains the tests/exercises for each module.

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 10.

COURSE MAP



COURSE RESOURCES

Required References

1. IAS/RSX-11 I/O Operations Manual (AA-M176A-TC)
2. IAS/RSX-11 System Library Routines Reference Manual (AA-5580A-TC)
3. RSX-11M Mini-Reference (AV-5570D-TC)
4. RSX-11M/M-PLUS Executive Reference Manual (AA-L675A-TC)
5. RSX-11M/M-PLUS I/O Drivers Reference Manual (AA-L677A-TC)
6. RSX-11M/M-PLUS Task Builder Manual (AA-L680A-TC)

Optional References

1. PDP-11 MACRO-11 Language Reference Manual (AA-5075B-TC)
2. PDP-11 Processor Handbook (EB-19402-20/81)
3. RMS-11 User's Guide (AA-D538A-TC)
4. RMS-11 MACRO-11 Reference Manual (AA-H683A-TC)

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, system library routines, and File Control Service calls.

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 [202,1] 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 [2Ø2,1] 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 [2Ø2,1] will remain intact for other students.

Each example program contains the following:

- Source code, with line numbers added
- A sample run session
- Bulleted items which are described in the text.

Line numbers have been added to the source code to ease referencing during a group discussion. These line numbers are not part of the actual source file. The source code also 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 assemble 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, assemble 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 FORTRAN course. Do all tests/exercises except those which specifically say in FORTRAN. All exercises have solutions in the Tests/Exercises book. In addition, any solutions involving programs should be available on-line, in UFD [2Ø2,2]. 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, plus a glossary. The appendices are:

Appendix A - Supplied Macros. This appendix contains documentation on how to use the macros supplied with the course. In addition, it includes the source code for all of the macros and any subroutines which they require.

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-11. It also explains how to call such a subroutine from MACRO-11.

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 Examples. 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 [2Ø2,1]. They are included here in case they are not available on-line on your system.

Appendix H - Learning Activity Solutions. 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				

1

USING SYSTEM SERVICES

INTRODUCTION

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

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

The first step in learning to use these services is understanding what services exist, how you can request them from within your task, and how the services are delivered to you. These topics are explained in this module and the following module.

OBJECTIVES

1. To identify the facilities provided through system services
2. To list the ways in which system services may be provided to a task
3. To list the various system libraries and the facilities they provide.

RESOURCES

1. RSX-11M/M-PLUS Executive Reference Manual, Chapter 1
2. IAS/RSX-11 System Library Routines Reference Manual, Chapters 1 through 6

WHAT IS A SYSTEM SERVICES?

An RSX-11M system service is a function or service performed for a running task. It is performed during the task's execution. The software which provides the service is either in the Executive itself 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 a part of your programming language. Examples of this are:

1. Accessing shared resources in a properly synchronized way
2. Performing I/O operations in MACRO-11
3. Coordinating among multiple tasks
4. Controlling memory allocation and mapping
5. Interacting with the Executive
6. Performing often needed functions, such as:
   a. Numeric conversion of ASCII data typed in at a terminal to binary format for internal use
   b. Editing, and conversion, to produce suitable output messages which include data generated at run time.

To Ease Programming and Maintenance

DIGITAL provides the code to perform these services. Therefore, you will need less time to develop working programs. The supplied code has a well defined modular structure, which makes it easier to design your 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 your written code needs to be debugged. When maintenance is required in the code for the supplied system services, patches are released with clear-cut installation procedures.

To Increase Performance

The supplied code to perform system services is generally efficient MACRO-11, which assures minimum execution time. In addition, 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 given to show you the kinds of services which are available.

Note that a number of these services which are 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, as well as 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 Peripheral Devices

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

- Write data to or read data from a peripheral device.
- Attach a device for exclusive use by a task.
- Read or set variable characteristics of a device (e.g., for a terminal - baud rate or hold screen mode).

File and Record Access

You can access files, including individual records within a file. For example, you can:

- Create a file.
- Read blocks from or write blocks to a file on a block-by-block basis.

- Read records from or write records to a file. The records may be of different lengths, and not exactly one block long.

- Extend or truncate an existing file.

File and Record Access Systems

The two access systems available under RSX-11M are 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, the Task Builder) use FCS for their file interface. RMS offers all of the FCS functionality plus capabilities not available with FCS, such as indexed files (records that are accessible by a key field value) and more sophisticated file sharing. A more complete discussion of the facilities offered by FCS and RMS, and a comparison of the two, appears in Module 9, on File I/O.

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 only needed pieces of the program at 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:

- Save and restore all or a subset of the registers when writing a subroutine.
- Perform extended integer and double precision multiplication and division.
- Convert data from ASCII to internal binary.
- Convert and format output data produced at run time into printout and/or display messages.

These services are generally supplied as subroutines located in the system object library (LB:[1,1]SYSLIB.OLB). Most of the subroutines are documented in the IAS/RSX-11 System Library Routines Reference Manual. A few of the subroutines will be covered in detail in this course. However, most will not. Table 1-1 gives examples of specific functions performed by some of the subroutines.

LEARNING ACTIVITIES 1-1

1. Read the preface to the IAS/RSX-11 System Library Routines Reference Manual.
2. Look through Chapter 9 of the same manual.

See what functions are offered and how the subroutine calls are made. Also, familiarize yourself with the organization of the book so that you can easily find a subroutine to perform a specific function.

Table 1-1 Examples of Use of Other Services

User Task Requirement	Solution Using System Services
A user written subroutine must save and restore the registers RØ through R5	On entry to the subroutine, call $SAVAL. On return to the caller, the registers will be restored for you. (There are other routines for saving RØ through R2, R1 through R5, and R3 through R5.)
User wants to convert a number input from the terminal (in decimal ASCII) to binary for program use.	Call the subroutine $CDTB, passing it a pointer to the first ASCII character. On return, R1 contains the converted value.
User wants to output a message that includes data calculated at run time (e.g., 3 + 4 = 7, where the values aren't known until run time).	Rather than using a number of calls to a routine to convert each value to ASCII, a general purpose editing routine $EDMSG (Edit Message) is available. $EDMSG takes a template string and converts and fills in the data values, creating an output string suitable for display.

HOW SERVICES ARE PROVIDED

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 notes are keyed to the figure.

1. The user task makes a service request and invokes the Executive.

2. The Executive takes control and performs the service.

   - Calls device drivers as needed
   - Requests other tasks as needed

3. The Executive returns control to the user task, at the instruction following the service request.

Figure 1-2 shows a more complex version of method 1. In this case, Task A and Task B interact through the Executive.

Task A starts up and at some point needs Task B to do some work, for example, perform a calculation. Task A sends the data to Task B, requests that Task B 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 Figure 1-1 would actually be used several times in this example.

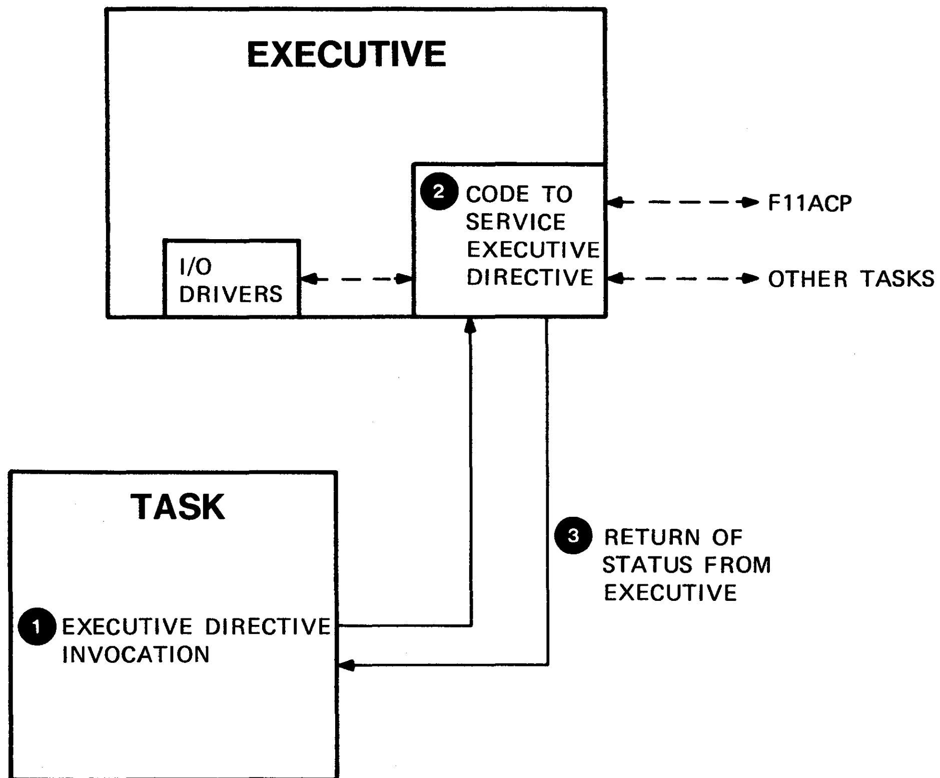


Figure 1-1 Using Executive Directives to Service a Task

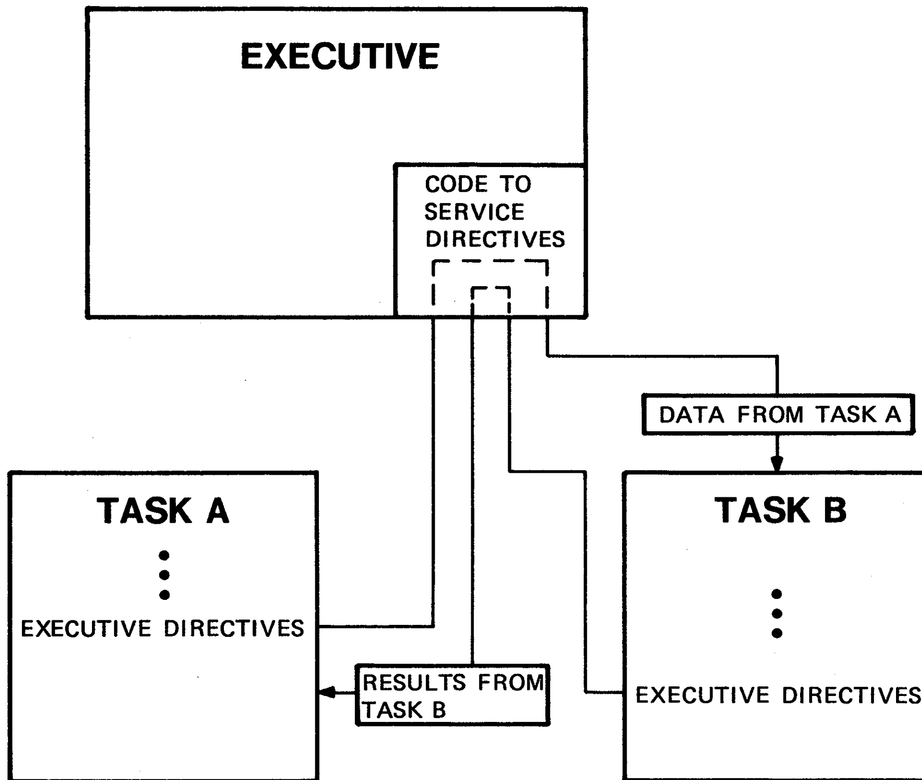


Figure 1-2 Using Executive Directives to Receive Services from Other Tasks

Code Inserted into Your Task Image

The second method for providing system services is illustrated in Figure 1-3. The code to perform the service is extracted from a system library and inserted directly in the user task. For system macros, the machine code resulting from the macro expansion is executed in place. 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 binary 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 with code inserted in the task 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 together 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 discussed 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 up your task in a special way to provide the service. We will discuss the techniques for accessing services with the Task Builder in later modules.

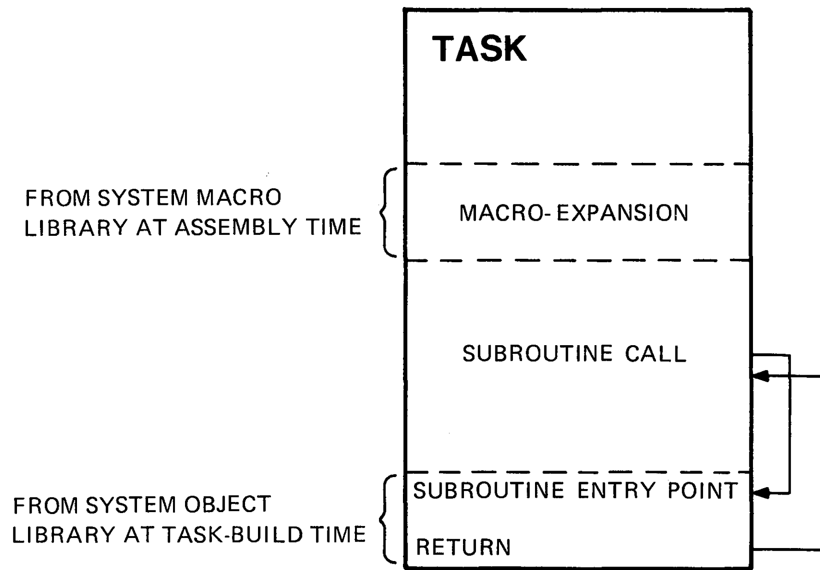


Figure 1-3 Code Inserted into Your Task Image

SYSTEM LIBRARIES

Table 1-2 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]. RSXMAC.SML is the system macro library searched by default by the MACRO-11 assembler. SYSLIB.OLB is the system object library searched by default by the Task Builder.

Table 1-2 Standard Libraries

Library	Languages Using Library (MACRO/FORTRAN)	Contents	Notes
RSXMAC.SML	MACRO-11	Executive directive calls for MACRO-11	Default macro library for the MACRO-11 assembler
		Storage and symbol definition macros to support executive directives	
		FCS calls, storage and symbol definition macros	
SYSLIB.OLB	MACRO-11 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	

Table 1-2 Standard Libraries (Cont)

Library	Languages Using Library (MACRO/FORTRAN)	Contents	Notes
RMSMAC.MLB	MACRO-11	RMS calls, storage and symbol definition macros	
RMSLIB.OLB	MACRO-11	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

Table 1-3 contains a list of the shareable resident libraries which may also be on your system, depending on your installation. You will learn how to use these resident libraries in Module 7, on Static Regions. 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 using the resident libraries.

Table 1-3 Resident Libraries

Resident Library	Routines Extracted from	Comments
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/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 than 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.

2

DIRECTIVES

INTRODUCTION

Once you know the various system services available, you need to know how to write programs which use them. This module explains more about the services available through executive directives and how to make various directive calls.

OBJECTIVES

1. To write programs in MACRO-11 which use directives
2. To use information returned by the Executive to perform error checking
3. To use event flags and asynchronous system traps (ASTs) with directives.

RESOURCES

1. RSX-11M/M-PLUS Executive Reference Manual, Chapter 1 and 2, and specific directives in Chapter 5
2. IAS/RSX-11 System Library Routines Reference Manual, Chapters 4 and 5

INVOKING EXECUTIVE DIRECTIVES FROM A USER TASK

Directive Processing

The sequence of steps outlined below details how a directive is invoked and processed. The following notes are keyed to Figure 2-1.

Executive Code	User Code
	1. The user creates a Directive Parameter Block (DPB) which contains all the information the Executive needs to process the directive.
	2. Either the Directive Parameter Block itself or its starting address is pushed onto the stack.
	3. The user task issues an EMT 377 instruction, causing a trap into the Executive.
4. A dispatcher routine retrieves the Directive Parameter Block, and checks it to find out which directive has been requested.	
5. The dispatcher routine calls the appropriate Directive routine to execute the directive.	

Executive Code	User Code
6 After executing the directive, the Executive returns control to the user task and returns directive status.	
	7 The user task checks the directive status and takes appropriate action.

Use macros in the system macro library, LB:[1,1] RSXMAC.SML to issue directives.

Most directives pass control back to the user task. Certain directives by their nature do not pass back to the user task. The Exit Task directive, for example, causes the task to exit. Control passes back to the user task only in the case of a directive error.

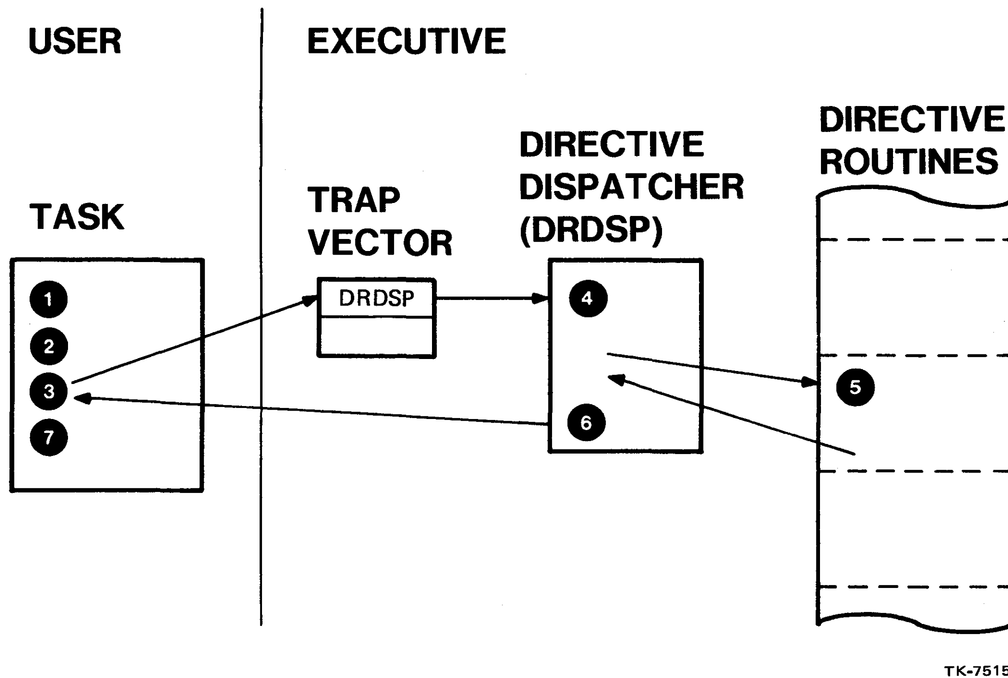


Figure 2-1 Directive Implementation

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 in each group, see section 5.1 on Directive Categories, in the RSX-11M/M-PLUS Executive Reference Manual.

Many of the functions available are discussed more fully in this module, and in the modules on Using the QIO Directive, Using Directives for Intertask Communication, and Dynamic Regions. No attempt is made to discuss every executive directive. You should, however, at the end of this course, know enough to be able to 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.

Table 2-1 Types of Directives

Type	MACRO-11	Description
Task Execution Control	ABRT$	Abort task
	EXIT$S	Exit task
	RQST$	Request task
	RSUM$	Resume task
	RUN$	Run task
	SPND$S	Suspend task
	STOP$	Stop task
	USTP$	Unstop task
Task Status Control	ALTP$	Alter priority
	DSCP$S	Disable checkpointing
	ENCP$S	Enable checkpointing
Informational	GPRT$	Get partition parameters
	GTIM$	Get time parameters
Event-Associated	CLEF$	Clear event flag
	CRGF$	Create group global flags
	ELGF$	Eliminate group global flags
	MRKT$	Mark time
	RDAF$	Read all event flags
	RDXF$	Read extended event flags
	SETF$	Set event flag
	WTSE$	Wait for single event flag
Trap-Associated	ASTX$S	AST Exit
	SREA$	Specify requested exit AST
	SVTK$	Specify task SST vectors
I/O and Intertask Communications	ALUN$	Assign LUN
	QIO$	Queue I/O request
	QIOW$	Queue I/O request and wait
	RCVD$	Receive data
	SDAT$	Send data
Memory Management	CRRG$	Create region
	MAP$	Map address window
Parent/Offspring Tasking	EXST$	Exit with status
	SPWN$	Spawn task

The Directive Parameter Block (DPB)

The Directive Parameter Block is set up as the first step in invoking an Executive directive. It contains all the information the Executive needs to perform the requested service. This includes a Directive Identification Code (DIC) which identifies the Executive directive being requested. See Figure 2-2 for a picture of the Directive Parameter Block layout.

The length of the DPB is included because its length varies depending on which directive is being invoked. The rest of the DPB is built from the arguments specific to the particular directive.

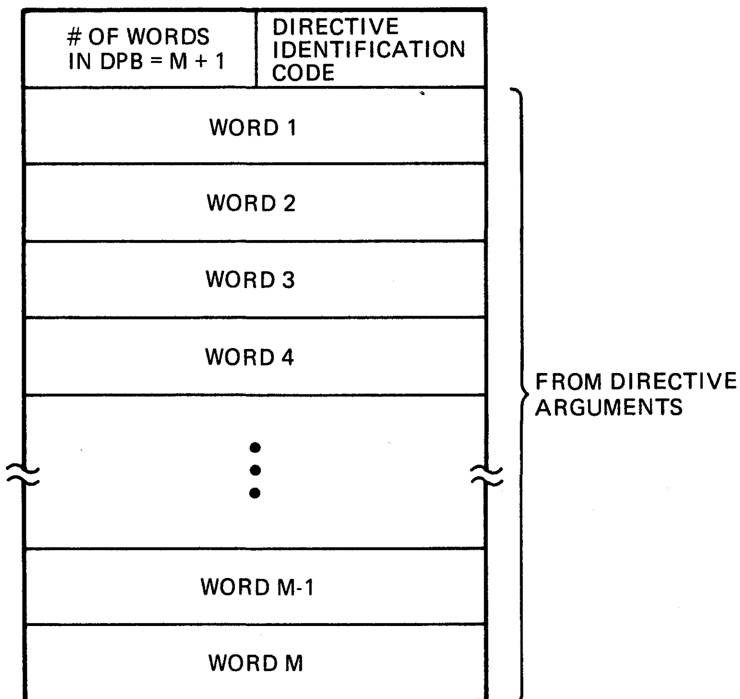


Figure 2-2 The Directive Parameter Block

Macros are provided in the system macro library [LB:[1,1]RSXMAC.SML] to set up the DPB and invoke each executive directive. The format of the macro call is as follows.

```
xxxx$x  arg1,arg2,arg3,...,argn
```

Example:

```
GLUN$C  4,BUFF
```

The macro name determines the DIC and the length of the DPB; the arguments in the macro call are used to build the rest of the DPB. The DPB for the example given is as shown below.

3	5
	4
BUFF	

For additional information on the macros for each directive, see the individual directives in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual.

The Executive preserves (saves and restores) all task registers when a task issues a directive.

The Directive Status Word (DSW)

Upon completion of directive processing, the Executive returns a code in the Directive Status Word which gives the status of the request. The DSW is located in the task header at location $DSW, a global symbol which can be used to reference the value directly. Successful completion is usually indicated by a DSW value of +1 (IS.SUC).

A negative value indicates an error. Different negative values correspond to different sources of 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 Mini Reference Manual. In addition, specific error values and any special meanings are documented with each individual Executive directive call in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual.

In addition to setting the DSW, the Executive clears the carry bit to indicate successful directive execution and sets the carry bit to indicate failure. You can check for errors using a BCC or BCS instruction immediately after the directive call. In that case, access the actual DSW value only if you need it.

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: EXIT$S (no arguments)

- Used to make a task inactive and to free up the system resources it uses.

The Request Task Directive

- Format: RQST$ tsk
  where tsk is the name of the task to be requested

  RQST$C TASKA

- Used to request the specified installed task

- Offers the same functionality as the DCL RUN (immediately) command for an installed task.

Before using any directive in a program, always read over the description of that directive in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual. Specifically, pay attention to the different directive parameters and their meanings. See section 5.3 (on System Directive Descriptions) for an explanation of the organization of the directive desciptions and what elements are included.

Each program example in this course contains the following:

- Source code, with line numbers added

- A sample run session

- Bulleted items which are described in the text.

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

The following comments are keyed to the example.

1. The macros for invoking directives must be specified to the assembler in a .MCALL statement.

2. A number of special macros have been supplied with this course to assist you in class. Since you don't yet know how to issue the QIO directive, which is covered in the next module, the TYPE macro is supplied to perform writes to TI:.

   Example:

   TYPE <HELLO THERE>

   issues a QIO directive to display the text "HELLO THERE" at your terminal.

   The use of the supplied macros is documented in Appendix A, along with the source code for all macros and any internal subroutines they call.

3. Invoke the Request Task directive. The task name must be the installed name (...PIP), not just PIP.

4. The carry bit is set by the Executive in the case of an error and is cleared in the case of success. Always check the status on return from an executive directive.

5. The only case in which control will return to the user task after an EXIT$S call is if a directive error occurs. This is very unlikely to happen.

6. This is an easy way to display the DSW value. The IOT instruction causes the Executive to abort the task and display all registers at TI:.

7. ON THE RUN SESSION. A run session is included for each example program.

The simple method for displaying directive error messages is used here to keep things simple. This technique may be useful in the early stages of debugging a program. Later, this code should be replaced with code which displays more readable error messages. Techniques for doing this are covered in the next module.

```
 1           .TITLE  REQUES
 2           .IDENT  /01/
 3           .ENABL  LC               ; Enable lower case
 4   ;+
 5   ; FILE REQUES.MAC
 6   ;
 7   ; This task displays a message, then requests PIP, and
 8   ; exits
 9   ;
10   ; Assemble and task-build instructions, to include
11   ; supplied macros and subroutines:
12   ;
13   ;    MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]REQUES
14   ;    LINK/MAP REQUES,PROGSUBS/LIBRARY
15   ;-
16           .MCALL  EXIT$S,RQST$C    ; External system macros
17   ;+
18   ; TYPE is a macro supplied in the macro library
19   ; LB:[1,1]PROGMACS.MLB for doing I/O. It issues QIO
20   ; directives for you. TYPE calls subroutines in the
21   ; object library LB:[1,1]PROGSUBS.OLB.
22   ;-
23           .MCALL  TYPE             ; External supplied macro
24   ;
25   ; Display startup text
26   START:  TYPE <REQUES HAS STARTED AND WILL REQUEST PIP>
27                                    ; Display message
28   ; Request PIP
29           RQST$C  ...PIP           ; Request PIP
30           BCS     ERR              ; Branch on directive
31                                    ;  error
32           EXIT$S                   ; Exit
33   ; Error code
34   ERR:    MOV     $DSW,R0          ; Move DSW for display
35           IOT                      ; Trap and display
36                                    ;  registers
37           .END START
```

Run Session

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

Example 2-1 Requesting a Task

DIFFERENT FORMS OF THE DIRECTIVE CALLS

There are three different forms for each directive call, which correspond to three different methods for setting up the DPB and invoking the directive. For each directive call in a program, you may select which form to use.

With two forms, the $ and the $C, the DPB is set up in a data area of your task at assembly time. In the $ form, you use one system macro to set up the DPB, and another system macro at run time to invoke the directive. In the $C form, you use just one macro to both set up the DPB and invoke the directive. The assembler separates the DPB setup into a data area for you. In the $S form, the DPB is set up on the user stack at run time and the directive is invoked immediately afterwards. As in the $C form, only one system macro is needed to both set up the DPB and invoke the directive.

Decide which form of each directive call to use based on the following.

- Task size
- Run time efficiency
- Programming ease
- Knowledge of directive parameters, whether known at assembly time or at run time
- Requirements for reentrant code (e.g., if the code is contained in a shareable library).

Each of the three forms is further described below, using the Set Event Flag directive (SETF$) as an example.

The $ Form

Figure 2-3 shows the $ form, so named because the last character in the macro name is '$' (e.g., RQST$, ABRT$, etc.). In the source code, use a system macro to set up the DPB in a data area, specifying a label to identify it. In the example, LABEL is the label for the DPB set up by the macro SETF$. The DPB is set up at assembly time. The first word of the DPB contains the DPB length in the high-order byte and the DIC in the low-order byte. The next word contains the event flag number argument. Any additional arguments would appear in successive words.

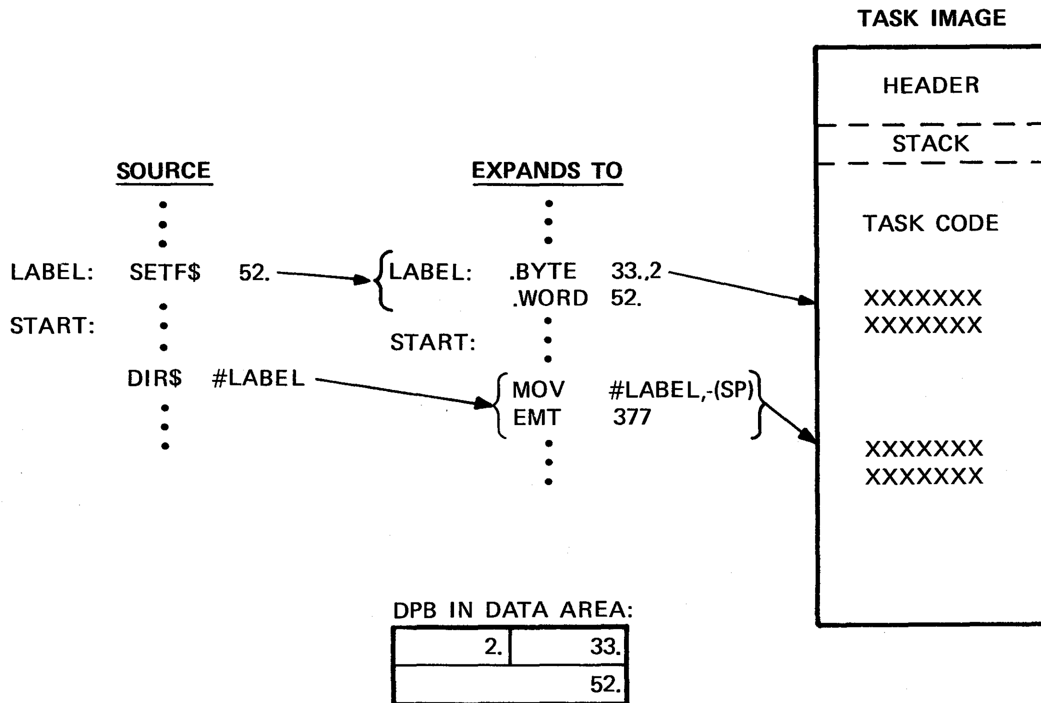


Figure 2-3 The '$' Form

- Use the $ form of the directive macro to set up the DPB in the data area at assembly time.
- Use DIR$ macro to initiate the directive at run time.
- The DIR$ macro pushes the DPB starting address onto the stack, then traps to the Executive.
- Arguments in the $ form must be valid for .BYTE, .WORD, or .RAD50 assembler directives.

  valid arguments: 14.*,204,TASKA,BUFF

  invalid arguments: #14.,#204,#TASKA,@BUFF,R2

* Throughout this course, a decimal point following a numeral indicates that it is in base 10 notation. If no decimal point follows a numeral, it is usually in base 8 notation. The exception is when base 10 is clear from the context; e.g., 16 bits.

Use the separate system macro DIR$ at run time to invoke the directive, specifying the label of the DPB. This macro pushes the starting address of the DPB onto the stack and then traps to the Executive. The label LABEL, which corresponds to the starting address of the DPB, is specified in the DIR$ call. If other directives are invoked in the same task, DIR$ is used each time, with the appropriate address (or label) specified.

Arguments in the $ form of the directive must be valid arguments for .BYTE, .WORD, or .RAD50 Assembler directives. This is necessary because the macros contain .BYTE, .WORD, or .RAD50 assembler directives. See the examples that accompany Figure 2-3.

This form of the directive is run time efficient. In addition, if the same directive is used later in the program to clear another event flag (e.g., 53.) it is possible to use the same DPB for both calls. Offsets within the DPB are defined by global symbols. Hence, at run time, the instructions INC LABEL+C.LEEF or MOV #53.,LABEL+C.LEEF would change the existing DPB for reuse, using another DIR$ #LABEL call. This saves on task space, especially for directives with many arguments.

One drawback of this method is that it is harder to use because two separate macros are needed for each directive invocation. Another is that it is not reentrant if the DPB is changed at run time. For example, reentrant code is required in shareable subroutines.

The $C Form

Figure 2-4 shows the $C form, so named because the last characters in the macro name are '$C' (e.g., RQST$C, ABRT$C, etc.). This form functions similarly to the $ form, but it is easier to use because the DPB setup and actual directive invocation are combined into one macro call. The assembler separates the DPB setup into a data area in a separate Psect named $DPB$$. At run time, a pointer to the DPB is pushed onto the stack when the directive is invoked, as in the $ form.

Arguments for the $C form must also be valid arguments for .BYTE, .WORD, or .RAD5Ø assembler directives. Also, there is an additional optional argument for all $C form calls which is only necessary if a call is made from a Psect other than the default blank Psect. This argument specifies the Psect from which the call is made. This allows return to this Psect for the directive invocation and other code. In Figure 2-3, the Psect PROGS contains the main code.

An advantage of this method is that it is easier to use than the $ form and is just as efficient at run time. One restriction is that a given DPB cannot be accessed and modified at run time. Therefore, to clear event flag 53., a separate CLEF$C 53. directive is required, which generates a separate DPB. So for repeated use of a directive, the $C form requires more task space. Another restriction, due to the inaccessibility of the DPB at run time, is that all directive arguments must be known at assembly time. One other advantage of the $C from is that it is always reentrant, since the DPB cannot be changed.

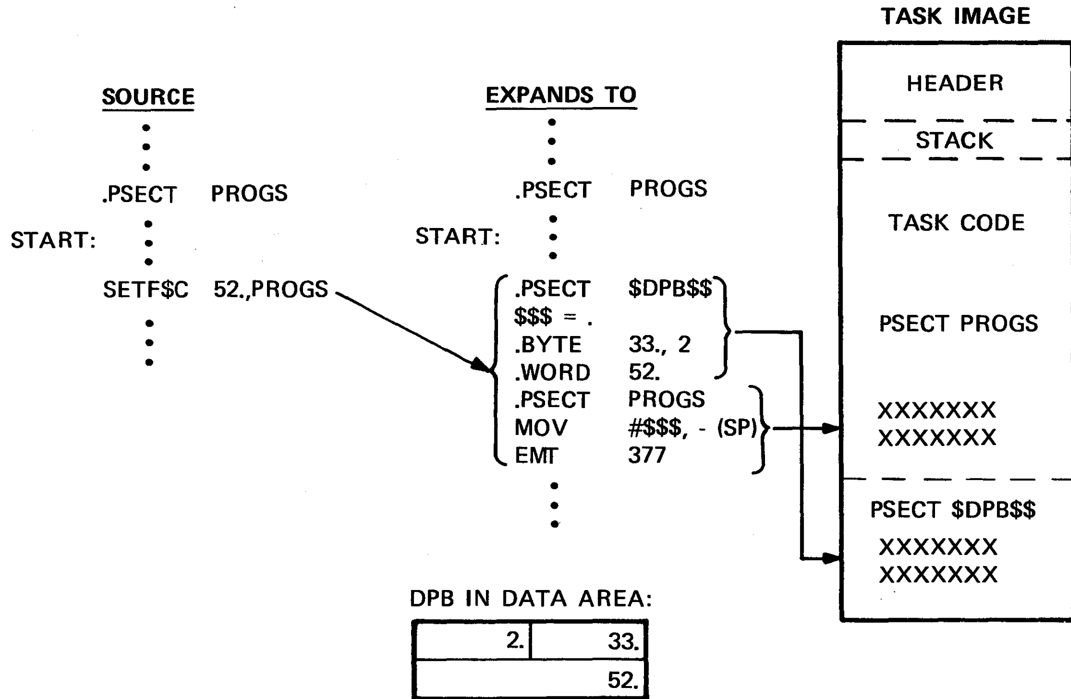


Figure 2-4 The $C Form

Using the $C Form:

- Needs only one macro call.
- Sets up the DPB in the data area at assembly time.
- The $C form, as in the $ form, also pushes the DPB address onto the stack and traps to the Executive at run time.
- Optional argument specifies the current Psect if other than the blank Psect.
- Arguments must also be valid for .BYTE, .WORD, or .RAD50 assembler directives.

The $S Form

Figure 2-5 shows the $S form, so named because the last characters in the macro name are '$S' (e.g., RQST$S, ABRT$S, etc.). In this form, the DPB setup and the directive invocation itself are combined into one macro call, as in the $C form.

However, unlike either the $ or the $C form, in the $S form, the DPB is built at run time instead of at assembly time, and it is built on the stack instead of in the task's data area. This means that all arguments must be valid source arguments for MOV or MOVB instructions. See the examples with Figure 2-5.

One advantage of this method is that the same call can be used with different arguments, since a new DPB is built with each executive directive macro call. Therefore, you can place parameters which aren't known until run time in registers or data areas. You can then specify the registers or the addresses of the data values as arguments in the directive call.

Another major advantage is that the code can be reentrant even if the directive arguments are modified. For example, a register may be used as an argument. Because each task has its own registers, each task has its own independent copy of the argument.

The major disadvantage of this form is that it executes the slowest of the three forms, because every word of the DPB must be pushed onto the stack immediately before invoking the directive. The more arguments the directive has, the longer it takes.

If a directive has no arguments (e.g., EXIT$), it is just as run-time efficient to use the $S form, because the complete DPB is only one word long. Therefore, it takes one instruction to push the complete DPB onto the stack in the $S form. It also takes one instruction to push the address of the DPB onto the stack in the $ and $C forms. Any directive which has no arguments (e.g., Exit Task, Suspend Task) is available with only the $S form.

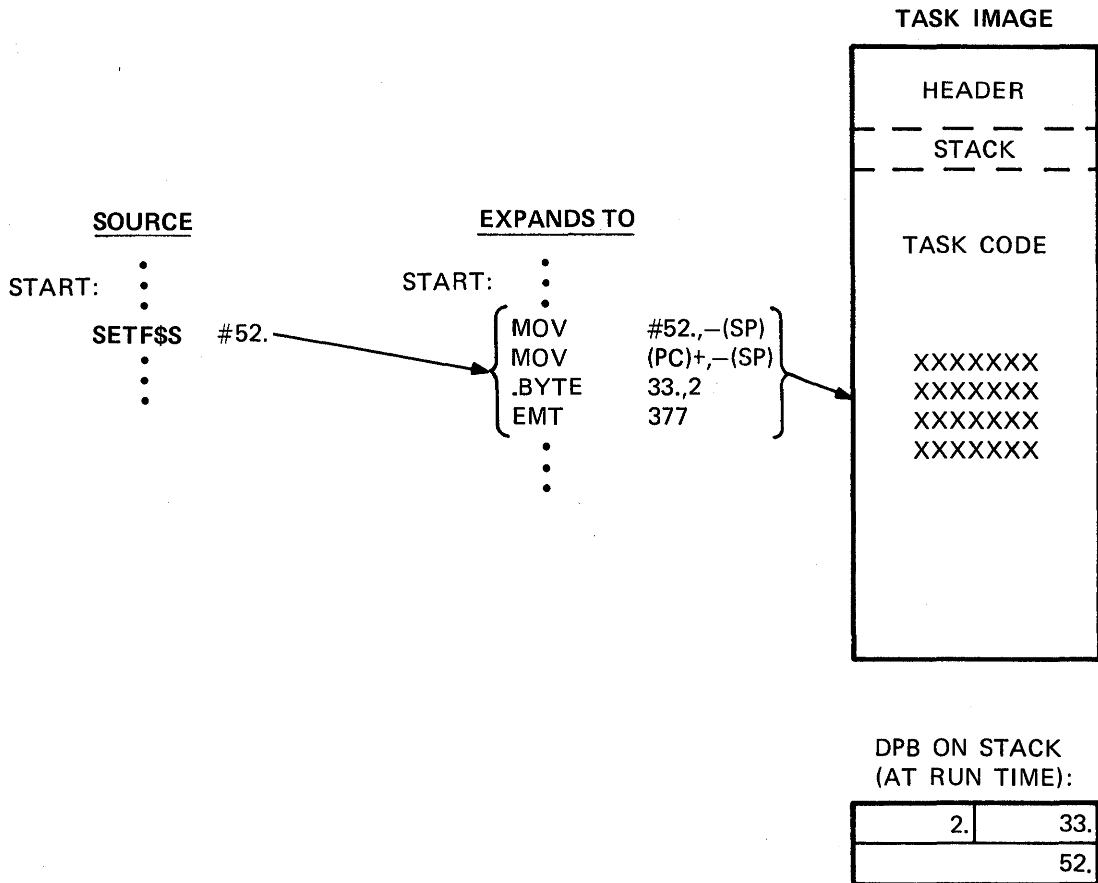


Figure 2-5 The $S Form

Using the $S Form:

- Needs only one macro call.
- The $S form pushes complete DPB onto the stack at run time, then traps to the Executive.
- Arguments must be valid source arguments for MOV instructions.

  valid arguments: #15.,#204,#BUFF,R1

  possible misused arguments: 15.,204,BUFF
  Use 15., 204 or BUFF only if you want the contents of those locations for the directive parameters.

One other disadvantage of using the $S form arises when task or partition names are specified as arguments. These arguments must be in Radix-50 format in the DPB. If the $C or $ form is used, the macro converts the ASCII name specified as an argument to Radix-50 format. If the $S form is used, you must place the name in a data area in Radix-50 format, then specify the address of the data in the macro call. You can either use a .RAD50 assembler directive at assembly time or the $CAT5 subroutine. See Appendix A of the IAS/RSX MACRO-11 Reference Manual for a description of the Radix-50 character set. Also, see 6.3.6 9 (on the .RAD50 assembler directive) in the same manual for a discussion of Radix-50 format.

Examples

Examples 2-2, 2-3, and 2-4 illustrate the use of the three forms of the directive calls. All three forms send a 13(10) = 13. word packet of data to a task RECEIV. The source code for RECEIV follows the code for Example 2-3. Don't worry yet about the actual mechanics of how to set up sender tasks and receiver tasks. These are discussed in the module on Intertask Communication. Just compare the uses of the different forms of directives. The following notes are keyed to all three examples.

1. The .MCALL statement declares the particular macro directive call or calls to be used, including the form.

2. Data area setup requirements:

   $ form: SDAT$ directive sets up the DPB in the data area.

   $C form: Nothing is set up separately. The Assembler sets up the DPB in a data area for you.

   $S form: Normally, nothing is set up in a data area. Task names are an exception, since they must already be in Radix-50 format. Therefore, the task name is set up in Radix-50 format in the data area. The argument in the $S call is the address of the task name.

❸ Executing the directive call.

$ form: Use the separate DIR$ macro.

$C form: Use the single SDAT$ call. The DPB is set up at assembly time by this macro. Just the directive invocation is performed at run time.

$S form: Use the SDAT$S call. The entire DPB is pushed onto the stack at run time and then the directive is invoked.

❹ ON THE RUN SESSION. First run the sender. Then run the receiver to receive and display the data.

Note the difference in the form of the arguments in the $S form. These arguments are source arguments for MOV or MOVB instructions. For the $ and $C forms, the arguments are arguments for .WORD, .BYTE, or .RAD50 Assembler directives.

```
1          .TITLE SEND
2          .IDENT  /01/
3          .ENABL  LC                 ; Enable lower case
4   ;+
5   ; FILE SEND.MAC
6   ;
7   ; This task sends a buffer of 13. words of data to the
8   ; task RECEIV for processing. It sets common event flag
9   ; 33. when the data is queued for RECEIV
10  ;
11  ; It uses the $ form of the Send Data directive
12  ;
13  ; Assemble and task-build instructions:
14  ;
15  ;          MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SEND
16  ;          LINK/MAP SEND,LB:[1,1]PROGSUBS/LIBRARY
17  ;
18  ; Install and run instructions:
19  ;
20  ;          This task does not have to be installed. RECEIV
21  ;          must be installed.
22  ;-
23         .MCALL  SDAT$,EXIT$S,DIR$ ; System macros
24         .MCALL  TYPE               ; Supplied macro
25  ;
26  BUFFER: .WORD   1,2,3,4,5,6,7,8.,9.,10.,11.,12.,13.
27                                    ; Data to send
28
```

(Callout ❶ marks line 23.)

Example 2-2 Using the $ Form of the Directives (Sheet 1 of 2)

```
   29    ; Create DPB separately in a data area for the $ form
❷  30    SEND:    SDAT$    RECEIV,BUFFER,33. ; Set up DPB for
   31                                        ; directive
   32    ;
❸  33    START:   DIR$     #SEND            ; Issue directive to
   34                                       ;  send data to RECEIV
   35             BCS      ERR              ; Branch on dir error
   36             TYPE     <DATA QUEUED TO RECEIV> ; Display
   37                                       ;  success message
   38             EXIT$S                    ; Exit
   39
   40    ERR:     MOV      $DSW,R1          ; Move DSW to R1 for
   41                                       ;  display
   42             IOT                       ; Trap and display
   43                                       ;  registers
   44             .END     START
```

Run Session

```
>INS RECEIV
>RUN SEND
DATA QUEUED TO RECEIV
>RUN RECEIV
   1    2    3    4    5    6    7    8    9   10   11   12   13
>
```

```
 1             .TITLE   RECEIV
 2             .IDENT   /01/
 3             .ENABL   LC               ; Enable lower case
 4    ;
 5    ; File RECEIV.MAC
 6    ;
 7    ; This task receives the data sent by SEND, SENDC, or
 8    ; SENDS and displays it at TI:
 9    ;
10             .MCALL   RCVD$C,EXIT$S
11             .MCALL   TYPE
12    ;
13    RBUFF:   .BLKW    15.              ; Buffer for data received
14    BUFF:    .BLKB    80.              ; Buffer for output message
15    FMT:     .ASCII   /%3S%D%4S%D%4S%D%4S%D%4S%D%4S%D%4S%D/
16             .ASCIZ   /%4S%D%4S%D%4S%D%4S%D%4S%D%4S%D/
17    ;
18    START:   RCVD$C   ,RBUFF           ; Receive from anyone
19             BCS      ERR1             ; Branch on directive error
20    ; Edit binary data into ASCII message for display
21             MOV      #BUFF,R0         ; Addr of output buffer
22             MOV      #FMT,R1          ; Addr of format string
23             MOV      #RBUFF+4,R2      ; Addr of data received,
24                                       ;  skip sender task name
25             CALL     $EDMSG           ; Edit ouput message
26             TYPE     #BUFF,R1         ; Display output message
27             EXIT$S                    ; Exit
28    ; Error code
29    ERR1:    MOV      $DSW,R0          ; Move DSW for display
30             IOT                       ; Trap and display
31                                       ;  registers
32             .END     START
```

Example 2-2 Using the $ Form of the Directives (Sheet 2 of 2)

```
  1             .TITLE SENDC
  2             .IDENT  /01/
  3             .ENABL  LC                ; Enable lower case
  4     ;+
  5     ; FILE SENDC.MAC
  6     ;
  7     ; This task sends a buffer of 13. words of data to the
  8     ; task RECEIV for processing. It sets common event flag
  9     ; 33. when the data is queued for RECEIV
 10     ;
 11     ; It uses the $C form of the Send Data directive
 12     ;
 13     ; Assemble and task-build instructions:
 14     ;
 15     ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SENDC
 16     ;
 17     ;       LINK/MAP SENDC,LB:[1,1]PROGSUBS/LIBRARY
 18     ;
 19     ; Install and run instructions:
 20     ;
 21     ;       This task does not have to be installed. RECEIV
 22     ;       must be installed.
 23     ;-
(1) 24          .MCALL  SDAT$C,EXIT$S,DIR$ ; System macros
 25             .MCALL  TYPE              ; Supplied macro
 26     ;
 27     ;
 28     BUFFER: .WORD   1,2,3,4,5,6,7,8.,9.,10.,11.,12.,13.
 29                                       ; Data to send
(3) 30  START:  SDAT$C  RECEIV,BUFFER,33. ; Issue directive to
 31                                       ;  send data to RECEIV
 32             BCS     ERR               ; Branch on dir error
 33             TYPE    <DATA QUEUED TO RECEIV> ; Display
 34                                       ;  success message
 35             EXIT$S                    ; Exit
 36
 37     ERR:    MOV     $DSW,R1           ; Move DSW to R1
 38             IOT                       ; Trap and display
 39                                       ;  registers
 40             .END    START
```

Run Session

```
>INS RECEIV
>RUN SENDC
DATA QUEUED TO RECEIV
>RUN RECEIV
   1    2    3    4    5    6    7    8    9   10   11   12   13
>
```

Example 2-3 Using the $C Form of the Directives

```
 1           .TITLE SENDS
 2           .IDENT  /01/
 3           .ENABL LC                 ; Enable lower case
 4   ;+
 5   ; FILE SENDS.MAC
 6   ;
 7   ; This task sends a buffer of 13. words of data to the
 8   ; task RECEIV for processing. It sets common event flag
 9   ; 33. when the data is queued for RECEIV
10   ;
11   ; It uses the $S form of the Send Data directive
12   ;
13   ; Assemble and task-build instructions:
14   ;
15   ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SENDS
16   ;       LINK/MAP SENDS,LB:[1,1]PROGSUBS/LIBRARY
17   ;
18   ; Install and run instructions:
19   ;
20   ;       This task does not have to be installed. RECEIV
21   ;       must be installed
22   ;-
23           .MCALL  SDAT$S,EXIT$S,DIR$ ; System macros
24           .MCALL  TYPE               ; Supplied macro
25   ;
26   BUFFER: .WORD   1,2,3,4,5,6,7,8.,9.,10.,11.,12.,13.
27                                      ; Data to send
28   ;
29   ; Task names must be specified in Radix-50 format for
30   ; the $S form
31   TASKNM: .RAD50  /RECEIV/
32   ;
33   START:  SDAT$S  #TASKNM,#BUFFER,#33. ; Issue directive to
34                                      ;  send data to RECEIV
35           BCS     ERR                ; Branch on dir error
36           TYPE    <DATA QUEUED TO RECEIV> ; Display
37                                      ; success message
38           EXIT$S                     ; Exit
39
40   ERR:    MOV     $DSW,R1            ; Move DSW to R1
41           IOT                        ; Trap and display
42                                      ;  registers
43           .END    START
```

Run Session

```
>INS RECEIV
>RUN SENDS
DATA QUEUED TO RECEIV
>RUN RECEIV
   1     2     3     4     5     6     7     8     9    10    11    12    13
>
```

Example 2-4 Using the $S Form of the Directives

Repeated Use of a Directive with Different Arguments

The following sections of code illustrate the use of the different directive forms when using a directive several times in a program. All three clear event flags 5. to 15., using the Clear Event Flag directive 11 times. Note in particular that the $ form uses the same DPB over and over again. The $C form macro calls result in 11 different DPBs in the data area of the task. The $S form uses a register as an argument and a new DPB is generated for each call; but on the stack, not in a data area.

NOTE

A discussion of event flags and their uses appears later in this module.

$ Form

Use the Executive directive first for event flag 5, then access and change the DPB for the other ten calls. In the example below, the DPB begins at CLEAR.

```
            .MCALL CLEF$, DIR$
               .
               .
               .
CLEAR:      CLEF$  5.
               .
START:         .
               .
            MOV    #5.,R0
AGAIN:      DIR$   #CLEAR
            BCS    ERR
            INC    R0
            CMP    R0,#15.
            BGT    DONE
            INC    CLEAR+C.LEEF
            BR     AGAIN
DONE:          .
               .
               .
```

$C Form

The $C form cannot access the DPB; so make 11 different calls with separate DPBs.

```
                  .MCALL CLEF$C
                     .
         START:      .
                     .
                  CLEF$C  5.
                  BCS     ERR
                  CLEF$C  6.
                  BCS     ERR
                  CLEF$C  7.
                  BCS     ERR
                  CLEF$C  8.
                  BCS     ERR
                  CLEF$C  9.
                  BCS     ERR
                  CLEF$C  10.
                  BCS     ERR
                  CLEF$C  11.
                  BCS     ERR
                  CLEF$C  12.
                  BCS     ERR
                  CLEF$C  13.
                  BCS     ERR
                  CLEF$C  14.
                  BCS     ERR
                  CLEF$C  15.
                  BCS     ERR
                     .
                     .
                     .
```

$S Form

A new DPB is pushed onto the stack for each call. Use a register value for an argument. Make the same call 11 times; update the register each time.

```
                .MCALL    CLEF$S
                   .
        START:     .
                   .
                MOV       #5,RØ
        AGAIN:  CLEF$S    RØ
                BCS       ERR
                INC       RØ
                CMP       RØ,#15.
                BLE       AGAIN
                   .
                   .
                   .
```

Table 2-2 gives a summary of the three forms of the directive call.

Table 2-2 Summary of the Directive Forms

	$ Form	$C Form	$S Form
DPB Location	Data area of current Psect	Data area of Psect $DPB$$	On task stack
DPB Creation	At assembly time	At assembly time	At run time
Advantages	DPB reusable, easily changed Run time efficient	Easily coded Run time efficient Reentrant	Arguments can be changed at run time Can be reentrant even if DPB changed
Disadvantages	Not reentrant if DPB changed Need two macro calls	DPB can't be changed Uses more space for multiple calls	Run time inefficient if directive has many arguments
Recommended Use	If parameters not set at assembly time and run time efficiency critical For repeated use with same arguments	Known directives used once Parameters all set at assembly time and directive used once	If directive has no arguments When parameters not set at assembly time and run time efficiency not critical or code must be reentrant
Notes	Arguments valid in .WORD or .BYTE	Return Psect may be needed in call Arguments valid in .WORD or .BYTE	Arguments valid for MOV or MOVB

ADDITIONAL DIRECTIVE CONSIDERATIONS

An Alternative Method for Error Checking

An additional argument can be used to specify the address of an error subroutine.

Format:

```
        $ Form                              $C Form

DCLEF:  CLEF$    53.
          .
          .
          .
        DIR$     #DCLEF,ERROR              CLEF$C  53.,,ERROR


                          $S Form


                          CLEF$S  #53.,ERROR
```

NOTES

The extra null argument in the $C form is for the optional Psect.

In the $S form, no '#' is needed on the address, since this becomes a JSR PC,ERROR. This argument is not moved to the stack.

In all three cases, the extra argument causes the following code to be generated:

```
;macro without error address
          .
          .
          .
;additional code
        BCC   .+6
        JSR   PC,ERROR
```

This results in a branch to the instruction following the directive macro if the directive is executed successfully, and a call to the subroutine ERROR if not. It is equivalent to including the following code yourself.

```
        DIR$    #LABEL
        BCC     OK
        JSR     PC,ERROR
OK:
```

Note that in case of an error the transfer to the error routine is with a JSR, **not** a JMP or BR. The result is that the return address is pushed onto the stack. If you generate an error message and exit, the JSR won't cause any problems because the stack isn't accessed.

If, on the other hand, you attempt to recover from the error, you must remember that the return point is on the stack. You must either use a RETURN (RTS PC) or clear the return address off the stack if you wish to branch to a different location.

Examples Using Other Directives

The following directives are used in Example 2-5.

- Suspend Task (SPND$S)
  - Used to suspend itself
  - Can be resumed by another task issuing a Resume task directive or by an operator using the DCL CONTINUE command
- Alter Priority (ALTP$)
  - Alters the running priority of an active task
- Disable Checkpointing (DSCP$S)
  - Disables checkpointing for a checkpointable task

- Enable Checkpointing (ENCP$S)
  - Enables checkpointing again after a DSCP$ directive
- Extend Task (EXTK$)
  - Modifies the size of the task by a positive or negative number of 32-word blocks.

The $S form of SPND$, DSCP$, and ENCP$ is recommended because each directive has no arguments.

Example 2-5 shows the use of a variety of directives. See the run demonstration below the source code. The following comments are keyed to the example.

1. R1 is a directive counter. When several directives are used in a program, the counter helps keep track of which directive caused an error. After an IOT, n in R1 means that there was an error on the nth directive. RØ contains the DSW value.

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

3. The task is loaded at physical address ØØ6152ØØ(8) to ØØ6172ØØ(8). SPN means the task is suspended.

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

5. Suspend again after you disable checkpointing and alter the running priority.

6. Note the change in running priority (PRI). CKD indicates the disabling of checkpointing.

7. Suspend again after you enable checkpointing, alter the priority back to 5Ø., and extend the task.

❽ Note the change in priority. Note also that the task was checkpointed and is now loaded at addresses Ø1Ø452ØØ(8) to Ø1Ø672ØØ(8). The new task size is 22ØØØ(8) bytes, compared to 2ØØØ(8) bytes before, as shown below. The extend is for 2ØØ(8) blocks, where each block is 1ØØ(8) bytes long, which means there are 2ØØØØ(8) extra bytes. See Appendix B for a conversion table of bytes to blocks and of octal to decimal.

Before:

```
 ØØ6172ØØ(8)
-ØØ6152ØØ(8)
------------
     2ØØØ(8) bytes
```

After:

```
 Ø1Ø672ØØ(8)
-Ø1Ø452ØØ(8)
------------
    22ØØØ(8)
```

```
 1           .TITLE  MISC
 2           .IDENT  /01/
 3           .ENABL  LC              ; Enable lower case
 4      ;+
 5      ; FILE MISC.MAC
 6      ;
 7      ; This task uses some miscellaneous Executive directives
 8      ; to suspend itself, alter its running priority, disable
 9      ; and enable checkpointing, and extend its task size.
10      ;
11      ; Task-build instructions:
12      ;
13      ;       LINK/CHECKPOINT/MAP MISC
14      ;       since the task must be checkpointable to allow
15      ;       disabling of checkpointing and extending its size
16      ;
17      ; Install and Run instructions:
18      ;
19      ;       Install the task. Then Run it to start it up.
20      ;       The task will suspend itself several different
21      ;       times. Each time, use the command
22      ;       SHOW TASKS:MISC/ACTIVE/FULL (MCR ATL MISC)
23      ;       to examine the changes. Use the command
24      ;       CONTINUE MISC (MCR RESUME MISC)
25      ;       to resume the task.
26      ;-
27      ;
28              .MCALL  SPND$S,ALTP$C,DSCP$S,ENCP$S
29              .MCALL  EXTK$C,EXIT$S
30      ;
(1) 31  START:  CLR     R1         ; Directive counter for errors
(2) 32          SPND$S             ; Suspend to allow status check
    33          BCS     ERR1       ; Branch on directive error
    34  ; Make some changes and then suspend again
  ┌ 35          DSCP$S             ; Disable checkpointing
  │ 36          BCC     OK         ; Branch on good directive
  │ 37          JMP     ERR2       ; Jump to error code
(5)│ 38  OK:     ALTP$C  ,10.       ; Alter running priority
  │ 39          BCC     GOOD       ; Branch on good directive
  │ 40          JSR     PC,ERR3    ; Call error subroutine
  └ 41  GOOD:   SPND$S  ERR4       ; Suspend to allow status check
    42  ; Make some other changes and then suspend again
  ┌ 43          ENCP$S             ; Enable checkpointing again
  │ 44          BCS     ERR5       ; Branch on directive error
(7)│ 45          ALTP$C  ,,,ERR6    ; Return priority to original
  │ 46          EXTK$C  200        ; Extend task size by 200(8)
  └ 47                             ;  blocks
```

Example 2-5 Using Several Directives (Sheet 1 of 2)

```
7  48             BCC      ALSOOK   ; Branch on good directive
   49             CALL     ERR7     ; Call error subroutine
   50   ALSOOK:   SPND$S            ; Suspend again
   51             BCC      AGNOK    ; Branch on directive ok
   52             BR       ERR8     ; Branch on directive error
   53   AGNOK:    EXIT$S            ; Exit
   54   ; Error handling
1  55   ERR8:     INC      R1       ; 8 means error on 3rd SPND$S
   56   ERR7:     INC      R1       ; 7 means error on EXTK$C
   57   ERR6:     INC      R1       ; 6 means error on 2nd ALTP$C
   58   ERR5:     INC      R1       ; 5 means error on ENCP$S
   59   ERR4:     INC      R1       ; 4 means error on 2nd SPND$S
   60   ERR3:     INC      R1       ; 3 means error on 1st ALTP$C
   61   ERR2:     INC      R1       ; 2 means error on DSCP$S
   62   ERR1:     INC      R1       ; 1 means error on 1st SPND$S
   63             MOV      $DSW,R0  ; Move DSW for display
   64             IOT               ; Trap and display registers
   65             .END     START
```

Run Session

```
  >INS MISC
  >RUN MISC
3 >SHOW TASKS/ACTIVE FULL MISC
  MISC    055420   GEN    054500 00615200-00617200  PRI - 50.   DPRI - 50.
     STATUS:  SPN -PMD
     TI - TT11:  IOC - 0.  BIO - 0.  EFLG - 000000 000000  PS - 170000
     PC - 001264  REGS 0-6  000000 000000 011300 140130 000000 000000 001254
4 >CONTINUE MISC
6 >SHOW TASKS/ACTIVE FULL MISC
  MISC    055420   GEN    054500 00615200-00617200  PRI - 10.   DPRI - 50.
     STATUS:  CKD SPN -PMD
     TI - TT11:  IOC - 0.  BIO - 0.  EFLG - 000000 000000  PS - 170000
     PC - 001324  REGS 0-6  000000 000000 011300 140130 000000 000000 001254
  >CONTINUE MISC
8 >SHOW TASKS/ACTIVE FULL MISC
  MISC    055420   GEN    054500 01045200-01067200  PRI - 50.   DPRI - 50.
     STATUS:  SPN -PMD
     TI - TT11:  IOC - 0.  BIO - 0.  EFLG - 000000 000000  PS - 170000
     PC - 001400  REGS 0-6  000000 000000 011300 140130 000000 000000 001254
  >CONTINUE MISC
  >SHOW TASKS/ACTIVE FULL MISC
  ATL -- Task not active
```

Example 2-5 Using Several Directives (Sheet 2 of 2)

This example illustrates a number of techniques for directive error checking. At lines 33 and 44, a BCS is used. At lines 36, 39, 48, and 51, a BCC is used to branch past the transfer to the error handling code.

The transfers themselves also differ. At line 37, a JMP is used. At line 40, a JSR PC is used, while at line 49, a CALL which is equivalent to a JSR PC is used. At line 52, a BR is used. Finally, at lines 41 and 45, the address of the error routine is specified as the last argument of the directive macro call. This results in a JSR PC, generated as part of the macro expansion.

All of these get you to the error routines. They are all equivalent as long as you don't attempt to recover from the error. If you do recover, you must remember that a JSR PC or CALL pushes a return address onto the stack, as explained in the section on An Alternate Method for Error Checking.

Run Time Conversion Routines

As mentioned earlier, the system maintains task names, partition names, and certain other data in Radix-5Ø format to save space. There are times when conversions between ASCII and Radix-5Ø format need to be performed at run time.

You can modify Example 2-1 (REQUES.MAC) so an operator can type in the task name at run time. This ASCII name would then have to be converted at run time to Radix-5Ø format because the .RAD5Ø assembler directive can only be used at assembly time. The subroutine $CAT5 in SYSLIB.OLB is provided for performing this conversion. Its use is documented in Chapter 4 of the IAS/RSX-11 System Library Routines Reference Manual.

If the Get Task directive (GTSK$) is used to retrieve task information, the task name and partition name are returned in Radix-5Ø format. If you want to display these, you need to convert them to ASCII format. The subroutine $C5TA, also in SYSLIB.OLB and documented in Chapter 5 of the manual mentioned above, is provided for this purpose.

Additional subroutines are provided for converting between binary and octal ASCII (signed or unsigned) and between binary and decimal ASCII (signed or unsigned). See Chapters 4 and 5 of the IAS/RSX-11 System Library Routines Reference Manual for additional information.

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. The two methods for implementing synchronization are by using event flags and using 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(10)-96(10)).

Local event flags are provided for each task. There are 32(10) local event flags, numbered 1(10)-32(10). 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(10)-32(10) are reserved for system use and therefore 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(10) global event flags for the system, numbered 33(10)-64(10). These flags can be referenced by any task. Global event flags 57(10)-64(10) 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 ensure that it is not used for some other purpose.

There are only 32(10) global event flags available in the system. If additional event flags are needed, another set of event flags can be created for synchronization among different tasks. Group global event flags (32(10)), 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. Therefore, they can be used to synchronize tasks running under that group number. An additional advantage is that they cannot be referenced by tasks running under other group numbers.

Group global event flags are created using the DCL SET GROUP FLAGS CREATE (FLA /CRE in MCR) command or the Create Group Global Event Flags (CRGF$) directive. When users in a group don't need them anymore, the group global event flags can be marked for deletion using the DCL SET GROUP FLAGS DELETE (FLA /ELIM in MCR) command or the Eliminate Group Global Event Flags (ELGF$) 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 ACTIVITIES 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. This material will not be covered in this course. 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 acceptable in Examples 3 and 4?

Examples of the Use of Event Flags for Synchronization

Examples 2-6 and 2-7 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.

1. Create the group global event flags - The default used here creates them for the group number which 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.

   NOTE

   If the error address had been included in the macro directive call (CRGF$C ,,ERR1), two changes must be made to the code. First, the check for IE.RSU must be made at location ERR1. Second, in the case of the nonfatal error IE.RSU, the stack will have one extra word because the macro does a JSR PC,ERR1, not a BCS ERR1. Therefore, you would need to either use a RTS PC (synonym RETURN) or, if you want to branch to another location, you need to pop the return address off the stack before branching.

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. This allows WFLAG to become unblocked. SFLAG now exits.

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

```
     1            .TITLE  WFLAG
     2            .IDENT  /01/
     3            .ENABL  LC                 ; Enable lower case
     4   ;+
     5   ; FILE WFLAG.MAC
     6   ;
     7   ; This program creates the group global event flags,
     8   ; clears event flag 65. and waits for it to be set. When
     9   ; the flag is set it writes a message and exits.
    10   ;
    11   ; Assemble and task-build instructions:
    12   ;
    13   ;        MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]WFLAG
    14   ;        LINK/MAP WFLAG,LB:[1,1]PROGSUBS/LIBRARY
    15   ;
    16   ; Install and Run instructions:
    17   ;
    18   ;        Run WFLAG, then run SFLAG. At least one of the
    19   ;        tasks must be installed, or else the RUN command
    20   ;        will try to install both tasks under the same
    21   ;        name, TTnn.
    22   ;-
    23            .MCALL  EXIT$S,WTSE$C,CLEF$C,CRGF$C ; System
    24                                                ;  macros
    25            .MCALL  TYPE               ; Supplied macro
    26
    27   START:   CLR     R0                 ; R0 used to identify
    28                                       ;  the error
    29            TYPE    <WFLAG IS CREATING THE GROUP GLOBAL EVENT FLAGS>
(1)  30            CRGF$C                     ; Create group global
    31                                       ;  event flags
    32            BCC     OK                 ; Branch on directive ok
    33   ; If group global event flags already exist,
    34   ; Just display message and continue
(2) [35            CMP     $DSW,#IE.RSU       ; Check for efs already
    36                                       ;  in existence
    37            BNE     ERR1               ; Branch on any other
    38                                       ;  dir error
    39]           TYPE    <GROUP GLOBAL EVENT FLAGS ALREADY EXIST>
    40   OK:      TYPE    <CLEAR AND THEN WAIT FOR EF 65. TO BE SET>
(3)  41            CLEF$C  65.                ; Clear event flag 65.
    42            BCS     ERR2               ; Branch on directive
    43                                       ;  error
(4)  44            WTSE$C  65.                ; Wait for event flag 65
    45                                       ;  to be set
(6) [46            BCS     ERR3               ; Branch on directive
    47                                       ;  error
    48            TYPE    <EF 65. HAS BEEN SET. WFLAG WILL NOW EXIT>
    49]           EXIT$S
    50   ERR3:    INC     R0                 ; R0 = 3 if error on
    51                                       ;  wait for dir
    52   ERR2:    INC     R0                 ; R0 = 2 if error on
    53                                       ;  clear flag dir
    54   ERR1:    INC     R0                 ; R0 = 1 if error on
    55                                       ;  create group flags dir
    56            MOV     $DSW,R1            ; Place DSW in R1
    57            IOT                        ; Trap and dump registers
    58            .END    START
```

Example 2-6 Waiting for an Event Flag (Sheet 1 of 2)

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 2-6 Waiting for an Event Flag (Sheet 2 of 2)

```
      1              .TITLE   SFLAG
      2              .IDENT   /01/
      3              .ENABL   LC                 ; Enable lower case
      4      ;+
      5      ; FILE SFLAG.MAC
      6      ;
      7      ; This task sets event flag 65.  It assumes that the
      8      ; group global event flags have already been created.
      9      ;
     10      ; Assemble and task-build instructions:
     11      ;
     12      ;        MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SFLAG
     13      ;        LINK/MAP SFLAG,LB:[1,1]PROGSUBS/LIBRARY
     14      ;
     15      ; Install and Run notes:
     16      ;
     17      ;        First run WFLAG, then run SFLAG. At least one of
     18      ;        the tasks must be installed, or else the RUN
     19      ;        command will try to install both tasks under
     20      ;        the same name, TTnn.
     21      ;
     22      ;-
     23              .MCALL   EXIT$S,SETF$C      ; System macros
     24              .MCALL   TYPE               ; Supplied macros
     25      ;
     26      START:  TYPE     <EF 65. IS BEING SET. THEN SFLAG WILL EXIT.>
(5)  27              SETF$C   65.                ; Set event flag 65.
     28              BCS      ERR                ; Branch on dir error
     29              EXIT$S                      ; Exit
     30      ERR:    MOV      $DSW,R1            ; Save DSW
     31              IOT                         ; Trap and dump registers
     32              .END     START
```

Example 2-7 Setting an Event Flag in a Task

Asynchronous System Traps (ASTs)

Asynchronous System Traps (ASTs) are used to detect events that occur asynchronously to a task's execution. Two examples are the completion of an I/O transfer and a power up after a power failure. 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. If a task needs to do work while waiting for an event to occur, it can do so and periodically check an event flag to detect the event. However, this means that the task must 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. This technique is more efficient because the task doesn't have to do any checking. It also results in faster notification because the task is notified immediately after the event occurs. With checking of the flag, it may take a long time to notice an event that has occurred immediately after a check.

Several Executive directives allow the use of ASTs to handle synchronization. A complete list appears in the section 5.1.5 on Trap Associated Directives in the RSX-11M/M-PLUS Executive Reference Manual.

Figure 2-6 shows how an AST works. The following notes are keyed to this figure.

1. The user specifies an AST routine in an Executive 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 which corresponds to the specified AST routine has occurred, the Executive passes control to the AST routine and the task executes 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.

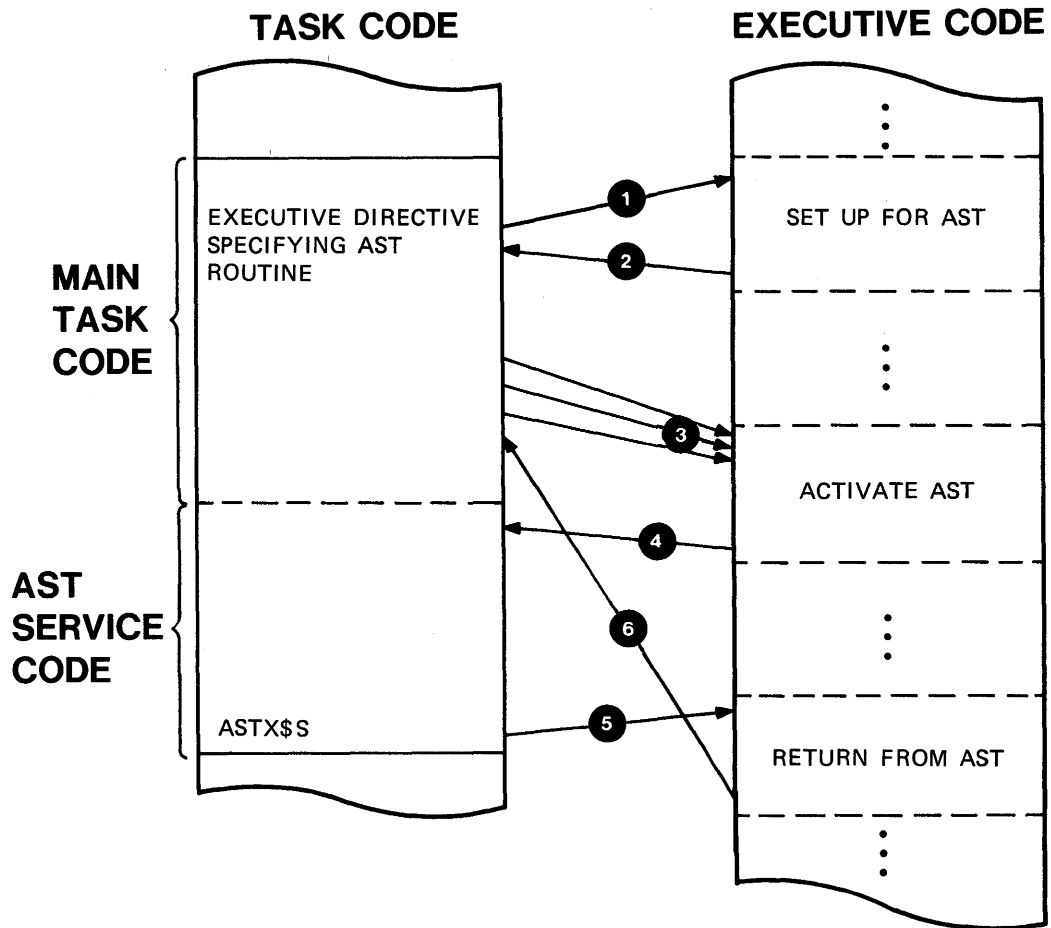


Figure 2-6 AST Mechanics

❹ The AST routine is a user written routine contained within the task. The user stack is set up in a special way by the Executive before the AST routine is entered, as shown in Figure 2-7. Notice that some ASTs have special words added to the stack. The AST routine may use these parameters to check on what caused the AST, and then take appropriate action. See section 2.3.4 on AST Service Routines in the RSX-11M/M-PLUS Executive Reference Manual for a discussion of the specific stack formats.

5. Finally, the AST routine uses the ASTX$S Executive directive to "return" control to the main task code via the Executive. When the ASTX$S directive is invoked, the Executive assumes that the stack contains only the standard first four AST stack words. The user AST routine must clear any additional AST specific parameters off the stack before issuing this directive.

6. The Executive checks for any other ASTs which may have occurred while the AST routine was executing. Any such additional ASTs are queued up in an AST pending queue in a first-in-first-out order. These ASTs are also serviced before the Executive "returns" to the "interrupted" state and code.

Note that the task's general purpose registers RØ through R5 and SP are not saved. Therefore, if you use these registers in an AST routine, you must save and restore them.

For additional information on ASTs, see sections 2.3.3 and 2.3.4 on ASTs and AST Service Routines in the *RSX-11M/M-PLUS Executive Reference Manual*.

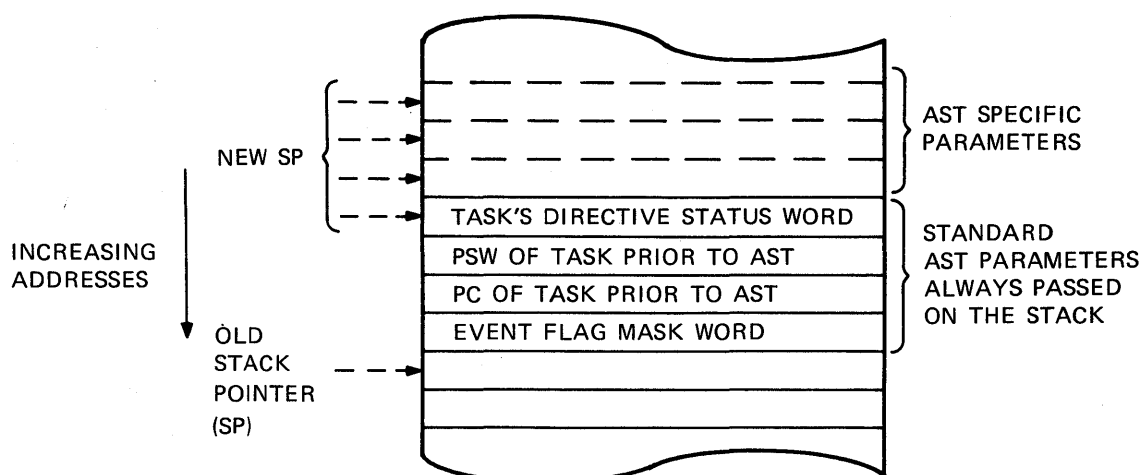


Figure 2-7 Stack as Set Up by the Executive for ASTs

Example 2-8 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 AST, a nonprivileged task enters the 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. There is no need to set up the stack for the AST return, because there are no AST specific parameters (only the four words expected by the Executive are on the stack). The AST exit causes the Executive to transfer control to the task back in the main code where it was "interrupted."

Another directive, SREX$, gives extended capabilities. An entry passed on the stack 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.

```
     1           .TITLE  ASTEX
     2   ;+
     3   ; FILE ASTEX.MAC
     4   ;
     5   ; This task sets up a Specify Request Exit AST routine.
     6   ; It then sits in a loop until someone tries to abort
     7   ; it. At that point, it enters the AST routine and sends
     8   ; out a message. It won't abort the first time. A second
     9   ; abort attempt will succeed because for this particular
    10   ; AST, the first abort AST cancels any further abort
    11   ; AST's.
    12   ;
    13   ; Assemble and task-build instructions:
    14   ;
    15   ;       >MACRO/LIST ASTEX=LB:[1,1]PROGMACS/LIBRARY,-
    16   ;       ->dev:[ufd]ASTEX
    17   ;       >LINK/MAP ASTEX,LB:[1,1]PROGSUBS/LIBRARY
    18   ;-
    19           .MCALL  SREA$C,ASTX$S    ; External system macros
    20           .MCALL  TYPE             ; External supplied macros
    21   ;
    22   START:  CLR     R0               ; Error count
(1) 23           SREA$C  REXAST           ; Set up Specify Exit AST
    24           BCS     ERR1             ; Branch on dir error
    25           TYPE    <ASTEX STARTING UP. WILL WORK UNTIL ABORTED.>
    26   ; Do some work.
    27           CLR     R2               ; Clear counter
(2) 28   LOOP:   INC     R2               ; Increment counter
    29           BR      LOOP             ; Loop back
    30   ; Error code
    31   ERR1:   INC     R0               ; Error count
    32           MOV     $DSW,R1          ; Move DSW for display
    33           IOT                      ; Trap and display
    34                                    ;  registers
    35   ; AST service routine
(3) 36   REXAST: TYPE    <TRYING TO ABORT ME, EH?> ; Display
    37           TYPE    <WE WON'T LET YOU THIS TIME!> ;  message
(4) 38           ASTX$S                   ; AST exit
    39           .END    START
```

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 2-8 Using a Requested Exit AST

Example 2-9 shows the use of an AST routine with the Mark Time (MRKT$) directive. The AST routine is entered after a 1Ø. second time period expires. The task starts the time period and then suspends itself until the 1Ø. seconds go by. The AST routine, when entered, resumes the task. Therefore, the task is unblocked and continues to execute when the AST routine exits. The "main" code then displays a message and exits.

The following notes are keyed to the example.

1. The Mark Time instructs the system to start the 1Ø. second interval. The two specifies seconds. After that, the AST routine at ASTSRT is entered. The missing first argument is for an event flag, which would, if specified, be initially cleared and then set when the 1Ø. seconds expired.

2. Task suspends itself. The AST routine is entered even though the task is suspended.

3. The AST routine resumes the task. Otherwise, the task would return to a suspended state upon exit from the AST routine.

4. This instruction cleans up the stack for the AST Exit directive. The extra word contains the event flag number of the event flag set, or zero (in this case) if none was specified. This word could be used to distinguish which MRKT$ directive had expired in the case of several MRKT$ directives, using different event flags but the same AST routine.

5. After the task is resumed by the AST routine, it starts here.

If a task uses the Mark Time directive to place a time limit on an operation, the Mark Time can be cancelled using the Cancel Mark Time directive if the operation completes before the time limit expires.

```
    1           .TITLE  MARK
    2           .IDENT  /01/
    3           .ENABL  LC                 ; Enable lower case
    4   ;+
    5   ; FILE MARK.MAC
    6   ;
    7   ; This program issues a mark time for 10 seconds and
    8   ; then stops itself.  When the mark time expires, an AST
    9   ; routine is invoked which unstops the task.
   10   ;
   11   ; Assemble and task-build instructions:
   12   ;
   13   ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]MARK
   14   ;       LINK/MAP MARK,LB:[1,1]PROGSUBS/LINRARY
   15   ;
   16   ; Install and run instructions:
   17   ;
   18   ;       The task must be installed under the name MARK in
   19   ;       order to run correctly
   20   ;-
   21           .MCALL  EXIT$S,MRKT$C,ASTX$S,SPND$S,RSUM$C
   22                                      ; System macros
   23           .MCALL  TYPE               ; Special supplied macro
   24
   25   START:  CLR     R0                 ; R0 is used to identify
   26                                      ;  errors
   27           TYPE    <'MARK' IS RUNNING AND WILL SUSPEND>
   28           TYPE    <ITSELF UNTIL AST RESUMES IT>
(1) 29           MRKT$C  ,10.,2,ASTSRT      ; Issue mark time
   30           BCS     ERR1               ; Branch on directive
   31                                      ;  error
(2) 32           SPND$S                     ; Suspend task
(5) 33           BCS     ERR2               ; Branch on directive
   34                                      ;  error
   35           TYPE    <'MARK' IS RESUMED AND WILL EXIT>
   36           EXIT$S                     ; Exit
   37   ERR3:   INC     R0                 ; R0 = 3 if error on
   38                                      ;  unstop
   39   ERR2:   INC     R0                 ; R0 = 2 if error on
   40                                      ;  mark time
   41   ERR1:   INC     R0                 ; R0 = 1 if error on
   42                                      ;  stop
   43           MOV     $DSW,R1            ; Save DSW
   44           IOT                        ; Abort task and dump
   45                                      ;  registers
```

Example 2-9 Using an AST in the Mark Time Directive
(Sheet 1 of 2)

```
46      ;
47      ;       AST SERVICE ROUTINE
48      ;
49      ASTSRT: TYPE    <AST ROUTINE EXECUTING AND WILL UNSTOP 'MARK'>
③ 50            RSUM$C  MARK                ; Resume task
51              BCS     ERR3                ; Branch on directive
52                                          ;  error
53      ; User must clean AST specific values off the stack so
54      ; that the Exec gets control with stack as expected
55      ; (with regular 4 AST words)
④ 56            TST     (SP)+               ; Clean off stack for
57                                          ;  AST return
58              ASTX$S                      ; Return to main code
59                                          ;  through ast exit
60              .END    START
```

```
Run Session

>INSTALL MARK
>RUN MARK
>
 'MARK' IS RUNNING AND WILL SUSPEND
ITSELF UNTIL AST RESUMES IT
AST ROUTINE EXECUTING AND WILL UNSTOP 'MARK'
'MARK' IS RESUMED AND WILL EXIT
```

Example 2-9 Using an AST in the Mark Time Directive
(Sheet 2 of 2)

Synchronous System Traps (SSTs)

There is another kind of system trap available on the system, generally used if you wish to handle trap producing errors yourself, rather than have the Executive handle them. They are called Synchronous System Traps (or SSTs). They detect certain events which occur when program instructions are executed (e.g., odd address traps and memory protect violations). They are synchronous because they always occur at the same point in the program, when a given trap-causing instruction is executed.

> LEARNING ACTIVITY 2-2
>
> Read sections 2.3.1 on Synchronous System Traps and 2.3.2 on SST Service Routines in the RSX-11M/M-PLUS Executive Reference Manual. Pay particular attention to the section on SST service routines.

To set up for user coded SSTs, you must set up a vector table in a data area that contains a list of SST service routine addresses. Each entry in the table corresponds to a specific SST which may occur. A zero in an entry indicates that the Executive should handle that trap. Refer to Figure 2-8, which shows the setup and use of an SST routine. The following comments are keyed to this figure.

1. At start-up, the task issues a SVTK$ or SVDB$ directive, specifying the vector table address, which causes the Executive to record that address, setting up for user SST service routines.

2. The Executive returns control to the task.

3. An instruction is executed which causes a trap. The Executive checks the SST vector table to see if the user has specified a routine to handle the trap. If one is specified, the Executive sets up the user stack and transfers control to the SST routine. If no SST routine is specified, the Executive aborts the task and displays an error message at TI:.

4. Once the task receives control again, it executes the SST routine as if in the main code. All system services are available to the task. To return to the main code, clean up the stack so it contains only the return PC and PSW, and execute an RTT or RTI instruction.

5. The RTT or RTI instruction causes the PC and PSW to be popped from the stack into the appropriate register, causing a return to the "interrupted" code.

Note that the general purpose registers RØ through R5 and SP are not saved. Therefore, if you use these registers in an SST routine, you must save and restore them.

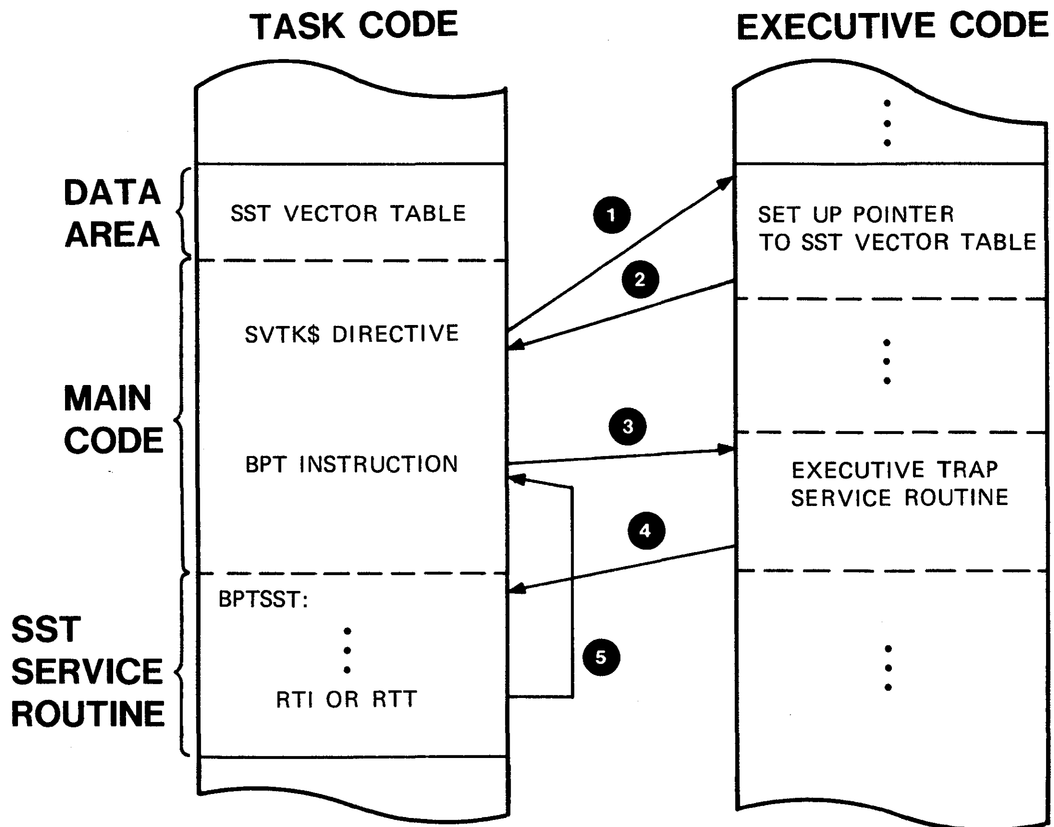


Figure 2-8 SST Sequence

Example 2-10 uses three SST service routines to handle BPT, IOT and memory protection violation traps in the user program. The following notes are keyed to this example.

1. Vector table containing the SST service routine addresses. See the documentation on SVTK$ in Chapter 5 of the RSX-11M/M-PLUS Executive Reference Manual for the order of words in the table.

2. Executive directive to permit the use of user SST service routines. You can also use SVDB$ to trap to an external debugger (e.g., ODT) instead of to the user code.

3. BPT causes a trap. The Executive checks the vector table; because a routine address is specified for BPTs, it sets up the stack and transfers control to location BPT.

4. The BPT SST routine displays a message, then returns from trap, to line 28.

5. The CLR 120000 causes a memory protect violation since the highest address used in this program is far below that (1627(8)). This causes another SST.

6. On a Memory Protect Violation SST, the Executive passes three more words on the stack in addition to the PC and the PSW. The details on these words are discussed in section 2.3.2 on SST Service Routines in the RSX-11M/M-PLUS Executive Reference Manual.

   We don't need the stack values in this routine, but we do need to pop them off the stack so that the RTI instruction works properly. The CMP and the TST are "dummy" instructions used to pop the three words off the stack.

7. IOT causes another SST.

8. In the IOT routine, we can alter the return PC (on the top of the stack), which changes the return point for the RTI to NEW.

9. The TRAP instruction causes an SST for which there is no user specified routine. Therefore, the Executive aborts the task and displays a message at TI:.

```
 1          .TITLE  SST
 2          .IDENT  /01/
 3          .ENABL  LC                ; Enable lower case
 4  ;
 5  ; FILE SST.MAC
 6  ;
 7  ; This task sets up an SST vector table to handle SST's
 8  ; for BPT, IOT, and odd address traps. It then executes
 9  ; instructions to cause these traps to occur. In each
10  ; SST routine, a message is displayed and then the task
11  ; continues. Finally, a TRAP instruction is executed.
12  ; Since no user SST routine is specified for TRAP, the
13  ; Executive aborts the task.
14  ;
15  ; Assemble and task-build instructions:
16  ;
17  ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SST
18  ;       LINK/MAP SST,LB:[1,1]PROGSUBS/LIBRARY
19  ;
20          .MCALL  SVTK$C,EXIT$S     ; External system macros
21          .MCALL  TYPE              ; External supplied macro
22  ;
23  VTABLE: .WORD   0,MPTVIO,BPT,IOT  ; SST vector table
24  ;
25  START:  SVTK$C  VTABLE,4          ; Have Executive set up
26                                    ; SST table
27          BPT                       ; BPT instruction
28          CLR     120000            ; Clear location 120000,
29                                    ;  causing a memory
30                                    ;  protect violation
31          IOT                       ; IOT instruction
32          EXIT$S                    ; Exit
33  NEW:    TRAP                      ; TRAP instruction
34  ;
35  ; SST routines
36  ;
37  MPTVIO: TYPE    <MEMORY PROTECT VIOLATION CAUGHT> ; Type
38                                    ;  message
39          CMP     (SP)+,(SP)+       ; Clean off three
40          TST     (SP)+             ;  specific stack words
41                                    ;  for memory protect SST
42          RTI                       ; Return from trap
43  BPT:    TYPE    <BPT CAUGHT>      ; Type message
44          RTI                       ; Return from trap
45  IOT:    TYPE    <IOT CAUGHT>      ; Type message
46          MOV     #NEW,(SP)         ; Change PC on stack so
47                                    ;  return from trap
48                                    ;  returns to NEW
49          RTI                       ; Return from trap
50          .END    START
```

Callouts in margin: 1 (line 23), 2 (line 25), 3 (line 27), 5 (line 28), 7 (line 31), 9 (line 33), 6 (lines 37–42), 4 (lines 43–44), 8 (line 46).

Example 2-10 Using SSTs (Sheet 1 of 2)

```
Run Session

>RUN SST
BPT CAUGHT
MEMORY PROTECT VIOLATION CAUGHT
IOT CAUGHT
14:07:50  Task "TT11  " terminated
          TRAP execution
          R0=001573
          R1=000012
          R2=000000
          R3=140312
          R4=144000
          R5=000000
          SP=001254
          PC=001312
          PS=170000
>
```

Example 2-10 Using SSTs (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, under UFD [202,2].

You will need the program READF.MAC to do question 1. It should be available on-line (probably under UFD [202,1]). 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 the QIO directive. 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 device that is not file-structured (e.g., a terminal)
2. To choose either synchronous or asynchronous I/O as the most effective method for a given application
3. To perform complete error checking upon I/O completion
4. To use formatting routines from the system subroutine library to improve the readability of output data.

RESOURCES

1. RSX-11M/M-PLUS Executive Reference Manual, specific directives in Chapter 5
2. RSX-11M/M-PLUS I/O Driver's Reference Manual, Chapters 1, 2 and 3
3. IAS/RSX-11 System Library Routines Reference Manual, Chapter 6

OVERVIEW OF QIO DIRECTIVES

All I/O operations under RSX-11M are performed using 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 ANSI magtape, and NETACP for DECNET.

The I/O packet is placed in an I/O queue for the service routine. The packets are queued up in order according to 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 only for I/O on non-file structured devices such as terminals. For file I/O, the File Control Services (FCS) or Record Management Services (RMS) are used, which in turn issue the appropriate QIOs for you.

When using QIOs, you need to 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). To specify additional information about the I/O operation (e.g., what buffer to write and how many characters), use 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 Executive directives.

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 device which is not file-structured, such as a terminal, it is converted to logical block I/O for you. 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 Function Codes

Global Symbol	Octal Value	Function
IO.ATT	001400	Attach device
IO.DET	002000	Detach device
IO.KIL	000012	Cancel I/O requests
IO.RLB	001000	Read Logical Block
IO.RVB	010400	Read Virtual Block
IO.WLB	000400	Write Logical Block
IO.WVB	011000	Write Virtual Block

Logical Unit Numbers (LUN)

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:

These default assignments may be overridden at task-build time by using the ASG option. Additional LUNs can be created (up to a maximum of 25Ø(1Ø)) 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 (GLUN$), and changed using the Assign LUN directive (ALUN$).

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 all synchronization. With asynchronous I/O, you must provide synchronization regarding the completion of the I/O operation itself.

When a task issues a synchronous I/O request, it doesn't get control back from the Executive until after:

1. The I/O packet is queued, and

2. 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, to wait until the specified event flag is set, signifying that the actual I/O operaton is complete.

Figure 3-1 shows the flow of instructions during the processing of a QIO directive. 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 QIOW$ directive to invoke this type of I/O.

Commentary to Figures 3-1 and 3-2:

1. User task executes QIO and Wait For directives.
2. Executive queues the I/O request.
3. Executive calls the driver.
4. Driver begins the I/O transfer.
5. Driver handles the I/O transfer as necessary.
6. I/O transfer completes.
7. Driver finishes its work and notifies the task that the I/O is completed.
8. User task continues.

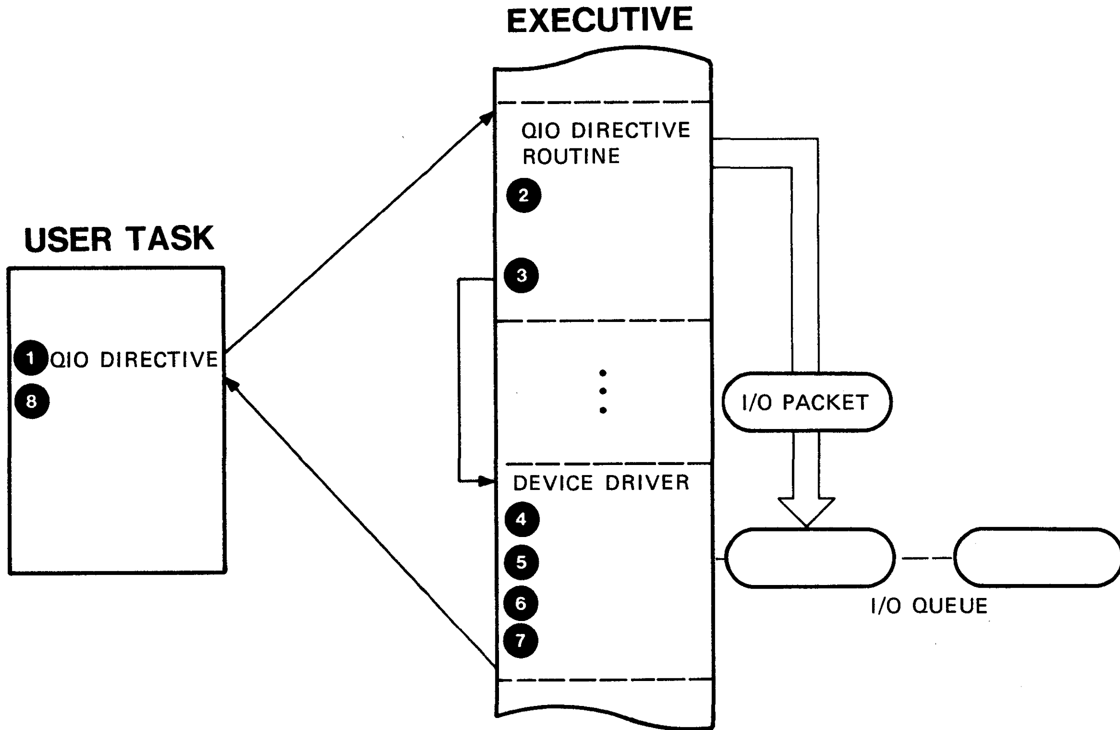


Figure 3-1 Execution of a Synchronous I/O Request

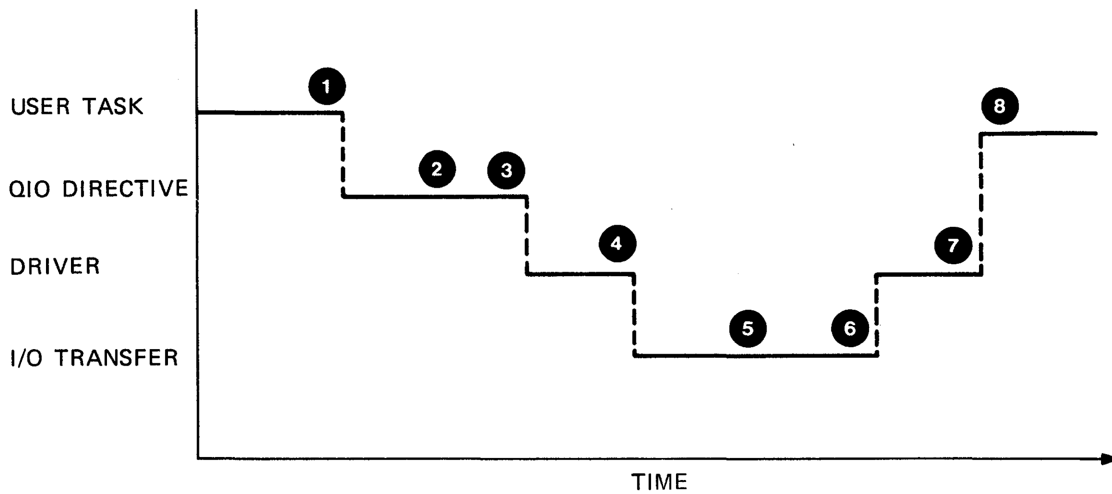


Figure 3-2 Events in Synchronous I/O

With asynchronous I/O, the Executive still queues the I/O request. 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 on such factors as how many other I/O packets are ahead of this one in the driver's I/O queue, and the speed of the device itself. The task executes in parallel with the I/O request.

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 step 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, asynchronous system traps, or both. The task shown in Figures 3-3 and 3-4 uses a Wait For Single Event Flag directive at step 5a. Use the directive QIO$ to invoke this type of I/O.

The advantage of asynchronous I/O is that a task can continue processing in parallel with the I/O transfer. For example, you can perform computations while waiting for a read or write 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 the QIO directive.

2. Executive queues the I/O request.

3. Executive calls the driver.

4. Driver begins the I/O transfer; Executive passes control back to the user task.

5. Driver handles the I/O transfer as necessary. User task executes in parallel with the I/O transfer.

   a. User task waits for the I/O operation to complete.

6. I/O transfer completes.

7. Driver finishes up and the Executive notifies the task that I/O is completed.

8. User task continues.

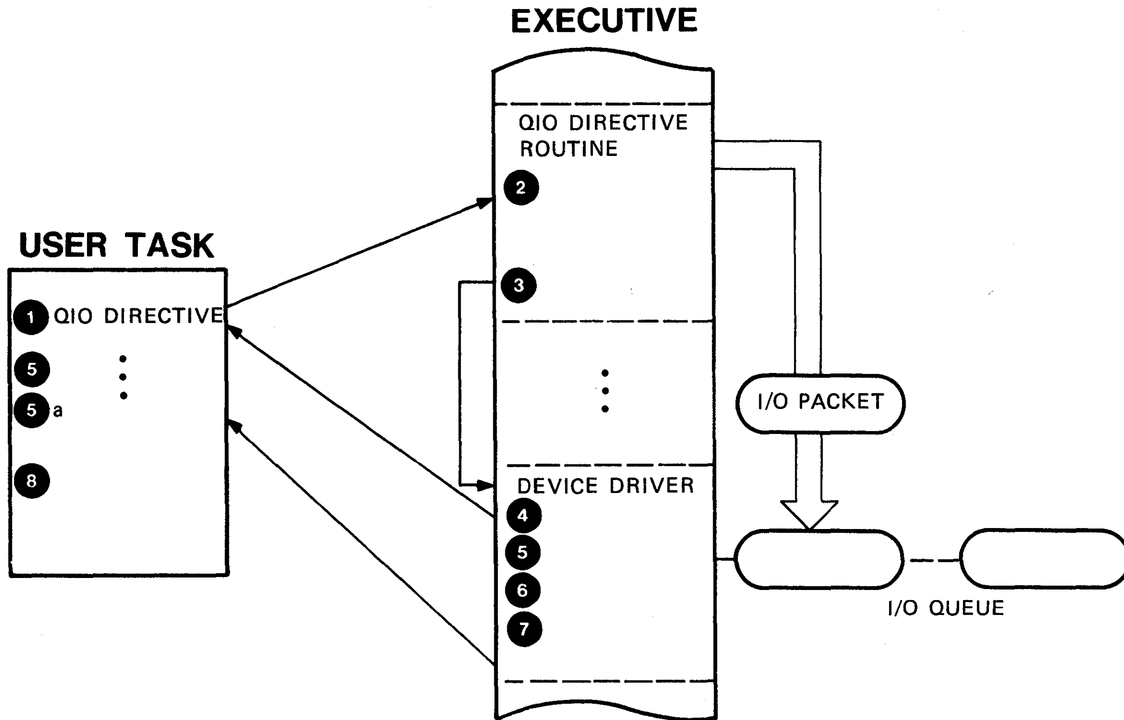


Figure 3-3 Execution of an Asynchronous I/O Request

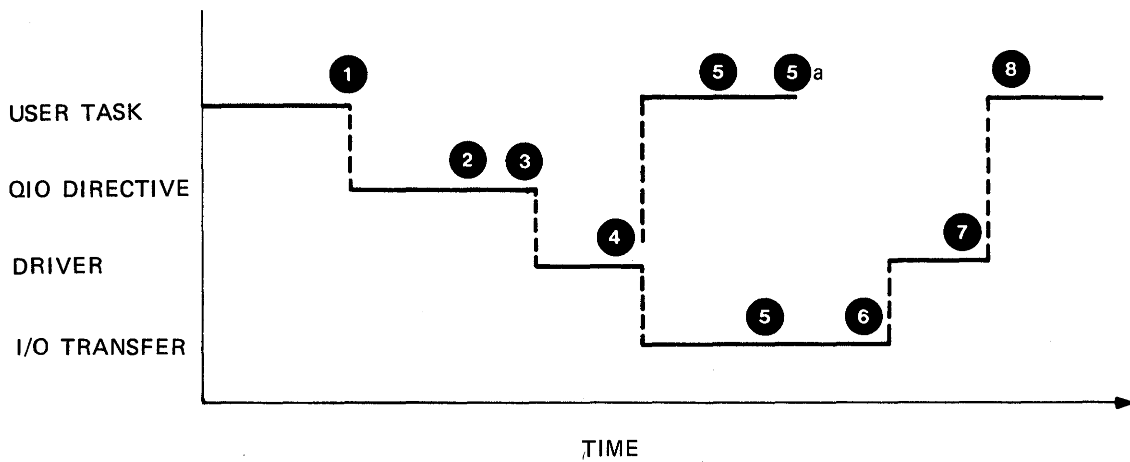


Figure 3-4 Events in Asynchronous I/O

MAKING THE I/O REQUEST

Specify the following information in the QIO$ or QIOW$ call 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) - two words set aside with .BLKW or .BLKB assembler directives. The IOSB is used to pass status and other information about the I/O operation back to the task.
- The address of an AST routine, if transfer to an AST routine is desired upon completion of the I/O transfer.
- The I/O parameter list (up to six words) which specifies information for the particular device and for the particular I/O function requested.

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. Table 2-3 (in section 2.3 on the QIO Macro) in 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 further, later in this module. If your RSX-11M system has the half-duplex terminal driver, Table 3-3 in section 3.3 on the QIO Macro 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 10 second intervals) NOTE: Word 3 is Only used if a special subfunction bit is set. See this module's 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:

Octal Value	ASCII Character	Meaning
040	Blank	Single space
060	0	Double space
061	1	Form feed
044	$	Prompting output-stay in same location after output
053	+	Overprint
000	Null	No implied format control - use internal control

Words 4, 5, and 6 - Unused

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 Driver's Reference Manual and also in the RSX-11M Mini-Reference.

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

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

<table>
<tr><td>Device Dependent</td><td>I/O Status</td></tr>
<tr><td colspan="2">Actual Number of Bytes Transferred</td></tr>
</table>

NOTE

The low-order byte of the first word of the I/O Status Block contains the I/O status code. This is a byte value, not a word value. A positive I/O status code (usually +1 = IS.SUC) indicates success. Again, 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 a write which ends after only some of the data is transferred. The device dependent byte usually contains information which is device dependent. For example, for a read from a terminal, it contains the character which was typed 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 right 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 section that follows, after an example of synchronous I/O.

THE QIO DIRECTIVES

Synchronous I/O

The format of the QIOW$ call is:

```
        QIOW$   ifn,lun,efn,pri,iosb,ast,iopl
```

where

```
ifn  - I/O function code
lun  - Logical unit number
efn  - Event flag number (required for synchronous I/O)
pri  - Priority (not used)
iosb - I/O status block address
ast  - AST routine address
iopl - I/O parameter list
```

Example using the $S form:

```
                .
            .MCALL QIOW$S
                .
                .
                .
    BUFF:   .ASCII /HERE IS THE MESSAGE/
    LBUFF:  =.-BUFF
            .EVEN
    IOSB:   .BLKW      2
                .
                .
                .
            QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<#BUFF,#LBUFF,#40>
                .
```

Explanation of QIO arguments:

Write Virtual Block
LUN 5 (TI:)
Event flag #1
Priority (always ignored)
I/O status block address = IOSB
AST routine address (none specified)
I/O parameter list
    Input buffer address = BUFF
    Buffer length = LBUFF
    Vertical format control = 40(8) for single space

Once again, the $, $C, or $S form of the directive may be used. 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.

ASTs are not generally used for synchronous I/O, because the Executive performs all synchronization for you. The I/O parameter list is a single directive parameter. Therefore, the list is enclosed in angle brackets, with the elements separated by commas. In fact, six words are always placed in the DPB for the I/O parameter list, whether or not all six words are specified.

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

1. As with other directives, the macro names must be specified in a .MCALL statement. Note that in this example, we use both the $C form and the $S form of the QIOW$ directive.

2. The two-word I/O status block for return of I/O status.

3. The buffer into which the data will be read, and also from which the data will be displayed.

4. R4 is used to indicate whether a QIO error is a directive error or an I/O error. A value of zero indicates that a directive error occurred (and that R3 will contain the DSW value). A value of -1(177777(8)) indicates that an I/O error occurred (and that R3 will contain the I/O status byte).

5. Issue the read request. We are using LUN 5, event flag 1, and IOSB is the label of the IOSB. The I/O parameter list is set up as a single parameter (hence the need for the angle brackets (< and >)). It specifies BUFF, the address of the buffer for the characters read and 80., the maximum number of characters to read. If input is terminated with a terminating character, such as a carriage return, before 80 characters are typed in, the number of characters actually read will be returned in the second word of the IOSB. Input will be terminated automatically after the 80th character, if 80 characters are typed. In that case, 80 will be returned as the number of characters read.

6. Check for directive error - indicating a failure in queueing the I/O packet.

7. With synchronous I/O, we don't get control again until after the I/O operation has completed, so also check the I/O status. A value less than zero indicates an error in the I/O transfer.

8. Get the count of characters typed in from the second word of the IOSB. We will only check on and convert that many characters.

9. Check each character to see if it is in the range ASCII A to ASCII Z. If so, convert to lowercase by adding 32(10) = 40(8) to that value, or else continue.

10. Write out the buffer BUFF, which has the converted message. This is a Write Virtual Block. We use the $S form instead of the $C form because we don't know how many characters to write until run time. The $ form would also work. Notice the difference in the format of the arguments for the $S form compared to the $C form. Note also that in the $S form, the lack of a '#' sign in IOSB + 2 means get the <u>contents</u> of that location, specifically the number of characters to write out. The third argument of the I/O parameter list, #40, is for vertical format control. Single linefeed before writing the characters is indicated by #40, or ASCII space.

11. Check for any directive or I/O errors.

12. See note 4. R5 is the directive counter, which will be one for the first QIO and two for the second QIO. We need to distinguish directive errors from I/O errors. In this example, we use R4 to distinguish the two type of errors. Zero in R4 means a directive error, and -1 (or 177777(8) in two's complement) in R4 indicates an I/O error. For directive errors, the DSW is placed in R3; for I/O errors, the I/O status byte is placed in R3.

The list of all error codes appears in Appendix B of the RSX-11M/M-PLUS I/O Drivers Reference Manual and in the RSX-11M Mini-Reference Manual. Of course, this simple error handling will normally be replaced with a text error message and the error code. You will learn how this is done later in the module.

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 by the Executive to a logical block request. If, for example, logical block writes are used and someone reassigns the LUN to a disk, the write may overwrite a block on the disk. If, on the other hand, write virtual blocks are used and someone reassigns the LUN to a disk, the write will only be allowed if a file is open on the disk. The write will fail in most cases if the program is writing to a terminal.

```
   1           .TITLE  SYNCHQ
   2           .IDENT  /01/
   3           .ENABL  LC                   ; Enable lower case
   4   ;+
   5   ; FILE SYNCHQ.MAC
   6   ;
   7   ; This task reads a line of text from the terminal,
   8   ; converts all upper case characters to lower case, and
   9   ; prints the converted message back at the terminal. It
  10   ; uses synchronous QIO directives.
  11   ;-
(1)12          .MCALL  QIOW$C,QIOW$S,EXIT$S ; External system
  13                                        ;  macros
  14
(2)15  IOSB:   .BLKW   2                    ; I/O Status Block
(3)16  BUFF:   .BLKB   80.                  ; Text buffer
  17
  18   START:  CLR     R5                   ; Error Count
(4)19          CLR     R4                   ; Error indicator - 0
  20                                        ;  means directive error
  21                                        ;  (DSW in R3), neg
  22                                        ;  means I/O error
  23                                        ;  (I/O status in R3)
(5)24          QIOW$C  IO.RVB,5,1,,IOSB,,<BUFF,80.> ; Issue
  25                                              ;  read
(6)26          BCS     ERR1                 ; Branch on dir error
(7)27          TSTB    IOSB                 ; Check for I/O error
   28          BLT     ERR1A                ; Branch on I/O error
(8)29          MOV     IOSB+2,R0            ; Get count of characters
  30                                        ;  typed in
  31           CLR     R1                   ; Offset into buffer to
  32                                        ;  character
(9)33  LOOP:   CMPB    BUFF(R1),#'A         ; Check for upper case
  34                                        ;  ASCII character
  35           BLT     NEXT                 ; Branch if below range
  36           CMPB    BUFF(R1),#'Z
  37           BGT     NEXT                 ; Branch if above range
  38   ; Here if upper case, move to register R2 and convert
  39           MOVB    BUFF(R1),R2          ; Move to register
  40           ADD     #32.,R2              ; Convert to lower case
  41           MOVB    R2,BUFF(R1)          ; Replace in message
  42   NEXT:   INC     R1                   ; Increment offset into
  43                                        ;  buffer to next char
  44           SOB     R0,LOOP              ; Decrement count of
  45                                        ;  characters left to check
(10)46         QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<#BUFF,IOSB+2,#40>
  47                                        ; Write text
```

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

```
❶ [48            BCS     ERR2            ; Branch on dir error
   49            TSTB    IOSB            ; Check for I/O error
   50            BLT     ERR2A           ; Branch on I/O error
   51            EXIT$S                  ; Exit
   52    ;
   53    ; Error code
   54    ;
❷ [55    ERR2A:  INC     R5              ; Up error count - 2nd QIO
   56    ERR1A:  INC     R5              ;                 - 1st QIO
   57            MOVB    IOSB,R3         ; I/O error. I/O status
   58                                    ;  to R3.
   59            DEC     R4              ; Negative value in R4
   60                                    ;  means I/O error
   61            IOT                     ; Trap and display
   62                                    ;  registers
   63    ERR2:   INC     R5              ; Up error count - 2nd QIO
   64    ERR1:   INC     R5              ;                 - 1st QIO
   65            MOV     $DSW,R3         ; Directive error. DSW
   66                                    ;  to R3, leave R4=0.
   67            IOT                     ; Trap and display
   68                                    ;  registers
   69            .END START
```

```
Run Session

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

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

Asynchronous I/O

The format of the QIO$ call is:

```
QIO$      ifn,lun,efn,pri,iosb,ast,iopl
```

where

```
ifn  - I/O function code
lun  - Logical unit number
efn  - Event flag number
pri  - Priority (not used)
iosb - I/O status block address
ast  - AST routine address
iopl - I/O parameter list (up to six words)
```

Example using the $C form:

```
               .
               .MCALL  QIO$C
               .
               .
               .
        IBUF:  .BLKB   80.
        IOSB:  .BLKW   2.
               .
               .
               .
               QIO$C   IO.RVB,5,1,,IOSB,,<IBUF,80.>
               .
```

Explanation of QIO arguments:

```
Read Virtual Block
LUN 5 (TI:)
Event flag 1
Priority (ignored)
I/O status block address = IOSB
AST routine address (not used here)
I/O parameter list
    Buffer address = IBUF
    Buffer length  = 80.
```

Synchronization With Asynchronous I/O

As mentioned earlier, event flags and asynchronous system traps 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.

Another possible technique for synchronization is to use ASTs (discussed in Chapter 2). The following techniques might be used with ASTs, after specifying an AST routine address in the QIO$ directive.

1. "Main" task does some work, then suspends or stops itself. AST routine resumes or unstops the task.
2. "Main" task works the entire time, periodically checking a cleared event flag or a cleared byte in a local data area. AST routine sets the flag or sets the byte to a nonzero value, thus notifying the "main" task that the I/O operation has completed. If an event flag is used, it will typically be different from the flag specified in the asynchronous I/O request.

A third technique which can be used is to monitor the contents of the I/O status byte of the I/O status block. The complete 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. Here we use QIO$C and QIO$S instead of QIOW$C and QIOW$S. WTSE$C is a Wait for Single Event Flag directive, used to synchronize the I/O operation.

2. A work buffer to be filled with values while the I/O transfer is going on.

3. Issue the read. QIO$C instead of QIOW$C. All arguments are the same. If ASTs were used for synchronization, an AST address would be specified. The Executive will clear Event Flag 1 when the I/O packet is queued and set it when the I/O operation completes.

4. Check again immediately for directive errors. Here, you are checking for an error in queueing the I/O packet.

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

6. When you are finished with your work, enter a Wait For state until the event flag specified in the QIO$ directive is set. It will be set when the I/O operation completes.

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

8. After converting the message, issue the write.

9. This time, wait for the flag to be set immediately after checking the directive status. You could do some more work here. If you choose to wait, it is simpler and more efficient to use synchronous I/O (QIOW$). Synchronous I/O is more efficient because you perform both functions (QIO$ and WTSE$) in just one Executive directive call.

10. Still use the error count to indicate the directive number. Since there are now extra directives (the WTSEs), adjust the counts accordingly.

```
    1           .TITLE ASYNCQ
    2           .IDENT  /01/
    3           .ENABL  LC                  ; Enable lower case
    4   ;+
    5   ; FILE ASYNCQ.MAC
    6   ;
    7   ; This task reads a line of text from the terminal,
    8   ; converts all upper case characters to lower case, and
    9   ; prints the converted message back at the terminal. It
   10   ; uses asynchronous QIO, using wait for event flag for
   11   ; synchronization.
   12   ;-
(1) 13          .MCALL  QIO$C,QIO$S,EXIT$S,WTSE$C ; External
   14                                       ;  system macros
   15
   16   IOSB:   .BLKW   2                   ; I/O Status Block
   17   BUFF:   .BLKB   80.                 ; Text buffer
(2) 18   K:      .BLKW   10.                 ; Array to fill while
   19                                       ;  waiting for I/O
   20
   21   START:  CLR     R5                  ; Error Count
   22           CLR     R4                  ; Error indicator: 0
   23                                       ;  means directive error,
   24                                       ;  -1 means I/O error
(3) 25          QIO$C   IO.RVB,5,1,,IOSB,,<BUFF,80.,40> ; Issue
   26                                       ;  read
(4) 27          BCS     ERR1                ; Branch on dir error
  [28   ; Now do some work
  |29           CLR     R0                  ; Offset into array K
  |30           MOV     #64.,R1             ; Value to place in array
  |31   PLACE:  MOV     R1,K(R0)            ; Place value in array
(5)|32          ADD     #2,R0               ; Point to next element
  |33                                       ;  in K
  |34           CMP     R0,#20.             ; At the end?
  |35           BHI     WAIT                ; Branch if done
  |36           ADD     #64.,R1             ; Compute next value
  [37           BR      PLACE               ; Place it in the array
   38   ; Now wait for I/O operation to complete
(6) 39   WAIT:   WTSE$C  1                   ; Wait for I/O to
   40                                       ;  complete
   41           BCS     ERR2                ; Check for dir error
(7)[42          TSTB    IOSB                ; Check for I/O error
  [43           BLT     ERR1A               ; Branch on I/O error
   44           MOV     IOSB+2,R0           ; Get count of characters
   45                                       ;  typed in
   46           CLR     R1                  ; Offset into buffer to
   47                                       ;  character
```

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

```
48    LOOP:    CMPB    BUFF(R1),#'A      ; Check for upper case
49                                       ;  ASCII character
50             BLT     NEXT              ; Branch if below range
51             CMPB    BUFF(R1),#'Z
52             BGT     NEXT              ; Branch if above range
53    ; Here if upper case, move to register R2 and convert
54             MOVB    BUFF(R1),R2       ; Move to register
55             ADD     #32.,R2           ; Convert to lower case
56             MOVB    R2,BUFF(R1)       ; Replace in message
57    NEXT:    INC     R1                ; Increment offset into
58                                       ;  buffer to next char
59             SOB     R0,LOOP           ; Decrement char count
60             QIO$S   #IO.WVB,#5,#1,,#IOSB,,<#BUFF,IOSB+2,#40>
61                                       ; Write text
62             BCS     ERR3              ; Branch on dir error
63    ; Could do some more work here too
64             WTSE$C  1                 ; Wait for I/O to
65                                       ;  complete
66             BCS     ERR4              ; Branch on dir error
67             TSTB    IOSB              ; Check for I/O error
68             BLT     ERR3A             ; Branch on I/O error
69             EXIT$S                    ; Exit
70    ; Error code
71    ERR3A:   INC     R5                ; R5=3, 2nd QIO
72             INC     R5                ;
73    ERR1A:   INC     R5                ; R5=1, 1st QIO
74             DEC     R4                ; Make R4 negative to
75                                       ;  indicate I/O error
76             MOVB    IOSB,R3           ; I/O status to R3
77             IOT                       ; Trap and display
78                                       ;  registers
79    ERR4:    INC     R5                ; R5=4, 2nd Wait For
80    ERR3:    INC     R5                ; R5=3, 2nd QIO
81    ERR2:    INC     R5                ; R5=2, 1st Wait For
82    ERR1:    INC     R5                ; R5=1, 1st QIO
83             MOV     $DSW,R3           ; Directive error. DSW
84                                       ;  to R3, leave R4=0.
85             IOT                       ; Trap and display
86                                       ;  registers
87             .END START
```

(Callouts: 8 at line 60, 9 at line 64, 10 at lines 79–85)

Run Session

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

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

Example 3-3 shows the use of ASTs for synchronization. In addition, it shows the use of some supplied macros for generating error reports. These macros are documented in Appendix A of this course. The following notes are keyed to the example.

1. This is the text for the messages to be written. The LEN=.-MES lets the assembler calculate the length of the message for you. A similar technique is used for the other messages.

2. The ASCII text may contain an odd number of characters. The .EVEN assembler directive assures that your first executable instruction is an even word boundary.

3. Issue the write request. The AST routine address is specified. Also specify the address of the buffer, MES, and its length LEN. You can use the $C form of the directive because all arguments are known at assembly time.

4. Suspend until the AST routine is activated upon I/O completion. Normally some other processing would be done here, in parallel with the I/O operation.

5. The Executive passes control to the AST routine when the I/O transfer completes. First check the I/O status. You do that here instead of in the main code because you will be issuing another write which will overwrite the IOSB. The I/O status check could otherwise be checked in the main code after the task is resumed.

6. Write out a message so the operator knows you are in the AST routine. This time you use synchronous I/O, since you aren't planning to do any work while the I/O transfer takes place. Again, check for errors.

7. Resume the task so it will be ready to run upon exit from the AST routine.

8. Pop the extra word off the stack (this AST is entered with five words on the stack instead of the standard four). Then use the ASTX$ directive to exit the AST routine via the Executive.

9. Check for directive errors on the SPND$. It's possible that you never suspended yourself.

10 Write another message synchronously, check for errors, and then exit.

11 The DIRERR and IOERR macros generate error messages for you. DIRERR generates a message with the following format.

```
DIRECTIVE ERROR
<user message>
DSW = <value> (in signed decimal)
```

IOERR generates a message of the following format:

```
I/O ERROR
<user message>
I/O STATUS BLOCK = <hb>,<lb>/<2nd word>
(in signed decimal)
```

hb is the high byte of the first word.
lb is the low byte of the first word.

Each of these macros then causes the task to exit. Later in this module you will learn how to generate such messages yourself.

```
 1          .TITLE  QIOAST
 2          .IDENT  /01/
 3          .ENABL  LC               ; Enable lower case
 4    ;+
 5    ; FILE QIOAST.MAC
 6    ;
 7    ; This program issues a QIO and then suspends itself.
 8    ; When the I/O operation completes, an AST routine is
 9    ; invoked which resumes the task.
10    ;
11    ; Assemble and task-build instructions:
12    ;
13    ;         MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]QIOASR
14    ;         LINK/MAP QIOAST,LB:[1,1]PROGSUBS/LIBRARY
15    ;
16    ; Install and run instructions: Install the task so that
17    ; the Resume directive works properly.
18    ;-
19          .MCALL  EXIT$S,QIO$C,QIOW$C,ASTX$S ; System
20          .MCALL  SPND$S,RSUM$C              ;  macros
21          .MCALL  IOERR,DIRERR     ; Supplied macros
22    IOSB:  .BLKW   2               ; I/O status block
23    MES:   .ASCII  /'QIOAST' IS STARTING/ ; Startup message
24    LEN    =       .-MES
25    MES1:  .ASCII  /'QIOAST' HAS BEEN RESUMED AND WILL/
26 (1)       .ASCII  / NOW EXIT/      ; Resumed message
27    LEN1   =       .-MES1
28    MES3:  .ASCII  /ASTRT IS EXECUTING AND WILL NOW/
29           .ASCII  / RESUME QIOAST/ ; AST message
30    LEN3   =       .-MES3
31 (2)       .EVEN
32 (3) START: QIO$C   IO.WVB,5,1,,IOSB,ASTRT,<MES,LEN,40>
33                               ; Issue write
34           BCS     ERR1        ; Branch on dir error
35 (4)       SPND$S              ; Suspend self
36 (9)       BCS     ERR2        ; Branch on dir error
37 (10)      QIOW$C  IO.WVB,5,1,,IOSB,,<MES1,LEN1,40> ; Issue
38                               ;  write
39           BCS     ERR3        ; Branch on dir error
40           TSTB    IOSB        ; Check for I/O error
41           BLT     ERR3A       ; Branch on I/O error
42           EXIT$S              ; Exit
43    ; Main code error handling, using supplied macros
44    ERR1:  DIRERR  <ERROR ON 1ST QIO BY QIOAST>
45    ERR1A: IOERR   #IOSB,<ERROR ON 1ST QIO BY QIOAST>
46 (11) ERR2: DIRERR <ERROR ON SUSPEND>
47    ERR3:  DIRERR  <ERROR ON 2ND QIO BY QIOAST>
48    ERR3A: IOERR   #IOSB,<ERROR ON 2ND QIO BY QIOAST>
```

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

```
   49   ;
   50   ; AST service routine - entered when the 1st QIO by the
   51   ; main code completes
   52   ;
 ┌53   ASTRT:  TSTB    IOSB              ; check I/O status on
5│54                                     ;  I/O operation
 └55           BLT     ERR1A             ; Branch on I/O error
6 56           QIOW$C  IO.WVB,5,1,,IOSB,,<MES3,LEN3,40> ; Issue
   57                                     ;  write
   58           BCS     ERR4              ; Branch on dir error
   59           TSTB    IOSB              ; Check for I/O error
   60           BLT     ERR4A             ; Branch on I/O error
7 61           RSUM$C  QIOAST            ; Resume task
   62           BCS     ERR5              ; Branch on dir error
 ┌63           TST     (SP)+             ; Pop AST specific word
8│64                                     ;  off stack
 └65           ASTX$S                    ; Leave AST state and
   66                                     ;  return to main code
   67   ; AST error handling code
  ┌68  ERR4:   DIRERR  <ERROR ON QIO BY AST ROUTINE>
11│69  ERR4A:  IOERR   #IOSB,<ERROR ON QIO BY AST ROUTINE>
  └70  ERR5:   DIRERR  <ERROR ON RESUME BY AST ROUTINE>
   71           .END    START
```

Run Session

```
>INSTALL QIOAST
>RUN QIOAST
>
 'QIOAST' IS STARTING
AST IS EXECUTING AND WILL NOW RESUME QIOAST
'QIOAST' HAS BEEN RESUMED
```

Example 3-3 Asynchronous I/O Using an AST for Synchronization (Sheet 2 of 2)

TERMINAL I/O

Device Specific Functions

Some device-specific function codes are listed in Table 3-3. Table 2-3 in section 2.3 (on the QIO macro) of the RSX-11M/M-PLUS I/O Drivers 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 bits. These bits may be ORed with standard or device-specific function codes to produce special functions. 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. Notice that some device-specific functions, such as Read No Echo (IO.RNE), have equivalents using subfunction bits (IO.RLB!TF.RNE). Read After Prompt (IO.RPR) on the other hand, has no equivalent using subfunction bits.

NOTE

When you OR subfunction bits with read or write functions, use Read Logical Block or Write Logical Block, not the Read Virtual Block or Write Virtual Block. If the Executive converts a virtual block operation to a logical block operation, any subfunction bit settings are lost.

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 here.

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 shows how CTRL/Z can be specially handled 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	001020	Read With No Echo (Same as IO.RLB!TF.RNE)	<stadr,size[,tmo]>
IO.RPR	004400	Read After Prompt	<stadr,size,[tmo],pradr,prsize,vfc>
IO.RST	001001	Read With Any Special Terminators (Same as IO.RLB!TF.RST)	<stadr,size[,tmo]>
IO.RTT	005001	Read With Specified Special Terminators	<stadr,size,[tmo],table>
IO.WBT	000500	Write Logical Block, through ongoing I/O (Same as IO.WLB!TF.WBT) Task must be privileged	<stadr,size,vfc>
None	001200	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. System overhead is lower because only one QIO directive is processed. In addition, there is no window during which a response to the prompt may be ignored. Such a window may occur if separate QIOs are used to write and read, and if there is a delay between the write of the prompt and the read. The I/O parameter list contains six parameters, three for the read, and three for the write. The following notes are keyed to Example 3-4.

1. Placing the buffer with "You typed:" just ahead of the buffer for the input text allows the use of a single QIO to write out the complete line of output text. FINMES is the starting addres of the output text and length is FLEN + n, where n is the number of characters typed in.

2. We assign the symbol IOLEN to the second word of the IOSB. This allows you to reference that word with IOLEN, instead of using IOSTAT + 2.

3. QIO for Read After Prompt. The function code is IO.RPR. 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.

4. We are going to display the message typed, preceded by the text "you typed in." By placing the input buffer BUFF immediately after the preceding text, we now have our output text as one string beginning at FINMES. The total length of the message to be displayed is the length of the preceding text plus the number of characters typed in.

5. Use a normal QIO with Write Virtual Block to display the output.

6. If the operator types a CTRL/Z, an error status is returned. In this case, simply exit normally. Therefore, you must check for this condition and handle it specially.

```
 1            .TITLE  PROMPT
 2            .IDENT /01/
 3            .ENABL LC                  ; Enable lower case
 4    ;+
 5    ; File PROMPT.MAC
 6    ;
 7    ; This task prompts the user for an input string and
 8    ; then echos the string to the terminal. It repeats this
 9    ; process until the user types a CTRL/Z.
10    ;
11    ; Assemble and task-build instructions:
12    ;
13    ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[uic]PROMPT
14    ;       LINK/MAP PROMPT,LB:[1,1]PROGSUBS/LIBRARY
15    ;-
16            .MCALL  EXIT$S,QIOW$S      ; System macros
17            .MCALL  DIRERR,IOERR       ; Supplied macros
18    ;
19    PROM:   .ASCII  /Please type anything: / ;Prompt
20    PLEN    =       .-PROM             ; Length of prompt
21    FINMES: .ASCII  /You typed: /      ; Echo prefix
22    FLEN    =       .-FINMES           ; Length of above
23    BUFF:   .BLKB   80.                ; Buffer
24            .EVEN                      ; Move to word boundary
25    IOSTAT: .BLKW   1                  ; I/O status block for
26                                       ;  QIOs.
27    IOLEN:  .BLKW   1                  ; 2nd word of I/O status
28                                       ;  block
29    ;
30    START:  QIOW$S  #IO.RPR,#5,#1,,#IOSTAT,,<#BUFF,#80.,,#PROM,#PLEN,#44>
31                                       ; issue QIO for Read
32                                       ; After Prompt
33            BCS     DERR               ; Branch on dir error
34            TSTB    IOSTAT             ; Check I/O status
35            BLT     IERR               ; Branch on I/O error
36            ADD     #FLEN,IOLEN        ; Add length of prefix
37                                       ;  to that of entered text
38            QIOW$S  #IO.WVB,#5,#1,,#IOSTAT,,<#FINMES,IOLEN,#40>
39                                       ; Write the new message
40            BCS     CERR               ; Branch on dir error
41            TSTB    IOSTAT             ; Check for I/O error
42            BLT     OERR               ; Branch on I/O error
43            BR      START              ; Start over again
44    ; Errors come here
45    DERR:   DIRERR  <Error in QIO to READ AFTER PROMPT>
46                                       ; Use macro to tell of
47    CERR:   DIRERR  <Error in QIO to WRITE> ;  dir error
48    IERR:   CMPB    #IE.EOF,IOSTAT             ; Check for ^Z
49            BNE     JERR                       ; Branch if not,
50                                               ; was I/O error
51            EXIT$S                             ; Normal exit
52    JERR:   IOERR   #IOSTAT,<Error in READ AFTER PROMPT>
53                                                 ; Use macro to
54                                                 ;  tell of
55    OERR:   IOERR   #IOSTAT,<Error in WRITE> ;  I/O error
56            .END    START
```

Example 3-4 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 3-4 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 3-5 uses this function. The following notes are keyed to the example.

1. The .NLIST BEX assembler directive instructs the assembler not to list binary code which takes up more than one line. This saves room in the listing for all the ASCII text. Return to listing binary extensions for the code by using a .LIST BEX assembler directive.

2. 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 thus doesn't move the cursor on the screen at all, precede the text with a carriage return (15(8)) to get the cursor back to the start of the line. If this is not done, the NO LONGER A SECRET WORD message will begin away from the left hand margin, below the : in "SECRET WORD".

3. Write prompting text, then leave cursor at that position for input (since 44(8) is used for vertical format control).

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

5. As in the previous example, add the length of the preceding text to the text typed in to determine the total length of the output message. Here, however, you do the calculation in a register instead of in the IOSB. Since the Read No Echo doesn't echo any characters back, it doesn't move the cursor on the screen. Therefore, precede the text with a carriage return (15(8)) to move the cursor back to the start of the line. Without it, the "NO LONGER A SECRET WORD" message will begin away from the margin, below and after "SECRET WORD: ".

You can combine the write of the prompt and the read into one QIO directive call using a Read After Prompt with the Read No Echo subfunction bit (IO.RPR!TF.RNE). If you want, imbed the carriage control characters in the message.

```
        1            .TITLE  NOECHO
        2            .IDENT  /01/
        3            .ENABL  LC                 ; Enable lower case
        4    ;+
        5    ; FILE NOECHO.MAC
        6    ;
        7    ; This task prompts for input, reads it without echo,
        8    ; displays the input text and exits.
        9    ;
       10    ; Assemble and task-build instructions:
       11    ;
       12    ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[uic]NOECHO
       13    ;       LINK/MAP NOECHO,PROGSUBS/LIBRARY
       14    ;-
       15            .MCALL  EXIT$S,QIOW$C,QIOW$S ; System macros
       16            .MCALL  DIRERR,IOERR       ; Supplied macros
(1)    17            .NLIST  BEX                ; Don't list binary
       18                                       ;  extensions
       19    MES:    .ASCII  /SECRET WORD: /    ; Prompt messege
       20    LEN     =       .-MES              ; Length of prompt
(2) [  21    BUFF:   .ASCII  <15>/NO LONGER A SECRET WORD: /
    [  22                                       ; Preceding remark
    [  23    BLEN    =       .-BUFF             ; Length of Remark
       24    BUF:    .BLKB   80.                ; Input buffer
       25            .EVEN                      ; Word align for IOSB
       26    IOSB:   .WORD   0                  ; IOSB is broken into
       27    LENT:   .WORD   0                  ;  two parts for
       28                                       ;  convenience.
       29            .LIST BEX                  ; List binary extensions
(3)    30    START:  QIOW$C  IO.WVB,5,1,,IOSB,,<MES,LEN,44>  ; Write
       31                                       ;  prompt
       32            BCS     DERR1              ; Branch on dir error
       33            TSTB    IOSB               ; Check for I/O error
       34            BLT     IERR1              ; Branch on I/O error
(4)    35            QIOW$C  IO.RNE,5,1,,IOSB,,<BUF,80.> ; Read Noecho
       36            BCS     DERR2              ; Branch on dir error
       37            TSTB    IOSB               ; Check for I/O error
       38            BLT     IERR2              ; Branch on I/O error
(5) [  39            MOV     LENT,R0            ; Get length of input
    [  40            ADD     #BLEN,R0           ; Add length of remark
       41            QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<#BUFF,R0,#40>
       42                                       ; Write out text
       43            BCS     DERR3              ; Branch on dir error
       44            TSTB    IOSB               ; Check for I/O error
       45            BLT     IERR3              ; Branch on I/O error
       46            EXIT$S                     ; Exit
       47    ; Errors come here
       48    IERR1:  IOERR   #IOSB,<Error on 1st WRITE> ; Display I/O
       49    IERR2:  IOERR   #IOSB,<Error on READ>      ;  message and
       50    IERR3:  IOERR   #IOSB,<Error on 2nd WRITE> ;  exit
       51    DERR1:  DIRERR  <Error in QIO on 1st WRITE> ; Display dir
       52    DERR2:  DIRERR  <Error in QIO on READ>      ;  message and
       53    DERR3:  DIRERR  <Error in QIO on 2nd WRITE> ;  exit
       54            .END    START
```

Example 3-5 Read No Echo (Sheet 1 of 2)

```
Run Session

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

Example 3-5 Read No Echo (Sheet 2 of 2)

Read with Timeout

Example 3-6 is a repeat of Example 3-1, only with a timeout on the read. The following notes are keyed to the example.

1. To invoke the timeout mechanism, TF.TMO is ORed with the read function (IO.RLB). You 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 10 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. On the Run Session - 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 (MRKT$). The timeout with a Mark Time is for the entire input, rather than for the next keystroke.

```
    1           .TITLE  QIOTIM
    2           .IDENT  /01/
    3           .ENABL  LC                  ; Enable lower case
    4   ;+
    5   ; FILE QIOTIM.MAC
    6   ;
    7   ; This task reads a line of text from the terminal,
    8   ; converts all upper case characters to lower case, and
    9   ; prints the converted message back at the terminal. It
   10   ; uses synchronous QIOs, with a timeout on the read.
   11   ;
   12   ;-
   13           .MCALL  QIOW$C,QIOW$S,EXIT$S ; System macros
   14
   15   IOSB:   .BLKW   2                   ; I/O Status Block
   16   BUFF:   .BLKB   80.                 ; Text buffer
   17
   18   START:  CLR     R5                  ; Error Count
   19           CLR     R4                  ; Error indicator - 0
   20                                       ;  means directive error,
   21                                       ;  (DSW in R3), neg means
   22                                       ;  I/O error (I/O status
   23                                       ;  in R3)
(1) 24          QIOW$C  IO.RLB!TF.TMO,5,1,,IOSB,,<BUFF,80.,1>
   25                                       ; Issue read
   26           BCS     ERR1                ; Branch on dir error
   27           TSTB    IOSB                ; Check for I/O error
   28           BLT     ERR1A               ; Branch on I/O error
   29           MOV     IOSB+2,R0           ; Get count of characters
   30                                       ;  typed in
   31           CLR     R1                  ; Offset into buffer to
   32                                       ;  character
   33   LOOP:   CMPB    BUFF(R1),#'A        ; Check for upper case
   34                                       ;  ASCII character
   35           BLT     NEXT                ; Branch if below range
   36           CMPB    BUFF(R1),#'Z
   37           BGT     NEXT                ; Branch if above range
   38   ; It is upper case, so move to register R2 and convert
   39           MOVB    BUFF(R1),R2         ; Move to register
   40           ADD     #32.,R2             ; Convert to lower case
   41           MOVB    R2,BUFF(R1)         ; Replace in message
   42   NEXT:   INC     R1                  ; Increment offset into
   43                                       ;  buffer to next char
   44           SOB     R0,LOOP             ; Decrement count of
   45                                       ;  characters left
   46           QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<#BUFF,IOSB+2,#40>
   47           BCS     ERR2                ; Branch on dir error
```

Example 3-6 Read With Timeout (Sheet 1 of 2)

```
48              TSTB    IOSB            ; Check for I/O error
49              BLT     ERR2A           ; Branch on I/O error
50              EXIT$S
51      ;
52      ; Error code
53      ;
54      ERR2A:  INC     R5              ; R5=2, 2nd QIO
55      ERR1A:  INC     R5              ; R5=1, 1st QIO
56              MOVB    IOSB,R3         ; I/O error. I/O status to R4.
57              DEC     R4              ; Negative value in R4
58                                      ;  means I/O error
59              IOT                     ; Trap and display registers
60      ERR2:   INC     R5              ; R5=2, 2nd QIO
61      ERR1:   INC     R5              ; R5=1, 1st QIO
62              MOV     $DSW,R3         ; Directive error. DSW
63                                      ;  to R3, leave R4=0.
64              IOT                     ; Trap and display registers
65              .END START
```

Run Session

```
  >RUN QIOTIM
  KJHKJJJJ
          kJhkJJJJ
❷ >RUN QIOTIM
  JJJafhkJfiur<RET>
  JJJafhkJfiur
  >
```

Example 3-6 Read With Timeout (Sheet 2 of 2)

Terminal-Independent Cursor Control

Terminal-independent cursor control is a SYSGEN option, provided only if selected. If it is selected, certain I/O requests are automatically converted by the terminal driver for the specific 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 can also of course be done by imbedding the terminal specific escape sequences into the write buffer. However, the advantage of using terminal-independent cursor control, is that the same program will work at different terminals (VT-52's and VT-100's, for example), without any need for modification.

All you need to do is place the proper value in the vertical forms control word of the I/O parameter list. If the high order byte is non-zero, then 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 10., column 25., place a 10. in the high order byte and a 25. in the low order byte. An easy way to do this is to let the assembler convert 10.*256.!25. for you. In general, X*256.!Y corresponds to position X,Y on the screen. In addition, if bit 15 (the most significant bit in the line number byte) is set, the screen is cleared before the cursor is moved.

Example 3-7 demonstrates the use of terminal-independent cursor control. The following notes are keyed to the example.

1. Parameters defined with symbols so that they can easily be changed.

2. Use the $ form of the mark time directive to allow reuse of a single DPB.

3. Issue a mark time directive for one minute to set event flag 3, allowing the task to exit after one minute.

4. Modify the DPB and use it over and over again, at line 34, to mark time for Z seconds before updating the display. The second mark time uses event flag 2, to avoid conflicts. This approach saves task space since the DPB is used again.

5. Issue the Z second mark time directive. We will wait for event flag 2 at line 5Ø. When one second goes by and the flag is set, check for one minute and update the display again if it hasn't yet gone by.

6. Get the time and date parameters in binary.

7. Use the System Library Route $DAT and $TIM to format the date and time for display. See Chapter 6 of the <u>IAS/RSX-11 System Library Routine Reference Manual</u> for documentation on these routines.

8. Calculate the length of the output message by subtracting the starting position in the buffer from the position after the last character in the buffer.

9. Issue the write. X*256!Y places the cursor before the write at line X, column Y. The TF.RCU subfunction bit instructs the terminal driver to save the cursor position before moving it, and then to restore it after writing the message. This allows you to continue typing in commands while the task runs.

10. Wait for z seconds to go by. The mark time directive will cause event flag 2 to be set.

11. Check event flag 3. If it is set, the one minute is up and you should exit. Use Clear Event Flag instead of Read All Event Flags so that the DSW will indicate whether the flag was clear or set before you cleared it. With Read All Event Flags, the settings of flags 1-16 are returned in a word in a buffer. You would then need to test the specific bit to check the flag setting, which is more work.

12. On the Run Session - The display actually will appear at line X, column Y on the screen, and is updated every z seconds.

```
     1          .TITLE  DATTIM
     2          .IDENT  /01/
     3          .ENABL  LC                  ; Enable lower case
     4  ;+
     5  ; FILE DATTIM.MAC
     6  ;
     7  ; This task places the date and the time at line X, column Y and
     8  ; then updates the display every Z seconds for 1 minute.
     9  ;
    10  ; Assemble and Link instructions:
    11  ;
    12  ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]DATTIM
    13  ;       LINK/MAP DATTIM,LB:[1,1]PROGSUBS/LIBRARY
    14  ;-
    15          .MCALL  QIOW$S,MRKT$,WTSE$C,GTIM$C ; External system macros
    16          .MCALL  EXIT$S,CLEF$C,DIR$         ;
    17          .MCALL  DIRERR,IOERR        ; External supplied macros
    18  ; Data
(1) 19          X=5.                        ; Line number
    20          Y=32.                       ; Column number
    21          Z=1.                        ; How often to update (in seconds)
    22
    23  TIMBUF: .BLKW   8.                  ; Buffer for return of system time
    24  TIMMSG: .BLKB   20.                 ; Buffer for creating output message
    25  IOSB:   .BLKW   2                   ; I/O status block
(2) 26  MRKTM:  MRKT$   3,1,3               ; DPB for mark time directive
    27  ; Code
(3) 28  START:  DIR$    #MRKTM              ; Set up to exit after 1 minute
    29          BCS     ERR1                ; Branch on directive error
    30  ; Set up for the other mark time directive
(4) 31          MOV     #2,MRKTM+M.KTEF     ; Change event flag #
    32          MOV     #Z,MRKTM+M.KTMG     ; Change time magnitude
    33          MOV     #2,MRKTM+M.KTUN     ; Change time units
(5) 34  AGAIN:  DIR$    #MRKTM              ; Schedule next update
    35          BCS     ERR2                ; Branch on directive error
(6) 36          GTIM$C  TIMBUF              ; Get system time and date
    37          BCS     ERR3                ; Branch on directive error
(7) 38          MOV     #TIMMSG,R0          ; Set up for call to $DAT
    39          MOV     #TIMBUF,R1          ;
    40          CALL    $DAT                ; Convert date for display
    41          MOVB    #' ,(R0)+           ; Insert space into output message
    42          MOV     #3,R2               ; Set up for call to $TIM, ask
    43                                      ;       for HH:MM:SS format
    44          CALL    $TIM                ; Convert time for display
(8) 45          SUB     #TIMMSG,R0          ; Compute character count
(9) 46          QIOW$S  #IO.WLB!TF.RCU,#5,#1,,#IOSB,,<#TIMMSG,R0,#X*256.!Y>
    47          BCS     ERR4D               ; Branch on directive error
    48          TSTB    IOSB                ; Check for I/O error
    49          BLT     ERR4I               ; Branch on I/O error
(10) 50         WTSE$C  2                   ; Wait for mark time to expire
    51          BCS     ERR5                ; Branch on directive error
```

Example 3-7 Terminal-Independent Cursor Control (Sheet 1 of 2)

```
    52    ; Check for 1 minute gone by
(11) 53          CLEF$C  3                ; Clear event flag to check setting
    54          BCS     ERR6             ; Branch on directive error
    55          CMP     $DSW,#IS.CLR     ; Check for flag already clear
    56          BEQ     AGAIN            ; If still clear, minute not up yet,
    57                                   ;       update display again
    58          EXIT$S                   ; Exit if 1 minute is up
    59    ; Error code
    60    ERR1:   DIRERR  <ERROR ON MARK TIME FOR 1 MINUTE>
    61    ERR2:   DIRERR  <ERROR ON MARK TIME FOR 1 SECOND>
    62    ERR3:   DIRERR  <ERROR ON GET TIME>
    63    ERR4D:  DIRERR  <ERROR ON WRITE>
    64    ERR4I:  IOERR   #IOSB,<ERROR ON WRITE>
    65    ERR5:   DIRERR  <ERROR ON WAIT FOR>
    66    ERR6:   DIRERR  <ERROR ON CLEAR EVENT FLAG>
    67          .END    START
```

Run Session

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

Example 3-7 Terminal-Independent Cursor Control (Sheet 2 of 2)

Formatting Output Data

The subroutine $EDMSG in SYSLIB.OLB provides a generalized output formatting capability for easily creating display messages. It is useful if some of the data is generated at run time. This allows you to combine a number of functions available with individual conversion routines (such as $CBDMG) for converting a single binary word to an ASCII octal string for display. It includes all of the following functions.

- Conversion of internal binary stored data to
  - ASCII signed or unsigned octal
  - ASCII signed or unsigned decimal
  - ASCII alphanumeric characters
- Conversion of time or date data into standard ASCII formats (hh:mm or dd-mmm-yy)
- Formatting of converted characters for display, by themselves or intermixed with other text.

For a complete discussion of the use of $EDMSG, see Chapter 5 of the <u>IAS/RSX-11 System Library Routine Reference Manual</u>.

To invoke $EDMSG, use the following procedure.

1. Set up the output buffer, the format string, and the argument block.
2. Set up the input parameters.

   R0 - starting address of output buffer

   R1 - starting address of format string

   R2 - starting address of argument block, containing the data to be converted
3. Call $EDMSG.

On return, the converted/formatted string is in the output buffer. The output parameters are:

RØ - Address of next available byte in the output buffer

R1 - Length (in bytes) of the output string

R2 - Address of the next argument in the argument block.

NOTE
The output parameters make it possible to concatenate messages using multiple calls to $EDMSG.

The output buffer is a buffer in which $EDMSG generates the output message. It is typically set up using the .BLKB or .BLKW assembler directive. The format string is set up using a combination of ASCII text and editing "directives." It must be in ASCIZ format, meaning that it is terminated by a Ø(8). The editing "directives" are in one of three formats, as follows.

%d - Means perform directive d once

%nd - Means perform directive d n-times

%Vd - Means perform directive d V-times, where V is an argument in the argument block.

For example, if %O means convert binary word to ASCII signed octal, %O means convert the next word in the argument block to ASCII signed octal in the output buffer. %3O means convert the next three words to ASCII signed octal in the output buffer, separated by tabs. %VO means get the binary word in the argument block and convert that many words in the argument block to signed octal in the output buffer.

Table 3-4 shows many of the editing directives available with $EDMSG. An example follows the table.

Table 3-4 Sample Editing Directives for $EDMSG

Conversion Directive	Directives Meaning	Formatting Directive	Directives Meaning
D	Convert binary word in binary block to ASCII signed decimal.	F	Form feed
U	Convert binary word in argument block to ASCII unsigned decimal.	N	Line feed
O	Convert binary word in argument block to ASCII signed octal.	S	Space
P	Convert binary word in argument block to ASCII unsigned octal.		
R	Convert RAD5Ø word in argument block to ASCII.		
Y	Convert three words to dd-mmm-yy format.		
Z	Convert time parameter(s) to hh:mm:ss,s format (or some part - hh only, etc.).		
A	Move the ASCII character at the address pointed to in the argument block.		

Example:

```
FORMAT:   .ASCIZ    /%10$NAME IS %5A AND # IS %D/
          .EVEN
OUTBUF:   .BLKW     80.
DATA:     .WORD     ADRNAM
          .WORD     234
ADRNAM:   .ASCII    /BILLY/
          .EVEN
             .
             .
             .
          MOV       #OUTBUF, R0
          MOV       #FORMAT, R1
          MOV       #DATA, R2
          CALL      $EDMSG
```

The resulting string in OUTBUF would display as:

```
NAME IS BILLY AND # IS 156
```

Explanation:

%10S in the format string - Produces 10 spaces in the output buffer.

NAME IS - Placed in the buffer as is.

%5A - Get five bytes and convert to ASCII. Because the argument block is set up on a word-by-word basis, place the address of the ASCII characters (ADRNAM), instead of the ASCII characters themselves, in the argument block.

AND # IS - Moved to the output buffer as is.

%D - Get the next binary word in the argument block and convert it to signed decimal in the output block. 234(8) is converted to +156(10).

No decimal point is appended to decimal numbers unless you specify %D. (including the ".") in the format string.

Three examples follow which demonstrate the use of the $EDMSG routine.

Examples of Formatting Numeric Data

Example 3-8 shows the use of $EDMSG for formatting numeric data. The following notes are keyed to the example.

1. This is the argument block, which must be a set of contiguous words.

2. This example uses the $ form of the QIO directive. The length of the buffer to be written out will be filled in at run time.

3. The output buffer starts at BUF and is 80. bytes long. This buffer should be long enough for at least the longest message that you might generate.

4. The format string. Note that three words will be converted to signed decimal ASCII using $EDMSG. The .ASCIZ assembler directive assures that the format string ends with an octal 0.

5. Set up input parameters for call to $EDMSG. The addends and the sum are already in the argument block.

6. Invoke $EDMSG. The output string is returned at BUF. R1 contains the count of characters in the output string.

7. Move the count of characters to be written into the DPB of the QIO$ directive.

8. Write the results out at the terminal.

Normally, the addends might be placed in the format string if they are known at assembly time. Only data generated at run time would be converted using $EDMSG.

```
 1            .TITLE  NUMER
 2            .IDENT  /01/
 3            .ENABL  LC                 ; Enable lower case
 4    ;+
 5    ; FILE NUMER.MAC
 6    ;
 7    ; This task does a simple addition and outputs the
 8    ; results. It demonstrates the use of $EDMSG for
 9    ; formatting messages with numeric data
10    ;-
11            .MCALL  QIOW$,EXIT$S,DIR$  ; System macros
12            .NLIST  BEX                ; Do not list binary
13                                       ;  extensions
14    ; Data
15    A:      .WORD   10                 ; 1st addend and start
16                                       ;  of argument block
17    B:      .WORD   22                 ; 2nd addend
18    C:      .BLKW   1                  ; Location for sum
19    ;
20    OUT:    QIOW$   IO.WVB,5,1,,IOSB,,<BUF,,40> ;QIO for
21                                       ;  output message
22    IOSB:   .BLKW   2                  ; I/O status block
23    ;
24    ; Set up for $EDMSG
25    ;
26    BUF:    .BLKB   80.                ; Output buffer
27    FMES:   .ASCIZ  /%D. WAS ADDED TO %D., GIVING %D./
28                                       ; Format string
29
30            .LIST   BEX                ; List binary extensions
31            .EVEN                      ; Move to word boundary
32    START:  MOV     A,C                ; Move 1st addend to sum
33                                       ;  word
34            ADD     B,C                ; Add 2nd addend to form
35                                       ;  sum
36    ; Set up for call to $EDMSG
37            MOV     #BUF,R0            ; Addr of output buffer
38            MOV     #FMES,R1           ; Addr of format string
39            MOV     #A,R2              ; Addr of argument block
40            CALL    $EDMSG             ; Make call, character
41                                       ; count returned in R1
42            MOV     R1,OUT+Q.IOPL+2    ; Place # of characters
43                                       ;  to write into IOPL
44                                       ;  in QIO DPB
45            DIR$    #OUT               ; Write output message
46            BCS     ERR1D              ; Branch on dir error
47            TSTB    IOSB               ; Check for I/O error
48            BLT     ERR1I              ; Branch on I/O error
49            EXIT$S
```

(Callouts: 1 = lines 15–18; 2 = line 20; 3 = line 26; 4 = line 27; 5 = lines 37–39; 6 = line 40; 7 = line 42; 8 = line 45)

Example 3-8 Formatting Numeric Data (Sheet 1 of 2)

```
50    ; Error handling
51    ERR1D:  MOV     $DSW,R0         ; Move DSW for display
52            CLR     R1              ; Indicate dir error, by
53                                    ;  0 in R1
54            IOT
55    ERR1I:  MOVB    IOSB,R0         ; Move I/O status for
56                                    ;  display
57            MOV     #-1,R1          ; Indicate I/O error by
58                                    ;  -1 in R0
59            IOT
60            .END    START
```

Run Session

```
>RUN NUMER
8. WAS ADDED TO 18., GIVING 26.
>
```

Example 3-8 Formatting Numeric Data (Sheet 2 of 2)

Example 3-9 shows how to use $EDMSG to generate error messages for display. This is a modification of Example 3-1 (SYNCHQ.MAC). These error routines will typically replace trap routines which might be used early in the debugging stage of an application. The supplied macros DIRERR and IOERR have performed similar functions for you. The following notes are keyed to the example.

❶ This is the assembly time setup for $EDMSG. ARG is the start of the one word argument block. EBUFF is the start of the buffer in which error messages are to be built. FMT1, FMT1A, FMT2, FMT2A are the format strings for the various error messages. FMT1 and FMT2 are for directive errors; FMT1A and FMT2A are for I/O errors. The quotation marks are used as delimiters in two of the format strings because the strings contain slashes (/).

❷ The main code is the same as before. Only the error handling is different.

❸ For each error, move the address of the appropriate format string into R1 (for the call to $EDMSG). Then move the DSW into the argument block for directive errors, and the I/O status into the argument block for I/O errors. Because the I/O status is a byte, move it to R1 first and then to the argument block, in order to extend the sign bit to the high order byte (see lines 0064 and 0065). Then branch to the final setup for $EDMSG at EDAWT.

4. Move the address of the output buffer to RØ and of the argument block to R2. Then call $EDMSG.

5. Finally, write the formatted message out at the terminal and exit.

6. On the Run Session - The first run shows a successful read. The second run shows an error caused by a ^Z.

```
 1          .TITLE SYNQER
 2          .IDENT  /01/
 3          .ENABL  LC                    ; Enable lower case
 4  ;+
 5  ; FILE SYNQER.MAC
 6  ;
 7  ; This task reads a line of text from the terminal,
 8  ; converts all upper case characters to lower case, and
 9  ; prints the converted message back at the terminal. It
10  ; uses synchronous QIO. It also uses $EDMSG to generate
11  ; error messages
12  ;-
13          .MCALL  QIOW$C,QIOW$S,EXIT$S ; System macros
14
15  IOSB:   .BLKW   2                     ; I/O Status Block
16  BUFF:   .BLKB   80.                   ; Text buffer
17
18  ; Set up for error messages using $EDMSG
19
20  ARG:    .BLKW   1                     ; Argument block
21  EBUFF:  .BLKB   80.                   ; Output buffer
22  FMT1:   .ASCIZ  /DIRECTIVE ERROR ON READ, DSW = %D/
23  FMT1A:  .ASCIZ  'I/O ERROR ON READ, I/O STATUS = %D'
24  FMT2:   .ASCIZ  /DIRECTIVE ERROR ON WRITE, DSW = %D/
25  FMT2A:  .ASCIZ  'I/O ERROR ON WRITE, I/O STATUS = %D'
26          .EVEN
27
28  START:  QIOW$C  IO.RVB,5,1,,IOSB,,<BUFF,80.,40> ; Issue
29                                        ;  read
30          BCS     ERR1                  ; Branch on dir error
31          TSTB    IOSB                  ; Check for I/O error
32          BLT     ERR1A                 ; Branch on I/O error
33          MOV     IOSB+2,R0             ; Get character count
34          CLR     R1                    ; Offset into buffer to
35                                        ;  character
36  LOOP:   CMPB    BUFF(R1),#'A          ; Check for upper case
37                                        ;  ASCII character
38          BLT     NEXT                  ; Branch if below range
39          CMPB    BUFF(R1),#'Z
40          BGT     NEXT                  ; Branch if above range
41  ; Here if upper case, move to register R2 and convert
42          MOVB    BUFF(R1),R2           ; Move to register
43          ADD     #32.,R2               ; Convert to lower case
44          MOVB    R2,BUFF(R1)           ; Replace in message
45  NEXT:   INC     R1                    ; Increment offset into
46                                        ;  buffer to next char
47          SOB     R0,LOOP               ; Decrement count of
48                                        ;  chars left to check
49          QIOW$S  #IO.WVB,#5,#1,,#IOSB,,<#BUFF,IOSB+2,#40>
50                                        ; Write text
51          BCS     ERR2                  ; Branch on dir error
52          TSTB    IOSB                  ; Check for I/O error
53          BLT     ERR2A                 ; Branch on I/O error
54          EXIT$S                        ; Exit
```

(Callout 1 brackets lines 20–25; callout 2 brackets lines 28–54.)

Example 3-9 Formatting Directive and I/O Error Messages (Sheet 1 of 2)

```
   55    ;
   56    ; Error code
   57    ;
  [58    ERR1A:  MOV     #FMT1A,R1          ; Format string for
  |59                                       ;  1st I/O error message
  |60            BR      ERRGOA             ; Branch to common I/O
  |61                                       ;  error code
  |62    ERR2A:  MOV     #FMT2A,R1          ; Format string for 2nd
  |63                                       ;  I/O error message
  |64    ERRGOA: MOVB    IOSB,R0            ; Extend sign on I/O
  |65            MOV     R0,ARG             ;  status byte by moving
  |66                                       ;  it through R0 to the
3 |67                                       ;  argument block
  |68            BR      EDAWT              ; Branch to common edit
  |69                                       ;  and write code
  |70    ERR1:   MOV     #FMT1,R1           ; Format string for 1st
  |71                                       ;  directive error
  |72            BR      ERRGO              ; Branch to common
  |73                                       ;  directive error code
  |74    ERR2:   MOV     #FMT2,R1           ; Format string for 2nd
  |75                                       ;  directive error
  [76    ERRGO:  MOV     #$DSW,ARG          ; Move DSW to arg block
   77    ; Finish setting up for $EDMSG
  [78    EDAWT:  MOV     #EBUFF,R0          ; Output buffer address
4 |79            MOV     #ARG,R2            ; Argument block address
  [80            CALL    $EDMSG             ; Edit output string
5  81            QIOW$S  #IO.WVB,#5,#1,,,,<#EBUFF,R1,#40> ; Write
   82                                       ;  out message
   83            EXIT$S                     ; Exit
   84            .END START
```

Run Session

```
>RUN SYNQER
SKJDSHKJKHkJhkJhkJhkyutduyeJherwerJwl12
skJdshkJkhkJhkJhkJhkyutduyeJherwerJwl12
>RUN SYNQER
dhfiooiJKLHJGHJGJHG^Z
I/O ERROR ON READ, I/O STATUS = -10
```

Example 3-9 Formatting Directive and I/O Error Messages (Sheet 2 of 2)

Formatting ASCII Data

Example 3-9 demonstrates the use of $EDMSG for formatting ASCII data. The only real difference between formatting ASCII data as compared to numeric data is that the argument block contains a pointer to the ASCII characters, rather than to the ASCII data itself. The following notes are keyed to the example.

1. The argument block contains four words. Only the address of the number to be typed is known at assembly time. The other values will be filled in at run time. In the format string, we are using %VA twice. The V tells $EDMSG to use the next word in the argument block as the number of times to perform the directive A. The directive A means move an ASCII character to the output buffer. This allows you to generate messages of different lengths at run time using the same format string.

2. An alternative to using TSTB is to use a CMPB instruction. IS.SUC is a global symbol equal to +1.

3. The number typed is in ASCII. We need to convert to binary before dividing by two.

4. Come here if the number is even. Place the address of the message and its length into the argument block. Then branch to the common code to generate the message.

5. This is the same as in note 4, but for an odd number.

6. Move the number of digits in the number entered by the operator to the argument block; so you display that many digits.

7. Now set up for $EDMSG, format the output message, write it, and exit.

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 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.

```
 1            .TITLE   FORMAT
 2            .IDENT   /01/
 3            .ENABL   LC                  ; Enable lower case
 4   ;+
 5   ; FILE FORMAT.MAC
 6   ;
 7   ; This task asks the user for an integer. It then
 8   ; decides whether the value is even or odd, and prints
 9   ; an appropriate message. It demonstrates the use of
10   ; $EDMSG for ASCII data
11   ;
12   ; Assemble and task-build instructions:
13   ;
14   ;        MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]FORMAT
15   ;        LINK/MAP FORMAT,LB:[1,1]PROGSUBS/LIBRARY
16   ;-
17            .MCALL   EXIT$S,QIOW$C,QIOW$S ; System macros
18            .MCALL   DIRERR,IOERR        ; Supplied macros
19
20   MES:     .ASCII   /INPUT A DECIMAL INTEGER BETWEEN 1 AND 9999/
21                                         ; Prompt text
22   LEN      =        .-MES               ; Length of prompt text
23            .EVEN
24   NUM:     .BLKB    4                   ; Buffer for ASCII # input
25            .EVEN
26   IOSB:    .WORD    0                   ; I/O Status Block
27   NUMB:    .WORD    0                   ; 2nd word of I/O Status
28                                         ;  Block - for return of
29                                         ;  # of characters read
30   ; Setup for $EDMSG
31   ARGBLK:
32   LNUM:    .WORD    0                   ; Count of characters
33                                         ;  in number
34   ANUM:    .WORD    NUM                 ; Pointer to number in ASCII
35   LWORD:   .WORD    0                   ; Count of characters
36                                         ;  in ODD or EVEN
37   AWORD:   .WORD    0                   ; Pointer to ODD or EVEN
38                                         ;  in ASCII
39   BUF:     .BLKB    80.                 ; Output buffer
40   OUT:     .ASCIZ   /%NTHE NUMBER %VA is %VA./ ; Format string
41   MESE:    .ASCII   /EVEN/              ; ASCII text for EVEN
42   LMESE    =.-MESE                      ; Length of message
43   MESO:    .ASCII   /ODD/               ; ASCII text for ODD
44   LMESO    =.-MESO                      ; Length of message
45            .EVEN
46   ;
47   START:   QIOW$C   IO.WVB,5,1,,IOSB,,<MES,LEN,40>   ; Write
48                                         ;  prompt text
49            BCS      ERR1D               ; Branch on dir error
50            CMPB     #IS.SUC,IOSB        ; Check I/O Status
51            BNE      ERR1I               ; Branch on I/O error
52            QIOW$C   IO.RVB,5,1,,IOSB,,<NUM,4> ; Read input
53            BCS      ERR2D               ; Branch on dir error
54            TSTB     IOSB                ; Check on I/O error
55            BLT      ERR2I               ; Branch on I/O error
```

Example 3-10 Formatting ASCII Data (Sheet 1 of 2)

```
3  56             MOV      #NUM,R0            ; Set up to convert
   57                                         ;  dec ASCII to binary
   58             CALL     $CDTB              ; Convert, result in R1
   59    ; Set up for divide. Dividend in R0,R1 combined
   60             CLR      R0                 ; Clear high order 16 bits
   61             DIV      #2,R0              ; Divide, quotient in R0,
   62                                         ;  remainder in R1
   63             CMP      R1,#0              ; Check remainder
   64             BNE      ODD                ; Branch if not 0
4  65             MOV      #LMESE,LWORD       ; Move length of EVEN
   66                                         ;  into argument block
   67             MOV      #MESE,AWORD        ; Move pointer to EVEN
   68                                         ;  into argument block
   69             BR       CONT               ; Branch to common code
5  70    ODD:     MOV      #LMESO,LWORD       ; Move length of ODD
   71                                         ;  into argument block
   72             MOV      #MESO,AWORD        ; Move pointer to ODD
   73                                         ;  into argument block
   74    CONT:    MOV      NUMB,LNUM          ; Move # of characters
   75                                         ;  in number to arg block
7  76             MOV      #BUF,R0            ; Set up for call to $EDMSG
   77             MOV      #OUT,R1
   78             MOV      #ARGBLK,R2
   79             CALL     $EDMSG             ; Edit output message
   80             QIOW$S   #IO.WVB,#5,#1,,#IOSB,,<#BUF,R1,#40>
   81                                         ; Write output message
   82             BCS      ERR3D              ; Branch on dir error
   83             CMPB     #IS.SUC,IOSB       ; Check for I/O error
   84             BNE      ERR3I              ; Branch on I/O error
   85             EXIT$S                      ; Exit
   86    ; Error handling
   87    ERR1D:   DIRERR   <ERROR ON WRITE OF PROMPT TEXT>
   88    ERR1I:   IOERR    #IOSB,<ERROR ON WRITE OF PROMPT TEXT>
   89    ERR2D:   DIRERR   <ERROR ON READ>
   90    ERR2I:   IOERR    #IOSB,<ERROR ON READ>
   91    ERR3D:   DIRERR   <ERROR OUTPUTTING ANSWER>
   92    ERR3I:   IOERR    #IOSB,<ERROR OUTPUTTING ANSWER>
   93             .END     START
```

Run Session

```
>RUN FORMAT
INPUT A DECIMAL INTEGER BETWEEN 1 AND 9999
600

THE NUMBER 600 IS EVEN.
>RUN FORMAT
INPUT A DECIMAL INTEGER BETWEEN 1 AND 9999
2349

THE NUMBER 2349 IS ODD.
>
```

Example 3-10 Formatting ASCII Data (Sheet 2 of 2)

USING DIRECTIVES FOR INTERTASK COMMUNICATION

4

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

1. To use directives which control task execution to synchronize cooperating tasks
2. To use the send/receive directives to pass data between tasks.
3. 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. Using 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 example, that you need to simulate customer transactions at a bank. There are five windows, up to 15 customers can physically stand in each line at a time, given the size of the waiting area. You 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 types of transactions to different lines) can easily be simulated by simply modifying the task that simulates that line.

An RSX-11M programmer typically uses a mix of the following four multitasking methods.

1. Common or Group Global event flags, together with synchronization and task scheduling directives, are used to synchronize tasks.

2. Resident commons are used to share data in memory.

3. Memory management directives are used to create and/or share data areas dynamically at run time.

4. File handling routines are used to open disk files for shared access.

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

Directives

Table 4-1 lists the various task control directives which are available for task synchronization. (Most of these 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 in a stopped state which effectively lowers the task priority to zero, 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 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. As discussed in Module 2, the Clear Event Flag directive (CLEF$) can be used instead of the Read All Event Flags (RDAF$) or the Read Extended Event Flags (RDXF$) directives, to check whether a single event flag is set (since the DSW indicates whether the flag was initially clear or set). This saves having to check the specific bits in the event flag mask word. Checking specific bits is necessary with RDAF$ and RDXF$ because they return several event flag mask words, each containing the settings of 16 flags, one flag per bit.

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

Directive	Example of Use for Synchronization
MACRO	
RQST$ RUN$	Issuing task activates target task. Target task then performs some operation for issuing task.
ABRT$	Issuing task aborts target task.
SPND$ 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.
RSUM$ USTP$	A task resumes or unstops another task which has suspended or stopped itself while waiting for that task 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 - Operator use of a DCL CONTINUE command (RESUME in MCR). A task can be unstopped by: - Its own AST routine - Operator use of 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 zero.	Priority remains unchanged.
Task can be checkpointed by any other task (if checkpointable).	Task can be checkpointed only by tasks of higher priority.
Chance of being checkpointed increases.	Chance 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
MACRO	
CLEF$	A task clears the event flag, then waits for it to be set by another task.
SETF$	A task performing an operation for another task sets an event flag to signify completion of the operation.
WTSE$ STSE$	A task waits for completion of an operation by another task by waiting or stopping for that task to set an event flag.
STLO$ WTLO$	A task waits or stops for the completion of the first of some set of operations.
RDAF$ RDXF$	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 (RQST$), Suspend (SPND$), and Resume (RSUM$) directives for synchronization. Notes 1, 2, 3 and 6 refer to TASKA. Notes 4 and 5 refer to TASKB.

1. The supplied macros are used to allow you to concentrate on the synchronization techniques. If you want, these macros can be replaced with QIO's and other code.

2. 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.

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

4. TASKB types out a message and then resumes TASKB. 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.

5. TASKB displays another message and then exits.

6. 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), steps 5 and 6 may be reversed on the run session.

```
    1           .TITLE  TASKA
    2           .IDENT  /01/
    3           .ENABL  LC                  ; Enable lower case
    4   ;
    5   ; FILE TASKA.MAC
    6   ;
    7   ; This task requests TASKB to run, and then suspends
    8   ; itself. TASKB resumes this task and exits.
    9   ;
   10   ; Assemble and task-build instructions:
   11   ;
   12   ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]TASKA
   13   ;       LINK/MAP LB:[1,1]TASKA,PROGSUBS/LIBRARY
   14   ;
   15   ; Install and run instructions: Both tasks must be
   16   ; installed. Just run TASKA.
   17   ;
   18           .MCALL  RQST$C,SPND$S,EXIT$S ; System macros
(1) 19          .MCALL  TYPE,DIRERR         ; Supplied macros
   20   ;
   21   START:  TYPE    <TASKA BEGINS AND REQUESTS TASKB>
   22                                       ; Display message
(2) 23          RQST$C  TASKB               ; Request TASKB
   24           BCC     OK1                 ; Branch on directive ok
   25           DIRERR  <TASKA UNABLE TO REQUEST TASKB> ; Display
   26                                       ; error message and exit
   27   OK1:    TYPE    <TASKA IS SUSPENDING ITSELF> ; Display
   28                                                ;  message
(3) 29          SPND$S                      ; Suspend self
   30           BCC     OK2                 ; Branch on directive ok
   31           DIRERR  <TASKA UNABLE TO SUSPEND> ; Display
   32                                       ;  error message and exit
(4) 33  OK2:    TYPE    <TASKA HAS BEEN RESUMED> ; Display
   34                                            ;  message
   35           EXIT$S                      ; Exit
   36           .END    START
```

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

```
 1            .TITLE  TASKB
 2            .IDENT  /01/
 3            .ENABL  LC                 ; Enable lowercase
 4      ;
 5      ; FILE TASKB.MAC
 6      ;
 7      ; This task is activated by TASKA.  It performs its
 8      ; operation and resumes TASKA, which has suspended
 9      ; itself.
10      ;
11      ; Assemble and link instructions:
12      ;
13      ;         MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]TASKB
14      ;         LINK/MAP TASKB,LB:[1,1]PROGSUBS/LIBRARY
15      ;
16      ; Install and run instructions: Both tasks must be
17      ; installed. Just run TASKA.
18      ;
19            .MCALL  RSUM$C,EXIT$S      ; System macros
20            .MCALL  TYPE,DIRERR        ; Supplied macros
21      ; Must enable local symbol blocks because we use local
22      ; symbols and DIRERR has .PSECT statements
23            .ENABL  LSB                ; Enable local symbol
24                                       ;  block.
25      ;
26      ; Any operation could be performed here, but in this
27      ; case it's only a typeout.
28 (4)  START:  TYPE    <TASKB IS ALIVE AND RUNNING> ; Display
29 (4)                                                ;  message
30 (4)          RSUM$C  TASKA              ; Resume TASKA
31              BCC     1$                 ; Branch on directive ok
32              DIRERR  <TASKB UNABLE TO RESUME TASKA> ; Display
33                                         ;  error message and exit
34 (5)  1$:     TYPE    <TASKB HAS RESUMED TASKA AND IS EXITING>
35 (5)                                     ; Display message
36 (5)          EXIT$S                     ; Exit
37              .END    START
```

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, the Stop For Single Event Flag is used rather than the Wait For Single Event Flag. The difference between them 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. Notes 1, 2, 3 and 6 refer to TASKC. Notes 4 and 5 refer to 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. Specifically, steps 3 and 4 above may be reversed, as well as 5 and 6 .

The event flag must be common or group global, not local. 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 scans the Active Task List and schedules tasks for CPU time only after a significant event. Setting an event flag does not 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 begin executing sooner than that, TASKD should issue the Declare Significant Event directive (DECL$), which causes the Executive to reschedule tasks. For a discussion of significant events, see Chapter 2 of the RSX-11M/M-PLUS Executive Reference Manual.

```
      1          .TITLE  TASKC
      2          .IDENT  /01/
      3          .ENABL  LC                 ; Enable lower case
      4   ;+
      5   ; FILE TASKC.MAC
      6   ;
      7   ; This task clears an event flag, requests TASKD to run,
      8   ; and then stops until the flag is set by TASKD.
      9   ;
     10   ; Assemble and task-build instructions:
     11   ;
     12   ;         MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]TASKC
     13   ;         LINK/MAP TASKC,LB:[1,1]PROGSUBS/LIBRARY
     14   ;
     15   ; Install and run instructions: TASKD must be installed.
     16   ; Just run TASKC.
     17   ;-
     18          .MCALL  CLEF$C,RQST$C,STSE$C,EXIT$S ; System macros
     19          .MCALL  TYPE,DIRERR        ; Supplied macros
     20   ;
     21   FLAG=33.                          ; Event flag to be used
     22   ;
     23   START: TYPE    <TASKC BEGINS AND REQUESTS TASKD>
     24                                     ; Display message
 (1) 25          CLEF$C  FLAG               ; Clear event flag
     26                                     ;  before stopping
     27          BCC     OK1                ; Branch on directive ok
     28          DIRERR  <TASKC UNABLE TO INITIALIZE EVENT FLAG>
     29                                     ; Display error message
     30                                     ;  and exit
 (2) 31   OK1:   RQST$C  TASKD              ; Request TASKD
     32          BCC     OK2                ; Branch on directive ok
     33          DIRERR  <TASKC UNABLE TO REQUEST TASKD> ; Display
     34                                     ;  error message and exit
     35   OK2:   TYPE    <TASKC IS STOPPING FOR EVENT FLAG>
     36                                     ; Display message
 (3) 37          STSE$C  FLAG               ; Stop for event flag
     38                                     ;  to be set
     39          BCC     OK3                ; Branch on directive ok
     40          DIRERR  <TASKC'S STOP REQUEST REJECTED> ; Display
     41                                     ;  error message and exit
    [42   OK3:   TYPE    <TASKC HAS BEEN UNSTOPPED AND WILL NOW EXIT>
 (6)|43                                     ; Display message
    [44          EXIT$S                     ; Exit
     45          .END    START
```

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

```
     1              .TITLE   TASKD
     2              .IDENT   /01/
     3              .ENABL   LC                 ; Enable lower case
     4      ;+
     5      ; FILE TASKD.MAC
     6      ;
     7      ; This task is activated by TASKC.  It sets the flag for
     8      ; which TASKC is stopped.
     9      ;
    10      ; Assemble and task-build instructions:
    11      ;
    12      ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:,[ufd]TASKD
    13      ;       LINK/MAP TASKD,LB:[1,1]PROGSUBS/LIBRARY
    14      ;-
    15              .MCALL   SETF$C,EXIT$S      ; System macros
    16              .MCALL   TYPE,DIRERR        ; Supplied macros
    17      ;
    18      FLAG=33.                            ; Event flag
    19      ;
    20      ;
    21      ; Any operation could be performed here, but in this
    22      ; case it's only a typeout.
   [23      START:  TYPE     <TASKD IS ALIVE AND RUNNING> ; Display
(4) 24                                                    ;  message
   [25              SETF$C   FLAG               ; Set the flag to allow
    26                                          ;  TASKC to be unblocked
    27              BCC      OK                 ; Branch on directive ok
    28              DIRERR   <TASKD UNABLE TO SET EVENT FLAG>
    29                                          ; Display error message
    30                                          ;  and exit
   [31      OK:     TYPE     <TASKD HAS SET THE EVENT FLAG AND IS EXITING>
(5) 32                                          ; Display message
   [33              EXIT$S                      ; Exit
    34              .END     START
    35
```

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 differences among the Receive Data directives concern what happens if there are no data packets in the receiver's receive queue.

All receive directives receive 15(10) words, including the sender task name (in Radix-50 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	SDAT$	Sends a 13(10) word buffer to receiver. Event flag (if used) set when packet queued to receiver.
Receive Data	RCVD$	Error if no data packets queued.
Receive Data or Exit	RCVX$	Exit if no data packets queued.
Receive Data or Stop	RCST$	Stop if no data packets queued.

Synchronizing Send Requests with Receive Requests

You can use event flags 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. Therefore, 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.

You can also use an AST for synchronization. To request entry into an AST routine whenever a data packet is received, use the Specify Receive Data AST directive (SRDA$). Typically, this directive is issued at the beginning of task execution. From that point on, the AST routine is entered when the first data packet has been placed in the task's receive queue. Only one receive data AST is queued, even if more than one data packet is received at a time. Therefore, you should keep receiving until you get a no data packets queued error to ensure that you have received all of the data packets in the queue.

After the run, use ASTX$S to exit from the AST routine. After exiting from the AST routine, the AST routine will be entered again if a new data packet is received. This continues until the task exits, or until receive data AST's are canceled, using the Specify Receive Data AST directive (SRDA$) with no AST routine specified. It is also possible to temporarily disable all AST recognition using the Disable AST Recognition directive (DSAR$).

In addition, you can use the task control directives for synchronization. Table 4-5 summarizes the various synchronization techniques which might be used. Keep in mind that a Receive Data directive (RCVD$) causes an error condition directive, which is inconsistent with task state (DSW = -8, IE.ITS) if there is no data packet in the receive queue. Receive Data or Stop (RCST$) and Receive Data or Exit (RCVX$), on the other hand, cause the task to stop or exit, respectively, if there is no data queued. For further information about possible synchronization problems, see the writeup on the Receive Data directive (RCVD$) 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 receiver suspends or stops.

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 notes below are keyed to Example 4-3. Note 1 refers to SEND1 and RECV1. Notes 5, 6 and 7 refer to SEND1. Notes 2, 3, 4 and 8 refer to 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. 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. Initialize the message counter. You 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. Skip the first two words (four bytes) of the buffer, which contain the name of the sender task in Radix-50 format.

10. Decrement the message counter. Branch back to clear the event flag and receive again if you have not yet received three messages. If you have, display a message and exit.

```
      1              .TITLE   RECV1
      2              .IDENT   /01/
      3              .ENABL   LC                  ; Enable lower case
      4      ;+
      5      ; FILE RECV1.MAC
      6      ;
      7      ; This task receives data from any sender task
      8      ; (e.g., SEND1). It prints the data on TI:.  Then it
      9      ; waits for another data packet. It does this until it
     10      ; has received 3 messages and then exits.
     11      ;
     12      ; This task synchronizes with its sender through an
     13      ; event flag.
     14      ;
     15      ; Assemble and task-build instructions:
     16      ;
     17      ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]RECV1
     18      ;       LINK/MAP RECV1,LB:[1,1]PROGSUBS/LIBRARY
     19      ;
(1) [20      ; Install and run instructions: RECV1 must be installed
    [21      ; and run before running SEND1.
     22      ;-
     23              .MCALL   CLEF$C,WTSE$C,RCVD$C,EXIT$S ; System macros
     24              .MCALL   TYPE,DIRERR         ; Supplied macros
     25      ;
     26      EFN = 33.                            ; Event flag
     27      ;
     28      RBUFF:  .BLKW    15.                 ; Receive buffer
     29      ;
     30              .ENABL   LSB                 ; Enable local symbol
     31                                           ;  blocks
     32      ;
(2)  33      START:  MOV      #3,R5               ; Initialize message
     34                                           ;  counter
(3)  35      AGAIN:  CLEF$C   EFN                 ; Initialize
     36                                           ;  synchronizing flag
     37              BCC      WAIT                ; Branch on directive ok
     38              DIRERR   <ERROR INITIALIZING FLAG> ; Display
     39                                           ;  error message and exit
(4)  40      WAIT:   WTSE$C   EFN                 ; Wait for a send
     41              BCC      3$                  ; Branch on directive ok
     42              DIRERR   <WAIT DIRECTIVE FAILED> ; Display error
     43                                           ;  message and exit
     44      ; We get here when the flag is set
(8)  45      3$:     RCVD$C   ,RBUFF              ; Receive from anyone
     46              BCC      5$                  ; Branch on directive ok
```

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

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. Therefore, if SEND1 sent four messages before RECV1 was run, the fourth message would be lost.

If you want 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. The process 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 you 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. Use a flag word (BEFORE) to distinguish whether you 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 to see if the event flag is 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 yet been run, 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 you started. Clear the BEFORE flag, because a Receive failure after the flag is set again is a fatal error.

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

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

```
 1              .TITLE   RECV1S
 2              .IDENT   /01/
 3              .ENABL   LC                  ; Enable lower case
 4      ;+
 5      ; FILE RECV1S.MAC
 6      ;
 7      ; This task receives data from any sender task
 8      ; (e.g. SEND1). It prints the data on TI:.  Then it
 9      ; waits for another data packet. It exits after
10      ; receiving and displaying 3 messages.
11      ;
12      ; This task synchronizes with its sender through an
13      ; event flag. Because of this synchronization, and the
14      ; care we take on startup to get messages already
15      ; sent, the tasks can be run in any order, with any
16      ; relative priorities.
17      ;
18      ; Assemble and task-build instructions:
19      ;
20      ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]RECV1S
21      ;       LINK/MAP RECV1S,LB:[1,1]PROGSUBS/LIBRARY
22      ;
23      ; Install and run instructions: RECV1S must be installed
24      ; under the name RECV1 to work with SEND1.
25      ;-
26              .MCALL   CLEF$C,WTSE$C,RCVD$C,EXIT$S ; System macros
27              .MCALL   TYPE,DIRERR         ; Supplied macros
28      ;
29      EFN = 33.                            ; Event flag
30      ;
31      ; "Before" flag, used to keep track of whether we are
32      ; receiving messages sent before RECV1 started up. If
33      ; the event flag is set at startup time, keep receiving
34      ; until we get a failure. We then wait until the flag is
35      ; set to receive again. 1 means receiving messages sent
36      ; before startup, 0 means finished receiving them.
37      BEFORE: .WORD    1                   ; Assume there are messages
38      RBUFF:  .BLKW    15.                 ; Receive buffer
39      ;
40              .ENABL   LSB                 ; Enable local symbol blocks
41      ;
42      START:  MOV      #3,R5               ; Message counter
43              CLEF$C   EFN                 ; Initialize synchronizing
44                                           ;  flag
45              BCC      1$                  ; Branch on directive ok
46              DIRERR   <ERROR INITIALIZING FLAG> ; Display
47                                           ;  error message and exit
```

Example 4-4 A Receiving Task Which Can be Run Before or After the Sender (Sheet 1 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. The technique will be explained first as it applies to the case of running RECV2, before you run SEND2. A discussion of the other possibilities will follow. Notes 2, 3 and 4 refer to SEND2. Notes 1, 5, 6, 7, 8 and 9 refer to RECV2.

1. Issue a Receive Data or Stop directive. If there is no data packet queued, RECV2 stops and must be unstopped by SEND2. If, on the other hand, there is a data packet queued, you would 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, stop.

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

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.

```
     1              .TITLE  RECV2
     2              .IDENT  /01/
     3              .ENABL  LC                 ; Enable lower case
     4      ;
     5      ; FILE RECV2.MAC
     6      ;
     7      ; This task receives data from another task. It prints
     8      ; the data, along with a header, on TI:.  Then it waits
     9      ; for another data packet, continuing this until it has
    10      ; received 3 messages.
    11      ;
    12      ; This task synchronizes with its sender using RCST$.
    13      ; Because of this synchronization, the tasks can be run
    14      ; in any order, with any relative priorities.
    15      ;
    16      ; Assemble and task build instructions:
    17      ;
    18      ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]RECV2
    19      ;       LINK/MAP RECV2,LB:[1,1]PROGSUBS/LIBRARY
    20      ;
    21      ; Install and run instructions: RECV2 must be installed.
    22      ;
    23              .MCALL  RCST$C,RCVD$C,EXIT$S ; System macros
    24              .MCALL  TYPE,DIRERR        ; Supplied macros
    25      ;
    26      RBUFF:  .BLKW   15.                ; Receive buffer
    27      ;
    28              .ENABL  LSB                ; Enable local symbol
    29                                         ;  blocks
    30      ;
    31      START:  MOV     #3,R5              ; Set up message counter
(1) 32      RECEIV: RCST$C  ,RBUFF             ; Receive from anyone
    33              BCC     5$                 ; Branch on directive ok
    34              DIRERR  <RECEIVE DIRECTIVE FAILED IN "RECV2">
    35                                         ; Display error message
    36                                         ;  and exit
    37      ; Successful receipt or unstopped by another task. First
    38      ; check for unstopped after being stopped, in which case
    39      ;  we have to receive the data
(5) 40      5$:     CMP     $DSW,#IS.SET       ; Were we stopped due to
    41                                         ;  no data
    42              BNE     6$                 ; If not, we have a data
    43                                         ;   packet
(6) 44              RCVD$C  ,RBUFF             ; Now get the packet
    45              BCC     6$                 ; Branch on directive ok
(7) 46              DIRERR  <RECEIVE DIR FAILED AFTER "RECV2" UNSTOPPED>
    47                                         ; Display error message
    48                                         ;  and exit
(8) 49      6$:     TYPE    <DATA RECEIVED BY "RECV2":> ; Display
    50                                                  ;  text and
    51              TYPE    #RBUFF+4,#26.               ;  data sent
(9) 52              SOB     R5,RECEIV          ; Decrement message
    53                                         ;  counter. Receive again
    54                                         ;  if haven't received 3
    55                                         ;  yet
    56
    57              TYPE    <"RECV2" HAS RECEIVED 3 MESSAGES AND WILL NOW EXIT>
    58                                         ; Type exit message
    59              EXIT$S                     ; Exit
    60              .END    START
```

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

Using Send/Receive Directives for Synchronization

If you want 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 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 (RQST$). 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 task 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 you want 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 acquires the same privileges as the sending task and can do I/O directly to the sending task's terminal (through TI:). To build a task as a slaved task, either task-build or install with the /SLAVE qualifier.

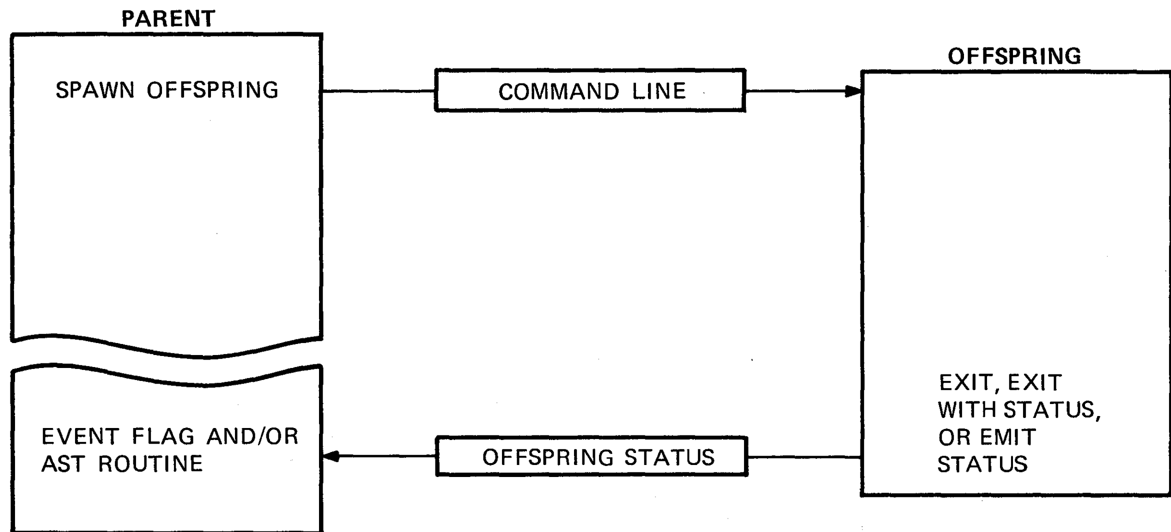


Figure 4-1 Parent/Offspring Communication Facilities

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, 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 only sends data and connects if the task is already active. Spawn is rejected if the task is already active, unless the task is a Command Line Interpreter (CLI).

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

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*	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 exchange data using Send/Receive. The table above indicates that the passing of data from parent to offspring is not a capability of the Connect directive, in and of itself.

LEARNING ACTIVITY 4-1

Chapter 4 of the _RSX-11M/M-PLUS Executive Reference Manual_ contains a good discussion of the Parent/Offspring directives and gives a number of possible uses for them. The various uses will not be discussed anywhere in this course.

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

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 before converting the code to signed decimal for display.

6. Use $EDMSG to produce a status message. Display the message and then exit.

7. 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.

```
53    ; Error handling code - ; Display error message and exit
54    ERR1D:   DIRERR   <ERROR WRITING STARTUP MESSAGE>
55    ERR1I:   IOERR    #IOSB,<ERROR WRITING STARTUP TEXT>
56    ERR2:    DIRERR   <ERROR SPAWNING PIP>
57    ERR3:    DIRERR   <ERROR WAITING FOR EVENT FLAG>
58    ERR4D:   DIRERR   <ERROR WRITING PIP'S EXIT STATUS>
59    ERR4I:   IOERR    #IOSB,<ERROR WRITING PIP'S EXIT STATUS>
60             .END     START
```

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.
>

>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.
>
```

7

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

```
 1              .TITLE   GSPAWN
 2              .IDENT   /01/
 3              .ENABL   LC                    ; Enable lower case
 4      ;+
 5      ; FILE GSPAWN.MAC
 6      ;
 7      ; This task prompts at ti: for a task name and command
 8      ; line, then spawns the specified task and passes it the
 9      ; command line. After that it waits until the offspring
10      ; task exits and displays its exit status.
11      ;
12      ; Assemble and task-build instructions:
13      ;
14      ;            MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]GSPAWN
15      ;            LINK/MAP GSPAWN,LB:[1,1]PROGSUBS/LIBRARY
16      ;
17      ; Run instructions: The name of the task to be spawned
18      ; must be typed in using all upper case characters.
19      ;-
20              .MCALL   SPWN$S,EXIT$S,EXST$,DIR$,WTSE$C ; System
21                                                       ;   macros
22              .MCALL   INPUT,TYPE,DIRERR ; Supplied macros
23      ;
24      ; I/O buffer - initialize first 6 bytes to blanks to pad
25      ; short task names
26  (1) BUFFER: .ASCII   /      /
27  (1)         .BLKB    74.
28      TSKNAM: .BLKW    2                     ; Task name in RAD50
29      BOMB:   EXST$    EX$SEV                ; Directive for fatal
30                                             ;  error
31      BUFF:   .BLKW    80.                   ; Output buffer for exit
32                                             ;  status display
33      FMT:    .ASCIZ   /%N%10STASK EXITED. STATUS WAS %D.%N/
34              .EVEN
35      EXSTAT: .BLKW    8.                    ; Status block
36      ;
37              .ENABL   LSB
38  (2) START:  TYPE     <TASK NAME?>          ; Display prompt
39  (2)         INPUT    #BUFFER,#80.          ; Get input (buffer addr
40                                             ;  returned in R0)
41              BCC      1$                    ; Branch on directive ok
42              TYPE     <INPUT FROM TI: FAILED>
43              DIR$     BOMB                  ; Fatal error
```

Example 4-7 A Generalized Spawning Task (Sheet 1 of 3)

```
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?
CLI...
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
DCL>ABORT/TASK DIR
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
 (9)      Aborted via directive or CLI
          And with pending I/O requests

          TASK EXITED. STATUS WAS 4.
```

Example 4-7 A Generalized Spawning Task (Sheet 3 of 3)

Chaining of Parent/Offspring Relationships

An offspring can chain or pass on its parent/offspring connection 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, and a new connection is established between the first offspring and the new offspring.

Table 4-10 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 executes successfully even if the task is already active.

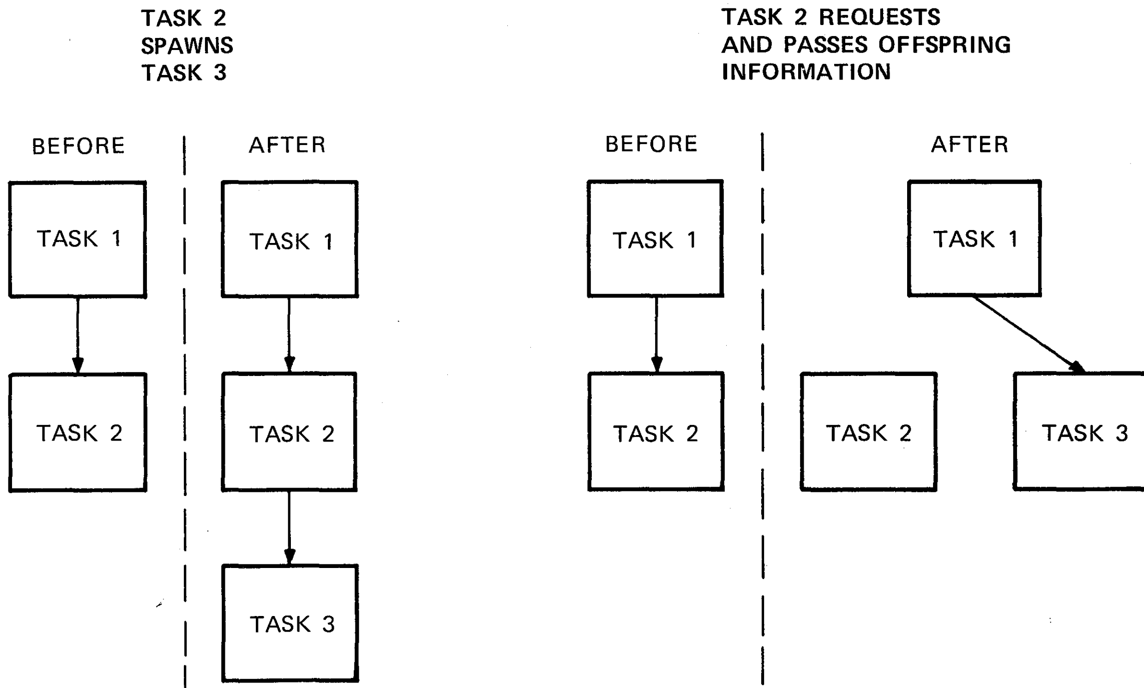


Figure 4-2 Spawning Versus Chaining
(Request and Pass Offspring Information

The following notes are keyed to Example 4-8.

1. Use RPOI$ instead of SPWN$. No event flag is needed nor is a status block set up since this task won't receive status from ...PIP. RP.OAL specified means that all (in this example there is only one) parent connections are passed on. 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. You don't need to use $EDMSG because this task doesn't receive exit status.

3. 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 EXIT$S is used, which results in a success code (+1) being sent as the exit status.

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

5. 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 RP.OAL in the RPOI$ directive call. If RP.OAL 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.

```
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


            TASK EXITED. STATUS WAS 1.
```

(4)

```
>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

            TASK EXITED. STATUS WAS 4.
>
```

(4)

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

Other Parent/Offspring Considerations

Retrieving Command Lines in Spawned Tasks - Use the Get MCR Command Line directive (GMCR$). The passed command is returned, beginning at offset G.MCRB within the DPB for the GMCR$ directive. Therefore, if you use the $ form of the directive and if the DPB starts at location DPB1, the first character of the command line is at location DPB1+G.MCRB.

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).

Any task is spawnable, but only tasks installed under a name of the form ...tsk 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 a greater chance of getting a task already active failure if you spawn a utility directly using the name ...tsk, than there is if you 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 already 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.

```
 1              .TITLE SPWNED
 2              .IDENT  /01/
 3              .ENABL LC                    ; Enable lower case
 4      ;+
 5      ; This task uses the GMCR$ directive to get a command
 6      ; line from either TI: or the parent task. It then
 7      ; echoes the command line and does an add or multiply,
 8      ; types out the answer and emits status on exit
 9      ;
10      ; Assemble and link instructions:
11      ;
12      ;       MACRO/LIST LB:[1,1]PROGMACS/LIBRARY,dev:[ufd]SPWNED
13      ;       LINK/MAP SPWNED,LB:[1,1]PROGSUBS/LIBRARY
14      ;
15      ; Install and run instructions: To make this task MCR
16      ; spawnable, install it under the name ...SPW. Commands
17      ; should be of the form SPW n, where n is a function.
18      ; The valid functions are 1 (for add) and 2 (for multiply).
19      ;-
20              .MCALL  EXST$S,GMCR$,DIR$,QIOW$S ; System macros
21              .MCALL  TYPE,DIRERR,IOERR ; Supplied macros
22      ;
23 (1)  GMCR:   GMCR$                        ; DPB for Get MCR Command
24                                           ;  Line directive
25      BUFF:   .BLKB   80.                  ; Output buffer
26      FMT:    .ASCIZ  /%D %A %D = %D./ ; Format string
27              .EVEN
28      IOSB:   .BLKW   2                    ; I/O status block
29      DATA:
30      NUM1:   .WORD   5                    ; 1st operand
31              .WORD   OP                   ; address of operation
32                                           ;  sign in ASCII
33      NUM2:   .WORD   2                    ; 2nd operand
34      ANS:    .BLKW   1                    ; answer to operation
35      OP:     .BLKB   1                    ; operand in ASCII
36              .EVEN
37      ;
```

Example 4-9 A Spawned Task Which Retrieves a Command Line (Sheet 1 of 3)

```
Run Session

>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                        (11)
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 3 of 3)

Table 4-11 Task Abort Status Codes

Mnemonic	Value	Exit Status	Meaning
S.CEXT	-2(10)	EX$SUC=1	Task exited normally
S.COAD	0	EX$SEV=4	Odd address and traps to four
S.CSGF	2(10)	EX$SEV	Segment fault
S.CBPT	4(10)	EX$SEV	Break point or trace trap
S.CIOT	6(10)	EX$SEV	IOT instruction
S.CILI	8(10)	EX$SEV	Illegal or reserved instruction
S.CEMT	10(10)	EX$SEV	Non-RSX EMT instruction
S.CTRP	12(10)	EX$SEV	Trap instruction
S.CFLT	14(11)	EX$SEV	11/40 floating-point exception
S.CSST	16(10)	EX$SEV	SST abort - bad stack
S.CAST	18(10)	EX$SEV	AST abort - bad stack
S.CABO	20(10)	EX$SEV	Abort via directive or CLI command
S.CLRF	22(10)	EX$SEV	Task load request failure
S.CCRF	24(10)	EX$SEV	Task checkpoint read failure
S.IOMG	26(10)	EX$SEV	Task exit with outstanding I/O
S.PRTY	28(10)	EX$SEV	Task memory parity error
S.CPMD	30(10)	EX$SEV	Task aborted with PMD request
S.CINS	32(10)	EX$SEV	Task installed in two different systems

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 needed.

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 Test/Exercises book. They are all lab problems. Check your answers against the solutions provided, either in that book or 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

5

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 2000K 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-11/03 and PDP-11/04 are always unmapped. The KT-11 unit is available as an option on some medium sized processors, including the PDP-11/35 and PDP-11/40. It is a standard feature on large and newer processors such as the PDP-11/70, 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.

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 an 18-bit UNIBUS address 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.

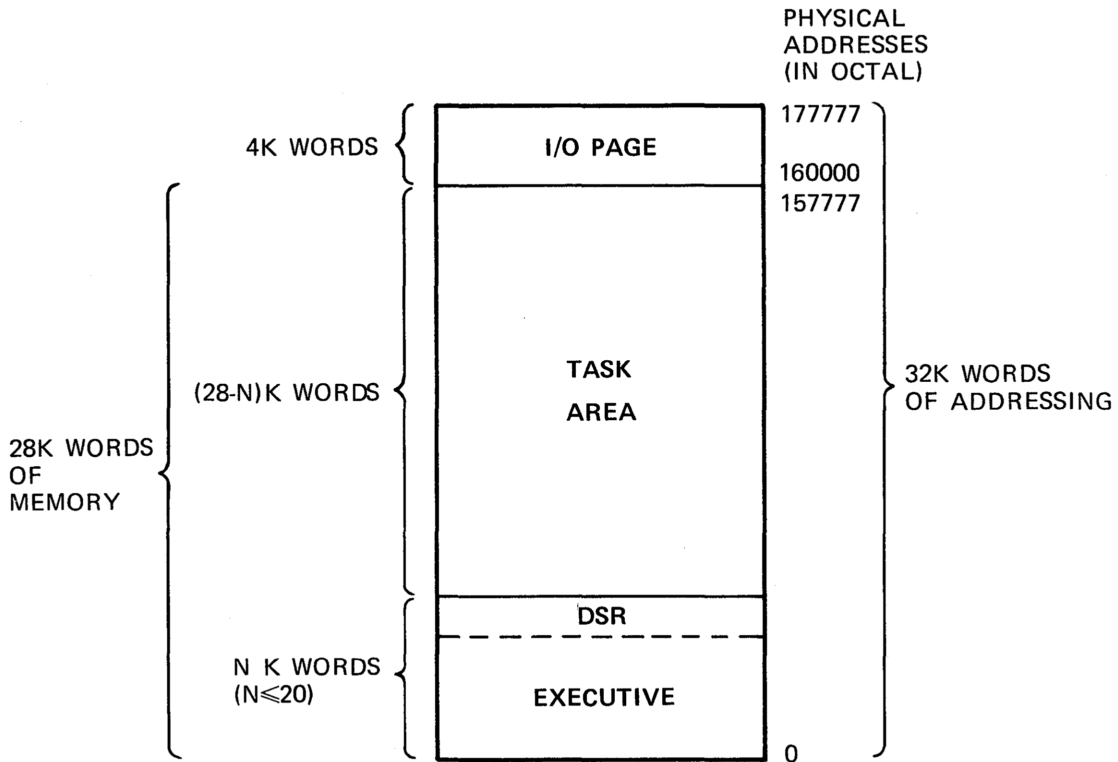


Figure 5-1 Physical Address Space in an Unmapped System

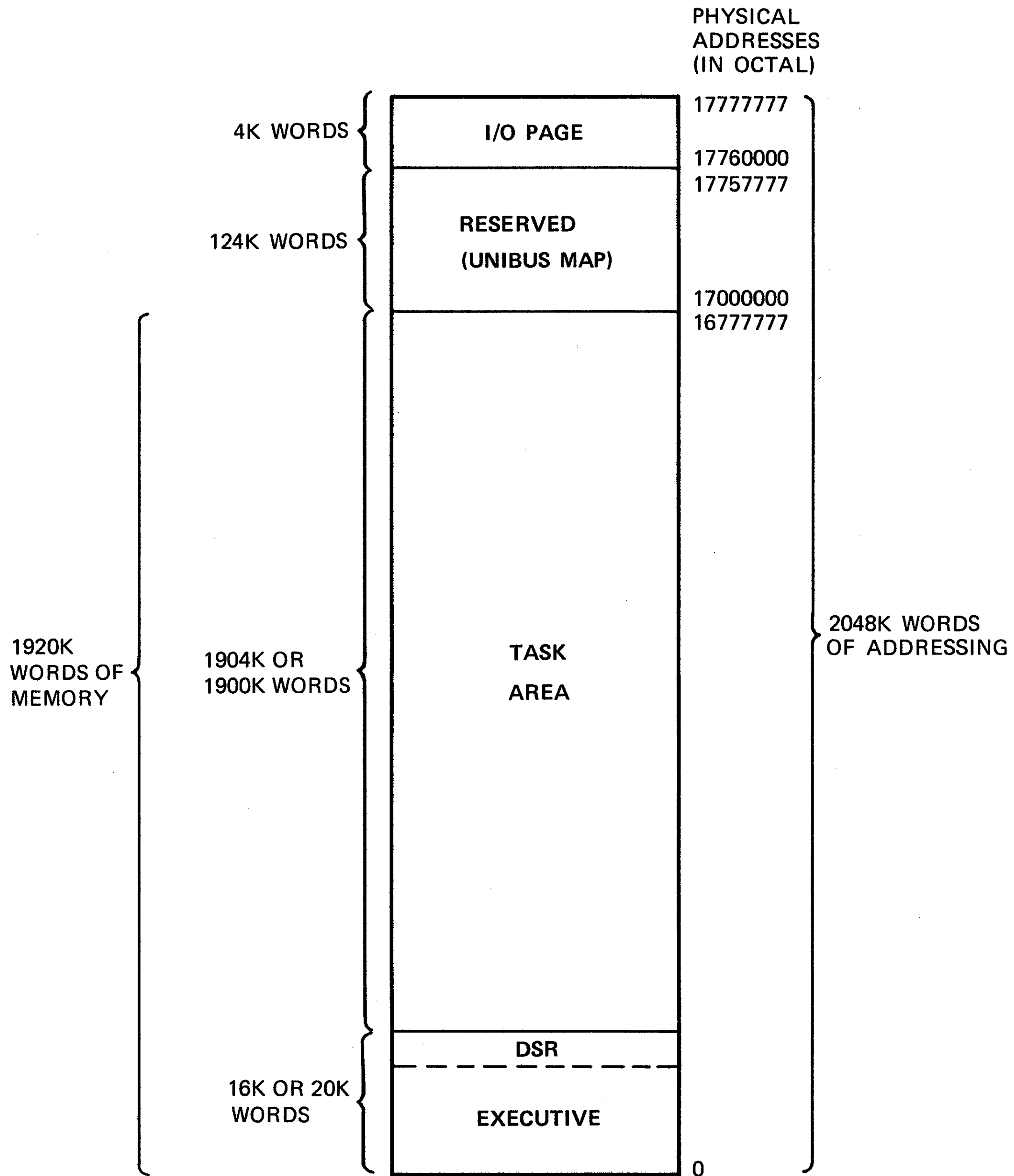


Figure 5-3 Physical Address Space in a 22-Bit Mapped System

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.

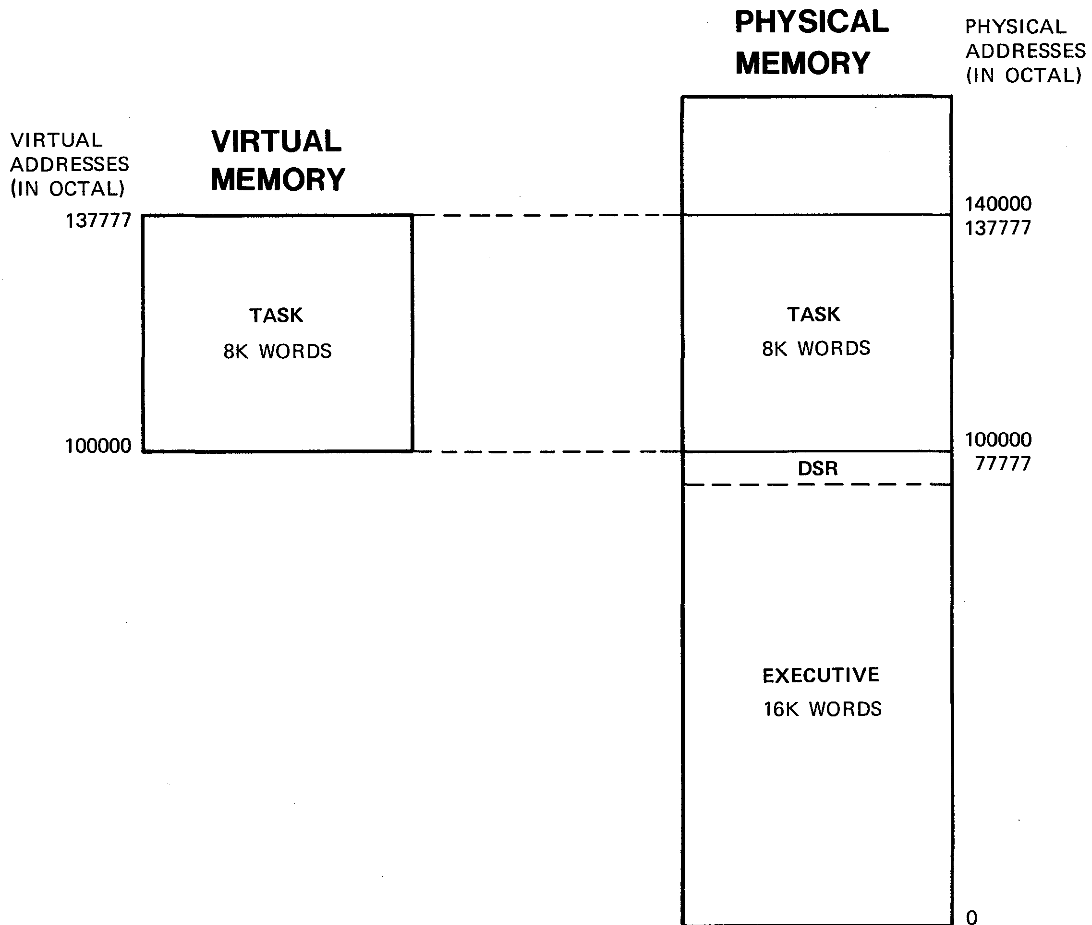


Figure 5-4 Virtual Addresses Versus Physical Address on an Unmapped System

The KT-11 Memory Management Unit

Mode Bits - Bit 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 (APR's) 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.

All virtual addresses within the main task area are mapped to physical addresses beginning at location 00432400(8). This means in effect that each virtual address corresponds to an offset from location 00432400(8). The page descriptor registers, not illustrated, indicate that APRs 0, 1, and 2 map 4K words each, but that APR 3 maps only 2K words.

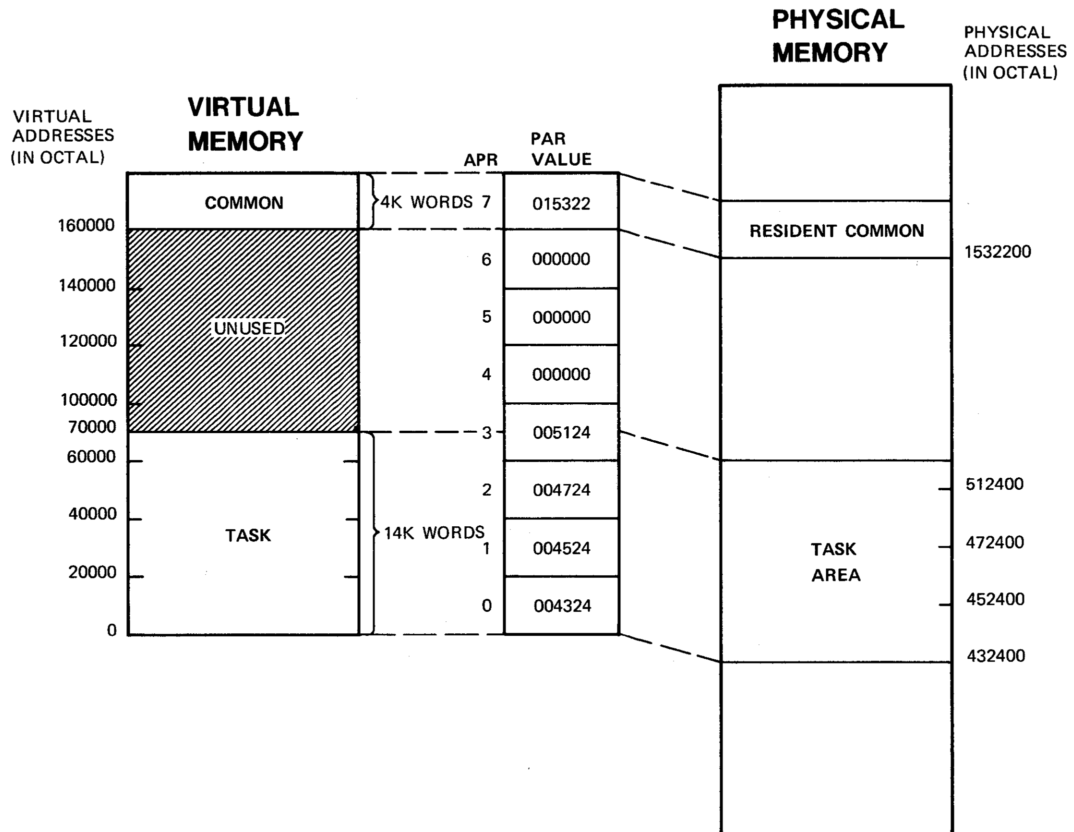


Figure 5-6 Page Address Registers Used in Mapping a Task

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 <u>PDP-11 Processor Handbook</u> discusses this conversion process in more detail.

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.

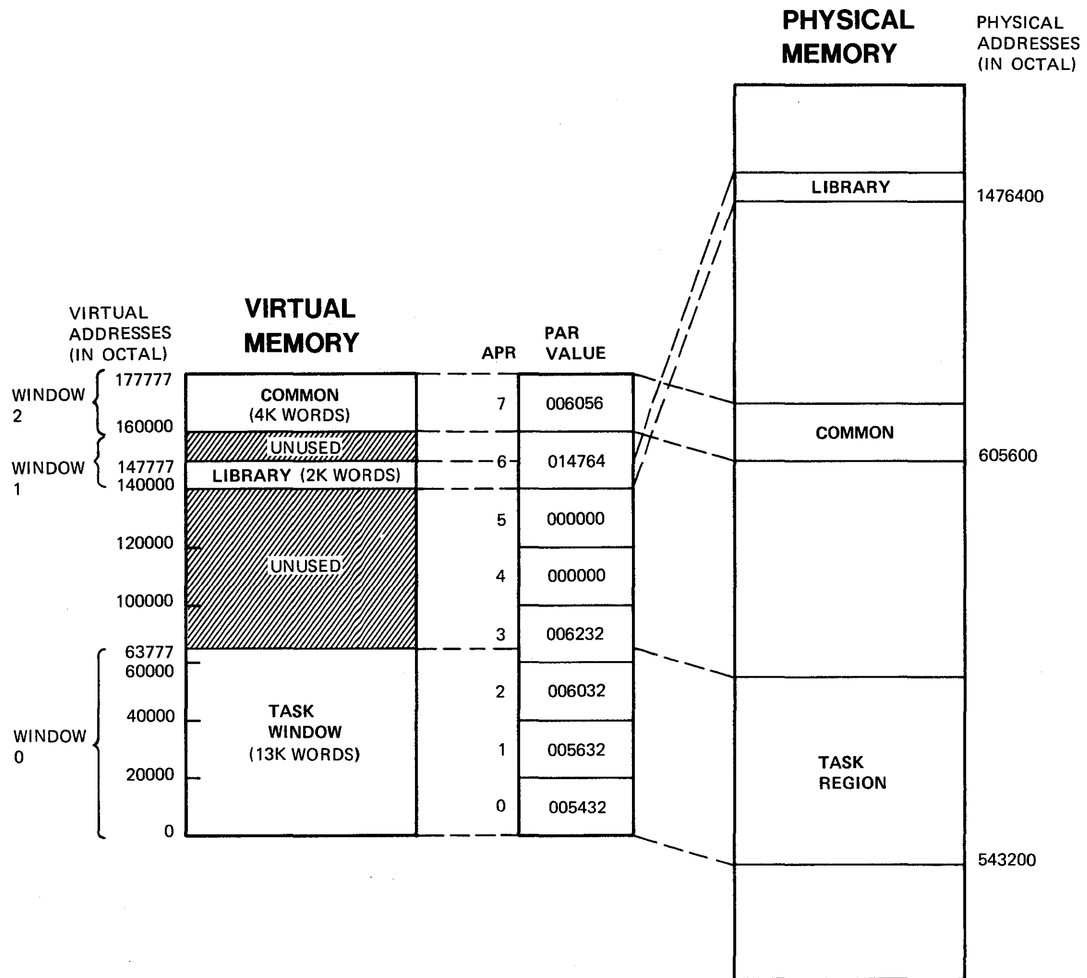


Figure 5-7 A Task with Three Windows Mapped to Three Regions

OVERLAYS

6

INTRODUCTION

Overlays are used to allow a task to be developed and run if there is not enough available physical or virtual memory for a task. This module explains the various overlay techniques and how to use them.

OBJECTIVES

1. To determine whether to use a disk-resident or memory-resident overlay in a given situation
2. To construct overlay structures using the overlay descriptor language
3. To write tasks using overlays.

RESOURCE

- RSX-11M/M-PLUS Task Builder Manual, Chapters 3 and 4

CONCEPTS

A task may be too large to fit in the available memory. This may happen because it is larger than the total amount of memory on the system. More likely, it is because the task is larger than the partition it is to run in, or the available space within the partition. The partition is probably used by other tasks at the same time; therefore, the available space may be considerably less than the full partition.

For example, a 2ØK word task may have to fit in 15K words of memory. The task can use overlays and load only portions of the code at a time, and use just 15K words of memory.

Typically, the pieces which overlay each other contain subroutines. As an example, consider a task with main code and two subroutines, G and H, which overlay each other. The main code calls subroutine G first, causing G's code to be read into memory. Later, the main code calls subroutine H, causing H's code to be read into the same memory locations, overlaying subroutine G. If the main code later calls G, G's code overlays subroutine H. As the task executes, overlaying is performed whenever necessary. You can choose to have all loading of overlay segments done automatically, or you can load them manually with specific calls to a loading routine.

In addition to physical memory limitations, tasks on PDP-11 systems have virtual memory limitations. As discussed in the last module, a task can only use a maximum of 32K words of virtual addresses at a time. A task may require more than 32K words of physical and also virtual memory. For example, a task may need 4ØK words of physical memory, exceeding the virtual addressing limit. This means that the task can't address all of its code. Overlays loaded from disk permit this task to run in 32K words or less of physical memory, and allow all of the code loaded at any given time to be addressed. Therefore, 32K words of code or less are loaded and addressed at any one time, satisfying the virtual address limit.

Another method is to use special kinds of overlays. With these, all 4ØK words of code can be loaded into memory, but the task maps only 32K words of code at a time. This means that the task stays within the virtual addressing limits even though it uses 4ØK words of physical memory.

These special kinds of overlays are called memory-resident overlays. They overlay by remapping, rather than reloading, code into memory.

Main Segment:	PROG
PROG calls:	SUB1, SUB2, SUB3
SUB1 calls:	A, B
SUB2 calls:	none
SUB3 calls:	C, D, E

Segment	Size in Words
PROG	4K
SUB1	2K
SUB2	3K
SUB3	1K
A	1K
B	2K
C	1K
D	2K
E	1K
Total	17K

Example 6-1 Description of an Overlaid Task

STEPS IN PROGRAM DEVELOPMENT USING OVERLAYS

Use the following steps to develop a task which uses overlays.

1. Assemble each module, producing an .OBJ file for each
2. Use the editor to create an overlay descriptor file (defines the overlay structure for the Task Builder).
3. Task-build using the overlay descriptor file as the input file.

THE OVERLAY DESCRIPTOR LANGUAGE (ODL)

The overlay descriptor language (ODL) is a fairly simple language which is used to define the overlay structure for the Task Builder. Statements are placed in a text file which has a file type 'ODL' (e.g., EXAMPLE.ODL). This text file is identified to the Task Builder as a special file by using the /OVERLAY_DESCRIPTION input file qualifier (/MP in MCR) in the task-build command line.

ODL Command Line Format

The ODL command lines use the format that follows.

label: directive argument-list ;comment

where:

label - A one- to six-character symbolic, required only on an .FCTR directive.

directive - one of the following:

.ROOT - indicates the start of the overlay tree

.END - indicates the end of input

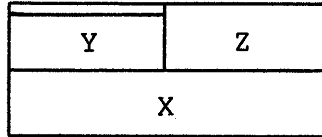
.FCTR - allows naming of subtrees

.NAME - allows naming a segment and assigning attributes

.PSECT - allows special placement of a global program section (Psect).

Examples of ODL

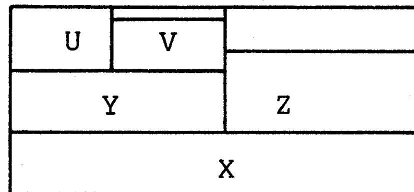
1. X, the root of a task, calls subroutines Y and Z.




```
.ROOT    X-(Y,Z)
.END
```

Explanation: X is the root segment, Y and Z are each overlay segments. Virtual addresses are assigned to X first. Starting after that, Y and Z begin at the same virtual address. Either Y or Z (never both) is loaded and mapped using those virtual addresses.

2. Using the information from Example 1, Y calls subroutines U and V.




```
.ROOT    X-(Y-(U,V),Z)
.END
```

Explanation: Add to Example 1. U and V are overlay segments which overlay each other. After the last address for Y, virtual addresses begin for U and V.

TYPES OF OVERLAYS

There are two types of overlays available, disk-resident overlays and memory-resident overlays. In fact, both are loaded from disk. The distinction is that disk-resident overlays are loaded from disk every time they are needed, while memory-resident overlays are loaded from disk only the first time they are needed. After that, they remain in memory and remapping is used to overlay segments as needed.

Disk-Resident

Disk-resident overlays are available on all RSX-11M systems. An example of a task with a root segment and three disk-resident overlays is shown in Figure 6-3.

On initial load, only the root segment MAIN is loaded. Overlay segments are loaded from disk whenever required. This typically occurs when a subroutine in the segment is called. So if the root segment MAIN contains a call for subroutine A, for example, segment A is loaded from disk prior to the transfer of control to A.

If, after the subroutine returns control to MAIN, a call is made to subroutine B, segment B is loaded into memory right over segment A. If a call is later made to subroutine C, segment C is loaded right over segment B. This loading of overlay segments is performed whenever necessary. The subroutines may be called in any order, and each subroutine may be called any number of times in the course of task execution.

The same starting virtual address is assigned to all three overlay segments, A, B, and C, beginning at the next 32(10) word boundary after the code for MAIN. So A, B, and C use the same virtual addresses and are loaded starting at the same physical address. One virtual address window maps the entire task; just the code in memory is changed when an overlay is loaded.

This technique is useful when the entire task is too large to fit into the space allowed for it. In the example in Figure 6-3, a 22K word task runs in 15K words of physical memory. Disk-resident overlays are the default overlay type. The ODL examples in the previous section all produce disk-resident overlays.

Memory-Resident

Memory-resident overlays are available only on mapped systems which support the memory management directives. Figure 6-4 shows the same task as in Figure 6-3, this time with memory-resident overlays. On initial load, again only the root segment MAIN is loaded. The first time an overlay segment is needed it is loaded from disk. However, once a segment is loaded, it remains in memory and is not reloaded from disk.

If subroutine A is called first, overlay segment A is loaded and virtual address window 1 is mapped to A. If, after the subroutine returns control to MAIN, a call is made to subroutine B, then segment B is loaded, but not directly over A. Instead, it is loaded into another area of memory, and then virtual address window 1 is mapped to B. If a call is later made to subroutine C, segment C is loaded into another area of memory, and virtual address window 1 is mapped to C.

The real gain in run time efficiency is made when an overlay is needed again. If another call is made to A, overlay segment A does not have to be loaded again from disk. It is already memory-resident. Therefore, virtual address window 1 is simply remapped from segment C to segment A. Any additional overlaying is performed by remapping, with no further loading of overlay segments necessary. Again, the subroutines may be called in any order and each subroutine may be called any number of times.

The advantage of this approach is that after the first load, it is much faster than using disk-resident overlays. However, there are no savings in the use of physical memory. In fact, a bit more memory is required than with a non-overlaid task. So the main use of memory-resident overlays is for overcoming the 32K word virtual address limit when execution time efficiency is important. A 44K word task can use memory-resident overlays if there is enough memory available and the time necessary for loading disk-resident overlay segments is unacceptable.

The root segment uses one window, plus each overlay area requires a separate window. That means that virtual addresses for each overlay segment begin at the starting virtual address for the next highest APR, corresponding to a 4K word boundary. Notice that A, B, and C all begin at virtual address 60000(8), for APR3, because MAIN is 9K words long. MAIN uses all 4K words in APRs 0 and 1, plus 1K word in APR2 (virtual addresses 40000(8) through 43777(8)).

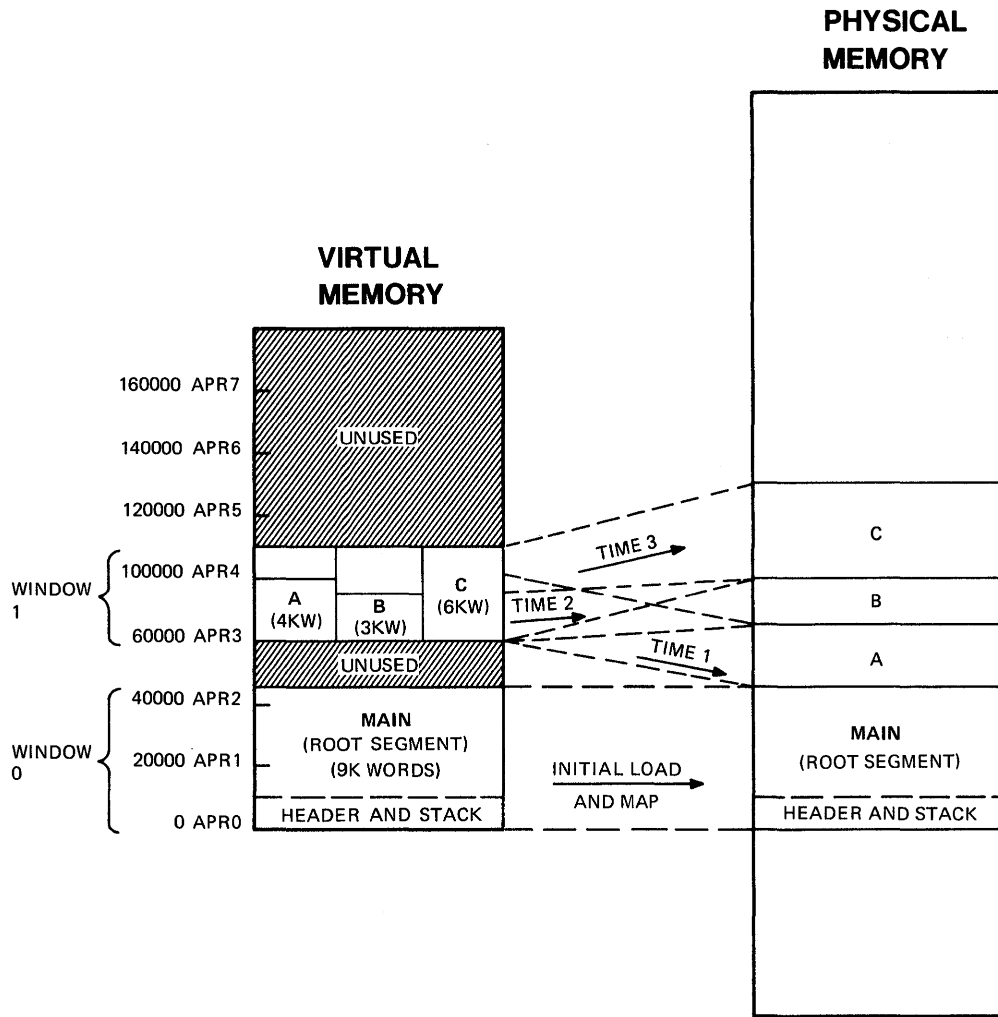


Figure 6-4 An Example of Memory-Resident Overlays

LOADING METHODS

There are two loading methods, autoload and manual load. With autoload, any necessary loading and/or remapping (in the case of memory-resident overlays) is done automatically and is transparent to the program. With manual load, the overlay segments are loaded by specific user calls to a loading routine. Autoload and manual load cannot be mixed in the same task.

Autoload

When a call is made to a subroutine in an overlay segment, an autoload routine takes control before the transfer to the subroutine is made. It checks to find out whether the required segment is already loaded, or loaded and mapped. It performs any necessary loading and/or remapping. After that, the transfer to the called subroutine is made.

Autoload is path loading, meaning that all segments along the path to the required overlay segment are loaded. For example, in example 2 in the previous section, with root X and subroutines Y, U, V, and Z, if a call from segment X is made to subroutine U, both Y and U are loaded. Note that autoload loads only overlay segments along the path which are not already loaded.

Autoload is indicated by an asterisk (*) before an overlay specification in an ODL line. An asterisk outside a set of parentheses applies to all levels inside the parentheses.

The advantages of autoload are that it is easy to use and does not require changes in the source code. One disadvantage is that it increases the size of the segments because the autoload code plus its data structures must be included in the task. Another is that it executes slower than manual load, because the autoload code has to check for whether the required segment is available or not each time an autoloadable segment is called. In addition, autoload must be performed synchronously. See section 4.1 on Autoload in the RSX-11M/M-PLUS Task Builder Manual for more information.

Manual Load

With manual load, you must call the subroutine $LOAD to load and/or map any required overlay segment before calling a subroutine in that segment. You must also keep track of which segments are currently available, to avoid a transfer of control to an incorrect segment and to avoid unnecessary calls to the loading subroutine. Manual load is not path loading. In Example 2 of the previous section, if X calls U, it can load just segment U, without loading segment Y, unless it is desirable to load both. See section 4.2 on Manual Load in the RSX-11M/M-PLUS Task Builder Manual for more information.

Manual load is the default loading method. Whenever there are no asterisks (*) in an ODL file, manual load is used.

The advantages of using manual load are that it results in smaller overlay segments, is usually more run time efficient, and overlay segments can be loaded either synchronously or asynchronously. The disadvantages are that you must keep track of which overlay segments are loaded and use special code in the source program.

Comparison of a Task With No Overlays, to One With Disk-Resident Overlays, and One With Memory-Resident Overlays

Example 6-1, shown earlier in the module, and repeated below for convenience, shows a main program which calls a subroutine, which in turn calls another subroutine, etc. Note that the sizes shown for the various parts of the task are only approximate.

Task-build command:

LINK/MAP PROG,SUB1,A,B,SUB2,SUB3,C,D,E

```
Partition name : GEN
Identification : 01
Task  UIC      : [305,301]
Stack     limits: 000254 001253 001000 00512.
PRG xfr address: 021254
Total address windows: 1.
Task  image  size  : 17792. words
Task address limits: 000000 105357
R-W disk blk limits: 000002 000107 000106 00070.


*** ROOT SEGMENT: PROG


R/W mem  limits: 000000 105357 105360 35568.
Disk blk limits: 000002 000107 000106 00070.
```

Example 6-2 Map File of Example 6-1 Without Overlays

PROG.ODL file:

```
.ROOT PROG-*(SUB1-(A,B),SUB2,SUB3-(C,D,E))
.END
```

Task-build command:

```
LINK/MAP PROG/OVERLAY_DESCRIPTION
```

```
Partition name : GEN
Identification : 01
Task  UIC      : [305,301]
Stack    limits: 000260 001257 001000 00512.
PRG xfr address: 021260
Total address windows: 1.
Task  image  size  : 8800. words
Task address limits: 000000 042237
R-W disk blk limits: 000002 000120 000117 00079.
```

```
EX63.TSK Overlay description:

Base    Top        Length
----    ---        ------
000000  022177  022200  09344.      PROG
022200  032233  010034  04124.          SUB1
032234  036237  004004  02052.               A
032234  042237  010004  04100.               B
022200  036203  014004  06148.          SUB2
022200  026247  004050  02088.          SUB3
026250  032253  004004  02052.               C
026250  036253  010004  04100.               D
026250  032253  004004  02052.               E
```

Example 6-3 Map File of Example 6-1 With Disk-Resident Overlays

PROG.ODL file:

```
.ROOT PROG-*!(SUB1-!(A,B),SUB2,SUB3-!(C,D,E))
.END
```

Task-build command:

```
LINK/MAP PROG/OVERLAY_DESCRIPTION
```

```
Partition name : GEN
Identification : 01
Task  UIC      : [305,301]
Stack    limits: 000320 001317 001000 00512.
PRG xfr address: 021320
Total address windows: 3.
Task  image  size  : 18464. words
Task address limits: 000000 077777
R-W disk blk limits: 000003 000122 000120 00080.


EXDOVR.TSK Overlay description:

Base    Top       Length
----    ---       ------
000000  023077  023100  09792.        PROG
040000  050077  010100  04160.             SUB1
060000  064077  004100  02112.                  A
060000  070077  010100  04160.                  B
040000  054077  014100  06208.             SUB2
040000  044077  004100  02112.             SUB3
060000  064077  004100  02112.                  C
060000  070077  010100  04160.                  D
060000  064077  004100  02112.                  E
```

Example 6-4 Map File of Example 6-1 With Memory-Resident Overlays

Table 6-1 Comparison of Overlaying Methods (Cont)

Method	Task Size	Windows	Advantages and Disadvantages
Memory-Resident	18464(10) words of memory 80(10) blocks on disk 100000(8) virtual addresses used	3	Advantages Faster execution than disk-resident overlays Task resident in memory at one time Disadvantages Uses the most memory and disk space May waste virtual address space Requires space in memory to hold all of the task

Table 6-1 compares the three overlaying methods. In addition to the various sizes, it lists the advantages and disadvantages of each approach.

Remember that it is also possible to mix memory-resident and disk-resident overlays in a task. For example, the first level (SUB1, SUB2, and SUB3) could be memory-resident, and either or both second levels (A, B or C, D, E) could be disk-resident.

Include needed modules from FOROTS.OLB in the root segment in segment A, and in segment B. You should specify the library in each segment which may need it. Otherwise, if segment A needs a library module not already included for the root segment, the library is not searched again for module A.

An Overlay Example

Example 6-5 is a simple task with a root segment ROOT and two overlay segments, P and Q. The following calling sequence is used during the execution of the task.

ROOT calls P
ROOT calls Q

Figure 6-5 shows an overlay tree and a memory allocation diagram for this task.

The code for Example 6-5 is separated into three different modules, one for each segment. The source file for the root segment ROOT contains the startup code and controls the overlay loading by calls to the subroutines. The source file for each overlay segment, P and Q, contains the subroutine code.

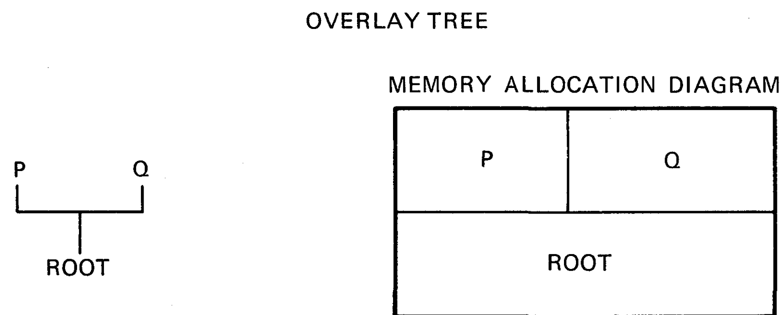


Figure 6-5 Task With Two Overlay Segments

The notes below are keyed to Example 6-5.

1. On initial load only the root segment ROOT is loaded.

2. With autoload, the call to subroutine P causes the autoload routine to load overlay segment P from disk, and then transfer control to the subroutine.

3. Subroutine P displays a message and returns.

4. The call to subroutine Q causes the autoload routine to load overlay segment Q from disk over segment P, and then transfer control to the subroutine.

5. Subroutine Q displays a message and returns.

If another call were added to subroutine Q, the autoload routine would check to make sure that overlay segment Q is already loaded, and would then transfer control to Q. If another call were added to subroutine P, the autoload routine would check and find that overlay segment P is not loaded. It would then load segment P over segment Q and transfer control.

To use manual load, use additional code to load the segments into the root segment ROOT. Also, modify the .ODL file, omitting the asterisk (*). The files MLROOT.MAC and MLEXDOVR.ODL on the tape provided with this course are modifications of ROOT.MAC and EXDOVR.ODL for manual load. Check UFD [202,3] for these files. See your course administrator if you have difficulty finding them.

```
 1           .TITLE Q
 2           .IDENT  /01/
 3           .ENABL  LC                  ; Enable lower case
 4     ;+
 5     ; FILE Q.MAC
 6     ;
 7     ; This subroutine displays a message and returns.
 8     ;-
 9           .MCALL  QIOW$C              ; External system macros
10     ;
11     MES:   .ASCII  /SEGMENT Q IS NOW LOADED. SUBROUTINE Q/
12           .ASCII  / IS EXECUTING./
13           LMES = . - MES
14           .EVEN                       ; Move to word boundary
15     ;
16     Q::    QIOW$C IO.WVB,5,1,,,,<MES,LMES,40> ; Display
17                                             ;  message
18           RETURN                      ; Return
19           .END
```

5 (lines 16–18)

Run Session

```
>RUN EXDOVR
THE MAIN SEGMENT IS RUNNING AND WILL CALL P.
SEGMENT P IS NOW LOADED. SUBROUTINE P IS EXECUTING.
THE MAIN SEGMENT WILL NOW CALL Q.
SEGMENT Q IS NOW LOADED. SUBROUTINE Q IS EXECUTING.
THE MAIN SEGMENT WILL NOW EXIT.
>
```

Example 6-5 A Task With Two Overlay Segments (Sheet 2 of 2)

Table 6-2 How Global Symbols Are Resolved

	Reference In	Resolved To
Q is defined in AØ and BØ	A22	AØ
	A1	AØ
	B1	BØ
R is defined in A2	A22	A2
	A1	Undefined
	CNTRL	A2
	(If autoload, defined through an autoload vector.)	
S is defined in AØ and BØ	A1	AØ
	A21	AØ
	A22	AØ
	B1	BØ
	B2	BØ
	CNTRL	Ambiguous
T is defined in AØ and A21	Symbol multiply defined.	

Subroutine Calls

With manual load, since the global symbols are resolved directly to the virtual address corresponding to the symbol, the transfer is directly to the subroutine. The program must ensure that the correct overlay segment is loaded before making the call. Otherwise, the transfer will transfer control to that virtual address in the wrong code, causing unexpected results.

With autoload, the global symbols are resolved directly for calls downward toward the root. This works because path loading ensures that the segments along the path closer to the root are in fact loaded. The calls to subroutines away from the root are resolved through autoload vectors. This causes the call to the subroutine to transfer control first to the autoload routine, allowing it to check and load any needed overlay segments before transferring control to the virtual address of the subroutine.

Data References

The safest place for all data is in the root segment. Data placed in an overlay segment is only accessible when the overlay segment is loaded and the reference is resolved directly. This means that with manual load, the data is accessible when the segment is loaded.

With autoload, on the other hand, it's not that simple. References out from the root are usually not resolved directly, but through autoload vectors. For example, the reference to the global symbol A, a data label, is resolved to the label of an autoload vector within the same overlay segment. The actual virtual address of A is a value within the autoload vector. Therefore, a reference to A references the autoload vector, not the data itself. In addition, a reference to A does not cause the overlay segment to be loaded. It is loaded only on a call to a subroutine. Although there are some ways with autoload to get references resolved directly, it is difficult.

With disk-resident overlays, another problem arises with any data changed at run time. If the data is in an overlay segment, it is reinitialized every time the segment is reloaded from disk, since the original copy of the code is reloaded. This problem occurs with both manual load and autoload.

The Task Builder normally combines together allocations for Psects of the same name. If the Psects have the local (LCL) attribute, combining is only done within a single overlay segment. If the Psects have the global (GBL) attribute, combining is done across overlay segment boundaries. For Psects with the GBL attribute, by default, these allocations are collected in the segment specifying the Psect which is closest to the root segment. Therefore, if the Psect MYDATA is specified in the root segment and also in one or more overlay segments, the complete allocation is placed in the root segment. The OVR attribute tells the Task Builder to begin both allocations at the same virtual address. Consider Example 2 above. The local symbol M, defined locally in the overlay segment, corresponds to the beginning of the Psect in the root segment, the address of the first 2. The instruction INC M+2 again increments the second 2 to a 3.

See Appendix E for additional information on how the Task Builder uses the various Psect attributes. Also see section 3.2.4 (on Allocation of Program Sections in a Multisegment Task) in the RSX-11M/M-PLUS Task Builder Manual for a description of how the Task Builder allocates Psects in an overlaid task.

Two other methods can be used to place in the root a Psect which is not defined in the root. If a Psect has the SAV attribute, the Task Builder automatically places that Psect's allocation in the root. If the Psect does not have the SAV attribute, then the .PSECT Overlay Descriptor Language statement can be used to specify placement of a particular Psect in the root, overriding the default placement. See section 3.4.5 (on the .PSECT Directive) in the RSX-11M/M-PLUS Task Builder Manual for an example of the use of .PSECT ODL directive.

Example 6-6 is a more complex example of the use of overlays. It shows the use of both techniques for placing data in the root and referencing it from overlay segments. The program calling sequence is shown below.

The following notes are keyed to the example.

1. The Psect OTHER is set up for using overlaid Psects to reference the data. Since it is defined in the root, the entire allocation for OTHER is in the root segment. OP1, OP2, and ANS can be just locally defined, since the overlay segments define the locations as offsets from the start of the Psect. On the other hand, global symbols can be used instead, if desired. The data is an argument block for a call to $EDMSG.

2. The references to the data from overlay segment JOB1 are set up by specifying the Psect OTHER, then defining local symbols. .BLKW statements are used because you are just defining symbols and offsets. The local symbols NUM1, NUM2, and SUM correspond to OP1, OP2, and ANS, respectively, in MAIN.

3. The references to the data from overlay segment JOBXX are set up in a similar way. This time the same local symbols OP1, OP2, and ANS are used again.

4. The references to the data from overlay segment A are also set up in a similar way. This time only the starting address of the argument block is needed.

5. The grand total and the ASCII operand (for $EDMSG) are defined with the global symbols TOT and OP.

6. The reference to TOT and OP in JOB1, and JOBXX, are automatically resolved directly. No special coding is needed in the referencing segment. TOTAL also references TOT, this time from the root segment (because TOTAL is also in the root segment).

7. Note that data which is pure (read-only) and referenced within the overlay segment only, causes no problems when placed in an overlay segment. The references are direct and the data is only referenced while the segment is loaded.

8. The input buffer for the job number typed in by the operator, and the output buffer for displaying an operation's results are contained in an overlay segment and changed at run time. However, since the data is accessed only from within the overlay segment, and only while the segment is still loaded, no problems result. If, in fact, the MAIN segment referenced this data after a call to B was made, it would no longer work correctly because on reload, the data is reinitialized.

```
52    ; Set up for loop
53              MOV      #3,R4              ; Counter
54    LOOP:     QIOW$C   IO.WVB,5,1,,,,<MES3,LMES3,40> ; Write MES3
55              CLR      ANS                ; Clear answer in case
56                                          ;  of no operation
57              CALL     A                  ; Call subroutine A
58              SOB      R4,LOOP            ; Decrement counter and
59                                          ;  loop back until done
60              QIOW$C   IO.WVB,5,1,,,,<MES4,LMES4,40> ; Write MES4
61              CALL     TOTAL              ; Call routine to
62                                          ;  display grand total
63              QIOW$C   IO.WVB,5,1,,,,<MES5,LMES5,40> ; Write MES5
64              EXIT$S                      ; Exit
65              .END START
```

```
 1              .TITLE A
 2              .IDENT  /01/
 3              .ENABLE LC                  ; Enable lower case
 4    ;+
 5    ; FILE A.MAC
 6    ;
 7    ; This subroutine displays a message and then asks which
 8    ; of two jobs to do. It calls the appropriate subroutine
 9    ; to do the job, displays the results, and then returns
10    ; to the main program.
11    ;-
12              .MCALL   QIOW$C,QIOW$S      ; System macros
13              .NLIST   BEX                ; Do not list binary
14                                          ;  extensions
15              .PSECT   OTHER   D,GBL,OVR,REL,RW ; PSECT with data
16    ARG:      .BLKW    4                  ; Set address for start     (4: lines 16-17)
17                                          ;  of argument block
18              .PSECT                      ; Back to blank PSECT
19    MES:      .ASCII   <11>/SEGMENT A IS NOW LOADED. SUBROUTIN/   (7: lines 19-26)
20              .ASCII   /E A IS EXECUTING./
21              LMES=.-MES
22    PMES:     .ASCII   <11>/DO YOU WANT TO DO JOB 1 OR JOB 2? /
23              LPMES=.-PMES
24    EMES:     .ASCII   <15><11>/NO SUCH JOB. SORRY./
25              LEMES=.-EMES
26    OFMT:     .ASCIZ   <11>/%D %A %D = %D.%N/
27    OBUFF:    .BLKB    100.               ; Buffer for display of     (8: lines 27-29)
28                                          ;  job results
29    BUFF:     .BLKB    1                  ; Buffer for input char
30              .EVEN                       ; Move to word boundary
```

Example 6-6 Complex Example Using Overlays (Sheet 2 of 6)

```
7 [ 26    MES:      .ASCII   <15><11><11>/SEGMENT JOB1 IS NOW LOADED./
  [ 27              .ASCII   <15><12><11><11>/SUBROUTINE JOB1 IS EX/
  [ 28              .ASCII   /ECUTING./
    29              LMES=.-MES
    30              .EVEN
    31              .LIST    BEX                ; List binary extensions
    32
    33    JOB1::    QIOW$C   IO.WVB,5,1,,,,<MES,LMES,40> ; Display
    34                                                   ;  message
2 [ 35              MOV      NUM1,SUM           ; First operand to ans
  [ 36              ADD      NUM2,SUM           ; Add in other operand
6 [ 37              ADD      SUM,TOT            ; Add this answer to total
  [ 38              MOV      #'+,OP             ; Move operand for output
    39                                          ;  display
    40              RETURN                      ; Return
    41              .END
```

```
    1               .TITLE   JOBXX
    2               .IDENT   /01/
    3               .ENABL   LC                 ; Enable lower case
    4     ;+
    5     ; FILE JOBXX.MAC
    6     ;
    7     ; This subroutine performs a multiplication operation.
    8     ; It is assumed that local symbols OP1, OP2 and ANS
    9     ; correspond to the same local symbols in MAIN. The
   10     ; global symbol TOT, defined in MAIN, is the address
   11     ; where the grand total is maintained.
   12     ;-
   13               .MCALL   QIOW$C             ; External system macros
   14               .NLIST   BEX                ; Do not list binary
   15                                          ;  extensions
3 [ 16              .PSECT   OTHER    D,GBL,OVR,REL,RW ; Data PSECT
  [ 17    OP1:      .BLKW    1                  ; 1st operand
  [ 18              .BLKW    1                  ; Address of operation
  [ 19                                          ;  in ASCII
  [ 20    OP2:      .BLKW    1                  ; 2nd operand
  [ 21    ANS:      .BLKW    1                  ; Answer
   22
   23               .PSECT                      ; Back to blank PSECT
   24               .BLKW    1024.*2            ; Leave space to make
   25                                          ;  module larger
7 [ 26    MES:      .ASCII   <15><11><11>/SEGMENT JOBXX IS NOW/
  [ 27              .ASCII   / LOADED./<15><12><11><11>
  [ 28              .ASCII   /SUBROUTINE JOB2 IS EXECUTING./
   29               LMES=.-MES
   30               .EVEN
   31               .LIST    BEX                ; List binary extensions
```

Example 6-6 Complex Example Using Overlays (Sheet 4 of 6)

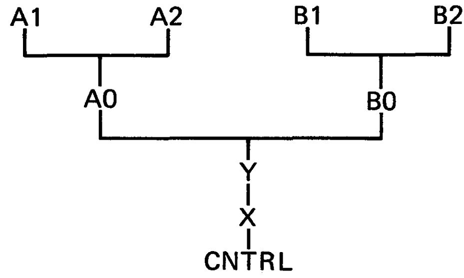
```
 1          .TITLE B
 2          .IDENT  /01/
 3          .ENABL  LC              ; Enable lower case
 4   ;+
 5   ; FILE B.MAC
 6   ;
 7   ; This subroutine displays a message and returns
 8   ;-
 9          .MCALL  QIOW$C          ; External system macros
10          .NLIST  BEX             ; Do not list binary
11                                  ;  extensions
12   MES:   .ASCII <11>/SEGMENT B IS NOW LOADED. SUBROUTINE/
13          .ASCII  / B IS EXECUTING./
14          LMES = . - MES
15          .EVEN                   ; Move to word boundary
16   ;
17   B::    QIOW$C IO.WVB,5,1,,,,<MES,LMES,40> ; Display
18                                             ;  message
19          RETURN                  ; Return
20          .END
```

(7) — callout marking lines 12–13

Run Session

```
>RUN MRMAIN
THE MAIN SEGMENT IS RUNNING AND WILL CALL A
        SEGMENT A IS NOW LOADED. SUBROUTINE A IS EXECUTING.
        DO YOU WANT TO DO JOB 1 OR JOB 2? 1
                SEGMENT JOB1 IS NOW LOADED.
                SUBROUTINE JOB1 IS EXECUTING.
        5 + 2 =  7

THE MAIN SEGMENT WILL NOW CALL B
        SEGMENT B IS NOW LOADED. SUBROUTINE B IS EXECUTING.
THE MAIN SEGMENT WILL NOW CALL A
        SEGMENT A IS NOW LOADED. SUBROUTINE A IS EXECUTING.
        DO YOU WANT TO DO JOB 1 OR JOB 2? 2
                SEGMENT JOBXX IS NOW LOADED.
                SUBROUTINE JOB2 IS EXECUTING.
        5 * 2 = 10

THE MAIN SEGMENT WILL NOW CALL A
        SEGMENT A IS NOW LOADED. SUBROUTINE A IS EXECUTING.
        DO YOU WANT TO DO JOB 1 OR JOB 2? 2
                SEGMENT JOBXX IS NOW LOADED.
                SUBROUTINE JOB2 IS EXECUTING.
        5 * 2 = 10

THE MAIN SEGMENT WILL NOW CALL A
      SEGMENT A IS NOW LOADED. SUBROUTINE A IS EXECUTING.
      DO YOU WANT TO DO JOB 1 OR JOB 2? 1
              SEGMENT JOB1 IS NOW LOADED.
              SUBROUTINE JOB1 IS EXECUTING.
       5 + 2 =  7

THE MAIN SEGMENT WILL CALL TOTAL
THE GRAND TOTAL IS  34.

THE MAIN SEGMENT WILL NOW EXIT
>
```

Example 6-6 Complex Example Using Overlays (Sheet 6 of 6)



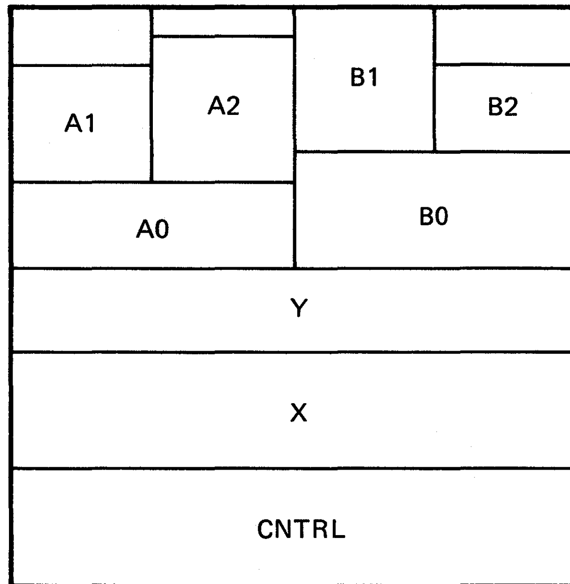


Figure 6-7 Task Without Co-Trees

Now do the tests/exercises for this module in the Tests/Exercises book. All but the first problem are lab problems. Check your answers against the provided solutions, either the on-line files (under UFD [2Ø2,2] or the printed copies in the Tests/Exercises book.

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