RSX

A Guide for Users

RSX

A Guide for Users

JOHN F. PIEPER

digital
DECbooks

Copyright © 1987 by Digital Equipment Corporation. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without prior permission of the publisher.

9 8 7 6 5 4 3 2 1

Printed in the United States of America
Order number EY-6708E-DP

DEC, DECsystem-10, DECSYSTEM-20, DECtape, DECwriter, the Digital logo, PDP-11, RSX, RSX-11, RSX-11M, RSX-11M-Plus, and Micro/RSX are trademarks of Digital Equipment Corporation. IBM-370 is a trademark of the International Business Machines Corporation. CP/M is a registered trademark of Digital Research, Inc. MS-DOS is a registered trademark of Microsoft Corporation.

Library of Congress Cataloging-in-Publication Data

Pieper, John F.
RSX, a guide for users.

(DECbooks)
1. RSX (Computer operating system) 2. PDP-11 (Computer)—Programming. I. Title. II. Series.
QA76.76.063P54 1987 005.4'44 86-29109
ISBN 0-932376-90-8

This book is dedicated across three generations:

to Gertrude, without whom none of this could have happened,

to Lee, who patiently did all the typing, changing and changing,

to Monica, who having decided I was finally serious about it, staunchly supported me all the way,

to Dawn, who waited for it all to get done,

and to Sean, who was too young to care.

Preface

In 1975 I met my first RSX. This was RSX-11D. Many changes and enhancements have been made by Digital Equipment Corporation since then. RSX-11M has supplanted "D"; "M"s big brother M-Plus is in the process of supplanting it. They have both been joined by Micro/RSX, which supports the increasingly popular Micro PDP-11. These changes notwithstanding, most all of the basic operating system commands I used then still work on your system today. Given the rapid growth in the computer industry—both hardware and software—this represents an impressive longevity.

When I was using RSX-11D, I did so at an extremely simple level. I wrote some scientific analysis programs, ran them, fixed them, put the answers in a report, kept a copy of the program somewhere—in short, I was a casual user. Several years later I was at a different job. When we bought our first computer (PDP-11/70 with RSX-11M Version 3.1), I found that I was the only one in the company who had ever seen RSX, in any shape or form, before. By definition, I was an expert. I decided that the only sensible way to handle the demands of a variety of users with a variety of questions was to write a brief manual explaining the basics of RSX. Thus, *RSX: A Guide for User* was born.

Things went more smoothly than expected during my stint as de facto system manager. Those who came with questions that were answered in the manual were told to read it first and sent away. Few came back. We rolled along, perhaps doing nothing fancy, but getting our work done. I decided that if this manual had been successful to our company, it should be of value to users at other RSX installations as well. Of course, a little rewriting and an extra section or two would be necessary before what was an in-house training manual would be good enough to give to a publisher. By the time I was done, the book had more than quadrupled in size. Then again, Digital's official documentation for the latest version of RSX-11M, binders and all, now fills a three-foot bookshelf. So be it.

As with any project of this nature, there are many acknowledgments to make. First, I would like to thank Dr. Julian J. Bussgang for graciously

giving me permission to use the material which I wrote as an employee of Signatron Inc. and which forms the nucleus of this book. Also, I would like to acknowledge the many helpful suggestions of those who reviewed the book in manuscript form: James McGlinchey, Thomas Parmenter, Craig Silver, and Cathy Ziegelmiller. Just as important as the creation is the preparation. In this regard, Lee Midthun has been all that one could ask for and more.

To all of these good friends—Thanks!

And to you the reader—Enjoy!

John Pieper
Reading, Massachusetts

Contents

Part I System Conventions

1

Some Introductory Remarks

This user's guide to RSX is intended to be a complete reference manual for the casual or normal RSX user. I consider a "casual" user to be someone who uses the computer system for relatively straightforward applications. A definition such as this can never be exact. Loosely speaking, if you write, edit, and run programs that do not run in real-time and that are not large enough to require special storage requirements, you are a casual or normal user. RSX also offers many features that allow the system level user to do more advanced things; these are outside the scope of this book and are not discussed. Recently, RSX has been expanded in its capabilities by the addition of networking; this too is outside the scope of this book.

Some of the statements herein are deliberate oversimplifications. These should be of no concern to the casual user and are accordingly not distinguished in the text from those statements that are strictly true. The system level user, who should be treating this manual as introductory or background material, is welcome to discover these half-truths and replace them with their more complex realities.

For the casual user, this manual is intended as an introduction and supplement to the official RSX documentation supplied by Digital Equipment Corporation. Its purpose is to introduce you to the various aspects of RSX, from basic concepts to advanced techniques, with emphasis on those things that you, as a casual user, will need to do and understand. I can never hope to achieve the level of detail offered in the RSX manuals. Once you have mastered the material in this book, you are encouraged to turn to these should you wish a better understanding of how things work. I nonetheless hope that, no matter how adept you become with RSX, this guide will continue to be a convenient and useful reference.

In this book I include many examples for which I follow several conventions. I use boldface to denote anything that you enter; responses made by the system to you are in regular type. I do this mainly in the examples themselves, but I also use boldface in the main text to highlight command names or other keywords that you enter. Anything that you must type in as shown is printed in upper case; a generic term, for which you substitute a specific value, is printed in lower case. Many commands consist of several parts with some special symbol used to separate one part from the next. Often this separator is a space. In other parts of the command you may use spaces if you wish. This is often recommended for clarity, especially in indirect command files. In the examples, I refrain from using spaces where they are not needed in order to accentuate those that are mandatory. When a generic term has two or more words in it, I use underscores instead of spaces between the words to avoid confusion with any mandatory spaces in the command. Finally, all command lines are terminated by a carriage-return (the RETURN key on your terminal); I do not show this. For example, the basic form of the command to run a task is shown as

```
RUN task
```

where the use of boldface indicates that this is something that you type in. Here, the space between **RUN** and **task** is clearly required. By using lowercase for **task**, I am indicating that you should replace this by the actual task name when you use this command. A specific example, such as the command to run a task called TEST1 is thus

```
RUN TEST1
```

An alternate form of the Run command allows specification of a task image file instead of a task; this is shown as

```
RUN task_image_file
```

Here, the underscores show that "task image file" is a single entity that you must similarly replace with an actual name when you issue the command.

Finally, I note that the English language does not offer any convenient way to say "he or she" or "his or her." It is traditional practice to use "he" and "his" when referring to a person of unspecified gender. (The aroused feminist may ponder the potential double implications of this.) I follow this practice, trusting that you will be understanding.

2

What Is RSX ?

RSX is intended as a real-time multiuser operating system. As such, it can accommodate several simultaneous users. Note that the definition of a user is not limited to the familiar notion of a person at a terminal, but also includes tasks that either have been previously activated by a user or are a part of the operating system itself. For example, RSX might be used as an operating system for a PDP-11 that is used to control (in real-time) an assembly line as well as to support (in nonreal-time) scientific users. Here, the various tasks to process interrupts from the assembly line would be users of the system, just the same as a job being run by someone at a terminal. Many of the features of RSX were created to supply this capability, so many of the commands and conventions are of little concern to you. Nonetheless, RSX offers a powerful multiuser capability, is relatively easy to use, is by now thoroughly debugged, and is accordingly a very useful operating system.

Three forms of the RSX operating system are commonly available—RSX-11M, RSX-11M-PLUS and Micro/RSX. For most purposes they are identical. The significant distinctions among them are discussed in this manual. Most of the time, I simply use RSX to denote any of these systems.

An older form of RSX, RSX-11D, is no longer offered but still exists on various installations. It is similar to the current forms of RSX, but it does not have the newer, more sophisticated features of these systems. Since RSX-11D has been discontinued, I do not distinguish those features that are not available with it.

A fifth form of RSX also exists. This is RSX-11S, a subset of RSX-11M that does not support any disk operations. It is intended to be an operating system for a small PDP-11 installation. Typically, in order to use RSX-

11S on a small system, you would also have available a larger system running under RSX-11M or RSX-11M-PLUS. You would do program development on the larger system, taking advantage of the file control services and other conveniences not available on the smaller system. Full compatibility between the two operating systems allows these programs to be transferred to the smaller system. If you have a small installation running under RSX-11S, you will probably use one of the larger forms of RSX to do most of your development anyway. I accordingly will not consider RSX-11S in this book.

RSX-11M or RSX-11M-PLUS may be used as an operating system on any of the PDP-11 series of computers of sufficient size (i.e., model 23 or larger). Micro/RSX is used on the Micro/PDP-11 computer. RSX can be adapted to support any variety of peripheral devices (disks, printers, plotters, analog interfaces, etc.) that might be included in a PDP-11 computer system. The basic parts of the operating system and the commands that enable you to use it are independent of the details of the particular computer system being used. This manual is thus applicable to any PDP-11 computer system being run under RSX.

Digital has upgraded RSX by releasing new versions of the operating system. These are identified by a major and a minor version number. A change in the major version number typically indicates a significant change in functionality whereas a change in the minor version number indicates that only corrections or lesser enhancements were made. As an example, version 2.0 of RSX-11M-PLUS was upgraded by the addition of some minor features to version 2.1. The next upgrade to RSX-11M-PLUS introduced more significant enhancements, and thus was denoted as version 3.0. The version of RSX that is on your system depends on when your system was installed and on whether your company has what is known as a Software Maintenance Contract (which entitles you to free upgrades). It is quite possible that you may not have the latest version of RSX. Then, some of the features discussed in this book will not be available to you. When I present a feature that was recently added to RSX, I identify the version under which it was added. You should accordingly determine which version of RSX is on your system.

Of the three operating systems, RSX-11M is the oldest. RSX-11M-PLUS was introduced as an enhanced form of RSX. The first version of RSX-11M-PLUS was released in 1979, coincident with the release of version 3.2 of RSX-11M. Since then, until the latest release, both products have been upgraded in parallel, with essentially the same improvements appearing in each. Micro/RSX was released in 1984 as a form of RSX for the Micro/PDP-11. You should not let this nomenclature fool you.

The "Micro" refers strictly to the type of PDP-11 system for which it was designed, not to the capabilities of the RSX implementation. Far from being a diminutive form, Micro/RSX was originally more powerful than either RSX-11M or RSX-11M-PLUS. In the January 1986 releases, RSX-11M-PLUS received a major upgrade, primarily to bring its capabilities up to those of Micro/RSX. This indicates a change in emphasis, with RSX-11M-PLUS and Micro/RSX now offering essentially the same features, with the only real distinction between them being the class of PDP-11 on which they run. (To highlight this similarity, the version number of the latest release of Micro/RSX jumped from 1.0 to 3.0 to make it the same as that of RSX-11M-PLUS; there never was a version 2 of Micro/RSX.)

The release history for the three systems is shown in Table 1.

Table 1. Release History of RSX Systems

Released	*RSX-11M*	*RSX-11M-PLUS*	*Micro/RSX*
Jun 1979	version 3.2	version 1.0	----
Jan 1982	version 4.0	version 2.0	----
Apr 1983	version 4.1	version 2.1	----
Apr 1984	----	----	version 1.0
Jan 1986	version 4.2	version 3.0	version 3.0

In this book, I do not consider the extremely old versions of RSX. I treat version 3.2 of RSX-11M (version 1.0 of RSX-11M-PLUS) as a baseline system. (It is very unlikely that your system will be older than this.) I identify any features that were introduced with later versions; when enhancements were simultaneously made to both RSX-11M and RSX-11M-PLUS, I typically identify just the RSX-11M version number. I also identify features that are unique to either RSX-11M-PLUS or Micro/RSX. Finally, if you have version 1.0 of RSX-11M-PLUS, you should see the comments in Chapter 3 concerning DCL.

I will occasionally discuss things which are not an official part of RSX. These are enhancements to RSX that are made available by DECUS, the Digital Equipment Computer Users Society. DECUS is an association of users of Digital products; it is supported by but not controlled by the Digital Equipment Corporation. DECUS has many Special Interest Groups (SIGs); one of these is the RSX SIG. Many useful programs are available (at a nominal charge) from the RSX SIG. Some of these are utilities that provide capabilities that will help you in your use of the system but that are not offered by RSX. Since these are not distributed by Digital, you might not have them on your system. If you do not, you should ask your system manager to get them.

3

MCR vs. DCL

As RSX has grown over the years, various changes and enhancements have been made to it. One of the biggest changes to RSX occurred with the introduction of DCL. Originally, DCL was an enhancement available only with RSX-11M-PLUS. It was introduced with version 1.0 of RSX-11M-PLUS, but was not added to RSX-11M until version 4.0. (DCL has always been available with Micro/RSX.) Prior to these versions of RSX, only MCR existed. What does all this mean?

In any operating system, when you enter a command, you are communicating with what is commonly known as the user interface. In RSX terminology, this is known as the Command Line Interpreter. This is a part of the operating system that interprets your command and either gives it to some other part of the operating system for actual processing, if it is determined to be a valid command, or else rejects it. Since your use of the operating system, especially on a casual basis, centers on the user interface, it is a common mistake to identify the two with each other. When there is only one user interface available, this blurring of identities is harmless. RSX is different from most other operating systems in that it offers you a choice of two user interfaces. The distinction then becomes important to understand.

Regardless of which interface you use, RSX is the operating system, and that is that. This is an important point. Much of our discussion in subsequent sections will be about various aspects of RSX itself (e.g., the file system). Any such discussion will apply to your use of the system, regardless of which user interface you choose to use.

The two available user interfaces or Command Line Interpreters for RSX are known as MCR and DCL. MCR is the Monitor Console Routine;

DCL is the Digital Command Language. MCR and RSX have grown up together; DCL is a recent addition. Of the two, MCR is the more powerful and also the more difficult to master. As an interface, it is aimed more at the experienced system user than the casual user. DCL, on the other hand, was designed to be an easy-to-use, general purpose interface. The name Digital Command Language signifies the effort made by Digital Equipment Corporation to develop a standard user interface that is independent of machine or operating system. (DCL, in almost identical forms, is available on the PDP-11 under both RSX and RT-11, on the VAX under VMS, and on the DECsystem-10 and the DECSYSTEM-20.) With the latest versions of RSX, some minor changes to certain DCL commands have been made to make RSX DCL compatible with VAX/VMS DCL. To preserve continuity with the earlier versions of DCL, these changes have been effected by adding new command forms, which are recommended over the older ones.

Most likely, you will find DCL easier to learn because it uses whole-word commands instead of abbreviations, and prompts you for missing fields should you enter only part of a command. As you become more familiar with RSX in general, however, you may find that this approach is, in places, verbose. MCR employs a much terser command structure (also more cryptic) which allows you do things with shorter commands. The choice of which you, as a user, prefer will be a matter of your own judgment. The old MCR standard still has many adherents, but DCL is undoubtedly the wave of the future.

You should realize that although you have a choice between MCR and DCL, it is not a one-time decision. Any time you want, you may switch from one user interface to the other. (We discuss this further in Section 7.3.) You are not necessarily encouraged to do this, as the difference in command structures leads to a schizophrenic style of computer usage. The significant point is that, initially, you should not worry a lot about whether to use DCL or MCR. Try both for a while, and then make your own choice.

Although you may find DCL preferable, my presentation of various commands will stress the MCR usage. There is a very simple reason for this—MCR is the only interface to the RSX operating system that really exists. That is, RSX understands MCR commands, but it does not understand DCL commands. DCL is actually a shell—it is a barrier erected around the true user interface to shield you from it and its vagaries. DCL translates the commands that you give it into the corresponding MCR commands; these are then processed as though you had directly

typed them in. Since it is MCR commands that actually are processed, if you really want to understand how to use RSX, you should understand MCR.

As we noted earlier, DCL was available on RSX-11M-PLUS before it was available on RSX-11M. With the change from version 3.2 to version 4.0 of RSX-11M, not only was DCL added, but it was also significantly changed. These changes also occurred with the switch from version 1.0 to version 2.0 of RSX-11M-PLUS. Thus, DCL under version 1.0 of RSX-11M-PLUS is somewhat of an anomaly in that it is noticeably different from all other RSX implementations of DCL. Unfortunately, these changes do not concern the addition of new features, which I could easily note when appropriate. Instead, they involve a redefinition of the basic form of the DCL commands. Some of these changes are as "trivial" as the addition of a slash before an optional field in a command. (This may seem trivial to you, but it is not taken so lightly by DCL.) Others change the keywords used within commands to signify various options, while still others change the names of commands themselves. For me to note all these special cases would unduly clutter the discussion. Instead, I will limit my presentation of the DCL command forms to cover only version 4.0 of RSX-11M (version 2.0 of RSX-11M-PLUS) or later. If you have the earliest version of DCL (if so, you have an operating system that is seven years old) you should expect that any command available with version 3.2 of RSX-11M will be available to you, but that the command syntax may be different from what I show. To be safe, you should look up the correct command syntax in the RSX-11M-PLUS Command Language Manual. If this applies to you, I apologize for the inconvenience, but strongly suggest that you solve the problem by upgrading your system.

Since DCL is a shell, it should be clear that there is nothing you can do in it that you cannot also do in the actual MCR command language. (Of course, the DCL commands might be easier, but that is merely a matter of convenience.) On the other hand, depending on how thoroughly the shell is implemented, there may be MCR commands for which there are no DCL counterparts. In this case, there will be things that you can only do in MCR. What you will find is that all the common operations will be possible from DCL; only some of the more arcane features are not provided. Unfortunately, to make all these features available from DCL, it has been necessary to clutter up the DCL command syntax to the point where the advantage it offers over MCR is, in many cases, moot.

As a final point, note that certain commands that you will more com-

monly use (such as those for getting a list of your files, or copying a file) are those that are typically cited as illustrating the advantage of DCL over MCR. It is certainly true that the DCL commands for these functions are much easier and more natural to use. Although you will find these DCL forms preferable, that does not necessarily mean that you will need to choose DCL as your command line interpreter in order to use them. Instead, depending on the details of your system, you may be able to use some or all of these commands directly from MCR. This depends on whether a system generation option known as the Catch All task is part of your system and, if so, on just what facilities it offers. As we noted above, when you enter a command, your Command Line Interpreter first checks to see if it recognizes the command. If it does not, it may reject the command immediately. Alternatively, your Command Line Interpreter may give the command to a special task known as the Catch All task. If Catch All recognizes your command, it will process it. On many RSX systems, Catch All is used as a means of adding the more useful DCL commands to the MCR vocabulary. This is especially true on the later versions of RSX. For example, although MCR does not understand the command **COPY**, the Catch All task can be written so that it will understand it and process it exactly as DCL would. What Catch All offers is the ability to use MCR as the Command Line Interpreter for more complicated commands (where the advantage of DCL is no longer clear) while providing the advantage of the DCL command style for simpler functions. Most likely, a Catch All capability of some sort will be available to you—you should check with your system manager to find out the details.

4

System Generation Options

As you read this book, you will probably be thinking in terms of your particular PDP-11 computer system operating under RSX. Your RSX need not be the same as someone else's, even if both are from the same release. RSX-11M and RSX-11M-PLUS, as supplied by Digital, contain the flexibility to be configured in one of several ways. The decisions implied by this flexibility are made by your system manager during a procedure known as system generation. Of necessity, we will at times discuss certain features that may not apply to your system. These will be identified as system generation options. (I have already mentioned one—the Catch All task.)

Unlike RSX-11M and RSX-11M-PLUS, Micro/RSX is delivered as a pregenerated system. What you receive is a ready-to-use operating system. No system generation is required; no modifications are possible. Typically, Micro/PDP-11 systems have limited disk storage; system generation and customization require much file space and are thus not made available with Micro/RSX. If your system is Micro/RSX, our occasional comments concerning system generation options will not apply to you.

One important system generation option is multiuser protection. This option is normally selected unless the computer system is intended to have very few users. Multiuser protection allows the system to control access to it by a potential user. In a system without this feature, anyone can use the system simply by entering commands at an available terminal. In a system with the protection feature, you must first properly identify yourself (through a procedure known as logging in) before being given access to the system. (Valid identifiers are issued by the system manager.) Once you have obtained access to an RSX system, there typ-

ically is no distinction between a system that supports the multiuser protection feature and one that does not. In particular, despite what the words "multiuser protection" may suggest, the ability of one user to access files belonging to another user or to interact with another user in other ways is the same in either type of system.

RSX systems without multiuser protection were presumably intended for dedicated applications (e.g., a PDP-11 used to monitor and control a manufacturing environment, with terminals located throughout the production area). Since its introduction, RSX has become increasingly popular as a multiuser timesharing system. This change in emphasis is reflected in the fact that on both RSX-11M-PLUS and Micro/RSX, multiuser protection is not an option—it is the only choice. On RSX-11M, it is still an option, but it is unlikely that your system will not have it.

5

User Privilege and Identification

RSX classifies users as either privileged or nonprivileged. Essentially, a nonprivileged user can do all sorts of normal things but none of the fancy things. This distinction was created to protect critical real-time functions from interference by other users.

In a multiuser protection system, individual users are declared by the system manager to be either privileged or nonprivileged. This distinction is noted when you log in to the system and determines your ability to perform privileged functions. In a system without multiuser protection, individual users are not identified and cannot be categorized in this way. In this case, each terminal in the computer configuration is declared to be either privileged or nonprivileged—this distinction is made during system generation. Your ability to perform privileged functions is then determined by the terminal you use—that is, you assume the privilege status of your terminal. For simplicity, we talk of you, the user, as being either privileged or not.

In a multiuser protection system, the privilege option is typically limited to system programmers. If you are reading this manual, you are probably a nonprivileged user. Let it not concern you.

In a multiuser protection system, each person authorized to use the system is identified by a User Identification Code (UIC). The UIC is of the form **[g,m]** where **g** is a group number and **m** is a member number, both of which are octal, between 000 and 377. Group numbers less than or equal to 010 are reserved for privileged users. Two users with UICs having the same group number are said to be in the same group. No significance is attached to UICs that have the same member number.

Only a privileged user (normally the system manager) can assign UICs. When you are given a UIC, you will also be given a password. This

information is maintained in a special file, known as the system account file. When you log in to the system, you must supply a valid UIC and a matching password. You may have more than one UIC, but you can use only one at a time. In addition to being used to control access to the system, the UIC is also used to classify and identify users to the system. The UIC you use when you log in establishes your file access and system privileges—these remain fixed as long as you are logged-in to the system.

6

The RSX File System

As with any modern computer operating system, the basic unit of data storage in RSX is the file. A file is restricted neither in size nor in the nature of its contents. A file is simply a string of bytes. (One byte is eight bits; on the PDP-11 series, this is half a computer word and is the most elemental unit of data storage or transfer.) Most of the bytes in a file are those that are of interest to you; some are inserted by the operating system for control purposes. The latter are transparent to you. A file may contain a source program, an executable task image, text, numeric data, a list of file names, or anything else. It is essentially impossible for you to do anything without referring to a file.

Although RSX shares the common concept of file storage, it has its own unique system for storing and handling files known as Files-11. As all files created under RSX are automatically in Files-11 format, the distinction is seldom a concern. It becomes important only when you wish to transfer files (usually program source files or data files) to or from a computer system that is not running under RSX.

Under Files-11, the basic unit of storage on a disk is the block. The size of a block is 512 bytes. Any file is stored as an integral number of blocks. When you obtain a directory listing (Section 14.1), part of the information that you may get is the size of each file. In some operating systems, a directory listing will show you the size of a file in bytes (i.e., the exact size); in RSX, the size is given in blocks. For example, if a file is shown as being 11 blocks long, you know that its size is somewhere between 5121 and 5632 bytes, but you cannot determine it more exactly.

We will at times talk of devices and volumes. A device is a unit that can write data onto and/or read data from a volume, which is some storage medium. Thus, a disk drive is a device and a disk platter is a

volume. Except for our discussion of backup volumes (Section 23.1), we will not need to be exact in distinguishing between the two. Normally, we will simply talk about a device, allowing that term to subsume the concept of volume as well.

Files may be transferred to any of a variety of devices (disk drive, magtape drive, line printer, terminal, etc.) and stored on various types of volumes (multiplatter disks, floppy disks, magtapes, etc.) Subject to obvious physical limitations, you can have as many files as you wish on any volume. The collection of files belonging to one user on one volume is known as his user area. It is convenient to picture the volume (usually a large disk) as being divided into distinct user areas. (Because files can be created or deleted at any time, this is clearly a dynamic allocation.)

The files within a user area are listed in a user file directory (UFD) which is itself a file on the particular volume. Strictly speaking, a user area is a collection of files, and a UFD is a special file containing a list of the files in a user area. This distinction is a fine one that we will normally not need to make. Since the UFD is a file, it has to have a name and it has to be in some user area. The name of each UFD file is formed from that of the corresponding user area and all UFD files are kept in a special system area known as the Master File Directory.

6.1 File Specification

In order to use a file, you must identify it. This is accomplished by using what is known as a file specifier. An individual file in Files-11 format is specified in one of several ways. In its full form, the specification is

```
device:[ufd]filename.type;version
```

(Note the punctuation shown here; the colon, brackets, period, and semicolon are all part of the required syntax.)

Changing any one of these specification fields is sufficient to specify a completely different file. Thus, to specify a particular file unambiguously, the complete specifier must be used. This can clearly be clumsy; fortunately, it is normally not necessary. Many of the fields in the identifier can usually be omitted, with a typical file reference being of the form

```
filename.type
```

or even more simply,

You must nonetheless understand the purpose and usage of each of these elements.

The **device** is the physical unit on which the file is (to be) stored. (Strictly speaking, files are stored on a volume, but it is to or from the device that the contents of the file are transferred. Following standard RSX convention, we will speak of the device, not the volume, on which a file is stored.) In a large computer system, more than one physical device may be available for file storage (e.g., there may be two disk drives). In this case, since files on one device are independent of files on the other device, it is necessary to specify the particular device to completely specify a file.

Although it is in general necessary to specify the device in order to unambiguously specify a file, you will normally not need to do so. A small system such as a Micro/PDP-11 running Micro/RSX will commonly have only one large disk drive. Even in a large system with many disk drives, you, as a casual user, will normally have only one drive available for your files. In cases such as these, it is unnecessarily complicated for you to have to specify the device every time you refer to a file. RSX provides for this by allowing a default specification. If no device is specified, a certain one is assumed. This default device is known as the system device; its identity is dependent on both the particular computer installation and the user. On a multiuser system, each user has his own definition of the system device; this is determined by the system manager. (The system device should be called the user device since that is what it really is, but the term "system device" is too deeply embedded in RSX to ever change, so you will have to adapt to it.) When the system device is to be used in a file specification, the **device** need not be specified; the colon is then also omitted. When any other device (magtape, terminal, or line printer) is to be used, the appropriate device code (including colon) must be used. We discuss how to specify device codes in Section 6.2.

Once you have specified a device (either explicitly or via default), you must next specify the user area on that device in which the file is (to be) stored. This is effected by the **ufd** portion of the file specifier, which identifies the corresponding User File Directory (UFD). Even if you have more than one user area, you will seldom work with files from more than one area at a time. RSX allows for simplification in this case as well by assigning you a default directory, which is used if you do not specify the **ufd**. When there is no need to specify the **ufd**, it (and the

brackets) may be omitted. We discuss user areas and default directories in Section 6.4.

In most cases, your system device and default directory will be appropriate, so that the **device** and **ufd** are not required. The file identifier can then be abbreviated to

```
filename.type;version
```

Here, **filename** is the actual name, **type** denotes the type of the file, and **version** distinguishes between several files with the same name and type.

The **filename.type** portion of the file specifier is, logically speaking, the most important. Files that have the same file name and type, differing only in the version, are said to be different versions of the same file. The purpose of the version is just what its name implies—whenever you change a file, you make a new version of that file. Each version is a distinct file; the version portion of the file specifier distinguishes between these files.

The concept of maintaining different versions of a file is somewhat special to RSX. It does not exist on older operating systems such as TOPS-10 or RT-11, nor does it exist on popular microcomputer operating systems such as CP/M or MS-DOS. It is a very powerful feature, especially for activities such as program development. For instance, it allows you to keep a set of program changes until you are sure the changes are correct. If necessary, you can undo these changes simply by going backward through the saved versions. The danger here is that this can easily lead to a large accumulation of files within your user area. A simple means of controlling this is described in Section 14.2.

The **version** is a number. In the older forms of RSX, you must treat the version number as an octal value. (Thus, for example, the next version after 7 is 10; there is no version 8.) This use of octal arithmetic directly reflects how the version number is stored by Files-11, but is unnatural from the user's viewpoint. In Micro/RSX, all version numbers are represented as decimal values. In version 3.0 of RSX-11M-PLUS, this capability is available as a System Generation option. (Thus, even if you have version 3.0 of RSX-11M-PLUS, you may still have to represent version numbers in octal—check with your system manager, or experiment to find out.) Decimal version numbers are not available with RSX-11M. The use of decimal version numbers certainly simplifies references to files, and is a welcome improvement to RSX.

The version number can usually be ignored because the system properly chooses default values for it. Nonetheless, you should understand

how this is done. When you first refer to a file (that is, the first time you make a file with some combination of device, ufd, file name and type), a file with these specified values and a version number of 1 is created. Thereafter, every time the file is recreated (a source file is edited, an object file is recompiled, a task file is rebuilt, etc.) another file is created with the next higher version number. Note that the creation of a new version of a file does not affect the current version of the file. When you specify a file with no version number (the semicolon is also omitted), a default version number is assumed. This is

```
for an input file:highest existing version number
for an output file:highest existing version number + 1
```

In either case, you can refer to the latest version of a file without having to keep track of the version number—the File Control Services do this bookkeeping for you. You also can specify a version number of 0 (when you specify no version number at all, a value of 0 is used) to refer to the latest version. Probably the only time that you will need to do this is to delete the latest version of a file, as is discussed in Section 14.2.

In most cases, you will want to refer to the latest version of your file; in this case you can omit the version portion of the file specifier. Then, the specification of a file reduces to the form

```
filename.type
```

In many cases, this is all you need to use as a file specifier.

The file name can never be omitted, as there is never a default for it. It can be from one to nine alphanumeric characters (the letters A–Z and the numerals 0–9). There is no distinction between upper and lower case alphabetic characters. The leading character in the file name can be either a letter or a numeral.

The file type is normally a three-character mnemonic but may more generally be from zero to three characters long. The type is normally set to a standard value to signify the nature of the file. The most common file types are listed below:

BAS a BASIC language source file
B2S a BASIC-PLUS-2 language source file
CBL a COBOL language source file
CMD an indirect command file
DAT a data file
FTN a FORTRAN-4 or FORTRAN-77 language source file
LST a listing file

MAC a MACRO-11 Assembler source file
OBJ an object (compiled source) file
OLB an object library file
TSK a task image (executable program) file
TXT a text file

Most system utilities or commands assume certain default file types. The type need not be specified when it is the same as the default for a particular command. In this case, the period is also omitted. This leads to a subtle distinction. Omitting the type but not the period does not specify a default file type. Instead, it specifies a null type. This is syntactically valid, but unless specifically desired, is a common source of error.

As an example of the use of default file types, the FORTRAN compiler reads an input file containing source code and produces an output file containing object code. The input file is assumed to have type FTN and the output file is assumed to have type OBJ. Thus, you may use the simplified file specifier TEST for both the input and output files; the input file specifier will be interpreted by the FORTRAN compiler as TEST.FTN and the output specifier as TEST.OBJ. You can, however, override either of these defaults by specifying the file type. Thus, if you specify TEST.OLD as the input file, it will be used by the FORTRAN compiler, even if a file TEST.FTN exists.

As an example of the use of defaults for file types and versions, consider execution of the following commands:

```
EDIT TEST.FTN
FORTRAN TEST
EDIT TEST.FTN
FORTRAN TEST
```

(These commands are in DCL form; we will worry about their exact structure later.) Assume that you have already made version 1 of the file TEST.FTN, that is, in your user area you already have the file TEST.FTN;1. The command **EDIT** uses your favorite editor to edit a file. In the first command above, the input to this editor is the file TEST.FTN;1, which stays unchanged; the output is a new file, TEST.FTN;2. The command **FORTRAN** uses the FORTRAN compiler to compile a source file. In the second command above, the input to the compiler is the file TEST.FTN;2, which stays unchanged; the output is a new file, TEST.OBJ;1. The third command performs another edit, resulting in the new file TEST.FTN;3. The last command compiles this file, producing as output the new file TEST.OBJ;2. At this point, a di-

rectory listing of all files with the name TEST would show the five files listed below:

```
TEST.FTN;1
TEST.FTN;2
TEST.OBJ;1
TEST.FTN;3
TEST.OBJ;2
```

It is often useful to refer to more than one file at a time. This may be done by using a wildcard in place of one or more of the file specification fields. This is not the same as using a default. Omission of a specification field directs the File Control Services to accept the default value for that field; use of a wildcard as a specifier directs it to accept any value for that field. A wildcard is specified by an asterisk. Wildcards are typically used in place of the file name, type, and/or version identifier. As an example, the file specification **TEST.FTN;*** specifies all files with a name of TEST and a type of FTN, regardless of version number. The specifier ***.FTN;*** may be used to refer to all FORTRAN source files. You may use a wildcard and a default together. For example, **TEST.*** specifies the latest version of any file with name TEST, regardless of file type. Similarly, **TEST.*;*** specifies all versions of any file with name TEST. (This specifier would have been used to obtain the list of file specifiers shown in the above example.) Wildcards are a very powerful feature—once you get used to them, you will use them for many purposes, such as obtaining a directory listing, and transferring, deleting, or purging files.

6.2 Devices

RSX allows you to refer to a number of possible devices. These may be either peripheral or pseudo devices.

A peripheral device is an actual input/output (I/O) unit. A peripheral device is designated in the form **ddnn:**, where **dd** is a two-letter device code, **nn** is an (optional) device number, and the colon (:) is required. The device codes you are likely to use are:

Terminal (any type)	TT
Line printer	LP
Magtape unit	MM, MT, MS

Floppy disk	DX, DY
Large disk	DB, DP, DR

Your system manager can tell you which devices are in your system. Device numbers are octal and start at zero. A reference to a peripheral device with no numeric identifier uses zero as a default value. For example, the device code for a line printer is LP. In a system with only one line printer, the complete device name is LP0:, but LP: is a sufficient designator. Similarly, two line printers would be identified as LP0: and LP1:. In our examples I will show the device number as **nn**, implying that it is a two-digit number. Actually, it may be as large as 377 octal. Unless your system has a very large number of terminals, you will not need a three-digit device number.

A pseudo device is not a physical unit; rather, it embodies the concept of I/O. Actual peripheral units are assigned to pseudo devices by the system. Thus, a reference to a pseudo device is translated into a reference to an actual unit and in these cases obviates the need for you to know the peripheral device codes. A pseudo device is designated as **xx:** where **xx** is a two-letter code. The pseudo devices that you are most likely to use are listed below. Of these, you will find **TI:** and **SY:** to be the most useful.

User terminal	TI:
User default device	SY:
System default (library) device	LB:
Null device	NL:
Console listing device	CL:

The pseudo device **TI:** is the pseudo input terminal. TI: refers to the particular terminal you are using. Different users may each refer to the pseudo input terminal; the system translates each reference into the correct terminal code and no confusion results. On the other hand, to use the actual peripheral device code, you must know whether the terminal you are using is known to the system as TT0:, TT1:, etc.

The other commonly used pseudo device is the user default device, SY:. This is the device on which your files are stored; it will be a disk drive of some sort. In a large installation, there may be several disk drives. In this case, the system will equate SY: to a particular one of these. If your system has multiuser protection, this assignment will be made when you log in to the system; SY: will then be set to be the disk on which your files are stored. On a system without multiuser protec-

tion, this automatic assignment of SY: is not possible, and you may have to assign SY: yourself. We discuss how to do this in Section 10.2.

Strictly speaking, whenever you refer to a file, you must include a device code in the file specifier. If you omit the device specification, SY: will be assumed by default. By having SY: assigned to the disk containing your files, you may refer to your files without having to bother with specifying a device. For this reason, we will often refer to SY: as being your system disk. Note that some other user will also have an SY: which will be his system disk; this need not be the same as yours.

The system default or library device is the device on which operating system files are stored. As with the user default device, this will be a disk drive of some sort. On a small system, SY: and LB: will refer to the same disk drive. On a large system, they will typically refer to different devices. You may occasionally need to refer to a file that is part of the operating system, hence this distinction should be noted. Normally, however, you will not need to use LB:.

Although SY is a mnemonic for "system," you should remember that SY: is the user default device, not the system default device. This is an easy source of confusion. The common use of "system disk" for SY: adds to this confusion. It may help you to think of the system device, LB:, as a mnemonic for "library," since one of the common reasons for referring to it is to access system library files. The choice of some other pseudo device name, such as US for user, would have been nicer, but so it goes.

The null device (NL:) does not exist. It allows you to have a syntactically valid input or output statement without performing any actual I/O operation. A write operation to the null device effectively aborts output without interfering with program execution. This is very useful in testing a program, as it can be used to direct portions of the output to a temporary void. Once the program has been verified, the output can be easily reassigned to an actual destination (e.g., LP:). The null device is often referred to as the bit bucket to indicate this deliberate loss of output. The null device can also be used for data input. In this case, a read from NL: returns an End-Of-File (EOF) indicator instead of any actual data. We discuss a common use of this device in Section 25.1.

The console listing device (CL:) is the default unit for certain types of printed output. It is normally assigned to a line printer; in a system with no printer, it may be assigned to a DECwriter or other hard-copy terminal. As with other pseudo devices, you may use CL: without knowing which peripheral device it is assigned to. Due to the use of print spooling (Chapter 15), however, you will normally neither need nor want to refer to CL:.

6.3 Logical Units

Ultimately, all I/O operations must refer to an actual peripheral device (except those that refer to the null device). As discussed in the previous section, it is possible to delay the choice of peripheral device by first referring to a pseudo device. For user programs, RSX provides yet another level of indirectness. In an I/O statement, no direct reference to any device is made. Rather, the I/O operation refers to a logical unit, which is identified by a decimal number in the range 0 to 255. The logical unit number (LUN) is later assigned (when your task is built) to either a peripheral or pseudo device. By referring to logical units, you may make your source program device independent. Under RSX, certain assignments are assumed by default. These are:

LUN 1 – 4 = SY:
5 = TI:
6 = CL:
7 or greater—not used

In many cases, the default assignments will be adequate for your needs. The multiple levels of assignment from logical unit to actual device may at first seem unduly complicated. When properly used, however, this interaction can provide a great degree of flexibility. We discuss this in greater detail in Section 25.1.

6.4 User Areas and Directories

As an RSX user, you may have many files on a particular disk volume. The collection of these files forms what is known as your user area. The association of all the files in this user area with you is effected through the User File Directory (UFD). A UFD is a special file maintained by RSX that contains the names of all the files in a user area, along with related information, such as where on the volume they are, how big they are, and when they were created. For each user area on a volume, there is a corresponding UFD; there is a simple one-to-one relation between these, based on the name of the user area. Strictly speaking, a user area is a collection of files and a directory is a file containing a list of all the files in a user area. The two concepts, however, are so closely interrelated that they are often used synonymously.

Since a user area is intended to contain files belonging to one user, that user is known as the owner of the user area. As we discussed in Section 5, each user is identified by his UIC, which is a two-part number

of the form **[g,m]**. Traditionally, UFDs have been identified in the same style as UICs. A UFD that is identified in this way (that is, by a group and a member number) is known as a numbered directory. Micro/RSX introduced a new concept, named directories, wherein the user area is identified by an alphanumeric name. This has also been made available in version 3.0 of RSX-11M-PLUS. A named directory is identified by a name of up to nine characters. (The rules for choosing this name are exactly the same as those for choosing a file name.) If your system does not support named directories, you must use numbered directories. If your system supports named directories, you can use either numbered or named directories. This allows compatibility among different RSX systems.

When you are entered as an authorized user into the system account file, you are assigned a user area in addition to your UIC. (As we discuss in Section 21.3, you may later obtain other user areas.) If your system has only numbered directories (any RSX-11M system and all RSX-11M-PLUS systems prior to version 3.0), this initial user area will be identified by your UIC. When you are given other user areas, they will typically have the same group number as your UIC, but different member numbers. If your system has named directories, you may choose any name you wish for your initial user area. You may similarly choose any name for subsequently assigned user areas.

Whenever you log in to the system, a user area is selected for you. This defines your default directory, which is used whenever you omit the **ufd** portion of a file specifier. (In Section 10.2, we discuss how to redefine your default directory.) This initial choice of default directory is the user area that was originally assigned to you when you were entered into the system account file. This is independent of whether you have numbered or named directories.

For example, suppose your UIC is [110,1] and your name is Tolman. If your system has numbered directories, your first user area will be identified as [110,1]. Subsequent user areas will also be in group 110, such as [110,2], [110,100], or [110,377]. Whenever you log in, your default directory will be set to [110,1]. If your system has named directories, you may request that your first user area be [TOLMAN], or [TOLMAN001], or [SYLVIA], or anything else you wish. Suppose you choose [TOLMAN] as your initial directory. You might later obtain directories called [T2], [PLOTTER], [TROPO], etc. Whenever you log in, your default directory will be set to [TOLMAN].

I stated earlier that a UFD was a file. As such, it has to have a name. For a numbered directory, the UFD is given the six-character name

gggmmm where ggg and mmm are the group and member numbers, each represented as three octal digits. For a named directory, the UFD is given the name of the user area. In either case, the file type of the UFD is DIR. In the examples above, the UFD names would be 110001.DIR and TOLMAN.DIR. Since a UFD is a file, it also must be in some user area. All UFDs are in a special system area, known as the Master File Directory (MFD), which is identified as [0,0].

7

Command Line Formats

Anything that you, as a casual user, do on a computer system is effected by issuing commands to its operating system. The things that you are allowed to do and the manner whereby you request them may differ drastically from one system to another, but, in general, you will always request a particular type of action, perhaps define what is to be acted upon, and optionally include some finer definitions of the action. For example, you cannot simply print; you have to print a particular file (or possibly several files), and instead of automatically getting one copy, you may further specify how many copies you want printed.

The commands that you may issue to RSX depend on which of the two command line interpreters, MCR or DCL, you are using. The general command style is quite distinct for these two user interfaces. In this chapter we look at the basics of the command syntax for each.

Before we go any further, you should realize that there is nothing magical about an operating system (not even RSX). It is a computer program, admittedly a big, complicated one, or, more precisely, a big collection of complicated programs, but that is all. When you ask RSX to do something for you, it first determines which program, or task, is the one designed to do what you have requested. Assuming that you have not asked for something undefined, RSX then gives the particulars of your request to the responsible task, which does what you requested. When one task does only one type of thing, this division of labor is very straightforward. When, however, one task is capable of performing many different functions, things are not so simple. All this leads to what is probably the biggest difference between MCR and DCL commands—the names of the commands themselves. In DCL, the command name is chosen to describe the action you request, whereas in MCR the com-

mand name is the name of the task that will actually do it. Presumably, you will want to think in terms of the operation to be done, but RSX needs to think in terms of the task required to do it for you. Thus, we can see that DCL was designed around the convenience of the user, while MCR was designed around the convenience of RSX.

7.1 MCR Commands

A typical MCR command directly names the task that is required to do whatever it is you want done. In some cases, there is an obvious relationship between the task name and the action, but in others (most noticeably, the file maintenance operations offered by PIP), there is not. Whether the name of the command itself makes sense to you or not, you will find that there is a common syntax to the rest of the command line.

In MCR, a command to the operating system consists of a command name, which identifies in a rather broad sense what is to be done, and normally one or more file specifiers, which identify the various files that are to be used. Some commands refer to no files at all. (A simple example of this is the command to find out what time it is.) Others refer to only one file or one group of files. (Some common examples are the commands to delete and print file(s).) For these commands, regardless of the command action, we will refer to the file(s) being acted upon as the input. Finally, there are those commands that refer to two (groups of) files. One of these will be known as the input to the command, the other as the output of the command. MCR has a well-defined syntax for this:

```
command output=input
```

You will find this syntax used in a variety of situations—copying a file, compiling a file, task building, etc. Although it might seem strange (it is somewhat counterintuitive) at first, this syntax is best remembered as an equation or as an assignment in a typical programming language—the output file(s) is set equal to the input file(s). Learn it well; you will see it often.

In many cases, an MCR command will offer you a variety of choices or options from which you may select. These are used to more finely specify what is to be done. These options may specify some detail of the file(s) to be used, or they may relate to the entire nature of the command. In MCR these finer details are specified via switches.

Switches are entered into the command line in the general form **/switch**, where the slash is syntactically required to identify the construct as a switch. In the basic format, the switch is denoted by a two-character switch code. (In some cases, the switch code is longer than two characters, but normally it is not.) In this form, you are requesting that the action identified by the switch be done. If you do not want this done, you negate the switch. There are two possible forms for this, **/-sw** or **/NOsw**. It is possible that once you have specified a switch, you may also have to specify a value to completely define the action. This is done in the form **/sw:value**—the switch code is followed by a colon and then the value. The value may be a number or a keyword. When the value is numeric, it will normally be interpreted as being octal unless you terminate it with a period, which declares it to be a decimal value. Unfortunately, MCR is not consistent on this; some commands assume switch values to be decimal.

Depending on the detail being specified, the switch is appended to either the appropriate file specifier or the command name itself. The older RSX commands use only file switches; some of the newer commands also allow command switches, reflecting the DCL influence. In a few cases, a switch may be used as either a command or a file switch. When this duality is possible, the distinction is typically made on the basis of whether the action specified affects all files (a command switch) or just one file (a file switch). You may want to specify more than one switch in a command. If two or more apply to the same item in the command (the command name for a command switch or a file specifier for a file switch), they are all appended to it, one after the other (in any order), each preceded by the mandatory slash that identifies it as a switch.

To examine some examples of how you do this, we will consider the FORTRAN-77 compiler. The MCR command to use it follows the general form given earlier of output = input. In addition to including an object file, the output can also specify a listing file. The general command form is

```
F77 obj,list=source
```

If you have a source file TEST.FTN and you wish to make an object file TEST.OBJ and a listing file (the default file type here is LST) TEST.LST, you would use the command

```
F77 TEST,TEST=TEST
```

When you generate a list file, you may request that it be automatically

printed by using the Spool switch, which is **/SP**. In this case, your command would be

```
F77 TEST,TEST/SP=TEST
```

Note that the Spool switch is appended to the specifier for the list file. Returning to our first example, when you ask for a list file, it will be generated in a certain format. You can use the List switch to modify this format. This switch is **/LI:value**, where **value** specifies the format you want. For FORTRAN-77, a value of 3 requests a listing with the greatest amount of detail. If this is what you want, your command will be

```
F77 TEST,TEST/LI:3=TEST
```

When a switch is possible in a command line, you do not necessarily have to specify that you want it (**/sw**) or that you do not want it (**/-sw**). Every switch has a default value that is used unless you override it. When we discuss the various switches possible for a command, I will state what the default assumptions are. Normally, these defaults are rigidly defined by RSX. Some of them, however, can be changed during system generation; in these cases I note that the default is system dependent. Let's consider again the Spool switch for a FORTRAN listing. The default is normally to spool the file. Some users make a listing file whenever they compile; automatic printing of these results in a lot of wasted paper. Thus, on your system, the default may have been changed to No Spool. If you are not sure of the default value for a particular switch, you can always specify what you want—the worst that will happen is that you will have typed a few characters more than are necessary in the command. Continuing with our example, if you want a listing with full detail but you do not want it to be printed, you would use both the Listing:Type 3 and the No Spool switches. Note that one switch directly follows the other (the order is immaterial), since both of these switches are appended to the listing file specifier:

```
F77 TEST,TEST/-SP/LI:3=TEST
```

A command such as the above may strike you as looking rather strange. Nonetheless, there is an indisputable logic underlying the format of the MCR commands. Once you become familiar with this, it will be easier to learn new commands than you might expect. Have faith—you will soon be entering commands with a multitude of switches just like an old pro.

7.2 DCL Commands

A typical DCL command directly names the action that you want done. The fact that the task required to do this may have an entirely unrelated name is a detail that is hidden from you by the DCL interface. This typically makes commands (especially the simple ones) easier to learn in DCL than in MCR. As a further help, you will find that there is a common syntax for all DCL commands.

In DCL, a command to the operating system consists of a command name, which identifies in a rather broad sense what is to be done, and normally one or more file specifiers, which identify the various files that are to be used. Some commands refer to no files at all. (A simple example of this is the command to find out what time it is.) Others refer to only one file. For these commands, regardless of the command action, we will refer to this file as being the input. Finally, some commands refer to two (groups of) files. One of these will be known as the input to the command, the other as the output of the command. It is for these commands that the DCL forms differ most from their MCR counterparts.

A DCL command that needs to refer to both input and output files will use one of two possible syntaxes. Some commands (such as the command to copy files) have a command form that assumes arbitrary specifiers for the input and output files. This form has the general syntax

```
command input output
```

Perhaps this reflects more closely the way that you think than does the MCR syntax. You should remember it as "from–to"—that is, the command goes from the input file(s) to the output file(s). Perhaps your biggest problem with this syntax will be the fact that it is the opposite of that used by MCR (at least for commands that use two files). If you are used to MCR and wish to switch to DCL, you will have problems for a while due to this reversal.

Other DCL commands (such as those for compiling a program) do not so readily offer the flexibility of this form. Instead, they use a form that assumes the output file to have a file specifier that is determined from that of the input file. For these commands, you do not normally specify the output file(s)—they are implicit to the command. When you accept these defaults, the output file specifier is formed for you as part of the translation into the MCR command form. This general command form is the following:

```
command input
```

If you want to use a different output file specifier, you have to use a special switch followed by the desired file name:

```
command input/switch:output
```

This illustrates an interesting aspect of the difference between MCR and DCL. For simple operations, where you do things in the "normal" manner, the DCL command form is simpler than the MCR form, but for other operations, where you do something that is not expected, the DCL command form is more complicated.

Just as in MCR (don't forget that DCL is translated into MCR), a command will in many cases offer you a variety of choices or options from which you may select. These are used to more finely specify what is to be done. These options may specify some detail of the file(s) to be used, or they may relate to the entire nature of the command. In official DCL terminology, these finer details are specified via qualifiers. Since DCL qualifiers translate directly into MCR switches, and to avoid alternating terminology when we discuss the format of a command under both MCR and DCL, we will typically refer to them as switches.

Switches are entered into the command line in the general form **/switch**, where the slash is syntactically required to identify the construct as a switch. In this form, you are requesting that the action identified by the switch be done. If you do not want this done, you negate the switch. There are two possible forms for this, **/-switch** or **/NOswitch**. It is possible that once you have specified a switch, you may also have to specify a value to completely define the action. This is done in the form **/switch:value**—the switch code is followed by a colon and then the value. The value may be a number, a keyword, or a file specifier. When the value is numeric, it will be interpreted as being decimal, which matches the way you think. Since the underlying MCR command will often want the value to be in octal (or may require a period to declare the value as being decimal), DCL will, depending on the exact command, translate the value for you.

In DCL, most switches may be placed either after the command (a command switch) or after a file specifier (a file switch). Some switches must be placed after the command as they are a part of the command name itself. Some switches have meaning only when used as a file switch. In a few cases, a switch may be used as either a command or a file switch, with different results being obtained. When this duality is

possible, the distinction is typically made on the basis of whether the action specified affects all files (a command switch) or just one file (a file switch). You may specify more than one switch in a command. If two or more apply to the same item in the command (the command name for a command switch or a file specifier for a file switch), they are all appended to it, one after the other (in any order), each preceded by the mandatory slash that identifies it as a switch.

In DCL, the name of a command or switch is an entire word, or even several words connected by underscores. This reflects DCL's use of the English language rather than MCR's use of abbreviations. Although easy to understand, long names are often annoying to enter. Thus, DCL allows you to shorten these commands or switches when you enter them. The DCL rule for abbreviating names is simple—you need only maintain enough initial characters of the word to distinguish it from any other word that DCL recognizes. Typically, the first three characters are sufficient for this; sometimes fewer and sometimes more are required. For example, **LIB** is recognized by DCL as an abbreviation for **LIBRARY**, but **LI** is not since it could also be an abbreviation for **LINK**. When I present DCL commands, I normally give the full form as well as typical abbreviations.

An important feature of DCL is command prompting. In DCL, you do not have to enter an entire command at once. Instead, you may enter it in pieces. All you have to do is enter (at DCL command level) the name of the command. DCL then prompts you for the remaining portions of the command. Note that you are prompted only for the mandatory parts of a command. If the command refers to only one file and you enter just the command name, you will be prompted for the input specifier; if the command refers to two files, you will be prompted for the input and then the output specifiers. You will not, however, be prompted for any switches (unless you end an input line with a slash). You may enter a switch as part of the command line or as part of your response to either the input or output prompt. Finally, if the command refers to two files, you may enter the command name and the input file specifier on one line; you will then be prompted only for the output specifier. At first, you may find command prompting useful. With it, you do not have to remember the syntax of the command. This is, however, a somewhat dubious advantage, since one of the main purposes of DCL is to offer an easy-to-remember command syntax. Also, once you become familiar with the various commands, you will find the prompting method slower than simply entering the whole command at once.

To examine some examples of how you do all this, let's consider the

FORTRAN-77 compiler. The DCL command to use it is one in which the output file specifiers are normally formed from the input specifier. In the simplest command form, you only need to name the input file, which, for this command, is the file containing your FORTRAN source code. The simplest command is

```
FORTRAN/F77 source
```

Here we use the full command name **FORTRAN**; it is more common to use the abbreviation **FOR**. Note the use of the command switch **/F77**. This must be a command switch, as it is actually part of the command name. Without this switch, the FORTRAN-IV compiler would be used instead of the FORTRAN-77 compiler.

In the form above, an object file is produced as output (since no name is specified, defaults are used), and no listing file is produced (because none was asked for, the default is to not make one). Thus, if you have a source file TEST.FTN and you wish to make an object file TEST.OBJ, you would use the command

```
FOR/F77 TEST
```

If you also wish to make a listing file (the default file type here is LST) named TEST.LST, the command would be

```
FOR/F77 TEST/LIST
```

Here, the switch **/LIST** specifies that a listing file should be generated. This switch may be either a file or a command switch, i.e., the above command could instead be entered as

```
FOR/F77/LIST TEST
```

These two examples are not identical; there is a difference concerning whether the listing file will be automatically printed or not. (We discuss this further in Section 16.1) This is one of the few examples in DCL of a switch whose meaning changes with its placement.

Returning to our first example, when you ask for a list file, it will be generated in a certain format. You may append other switches to the List switch to modify this format. The switch is then **/LI/format**, where **format** specifies the format that you want. (Note that we have again taken advantage of the ability in DCL to abbreviate command names or keywords by shortening **LIST** to **LI**.) A format code of **MACHINE_CODE**, typically shortened to **MACH** or **MAC**, requests a listing with the greatest amount of detail. If this is what you want, your command

will be

```
FOR/F77 TEST/LI/MAC
```

Under DCL, you may also use the FORTRAN-77 compiler in the prompting format. An example of this is

```
DCL>FOR/F77
File(s)?TEST
```

When a switch is possible in a command line, you do not necessarily have to specify that you want it (**/sw**) or that you do not want it (**/-sw**). Every switch has a default value that is used unless you override it. When we discuss the various switches possible for a command, I will state what the default assumptions are. Normally, these defaults are rigidly defined by RSX. Some of them, however, can be changed during system generation. In these cases, I note that the default is system dependent. If you are not sure of the default value for a particular switch, you can specify what you want—the worst that will happen is that you will have typed a few characters more than are necessary in the command.

We have already noted that since all DCL commands are translated into MCR commands, there is nothing that you can do in DCL that you cannot also do in MCR, but that the reverse is not true. This is also true for command line switches. Some switches that are accessible to you from MCR are hard-wired to certain values in DCL. An example of this is the switch in MCR for spooling the listing file made by FORTRAN-77. In our discussion of MCR command syntax in Section 7.1 we saw how you could make a listing file of arbitrary name and direct that it be spooled or not. In DCL you do not have this full capability, as there is no switch to control spooling directly. When you include the List switch, the resulting MCR command has the Spool switch set for you automatically. (Whether it is on or off and whether you can choose other than the default listing file name depends on where in the DCL command you include the List switch.) There are other examples as well (see, for example, Chapter 13). In general, however, most useful MCR switches will have direct DCL counterparts.

In this and the previous section you have seen a brief overview of MCR and DCL. From this, you have probably concluded that DCL commands are more meaningful and hence easier to learn, remember, and use. For simple operations, this is true. Do not take it to be the entire story. (Further, do not forget the Catch All capability that may be available to you from MCR.) Once you get past the simple operations, you

will find the DCL's advantage to be much less obvious. Certain DCL commands will, in fact, look stranger than their MCR counterparts. Read on!

7.3 Changing Your CLI

When you log in to your system, you will be given a Command Line Interpreter (either MCR or DCL) by default. The choice here will be made based on a decision made by your system manager when he set up your log-in account. (If your system does not have multiuser protection, you will get whatever CLI the last user on your terminal was using.) You are, however, in no way obligated to stick with this choice. Instead, you may change from MCR to DCL or from DCL to MCR whenever you want. As noted earlier, you should not do this frivolously, as the differences in command syntax lead to what is known as computer schizophrenia. There may, however, be times when it will be more convenient to change your CLI, or you may wish to try both to help you decide which one you like better.

Before going further, we must note an interesting point. In all forms of RSX, MCR is the part of the operating system that actually processes your commands. In both RSX-11M and RSX-11M-PLUS, MCR is fully supported as a CLI. In Micro/RSX, however, DCL is the only officially supported CLI. MCR is still there of course, but you are not expected to communicate with it directly. Indeed, version 1.0 of Micro/RSX was implemented in such a way that you could not give commands directly to MCR—you had to use DCL as your CLI. With version 3.0 of Micro/RSX, this has been corrected so that you may give commands to MCR. Nonetheless, Digital does not officially support the use of MCR with Micro/RSX. Further, if you are using DCL on Micro/RSX, you cannot easily change your CLI to MCR. If you want to use MCR on Micro/RSX, you have to have MCR set as your default CLI when you log in. (Your system manager can do this for you.)

On all forms of RSX, you may change your CLI from MCR to DCL. On RSX-11M and RSX-11M-PLUS you may similarly change your CLI from DCL to MCR. You may change your CLI by using the Set command. We will study various forms of the Set command in Section 19.2. For now, we simply present the commands that you need and ask you to bear with their rather strange syntax. If you are in MCR and wish to change to DCL, you use the command

```
MCR: SET /DCL=TI:
```

If you wish to switch from DCL to MCR, the corresponding command is

```
DCL: SET MCR
```

(I repeat that this will not work in Micro/RSX.) Note that the changes effected by these commands are temporary. The next time you log in to your system, you will again get your default CLI. If you wish to make a more permanent change, you will have to ask your system manager to change your default CLI as defined in the system log-in accounts file.

On an even more temporary level, you may change your CLI for one command. That is, if you are in MCR, you may enter a command in DCL syntax without actually changing your CLI to be DCL; you may similarly enter a single MCR command while in DCL. You do this by placing the name of the appropriate CLI in front of the command that you wish to execute. If you are in MCR and you wish to execute a DCL command, the form is

```
MCR: DCL dcl_command
```

This might be useful in avoiding some of the more clumsy MCR constructs. Similarly, if you are in DCL, you may execute an MCR command in this manner:

```
DCL: MCR mcr_command
```

For executing an MCR command from DCL, an alternative exists; you may preface the command with a period

```
DCL: .mcr_command
```

As we will later see, certain MCR commands do not have direct DCL counterparts; these command forms allow you to execute them from DCL. Note that both of these forms are available to you in Micro/RSX. If you log in to a Micro/RSX system and your default CLI is DCL, this will be the only way you will be able to execute MCR commands directly.

I have stated that DCL commands are translated into their MCR counterparts. Related to the switching from MCR to DCL (or vice versa) is the investigation of what the MCR commands are that result from DCL commands. If you are in DCL, you may determine this by using the DCL debug mode. When this mode is active, the MCR translation of each DCL command is displayed to you. The debug mode is enabled

by another form of the Set command. You may use one of two possible commands. If you enter

```
DCL>SET DEBUG
```

all subsequent DCL commands will be translated (the MCR commands will be displayed) but none will be executed. If you use the command

```
DCL>SET DEBUG/EXEC
```

after the translation into MCR is displayed, the command will be executed normally. In either case, you can disable debug mode with the command

```
DCL>SET NODEBUG
```

Debug mode is particularly useful for learning just how DCL works or for learning how to switch from DCL to MCR. If you are new to RSX, however, I recommend that you do not experiment with this—you will have enough to learn as is.

Part II Using the System

8

User/System Interaction

You will normally use the RSX system in an exclusively interactive manner. Your input to the system is entered via your terminal; output from the system may be directed to your terminal or to any of the other peripheral devices in the system configuration. Ignoring time delays due to other users, commands are effectively processed and executed immediately. It is also possible to use the system on a delayed basis. This might be done to schedule execution of a lengthy task for overnight or another slow period. This is discussed in Section 25.1. In the remainder of this section we consider the more typical interactive use of the system.

Interaction with the system is performed via a special task, known as the Command Line Interpreter (CLI). As I have already noted, version 4.0 or later of RSX offers two CLIs, MCR and DCL. (It is also possible to have additional CLIs; this is a system generation option. If your system does have a third CLI, it will be specific to your system, and you will have to ask your system manager about it.) Although there are significant differences between MCR and DCL, the concepts that we discuss here are common to both; to accentuate this, we will refer to a generic CLI. The CLI interprets your commands and schedules the appropriate system response. For example, suppose you have a task named A.TSK that you wish to run. You can do so by entering the command **RUN A**. (Note: this particular command has the same format for both MCR and DCL.) The CLI reads this line, interprets the command **RUN**, and causes the file A.TSK to be located, read from disk, and loaded into memory. RSX then passes control to this task. When task A terminates, control is returned to RSX, which then reactivates the CLI. At this point, the CLI is ready to process another command from you.

The passing back and forth of control brings us to an important point: under normal conditions, only one task can have control of a terminal. Here, by "task," I mean a user task or system program. In the previous example, task A may request you to enter the value of some parameter. Were the CLI to continue reading terminal input after the Run command, two different programs would be attempting to read your input from the terminal. This would lead to unpredictable results, with one program sometimes reading input intended for the other. The manner whereby this problem is avoided is somewhat complicated. Basically, if your task is waiting for input, anything that you type in will go to the task. If your task is not waiting for input, however, whatever you type in will go to the CLI. In either case, you normally can force your input to go to the CLI by using the system control character (Section 9.3). Thus, you can enter a new command before the previous one has finished.

What happens when you overlap commands in this manner? RSX has traditionally processed multiple commands in parallel. That is, a command is acted upon as soon as it is entered, even if the previous command has not finished. You could, for instance, type in a Compile command; before the compilation was finished, you could then type in some other command, such as one to edit a file. Then, while the compilation was proceeding, you could also be editing your file. As far as you are concerned, this allows you to do two things at once, although, of course, the computer is always doing only one thing at a time.

With versions 3.0 of Micro/RSX and RSX-11M-PLUS, this traditional parallel behavior has been modified by adding serial mode as a new way of responding to overlapped commands. With serial mode, one command is not started until the previous one has finished. This serial processing is available as an option to the traditional parallel form of command processing. When you log in to the RSX system, you will start in either the parallel or serial processing mode. In Micro/RSX you will be put into serial mode by default. In RSX-11M-PLUS you will be put into parallel mode by default. (In RSX-11M or in older versions of Micro/RSX or RSX-11M-PLUS you have no choice; serial mode is not available.) You may change from one mode to the other by using the Set Serial command. To go from parallel to serial mode, the command is

```
MCR: SET /SERIAL=TI:
DCL: SET TERM/SERIAL
```

Similarly, to go from serial to parallel mode, the command is

```
MCR: SET /NOSERIAL=TI:
DCL: SET TERM/NOSERIAL
```

The ability to enter multiple commands is one of the more powerful features of RSX, but it also offers significant potential for confusion. Until you become more familiar with using RSX and either MCR or DCL, you should not use this feature. If you have version 3.0 of either Micro/RSX or RSX-11M-PLUS, you might want to disable the traditional multiple command processing by setting your terminal to serial mode until you have gained more experience with RSX.

The CLI displays a prompt to you when it is ready to accept input. If you have used the system control character to call it, an explicit prompt (consisting of the name of the CLI followed by a greater-than symbol) is displayed. This is either MCR> or DCL>. Normally, you do not have to forcibly get the CLI's attention. If you simply wait for one command to finish, you will be given a default prompt when the CLI is ready to process your next command. In MCR and in the older versions of DCL, this default prompt consists of just the >. In the newest versions of DCL (version 4.2 of RSX-11M, version 3.0 of RSX-11M-PLUS, and version 3.0 of Micro/RSX) the default DCL prompt has been changed to a dollar sign ($). (As with other changes to the newest versions of DCL, the motivation is compatibility with DCL on VAX/VMS.)

When you run a system utility (other than the CLI) in an interactive manner, it will give you its own prompt, which is uti>, where uti is the three-character utility name. In some of our examples, I will show command sequences where you get prompts from both the CLI and a utility. In this case, I will show the explicit CLI prompt for clarity, although you might actually get the default prompt.

Throughout the rest of this book, I will discuss various ways in which you may interact with or use the RSX operating system. Of necessity, these discussions will depend on which Command Line Interpreter—MCR or DCL—you are using. Where things are common to both, I will simply use CLI to refer to a generic Command Line Interpreter. Where something is specific to either MCR or DCL, I will identify it as such. When I state the format of a command, I will show it as

command

when the format is the same for both MCR and DCL. When the formats are different, I will show them as a pair of commands in the form

MCR: **MCR command**

DCL: **DCL command**

When I show direct interaction with the CLI, e.g.,

MCR>**command**

I will rely on the CLI prompt to resolve any possible ambiguity. Finally, there will be times when the use of DCL will be either impossible or inappropriate. In these cases, I will state that I am considering only the use of MCR.

In general, interaction with the RSX system via either MCR or DCL is extremely easy and straightforward. You should not be afraid of it.

9

Use of a Terminal

All terminals, be they DECwriters, video displays, or teletypes, are conceptually the same. We will use the generic term "terminal" for any such device. Viewed as a peripheral device, your terminal is known to the system as TTnn:. To you, the pseudo-device TI: is a more meaningful reference.

Use of a terminal for input or output is straightforward. Remember that entering a character (i.e., depressing the key for that character) in the full-duplex mode (normal mode of operation) from a terminal does not directly result in the character being displayed or printed at the terminal. The character is transmitted from your terminal to the CPU and is processed by a special task known as the terminal driver. This program then transmits characters to your terminal, and these are what you see. This is known as echoing. Usually, the echo is the same as the input character; there are a few exceptions.

When a terminal (especially a DECwriter) is turned on or switched from offline (local) to online, switching transients may result in the transmission of random characters. These will cause whatever follows to be misinterpreted. In this case, you should enter a carriage return to terminate the current line of input. You can ignore the "Illegal Command" error message that follows and proceed normally.

The terminal driver interprets several characters in a special way. Since both MCR and DCL, as well as most other RSX system routines and user programs, use the terminal driver to access data from a terminal, these characters and the special responses they evoke can be used in almost any situation. (The most notable exception is the use of a video editor, in which case the special processing normally performed by the

terminal driver is bypassed.) Most of these special characters are known as control characters. These are formed by holding down the Control key while entering a letter. Note that if you depress the Control key, release it, and then depress a letter, you will not enter a control character; you will just enter the letter. In our discussions I will refer to a control character as CTRL/letter—e.g., CTRL/Z is the special character obtained by holding down the Control key while typing a Z. (It does not matter whether the letter is in upper- or lowercase.) When you enter a control character, it will be echoed to you as a caret (^) followed by the uppercase letter (e.g., ^Z). I will show this form in our examples. The special characters most commonly used are summarized in the following sections.

9.1 Input Control Characters

Line Feed (L/F)

Strictly speaking, a carriage return (C/R) positions the cursor (print head) at column 1 of the current line, and L/F advances the line without changing the cursor position. In most cases, the system generates a combination of C/R and L/F in response to the single entry of C/R. The L/F key normally has no use.

Carriage Return (C/R)

The carriage return is used to terminate a line of input. It is echoed as a carriage return and a line feed. Thus, just as you would with an electric typewriter, you need type only one character to end a line.

Enter

Most likely, the terminal you use will have an Enter key as part of the numeric keypad. In most cases, this key is identical in function to the Return key. There are instances, however, where the Enter and Return keys will produce different results. Two examples of this that you might encounter are in the use of certain editors and in the resetting of a VT200 series terminal.

Escape or Alternate Mode (ESC)

The Escape key (ESC, or, on very old terminals, ALT MODE) is used as a special command by certain utilities, most notably certain editors.

Also, you may use it instead of a carriage return to terminate a command to either MCR or DCL, although this is seldom done.

Delete or Rubout

The Delete (Rubout) key is used to correct mistakes made during the typing of the current line. Each time you press the Delete key, the last character is removed from the input line. To understand how this functions, you must realize that the terminal driver accepts input from a terminal one character at a time and stores these in a buffer until a terminator (C/R or ESC) is encountered. The accumulated input string is then passed to the program that requested input. Thus, the characters in a line are not really input until the line is completed; until then, the line can be arbitrarily changed. By typing a delete character, you can delete the last character entered; the terminal driver simply removes it from the input buffer. The terminal driver echoes each DELETE to allow you to be sure that the requested deletion has been effected. On a printing terminal, the deleted characters are printed in the order deleted (reverse of the order entered) and are enclosed in backslashes. On a video terminal, the deleted characters are simply erased from the display. Thus, a video terminal always shows the current contents of the input line, whereas a printing terminal presents a more confusing compendium of characters.

CTRL/R

When entered, this causes the current input line to be displayed. This is intended for use on printing terminals when you have deleted several characters; it shows you a clean copy of what is currently in the input buffer. The CTRL/R itself is not entered into the input line.

CTRL/U

This causes the entire input line to be deleted. It does not delete the request for input. Rather than repeatedly using the Delete key to correct an error in an input line, you can use CTRL/U and simply start the line over. If the input is being entered in response to a prompt, another prompt is not issued after the CTRL/U.

CTRL/I or Tab (TAB)

The Tab key functions in the same manner as a Tab key on a regular typewriter. In RSX, by definition, there is a tab-stop after every eighth

position in a line. TAB is especially useful for entering source language statements in most languages. Under RSX, most compilers have been modified to allow TABs to be used instead of blanks. Entering one TAB rather than several blanks is clearly easier; it also causes fewer characters to be stored per line, which can sometimes result in a significant reduction in the size of a source file.

CTRL/Z

The CTRL/Z character is used as an end-of-file (EOF) character. CTRL/Z is not a terminator for a single line of input as is C/R or ESC but rather is a terminator for a set of input lines. It also is used as a means of exiting from most system utilities.

9.2 Output Control Characters

When a task writes to your terminal, the output is sometimes small enough or slow enough so that you can simply let it go at its own chosen speed. Other times, you may want to be able to control things. You may do so by using the following output control characters.

CTRL/O

This is a print On/Off switch. Every time you enter CTRL/O, this switch changes state. It is initially set so that all printout is On—that is, the terminal driver actually directs output to your terminal. When printout is Off, the terminal driver discards the output. It is important to note that the program producing output merely requests the terminal driver to send the output to your terminal—it is unaware of the status of the print On/Off switch and continues to execute whether the output actually reaches your terminal or not. The CTRL/O switch is a system generation option and may not function on certain RSX systems. It also will not work with all tasks, as a task may disable this function.

CTRL/S

This is used to suspend output. CTRL/S effectively disables your terminal for reception of output. The terminal driver senses this "not ready" condition, and execution of the program requesting output to the terminal is suspended until the terminal is ready. Use of CTRL/S does not cause any output to be lost; it merely delays it.

CTRL/Q

This is the logical reverse of CTRL/S and reenables a terminal for reception of program output.

Scroll On/Off

Some video terminals (e.g., the VT100 series or the VT200 series) have this special key. It functions as an output suspension On/Off switch. Depressing this key the first time after the terminal is turned on or reset is equivalent to entering a CTRL/S, the next time a CTRL/Q, and so on.

Reset

An ill-advised command (e.g., listing a task image file) may result in the output of characters to your terminal that will be interpreted by the terminal as special commands. These may cause the terminal to enter a state wherein it will not respond to any normal inputs. This is especially likely for the more modern video terminals, since these have more special capabilities that can be activated by such commands. In this case, it is necessary to reset the terminal. For any type of terminal, this may be done by turning the power off and then back on. On a VT100 series video terminal, reset may be effected more conveniently by pressing the Setup key and then the Zero key (which has RESET written above it on the keyboard). On a VT200 series video terminal, reset is effected by pressing the Setup key which causes the Setup Directory to be displayed. You can then use the cursor keys to move the highlighted block to the function labeled "Reset Terminal." You must then press the Enter key; the Return key will not be accepted. Finally, you press the Setup key again to leave the Setup mode.

9.3 System Control Character

CTRL/C

Under RSX, CTRL/C is the system control character. You can use this to gain the attention of your CLI (either MCR or DCL) even if some other task is running at your terminal. The exact interaction is rather complicated, but it is nonetheless useful for you to have a general idea of how it works.

Anything that you enter at your terminal is processed by the terminal driver. This utility classifies your input as being either solicited or unsolicited. Solicited input occurs after some task has asked for input from your terminal. (Clearly, the terminal driver cannot determine whether you intended your input for the requesting task; it simply assumes that this is so.) If there are no requests for input at the time that you type, your input is considered to be unsolicited. Solicited input is always sent by the terminal driver to the task that asked for it. If no task is running, unsolicited input is always sent to the CLI. When another task is running, unsolicited input may either be sent to the CLI, held in a buffer, or discarded. The particular choice is a function of both the details of the operating system and the task.

Traditionally, no matter what else is running at your terminal, you may (normally) force the terminal driver to pass your input to your CLI by first typing a CTRL/C. (Note the normally; it is possible for a task running at your terminal to disable this capability.) When you enter a CTRL/C, the terminal driver interprets it as a request for immediate action. RSX is informed of this, whereupon the active task is interrupted and your CLI is given temporary control of your terminal. (This will happen even if the task is currently writing to your terminal.) The CLI gives you an explicit prompt, whereupon you may enter one command to it. One of the more common uses of this capability is to enter a command to the CLI to abort the task currently running on your terminal.

This immediate interaction with your CLI is the traditional RSX behavior in response to the system control character. With version 3.0 of Micro/RSX and RSX-11M-PLUS, Control C abort processing is available as an option, but only if you are using DCL. With Control C abort processing, when you enter a CTRL/C, all tasks active at your terminal are aborted. Following this, DCL regains control of your terminal. This certainly gets the attention of DCL; you might not appreciate the side effects.

In Micro/RSX, Control C abort processing is enabled by default. In RSX-11M-PLUS, the traditional immediate interaction with DCL is enabled by default. (In RSX-11M, or in older versions of Micro/RSX or RSX-11M-PLUS, or if you are using MCR, you have no choice; Control C abort processing is not available.) You may change from one mode to the other by using the Set Control C command. To disable Control C abort processing you use the command

```
DCL: SET TERM/NOCONTROL_C
```

Similarly, to enable Control C abort processing, you use the command

```
DCL: SET TERM/CONTROL_C
```

10

Identifying Yourself

Since RSX is a multiuser system, it is necessary for it to know who is using it. Thus, when you want to use the system, you must identify yourself to it. In a system with multiuser protection, this is intrinsic to the log-in procedure. If you can have only one possible identity, this initial identification will be sufficient. Under RSX, it is quite possible for you, even as a casual user, to have several identities. In this case, you may subsequently need to change your identity. If your system does not have multiuser protection, there will be no log-in procedure and you will need to explicitly set your initial identity. We discuss these procedures in the remainder of this chapter.

10.1 Logging In and Logging Out

The entire concept of logging in and logging out applies only to a system with multiuser protection. If your system does not have this feature, you do not need to use these commands (and you do not need this section of the book). You can simply find an available terminal, sit down, and enter commands.

Log-in to an RSX system is effected via the Log-In command. In MCR this is **HEL**; in DCL it is **LOGIN**, which may be abbreviated to **LOG**. Other than the name of the command itself, the DCL command is identical to the MCR command. Note that, with the exception of the Help command, no other system commands can be entered until you have successfully logged in to the system. (RSX allows you to enter a Help command even if you are not logged in so that you can refresh your memory about how to log in. The Help command is discussed further in Section 19.1.)

To log in to the system, you must first find an unused terminal. Note

that even when a terminal is unused (i.e., not assigned to a particular user), it is normally being monitored by the operating system for input. Thus, you can usually gain the attention of the system merely by entering the Log-In command appropriate to your CLI. If there is no response, a CTRL/C should be entered.

If only the Log-In command itself (**HEL** or **LOGIN**) is entered, the Log-In routine asks for your account or name. This may be either your UIC or your name, as assigned to your account by the system manager. The Log-In routine next requests your password. When the password is entered, print echo is suppressed by the system—that is, the password is not written back to your terminal. This feature is provided as part of the multiuser protection system to prevent one user from observing another's password. If your UIC (or user name) and password match an entry in the system account file, you will then be logged in. After the display of a system greeting and/or system information, a default prompt is issued, indicating that your particular CLI is ready to accept a command from you.

You can shorten the above procedure by entering

```
MCR: HEL uic/password
DCL: LOG uic/password
```

where **uic** is either the UIC or your name and the password is preceded by a slash. In this case, print echo of the password is not suppressed. Alternatively, you can use

```
MCR: HEL uic
DCL: LOG uic
```

The log-in routine then requests the password and suppresses print echo for it.

No matter which form of the Log-In command you use, you can enter the UIC in one of four forms: **[g,m]**, **g,m**, **[g/m]**, or **g/m**. The last two forms are used to avoid getting the log-in message every time you log in to the system. (Specifically, the only times that you will get a log-in message when you use a slash in the UIC are the first time you log in after midnight of any day or after the log-in message is changed.) This is the only case in which the UIC may be entered with a slash rather than a comma between the group and member numbers. Also, the Log-In command is the only command in which the UIC may be entered without the usual enclosing brackets.

As part of the log-in procedure, RSX determines your privilege status,

which it remembers for as long as you stay logged-in. RSX also assigns your default device (SY:) and default directory according to values specified in your log-in account. If you wish, you can subsequently change these as explained in Section 10.2.

After one user logs off a terminal and before the next user logs on to it, the terminal is not in use. If you attempt to use the system without first logging in, you will get a

```
NOT LOGGED ON
```

message. Similarly, an attempt to log in before a previous user has logged out will be greeted with an

```
OTHER USER LOGGED ON
```

error message and your command will be ignored.

Log-out is extremely simple; you use the Logout command. This consists solely of the command name:

MCR: **BYE**

DCL: **LOGOUT**

Note that any task initiated from your terminal that is still executing when you enter the Log-Out command will be aborted when this command is processed. To avoid this, your task must be installed and run with a time delay (see Sections 18.4 and 25.1). Otherwise, if you wish to run a long task, you must be sure not to log out—when a limited number of terminals is available, this is a poor technique. Also, it is easy under RSX to walk away from a terminal and not log out when you are finished. Common courtesy dictates that you remember to terminate your turn on the system so that another user may have your terminal next.

10.2 Changing Your Default Device or Directory

Whenever you interact with RSX, you have a particular disk device assigned as your system disk and a particular directory assigned as your default directory. Your system disk is used as a default for the device portion of the file specifier whenever you refer to a file. Similarly, your default directory is used as a default for the ufd portion of the file specifier. For you to be able to access your files, these must be set correctly.

On a system without multiuser protection, there is no log-in procedure

to establish the correct assignment of your system disk and default directory. When you find an available terminal, SY: and the default ufd will be as the last user left them. Although valid for him, they will probably be inappropriate for you. A system with multiuser protection makes these assignments for you when you log in; nonetheless, you may subsequently want to change them. The commands discussed in this section allow you to do this.

In MCR, two distinct commands are required: one to change your default device and one to change your default directory. In DCL, one general purpose command allows you to change either or both.

To change your default device in MCR, you use a form of the Assign command; in DCL you use a form of the Set Default command. These are

MCR: **ASN SY:=ddnn:**

DCL: **SET DEFAULT ddnn:**

where **ddnn:** is the peripheral device code for your disk.

In Section 6.4, we discussed the use of two approaches for identifying user areas—numbered and named directories. To repeat, on any version of RSX-11M and all versions of RSX-11M-PLUS prior to version 3.0, named directories are not available—you must use numbered directories. On any version of Micro/RSX and version 3.0 of RSX-11M-PLUS, named directories are available—you can use either numbered or named directories. If you have the choice of these two modes of operation, you must be able to pick one or the other. When you log in, a default mode will be chosen for you. In Micro/RSX, the default is to use named directories; in version 3.0 of RSX-11M-PLUS, the default is to use numbered directories. In either case, you can ask your system manager to modify your system account if you want this initial choice changed. Also, once you are logged in, you can switch between the numbered and named directory modes by using the Set Named command. To choose numbered directories, you use the command

MCR: **SET /NONAMED**

DCL: **SET DEFAULT/NONAMED**

Similarly, to choose named directories, you use the command

MCR: **SET /NAMED**

DCL: **SET DEFAULT/NAMED**

The command to change your default directory depends on whether

you are using numbered or named directories. Let's first consider the use of numbered directories, since this applies to all systems. In MCR, you use the Set UIC command,

```
MCR: SET /UIC=[uic]
```

where **uic** is the desired UFD, entered in the usual **g,m** form. Note that this command actually sets your current (or default) UIC. When you use numbered directories, the distinction between UICs and UFDs is blurred. Strictly speaking, you use this command to set your current UIC; RSX uses this as your default UFD. The distinction is important in that the command must be as given above; you cannot type instead SET /UFD=. In DCL, you use another form of the Set Default command,

```
DCL: SET DEFAULT [ufd]
```

(Although the name of this command may seem to be more appropriately chosen, it is translated into the MCR Set UIC command. Also, the Set UIC command has special ramifications for the privileged user, which we discuss in Sections 14.5 and 19.3.) If you are using named directories, you use the Set Default Directory command

```
MCR: SET /DEF=[ufd]
DCL: SET DEFAULT [ufd]
```

(Although the DCL command is the same for named and numbered directories, its effects are not identical for the privileged user.) Note that, whether you are using numbered or named directories, you can use the DCL Set Default command to change both your system disk and default directory in one command. In this form, the command is

```
DCL: SET DEFAULT ddnn: [ufd]
```

Let's look at some examples. First, let's consider the use of RSX-11M. In this case, named directories do not exist. Assume that you have logged in as UIC [110,1] and your system disk is DR1:. After doing some work in that area, you decide that you want to look at some files in another user's area. This is on disk DR2: and is identified by the UFD [222,3]. For each file that you want to use, you could specify **DR2:[222,3]** to override your defaults. This is a nuisance; it is easier to change your defaults. In MCR, you would enter these two commands:

```
MCR: ASN SY:=DR2:
     SET /UIC=[222,3]
```

In DCL, you would enter the single command

```
DCL: SET DEFAULT DR2: [222,3]
```

For a different example, suppose your system originally was version 2.1 of RSX-11M-PLUS. On that system, your UIC is [110,1]. In UFD [110,1], you have many files for small projects; you also have UFD [110,2] as a second user area, which is dedicated to one large project. Now, your system manager upgrades your system to version 3.0, making named directories available. The continued support for numbered directories allows him to keep all existing user areas and files without change. He sets your account so that when you log in, you will be put into numbered directory mode. Although named directory support is now available, you can ignore it and continue using the system exactly as you did before. Eventually, you are assigned a new project for which you decide that you need another user area. You decide to try a named directory this time. Your system manager creates one for you, called BIGJOB. When you log in, if you want to work in this new area, you must change your mode from numbered to named directories and then change your default directory from [110,1] to [BIGJOB]. In MCR, you use the two commands

```
MCR: SET /NAMED
     SET /DEF=[BIGJOB]
```

In DCL, you could similarly use two commands,

```
DCL: SET DEFAULT/NAMED
     SET DEFAULT [BIGJOB]
```

Alternatively, you could use another form of the Set Default command that combines these two functions,

```
DCL: SET DEFAULT [BIGJOB]/NAMED
```

After doing this, if you want to work in your numbered area [110,2], you could use commands corresponding to those above to change your mode from named to numbered directories and your default directory to [110,2],

```
MCR: SET /NONAMED
     SET /UIC=[110,2]
DCL: SET DEFAULT/NONAMED
     SET DEFAULT [110,2]
```

Note that if you are in MCR, the command for changing your default directory is either **SET /UIC** or **SET /DEF**, depending on whether you

are in numbered or named directory mode. Also note that for both MCR and DCL, you specify your directory as either **[g,m]** or **[name]**, again dependent on the same distinction.

When you are in named directory mode, you can refer to a numbered directory if you wish. In Section 6.4, we saw that any directory is simply a file (containing the names of all the files in the corresponding user area), and, as such, has a file name of at most nine characters. For a numbered directory, this name is **gggmmm**, where **ggg** and **mmm** are the group and member numbers, expressed as three octal digits with leading zeros if necessary. In the example above, you can remain in named directory mode but set your default to the numbered directory [110,2] by using its six-digit name,

MCR: **SET /DEF=[110002]**

DCL: **SET DEFAULT [110002]**

Whether you use this technique or the other, you will correctly change your default directory to [110,2]. The difference is in whether you want to be in numbered or named directory mode. This has no effect on file references that use your default directory (files for which you do not specify a ufd). It is only when you need to refer to a file in another directory that this matters. If you change back to numbered directory mode, you will not be able to refer to a named directory at all. If you stay in named directory mode, you will be able to refer to both numbered and named directories, but you must remember to use the six-digit form for numbered directories.

11

Introduction to System Functions

As a casual user, you may view your use of the computer system as one of simply running your own programs. That is, you have a problem that you wish to solve, so you write a program to solve it and then run the program. Of course, when you do this, you are running many other programs as well. These are not "your" programs—they are part of the operating system. Examples of these are the editor that you use to write your program, the compiler that you use to compile it, and the Task Builder. Although your emphasis will be on running your programs, you may well spend more time using these other programs. We will refer to these as system functions.

Although the concept of having and using system functions should be obvious, certain aspects of this under RSX require explanation. This chapter introduces the use of system functions; we return to this topic in greater detail in Chapter 18.

11.1 Installed Tasks

So far, we have referred to the concept of a task without defining just what this means. Under RSX, anything that can be executed (i.e., anything to which control of the CPU may be given) is known as a task. A user can build a task by first compiling a source program and then linking the resulting object code with the object code for the various required subroutines. This linking is done by using the Task Builder, which is a part of the RSX operating system. A task built by you in this manner is perhaps the most natural for you to picture. Note, however, that the entire operating system is itself a collection of tasks. Each of

these performs a certain function within the system. Some will be obvious—your CLI, an editor, a compiler, the task builder—but you will use others without ever realizing it. As far as you are concerned, there is no need to distinguish among these different types of tasks—they are all simply tasks.

The concept of linking, which we referred to in the preceding paragraph, is probably familiar to you. Let's discuss it in a little more detail. The Task Builder accepts as input a collection of object files—one corresponding to the main program, the others to required subroutines. The output of the Task Builder is a task image file. This is conceptually the same as any other file. It is stored on some device (typically the system device), but it is not kept in memory. As such, it cannot be directly executed by the computer. Instead, it must first be copied from the appropriate device into memory. Only then can control of the CPU be transferred to it, or, in more common terms, can it be run.

In RSX, an intermediate step is required in this process. Before a task image file can be copied into memory, the operating system must be made aware of its existence. This is done via a process known as installing the task. The installation of a task involves storing certain parameters pertinent to it (the location of the task image file, the name of the task, the priority of the task, and others) in a table known as the System Task Directory (STD). The word "task," strictly speaking, refers to any entry in the STD. Each task must correspond to a task image file, but the converse is not true. Note that whereas the name of a task image file is a file name and therefore can be nine alphanumeric characters, the name of a task can be only six characters in length.

From this discussion, we see that a "task" and a "task image file" are two distinct concepts. Most of the time (for a system level user as well as a casual user) it is convenient to forget this distinction and treat the two terms as synonymous. I have deliberately made this oversimplification several times already and will continue to do so throughout most of the rest of this book.

A task can be in one of two basic states—dormant or active. A dormant task is one that is not being used—it consists only of its STD entry and the corresponding task image file. An active task is one that is being used—that is, the operating system has received a request to run it, and has either already acted upon or is waiting to act upon this request by copying the task image file into memory and actually running it. A typical RSX system will have many installed tasks—at any point in time, almost all of these will be dormant and only a few will be active.

The process of installing a task prior to running it may seem unduly

complicated, and for most applications of interest to you, it is. This procedure was developed to facilitate the handling of real-time applications. By having a task already installed, the system need not waste time (using relatively slow disk reads) determining where on disk the task image file is located. Instead, it merely looks up the disk address of the file in the STD—because the STD is in the computer memory, this is extremely fast. Another unusual feature of RSX is that all task image files occupy a contiguous group of disk blocks. (Other files are composed of blocks that may be scattered throughout the disk.) As a task image file is contiguously stored on disk, it can be loaded into memory using a single Direct Memory Access (DMA) transfer. Thus, when a command is issued to run a task, RSX is able to give a very rapid response. For time-sensitive real-time applications, this can be an important point; to you, it is a feature of no benefit. As the process of task installation is seemingly unnecessary and thereby aggravating, RSX includes a special feature whereby this procedure is made invisible to you. This is discussed further in Chapter 18.

11.2 Invoking System Functions

You will need to use many system functions (tasks that are included as a part of RSX)—these include system utilities (used to create and maintain files), compilers, and the Task Builder. There is a common set of methods for invoking any of these system functions.

In MCR, you must invoke a system function by explicitly naming it. In DCL, you can often invoke a system function implicitly by asking for a certain action. For example, in DCL, when you enter the command **COPY**, you are actually invoking PIP, the Peripheral Interchange Processor. This indirect form is available for most actions that you will want. You may, however, directly invoke a system function in DCL. When you do, you must use the MCR command form for it. You might want to do this if you need to use a less common command for which there is no DCL counterpart or if you want to interact directly with the system function as discussed below.

The system utility PIP is very commonly used; we will use it as an example throughout this section. In MCR, if PIP has not been installed, it must be invoked in the form

```
MCR: RUN $PIP
```

In DCL, whether PIP has been installed or not, it must be invoked in this form. A dollar sign ($) as prefix indicates that the named task is a system, not user, task. Normally, the utilities that you will want to use will be installed. Then, if you are in MCR, you will not have to run them in this form (although you still may, if you wish). A simpler form is instead possible—you need merely enter the utility name directly,

```
MCR: PIP
```

In either case, the effect of the command is to have control transferred to the utility PIP, PIP then requests commands from you, PIP will remain in control until you terminate the command sequence by entering a CTRL/Z, and control then returns to your CLI. For example, a typical command sequence might look like this:

```
MCR>PIP          or     DCL>RUN $PIP
PIP>command             PIP>command
PIP>...                 PIP>...
PIP>^Z                  PIP>^Z
MCR>                    DCL>
```

When you use a utility in this manner, we will say that you have "entered" the utility. In general, this interactive use offers you the greatest flexibility and efficiency. For extremely simple commands, however, it can be unnecessarily complicated.

If you have only one line of command input, a simpler form may be used. If you are in MCR, you may invoke the utility and pass your command directly to it, as in

```
MCR: PIP command line
```

This results in execution of the command by PIP, with control then being automatically returned to MCR. You can use this abbreviated form only if the utility is installed. If you are in DCL, the above form can be used by declaring it to be an MCR command,

```
DCL: MCR PIP command line
```

Alternatively, you can often achieve the same effect by using a DCL command that implicitly invokes a system function. For example, from DCL you can copy a file by issuing a **COPY** command—this actually results in a single line command being issued to PIP.

Now suppose that you wish to execute several commands sequentially

that all use the same system function. There are two ways to do this. You can give the system function a succession of one line commands from the CLI, or you can first enter the function and then give the commands directly to it. Although you may achieve the same end result, these two techniques are not equivalent. Every time you invoke a system function, it must be loaded from disk into memory; after it executes your command, it is then removed by RSX from memory. If your next command again invokes the same utility, the loading has to be repeated. Thus, the two command sequences

```
MCR>PIP                 or      DCL>RUN $PIP
PIP>command 1                   PIP>command 1
PIP>command 2                   PIP>command 2
PIP>command 3                   PIP>command 3
PIP>^Z                          PIP>^Z
MCR>
```

and

```
MCR>PIP command 1       or      DCL>equiv command 1
MCR>PIP command 2               DCL>equiv command 2
MCR>PIP command 3               DCL>equiv command 3
```

are not equivalent. The second sequence of commands results in PIP being loaded into memory three times, whereas the first requires that this be done only once and is therefore more efficient.

All system functions have three-character abbreviations as their official names whereby they may be invoked. This is true regardless of whether you are in MCR or DCL; it is a function of RSX. (This is not true of version 1.0 of Micro/RSX, which is why it is virtually impossible to use MCR in it.) The actual name listed in the STD is preceded by three periods to make it a full six characters long (e.g., PIP appears as . . . PIP). When RSX processes a command to invoke a system function, it takes the first three characters as the name of the command and ignores any extra characters between these and either the first blank or carriage return. Thus, the two lines of command input

```
MCR>PIP command line
MCR>PIPSQUEAK command line
```

are equivalent, as are

```
DCL>RUN $PIP
DCL>RUN $PIPSQUEAK
```

11.3 More on File Specifier Defaults

In Section 6.1 we saw that certain portions of a file specifier normally assume default values if you omit them. In this section we consider this in greater detail for the special case of multiple input file specifiers.

When you enter commands to certain utilities, you can specify more than one input file. For example, you can use the Print command to print several files or the Delete command to delete several files. To do this, you enter the input specifier portion of the command as several file specifiers separated by commas. In this case, special rules apply for forming the defaults. These rules apply whether you invoke the utility directly from MCR or indirectly from DCL via a command that is translated into MCR. In general, the special defaults for multiple input files work as follows.

The default for the version number is unchanged from that mentioned earlier; if omitted, a value of zero is used, which is equivalent to the most recent version of the file. The device, ufd, file type, and sometimes even the file name, however, change their defaults when you have several input files. For the first file in the input list, these portions of the specifier assume their normal defaults. (For the first file, there is no default for the file name; you must specify it.) The device code defaults to SY:, and the ufd defaults to your default directory. If it is meaningful for the command, the file type also assumes a default. Of course, you can override any of these defaults by entering a value. Whether defaults are used or not, a full file specifier is constructed for the first input file. The device, ufd, file name, and file type portions of this are then used as defaults for the next file in the input list. Thus, if you do not override any defaults for the first file, the normal defaults will be used for the second file. When you do override a default, the value you enter will be used instead of the normal default for the next file. This process continues from one file to the next in the input list.

Let's consider some examples. (We will examine these commands in detail in later chapters.) You may use the Print command to print files on your system's line printer. For the simple examples that we consider here, the Print command is the same for both MCR and DCL; it is simply the command name **PRI** followed by the names of the files you want printed. The Print command assumes a default file type of LST. Suppose your system disk is DB1: and your default directory is [102,1]. The command

```
PRI A,B,C
```

will cause the latest versions of DB1:[102,1]A.LST, DB1:[102,1]B.LST and DB1:[102,1]C.LST to be printed. If you override the default file type in the first file specifier by using the command

```
PRI A.MAC,B,C
```

the files printed will instead be A:MAC, B.MAC, and C.MAC (all in DB1:[102,1]). If you want to print the MACRO source files for A and B but the listing file for C, you could use the command

```
PRI A.MAC,B,C.LST
```

Now, suppose file B.MAC is in someone else's user area (such as DB2:[211,3]). If you try to print all the source files via the command

```
PRI A.MAC,DB2:[211,3]B,C
```

files A and B will be printed, but file C will (presumably) not be found since what you have specified is actually the file DB2:[211,3]C.MAC.

Normally, you will not use defaults for the file name except perhaps with certain file maintenance commands, such as deleting files. For example, if you have various versions of the files A.MAC, A.OBJ, and A.TSK and you want to delete the object and task files but not the source files, you could use the command

```
MCR:PIP A.OBJ;*,.TSK;*/DE
DCL:DEL A.OBJ;*,.TSK;*
```

Here, the first file specifier sets the file name to be A, which is used for the second file specifier in the list.

These special file specifier defaults are a useful feature when you need to specify several input files. They are especially useful under RSX-11M, which limits a command line to a total of 79 characters, as they allow you to fit more file specifiers into one line. Most utilities for which it is meaningful to specify more than one input file will follow these special rules.

12

File Creation

Sooner or later, you will have a large variety of files. Some of these will be created by system functions acting upon other files (e.g., a compiler producing an object file from a source file, or the task builder producing a task image file from object files). Others will be created directly by you (e.g., source files, text files, or data files). We consider this latter form of file creation in this chapter.

Any file may be viewed as a string of bytes or words. A text file (whether the text be used as source statements, notes, data, or whatever) is interpreted by the various system functions that manipulate it as a string of bytes, where each byte contains one character. Characters are stored in ASCII format. To create a text file, you need a means of entering characters. Direct entry from the keyboard is possible by copying from a terminal to a file (see Chapter 13). Because this is effected via the utility PIP (whether you explicitly name PIP or use the DCL commands **COPY** or **CREATE**), and because PIP has no editing capability, mistakes cannot be corrected (except by using either the Delete or CTRL/U key) once entered. Thus, this method is not recommended except for very short files.

The typical means of creating a text file is by using an editor. An editor allows you to enter text or to change text that has already been entered. In general (unless a new file is being made), the editor accepts as input an existing file and produces as output the next highest version of that file. You communicate with the editor in an interactive manner, entering commands or text from the keyboard and obtaining responses as appropriate. The older generation of editors treats video terminals as though they are printing terminals. The newer generation takes advantage of the special capabilities of video terminals. In the video editing

mode, the video display is used to show you the current contents of a portion (typically 24 lines) of the file being edited. As you enter changes, they are immediately shown in the display. This is a very powerful technique of which the editors available with RSX have only recently taken full advantage.

Several editors are available under RSX. You may have some or all of these available on your system. Under very old versions of RSX, EDI (EDItor) was the standard editor. Under version 3.2 of RSX-11M, EDT (EDiTor) was recommended by Digital as being the standard editor, although this early version was rather limited in capability. Under version 4.0 of RSX, EDT was significantly enhanced, and further improvements were added with version 4.1. These later versions deserve their promotion as the standard editor to use on all Digital Equipment Corporation systems. Roughly concurrent with these changes to EDT, yet another editor, KED (Keypad EDitor), was offered. Almost forever, it seems, TECO (Text Editor and COrrector) has been the unofficial standard editor throughout the Digital community. Originally developed for the DECSystem-10, it is also available for most other Digital Equipment Corporation computers. Under RSX, TECO is not an officially supported product, but it is available from DECUS.

The most general view of a text file is as a string of characters. Perhaps a more common but less sophisticated conception of a text file is as a sequence of lines. Since each line is a string of characters, and since successive lines are separated by special characters (carriage return/line feed), these two viewpoints are somewhat equivalent.

An editor can treat a file in either manner depending on how it was designed. A line-oriented editor allows you to operate one line at a time—if changes are to be made, the entire line must be retyped. A character-oriented editor is less restrictive—you can operate on a string of characters, whether it is embedded in the middle of a line or encompasses several lines.

It is not my intent in this book to teach you how to use a particular editor, just as it is not my intent to teach you how to write computer programs. Therefore, I will limit our discussion to a brief comparison of the various editors available to you. (Although based on fact, these remarks also represent personal biases.)

EDI is the most primitive of the editors that you are likely to encounter. (The system user may deal with SLP [Source Language Processor], which makes EDI appear extremely flexible by comparison. Digital uses SLP for updates to the RSX source code where reliability

and accountability demand a simple-minded approach.) EDI is essentially a line-oriented editor, although a few character string capabilities exist. It does not offer a video editing mode.

EDT is, like EDI, essentially line-oriented with limited string capabilities. One major advantage of EDT over EDI is that it offers a video editing mode. The primary advantage of EDT is that it is now Digital's standard interactive editor. If you learn how to use it on the PDP-11 under RSX, you will know how to use it on other Digital computers or operating systems as well. EDT has changed considerably with different releases of the operating system. Many enhancements were made with version 4.0 of RSX-11M. One of these is the journal file, in which EDT automatically records every command you enter during an editing session. When your computer system crashes (they all do, sooner or later, often for no discernible reason), any work in progress is lost. This is very painful if you have just spent several hours making changes to your file. With the journal file, you can recover your work very easily.

The editor KED is specifically designed for video editing on a VT100 type terminal. (The more recently introduced VT200 series, although different in physical layout, is compatible with the VT100; thus, you may use KED on them as well.) K52 is a form of KED that is specifically designed for the VT52, which was the previous standard video terminal. In function, K52 is essentially the same as KED. Under version 3.2 of RSX-11M, KED was generally considered to be better than EDT for video editing. With the enhancements to EDT under version 4.0, the video editing capabilities of the two have become roughly the same. Since EDT also offers a nonvideo mode, there is no longer any real reason to use KED instead of EDT, and thus KED has fallen into disfavor.

TECO (Text Editor and COrrector) is the most sophisticated editor. It is a character-oriented editor with several line-oriented capabilities. It includes support for operation in a video editing mode. TECO is often described as being a programming language, not an editor. This is true. For example, you can write a TECO macro to perform some editing function (such as changing all occurrences of one string of characters into a second string) not just on one file but on every file (as defined by wildcards) in your user area. Macros for running TECO in a video editing mode are available from DECUS. These offer capabilities that are similar to those in EDT, but, if you need to do something fancy, you can get out of video mode and continue editing in normal TECO, from which you can do almost anything. The one big disadvantage of TECO is that it is slow in the video mode. This is probably due to the video editing

capabilities being implemented via a macro rather than being built into the editor, but it may also be due in part to the interface between TECO and the RSX terminal driver. Whatever the reason, it is likely that you will type faster than TECO in video mode can handle. (It is not likely that you will be able to do this with EDT.)

Regardless of which editor you use, a common syntax exists for invoking it. You enter a command to your CLI naming the editor you wish to use. Normally, you will also specify the file you wish to edit. If a version of this file does not already exist, the editor will create version 1 of it. Otherwise, the highest numbered version will be read into memory for you to modify. After you are done, the edited file will be saved as the next highest version; the original file will be unchanged. Modifications to this basic procedure are possible. For example, you can quit editing without saving your changes; you can call up the editor without naming a file and subsequently (from within the editor) do so; you can save your edited file under a different file name; or when you invoke TECO you can start an editing macro (this is how you use TECO for video editing). These are all functions of the particular editor that you use and typically represent a more advanced use of it. Here, we will examine only the standard commands for invoking an editor.

In MCR you invoke your favorite editor by using its name as given above (TECO is typically shortened to **TEC**). Editing commands are of the form:

```
MCR:EDT filespec
    EDI filespec
    KED filespec
    K52 filespec
    TEC filespec
```

In DCL you invoke your favorite editor by using the generic command **EDIT**. If you do not modify this command, you will get EDT, since this is the standard editor. To use one of the other editors, you use the command form **EDIT/editor**. For DCL, the editing commands corresponding to those listed above are:

```
DCL:EDIT filespec
    EDIT/EDI filespec
    EDIT/KED filespec
    EDIT/K52 filespec
    EDIT/TEC filespec
```

The editor with which an individual user will be most comfortable will change from one user to the next. It is unlikely that you will want

to use either EDI or KED unless you already know them and do not want to learn something new. If you are doing relatively simple editing, you will probably be happiest using EDT in video mode. If you commonly work with pieces of several files or perform other advanced tasks, you may well need to use TECO. It will be harder to master TECO than either EDI, EDT, or KED; concomitantly, if you do master it, you will be able to do things that will baffle your colleagues who are using one of the simpler editors. The best advice for you is to browse through the manuals for all the editors and to then start experimenting. Have fun!

13

File Copying

Almost everything you do on the computer involves using files. One of the basic things you can do with a file is to copy it. The actual copying of a file is done by the system utility PIP (the Peripheral Interchange Program), which is also used for a variety of other functions on user files. As its name implies, PIP can copy files from any peripheral device in the system to any peripheral device.

As with other utilities, if you are in MCR, you can use PIP in either a single-line command form or in an interactive manner; the actual command to PIP is the same. If you use DCL, you will use a different command; this is translated into the equivalent PIP command for you.

Copying a file is probably the most quoted example of the difference in command structure between MCR and DCL. Since the utility PIP does the copying, the MCR command requires that PIP be specifically named. In PIP, the general form of the Copy command is

```
output=input
```

Thus, in MCR, a complete single-line command to copy a file is

```
MCR: PIP output=input
```

This command style mimics that of an assignment statement in a language such as FORTRAN, and it may be best remembered that way—you are setting the output equal to the input. This style pervades MCR usage; it is also found in commands for compiling, task building, etc. In DCL, the same command is

```
DCL: COPY input output
```

This command reflects the way you would state things in normal conversation rather than in a program—you wish to copy from the input to the output. This "from–to" style is common to many DCL commands. If you wish, you can use the multiline command form, in which DCL prompts you for each file specifier. In the most general form, you first enter just the command **COPY**. DCL then asks for the input specifier, prompting you with "From?" After you supply this, DCL asks for the output specifier, prompting you with "To?" As with many other DCL commands, when you are learning them, this multiline form is convenient, but once you learn the command syntax (it happens faster than you might expect), you will find the single-line command form faster and easier.

Whether you use MCR or DCL, the **output** and **input** above are both file specifiers. Standard defaults apply to the device and ufd portions of the input and output specifiers. If the output file name and type are not specified, they are set the same as the input values. If the output version is not specified, it too is given a default; this depends, however, on whether you are in MCR or DCL, as we see later. It is possible to omit the output file specifier completely. You would do this if you wished to form the output file specification according to the above defaults. (Here, presumably, the input would be from a different user area or device; otherwise the output specifier would be the same as the input specifier.) In MCR you can use the form

```
MCR: PIP=input
```

The corresponding form does not work in DCL. If you enter

```
DCL: COPY input
```

DCL will assume that you are using the multiline command form and it will prompt you to enter the output file spec. At this point, you can enter a carriage return. The output file spec will be recognized as being empty, and the above defaults will apply. This is clumsy because it cannot be done in one line. If you wish to enter the entire command on one line (this is especially useful for indirect command files, which we discuss in Chapter 20), you must give part of the output file specifier. You do not have to supply all of it; either **SY:** or **[ufd]** or *.* is sufficient to force the above defaults to be used for the remainder of the output specifier. Note that whether you are in MCR or DCL, you must always give the input file name; you can, however, use wildcards.

As a nonprivileged user, you will normally copy files into your user

area. The files being copied may also be in your area, or they may be in someone else's area. Nothing in the Copy command itself restricts you to this—you could just as easily give a command to copy a file from your area into someone else's area. For this to work, however, you must be able to create files in that other user area. If you are not a privileged user, you will probably not be able to do this. Thus, although the Copy command may be syntactically correct, it will fail due to your lack of file access privileges. In our examples we will consider only the copying of files into your own area.

Let's now look at some examples of the basic copy function:

```
(1) MCR: PIP TEMP.MAC=PROC11.MAC
    DCL: COPY PROC11.MAC TEMP.MAC
```

Here, both the input and output files are within your user area. The input file is named PROC11.MAC, and you make an identical copy of the latest version of it, under the name TEMP.MAC. This might be useful if, for example, PROC11.MAC is a working program that you do not wish to destroy, but you want to experiment with some changes to it.

```
(2) MCR: PIP=DB2:[100,1]A.FTN
    DCL: COPY DB2:[100,1]A.FTN SY:
```

The latest version of FORTRAN source file A located on disk DB2: in user area [100,1] is copied into your current user area. The name of the new file is also A.FTN. Note that in the DCL form, SY: was used as a dummy output file specifier to enable the command to be given in one line. Without this, the command would have to be given in two lines:

```
DCL>COPY DB2:[100,1]A.FTN
To?
```

Here, the response to the "To?" prompt is simply a carriage return.

```
(3) MCR: PIP=[100,1]A.FTN,B.FTN
    DCL: COPY [100,1]A.FTN,B.FTN SY:
```

This is the same as example 2 except that two files are copied. They are copied separately, producing two corresponding output files.

```
(4) MCR: PIP B.FTN=[100,1]A.FTN
    DCL: COPY [100,1]A.FTN B.FTN
```

This is the same as example 2 except that the name of the output file is changed from that of the input file to B.FTN. Assuming that no files named B.FTN exist in your area, the version of the output file is set to 1.

In addition to the straightforward file copying discussed above, several modifications are possible. If you use MCR, these will be selected by adding switches to the basic PIP command. If you use DCL, you may need to add switches to the basic Copy command, or you may need to use a different command word entirely. The modifications that we consider here fall into two categories: combining several files into one, and specifying what is to happen when naming conflicts arise.

It is possible to combine several input files into one output file as part of the copying action. There are two ways to do this: merging the files and appending them. In either case, as with the basic Copy command, the input files are not affected by this command.

When two or more input files are copied, and an output file is explicitly named, the various input files are merged to form the single output file. This modification of the basic copying action is assumed by default and need not be specified. If you are in MCR, you may, however, explicitly specify that you want merging to occur; you do this by using the Merge switch (**/ME**). In DCL there is no way to explicitly specify this.

The Append command allows you to append one file to another. Appending involves copying the input file to the end of the output file. This is similar in action to the Merge command—the difference is that merging creates a new file whereas appending enlarges an existing file. Although the output file is changed, a new version is not created. Thus (except perhaps for the size of the output file) a listing of your directory will look the same before and after an Append command. The command is

```
MCR: PIP output=input/AP
DCL: APPEND input output
```

The input is appended to the output file; if two or more input files are specified, they are appended in the order in which they are listed in the command.

Let's look at some examples of these modified forms of the Copy command.

```
(5) MCR: PIP TEST.FTN=A.FTN,B.FTN/ME
    DCL: COPY A.FTN,B.FTN TEST.FTN
```

All files referred to are in your area. The latest versions of files A.FTN

and B.FTN are merged (concatenated) to form the single file TEST.FTN. Note that the difference between this example and number 3 is that in number 3, no output file was named. Thus, the files were copied individually, preserving their file names. When an output file is named (no wildcards) and more than one input file is listed, merging is assumed.

```
(6) MCR: PIP TEST.FTN=A.FTN
         PIP TEST.FTN=B.FTN/AP

    DCL: COPY A.FTN TEST.FTN
         APP B.FTN TEST.FTN
```

The net result of this example is identical to that of example number 5. The difference is that it involves two separate steps—a straight copy, and then an append.

The other special cases that need to be considered concern naming conflicts. A common scenario, as in the examples above, is that of copying a file from some other user area into your own. Typically this is done because you do not have the file in your area—that is, you do not have any versions of it at all. In this case, there can be no conflicts, and the copy operation will proceed without incident. What happens when this is not the case? The answer is somewhat messy because it depends on several factors.

Let's first consider the case where you give a complete output file specifier (including the version number) in the Copy command. There are two possibilities: either a file with the exact complete specifier already exists, or one does not. If one does not (the existence of other files with the same name and type but different versions is irrelevant here), the copying proceeds and that is that. If, however, you have a file in your area with the exact same file specifier (including the version number) as the file that you wish to copy into your area, the copy operation will fail. The assumption is that you must have made a mistake, and it is safer to refuse your copy request than it is to destroy the file you already have in your area. It is, of course, possible that you actually want to do what you said. In this case, you must specify that you want the copying to proceed, even if it requires that an existing file be destroyed. In MCR, you use the Supersede switch (/SU) for this; in DCL you use the Replace switch (/REP). Note that if you use this switch and the file does not already exist, nothing special happens—the switch is ignored, and a normal copy occurs.

Now, suppose you do not specify a version number for the output file. Instead, you wish to use the default version number for the file you are

making in your area. What happens in this case depends on whether you are in MCR or DCL. Let's first examine what happens in MCR. The version number of the output file is set by default to be that of the input file. If a file with the same name and version number already exists, you will have the same problem as discussed above for exactly the same reasons. A similar but somewhat more subtle problem occurs when you do not have a file with the exact same specifier, but you do have one or more files with the same file name and file type but different versions. Suppose, for example, that you already have versions 1, 4 and 5 of the file TEST.FTN in your area and the file that you want to copy into your area is TEST.FTN;2. Strictly speaking, there is no naming conflict as you do not have a version 2 of the file TEST.FTN. But what happens if the copy made in your area is given the number of the original version? What you may think of as being the latest version of the file is actually version 2, and FILES-11 will continue to consider version 5 to be the latest version. When you later refer to TEST.FTN, you will not get the copy that you just made. One way around this problem is to name the output file so that you can specify a high version, e.g., you could declare the output file to be TEST.FTN;10. This is a nuisance, however—you should be able to let RSX take care of such bookkeeping for you. You can do this by requesting a new version when you issue the Copy command. This means that you want PIP to perform the indicated copying but assign a new version number to the output. This is taken to be one greater than the highest version number of an existing file with the same name, or version 1 if no file exists with the same name. In MCR, you request a new version by using the New Version switch (**/NV**); the default is to not make a new version.

If you are in DCL, things are different. Since all DCL commands are translated into MCR commands, something similar to the above must happen. What is different about DCL is that when the output version is not specified, the Copy command is, by default, translated into a PIP command with the New Version switch set. Depending on which type of RSX you have, and what version it is, you may have no control over this. Specifically, if you have version 3.0 of RSX-11M-PLUS or version 3.0 of Micro/RSX, you may request that a new version not be created. If you have RSX-11M or any of the older versions of RSX-11M-PLUS or Micro/RSX, you do not have this choice. To prevent the creation of a new version, use the No New Versions switch, **/NONEWVERSION**. This may (somewhat confusingly) be abbreviated to **/NONE**.

In DCL, the only times that you can get a naming conflict are when you give a version number to the output file or when you request that

new versions not be made. Otherwise, you will always make a new version when you copy a file. This prevents some problems but can cause others. It can lead to confusion in the common situation where you do not have any versions in your area of the file being copied. Suppose that, as in example 2 above, you want to copy A.FTN into your area. Let the latest version of A.FTN be 22 (or 3 or 107 or anything else). The file that is made in your area will have version number 1, due to the action of the New Version switch which is forced upon you by DCL. The version number portion of the file specifier is useful for more than distinguishing one version of a file from another; it can also have an absolute meaning in and of itself. This meaning is lost to you when you use DCL. This is especially bad if you use the Copy command for making backup copies of your files, as the backup copies will not reflect the original versions of your files.

Let's now look at some examples. Suppose that in your directory, the only FORTRAN source files you have are

A.FTN;3
B.FTN;3

Also suppose that there are similar files in user area DB3:[100,1] with the latest versions being

A.FTN;2
B.FTN;3
C.FTN;4

Using this, we may consider several examples.

```
(7) MCR: PIP B.FTN;3=DB3:[100,1]B.FTN
    DCL: COPY DB3:[100,1]B.FTN B.FTN;3
```

These commands will fail since B.FTN;3 already exists in your directory.

```
(8) MCR: PIP B.FTN;3=DB3:[100,1]B.FTN/SU
    DCL: COPY/REPLACE DB3:[100,1]B.FTN B.FTN;3
```

Here, by including the Supersede (Replace) switch, you force the previous example to work.

```
(9) MCR: PIP=DB3:[100,1]B.FTN
    DCL: COPY DB3:[100,1]B.FTN *.*
```

The MCR command will fail since the output file specifier defaults to

B.FTN;3 and that file already exists. The DCL command will succeed since the New Version switch forces the output specifier to B.FTN;4. To achieve the same effect under MCR, the command should be entered with /NV:

```
MCR: PIP =DB3:[100,1]B.FTN/NV
```

or

```
MCR: PIP /NV=DB3:[100,1]B.FTN
```

```
(10) MCR: PIP=DB3:[100,1]A.FTN
     DCL: COPY DB3:[100,1]A.FTN SY:
```

Both of these commands will work. In MCR, the new file will still have version 2 and will thus appear to the File Control Services to be older than the already existing version 3. In DCL, the new file will have version 4 and will thus appear to be a new file. If you have version 3.0 of RSX-11M-PLUS or version 3.0 of Micro/RSX, you may perform the copy from DCL without creating a new version,

```
DCL: COPY/NONEW DB3:[100,1]A.FTN SY:
```

which will force the new file to retain the version number of the original.

So far, we have been concerned with copying from one file to another. As noted, PIP is the Peripheral Interchange Processor, and one of its major functions is copying a file from one peripheral device to another. In Section 23.3 we will study how to use this for making backups of your files. For now, let's consider some other examples in which either the input or the output is not a file on your default disk. One possibility is to copy from a file to your terminal (TI:). This is equivalent to typing the file on your terminal and is how you get a terminal listing of a file under RSX. Although you could issue a DCL command to copy to TI:, DCL allows the special command **TYPE** for this. Thus, to look at the latest version of the command file DOITNOW, you would use the command

```
MCR: PIP TI:=DOITNOW.CMD
DCL: TYPE DOITNOW.CMD
```

Note that for a video terminal, CTRL/S and CTRL/Q may be used if the file size is greater than the screen capacity (24 lines). You may similarly copy your file to another terminal (which might be useful if it is a hard-copy terminal) by specifying the TTnn: device name for it rather than TI:. In this case, you cannot use the **TYPE** command if you are in

DCL; you must use a Copy command. For example, to list the same file on terminal 7, the commands would be:

```
MCR: PIP TT7:=DOITNOW.CMD
DCL: COPY DOITNOW.CMD TT7:
```

It is also possible to reverse the above procedures and copy from your terminal into a file. In MCR, you specify TI: as the input. DCL also has a special command for this form of copying; it is **CREATE**. Following the command itself, everything you type is taken to be the "contents" of the "file" TI: until you enter an End-Of-File (CTRL/Z) character. If you want to write a quick note to yourself, you might enter the command

```
MCR: PIP NOTE.TXT=TI:
DCL: CREATE NOTE.TXT
```

As discussed in the previous chapter, this is the most primitive form of creating a file because it gives you essentially no editing capability. It is suitable only for very short files or for ones in which exact spelling or appearance do not matter.

As a final example, let's set the output to be the system printer, which may be identified as CL: (the Console Listing pseudo-device) or as LPnn: (Line Printer number nn). Copying a file to this is equivalent to getting a printout of the file. For example, to get listings of all your MACRO-11 source files, you would use the command

```
MCR: PIP CL:=*.MAC
DCL: COPY *.MAC CL:
```

As we discuss in Chapter 15, you will not use this command if your system has print spooling.

14

File Maintenance

In the last chapter we learned how to use the utility PIP for copying files. You can also use PIP for performing a wide variety of file maintenance functions. We look at these, as well as a few others, in this chapter.

With the exception of setting default file protection (Section 14.5), all the file maintenance functions that we examine here are performed by PIP. Thus, under MCR, these functions are requested by issuing various commands to PIP. Since something other than a copy is being done, a command line switch must be used to specify the exact function. Either the command form

```
PIP output=input/switch
```

or

```
PIP input/switch
```

is used, depending on whether output is generated. In DCL, for the various file maintenance functions, command names are used which have been chosen to directly signify the function to be performed. From DCL, you may not (and indeed need not) realize that most of these functions are performed by PIP. Once you have mastered these basic file maintenance functions, you may wish to turn your attention to Chapter 22 where we discuss some commands of a more advanced nature that augment the capabilities presented here.

14.1 Obtaining a File Directory

Once you have more than just a few files in your directory, you will find it easy to lose track of what you have. When this happens, you will want to get a list of the names of the files in your area. Such a listing is known as a file directory. RSX offers several possible forms of directory, differing in the amount of detail presented. We will first consider only the standard form; we will then show how to request the other forms.

In MCR you use the PIP Listing switch **/LI** to request a standard directory listing. In DCL you use the command **DIRECTORY**, which is typically shortened to **DIR**. These commands cause the standard directory listing to be generated. For each file, the file name (including version number), the number of disk blocks used to store the file, a file status code, and the date and time of file creation are printed. Following this, the total number of files in your directory and the total number of disk blocks that they occupy are printed.

Remember that the mechanism for creating a directory listing is the utility PIP and that the basic PIP command includes both an output and an input specifier. Thus, under MCR, the basic command form for obtaining a directory listing is

```
MCR:PIP output=input/LI
```

The output specifier directs where the directory listing is to go; if it is not given (the = is also omitted), TI: is assumed by default. This is by far the most common case, so the MCR command is normally just

```
MCR:PIP input/LI
```

which is a natural simplification of the more general form. In DCL, the command syntax is designed around the assumption that you will normally want to get the directory on your terminal. The command for this is

```
DCL:DIR input
```

which is indeed simple. To put the listing elsewhere, however, requires a special switch:

```
DCL:DIR/OUTPUT:output input
```

where the command switch **/OUTPUT:** may be shortened to **/OUT:** or

even /O:. Note that whether you are in MCR or DCL, the output specifier may refer to either an actual disk file (if it is to be in your user area, it will show up in the directory listing, but it will show as having zero blocks, since it is in the process of being made) or to a device such as a printer. Putting a directory list into a file is not as absurd as it might at first appear; it is useful for documenting the contents of your directory at a certain point in time, such as when you deliver programs to a customer.

The input specifier identifies those files that will be listed in the directory; if it is not given, a specifier of *.*;* (all files in your user area) is assumed by default. In this case, if the directory listing is to be printed on your terminal, the basic commands simplify even further:

MCR: **PIP /LI**

DCL: **DIR**

By including an input specifier in the command, you can restrict the files that appear in the listing. For example, if you want to see only the latest versions of all your files, you would use the command

MCR: **PIP *.*/LI**

DCL: **DIR *.***

To get a directory listing of the latest versions of all FORTRAN and MACRO-11 source files as of 23 July and to save this in a disk file, you might use the command

MCR: **PIP JULY23.DIR=*.FTN,*.MAC/LI**

DCL: **DIR/OUT:JULY23.DIR *.FTN,*.MAC**

Note that we have used multiple input file specifiers, separated by commas, as discussed in Section 11.3. As a final example, if you want to know when the latest version of the file BIGBOY.TSK was made, you could request a directory listing of just that one file:

MCR: **PIP BIGBOY.TSK/LI**

DCL: **DIR BIGBOY.TSK**

So far, we have considered only the standard form of the directory listing. This is the one that you will most likely find to be of the greatest use, but three other options are available. These are each selected by

adding a switch to the basic Directory command; the command syntax is otherwise unchanged. In a brief directory, only the file names are listed. In a summary directory, only the total disk block and file counts are given. A full directory listing contains greater detail than the standard format—the extra details will seldom be of interest to you. In MCR, the switches to request these three optional forms are: **/BR** (Brief), **/TB** (Total Blocks), and **/FU** (Full). In DCL, the switches are **/BRIEF**, **/SUMMARY**, and **/FULL**, which are typically shortened to **/BR**, **/SU**, and **/FU**. For example, to determine how much disk space you are using for your object files, you could use the command

```
MCR: PIP *.OBJ;*/TB
DCL: DIR/SU *.OBJ;*
```

This will list the total number of object files in your user area and the total number of disk blocks that they occupy, but it will not list details about the individual object files.

14.2 Deleting Files

Sooner or later you will find yourself with files in your user area that you no longer want. When this happens, you will need to know how to get rid of them. You can do this in one of two ways—deleting or purging. Deleting has the meaning that you would expect. Purging has meaning only in a system such as RSX that allows multiple versions of a file. Purging deletes the old versions of a file without disturbing the more recent versions. Both file deletion and purging are done by the utility PIP.

To delete a file in MCR, you use a PIP command with the Delete switch (**/DE**). In DCL you use the command **DELETE** which is commonly shortened to **DEL**. The form of this command is simply

```
MCR: PIP input/DE
DCL: DEL input
```

where **input** is one (or more separated by commas) file specifier. When using the Delete command, the version of the file to be deleted must be explicitly stated. This may be any of the following:

```
;*      to delete all versions
;0      to delete only the latest version
```

```
;-1      to delete only the earliest version
;n       to delete only the version #n
```

Thus, the command

```
MCR: PIP TEST.FTN/DE
```

will be rejected. This is a safety feature to guard against accidental deletion of the latest version of a file. Note, however, that in DCL the command

```
DCL: DEL TEST.FTN
```

may or may not be rejected. In older versions of RSX, it will be accepted but not as a normal Delete command. Instead, it will be translated into a Selective Delete command—we discuss this in Section 22.7. In the latest versions of RSX, the above command will be rejected. This change is part of the effort to remove nonstandard commands from RSX DCL.

For example, to delete all object and task image files for files A1 and A2, you could use the command

```
MCR: PIP A1.OBJ;*,A1.TSK;*,A2.OBJ;*,A2.TSK;*/DE
DCL: DEL A1.OBJ;*,A1.TSK;*,A2.OBJ;*,A2.TSK;*
```

To purge a file in MCR, you use a PIP command with the Purge switch (**/PU**). In DCL you use the command **PURGE**. The form of this command is simply

```
MCR: PIP input/PU
DCL: PURGE input
```

where **input** is one (or more separated by commas) file specifier. When using the Purge command, the version of the file to be deleted has no meaning and may not be stated.

The Purge command is more powerful than the Delete command because it does some of your bookkeeping for you. The versions to be deleted are determined as part of the purging action. Normally, all versions of each file specified in the command except the latest are deleted. The Purge command is especially convenient for cleaning out your user area after a series of edits and compiles. A common use of this command is

```
MCR: PIP *.*/PU
DCL: PURGE *.*
```

which results in the retention of only the latest version of each file in your area.

In its full form, the Purge command includes an integer value **n**. In MCR this is appended to the switch **/PU**; in DCL its inclusion requires use of the switch **/KEEP:**

MCR: **PIP input/PU:n**

DCL: **PURGE/KEEP:n input**

The form discussed above results from this with n = 1 being chosen by default. The integer n (greater than or equal to 1) specifies how many versions are to be saved. More specifically, if N is the latest version of the input file, then versions N, N − 1,..., N − n + 1 will not be deleted. Versions N − n and all lower will be purged. When all versions are present prior to the Purge command, this is equivalent to stating that n versions will remain after purging. This simplification can be misleading. Consider the command

MCR: **PIP TEST.FTN/PU:2**

DCL: **PURGE/KEEP:2 TEST.FTN**

If your directory contains versions 4, 2, and 1 of file TEST.FTN (version 3 having perhaps been previously deleted via a Delete command), PIP will delete versions 2 and 1, ignoring versions 4 and 3. Thus, following the purge, only version 4 will remain—the same result that would have been obtained with a **/PU:1** command. Another way of looking at this example is that the **/PU:2** command directs PIP to purge all but the two most recent versions—PIP's definition of the "two most recent versions" might not be the same as yours. Although confusing, this normally will not be a problem as you will most likely want to purge all old versions of your files.

14.3 Renaming a File

Another useful file maintenance activity is the renaming of an existing file. This does not result in the creation of a new file, nor does it change the contents of the existing file—it merely changes the directory entry associated with the file. Thus, it is significantly more efficient than copying the file (to get the new file name) and then deleting the original file (to get rid of the old name).

Renaming is done by using PIP with the Rename switch (**/RE**). If you are in DCL, you can use the command **RENAME**, which is commonly

shortened to **REN**. The form of the Rename command is:

```
MCR: PIP newname/RE=oldname
DCL: RENAME oldname newname
```

The **newname** and the **oldname** specifiers must refer to files on the same device; these need not, however, be in the same user area.

If a portion of the new specifier (ufd, file name, type, or version) is not explicitly stated, or if it is a wildcard, then it is set to the corresponding field in the input specifier—i.e., it is not changed. For the Rename operation to make sense, at least one of these four fields in the new specifier must be different from that in the old specifier, or else the "new" file name will be the same as the old.

Let's look at some examples. The command

```
MCR: PIP TEST.FTN;1/RE=A.FTN
DCL: REN A.FTN TEST.FTN;1
```

takes the latest version of A.FTN and changes its name to TEST.FTN;1. The command

```
MCR: PIP TEST.*;*/RE=A.*;*,B.*;*
DCL: REN A.*;*,B.*;* TEST.*;*
```

takes all files with a file name of A or B and changes the file name to TEST. The type and version specifiers are not changed. This last example can lead to an error, as PIP may attempt to assign the same new name to two different input files (e.g., if A.MAC;1 and B.MAC;1 existed, both could not be renamed to TEST.MAC;1).

The Rename command is especially useful when you are working with several user areas, as it allows you to "move" a file from one area to another. We discuss this in Section 21.3.

14.4 Fixing a File

Under the File Control Services (FCS), a file can become corrupted. This occurs when the file is opened for writing (as the result of a write statement) and not closed. Under RSX, normal task termination results in the automatic closing of any files that were opened. Thus, if your program terminates properly, you should not have any problems with your files. If, however, your program uses a file and is abnormally terminated, the file will not be properly closed. In this situation, the contents of your file may or may not be good.

Abnormal termination of your task can result from several causes. You (or a privileged user) may abort your task. This will normally happen if you notice something wrong with the way your task is running and you do not want to wait for it to finish. Your program may have an error that RSX considers to be especially bad (such as an Odd Address Trap). If this happens, you will have fun debugging your program. A third possibility is that your system may crash, in which case not only your task, but every other program running as well, will die.

If a task terminates abnormally with any files open for writing, and if the operating system is still there to do something about it (not a system crash), FCS will lock these files. A locked file cannot be read from or written into. The file is locked to inform you that, due to the abnormal program termination, the contents of the file may be garbage. You can determine whether or not a file is locked by obtaining a standard directory listing; a status code of L signifies a locked file. If you wish to use a locked file, you must first unlock it before you can do anything else with it.

To unlock a file in MCR you use PIP with the Unlock switch (**/UN**). In DCL you use the command **UNLOCK**. The form of the command is

```
MCR: PIP input/UN
DCL: UNLOCK input
```

Once you have unlocked a file, you must determine whether it has been corrupted or not. If you know that your program was only reading from the file, not writing to it, then you can trust the file to still be good. (In this case, you should, if possible, rewrite your program to open your file for reading only; then, even if your program terminates abnormally, FCS will not lock the file.) If you were writing to the file, most likely everything written until your task stopped will be in the file, but you will not be able to use it. Each file has a header associated with it. The file header contains various information about the file. One important piece of information is how big the file is. A new file has an initial size of zero blocks; the actual size is not put into the header until the file is closed. Thus, all your information may be in the file, but FCS will think that, since the file size is zero, nothing is there. You can detect this by looking at the file size in a directory listing. To fix this condition, you must use the End Of File command. This directs PIP to figure out where the end of your file is and to update your file header accordingly. Note that with older versions of RSX, this command is available only under MCR. Again you use PIP, this time with the End-of-File switch (**/EOF**).

The format of this command is

```
MCR: PIP input/EOF
```

With the latest versions of RSX (version 4.2 of RSX-11M, version 3.0 of RSX-11M-PLUS, and version 3.0 of Micro/RSX) this capability was added to DCL. In DCL, the Set File command was added to allow the setting of certain file attributes. One of these is the End_of_File attribute, which allows you to fix the size of a file after you have unlocked it. This command is

```
DCL: SET FILE/END_OF_FILE input
```

PIP cannot determine the exact end of your file. It will always give you everything that you put into the file, but normally it will also give you a little bit of junk at the end. This is certainly better than losing everything. You can remove the extraneous stuff by editing the file. Since this might involve working with strange characters, it may be necessary to use TECO.

For example, suppose your program is writing answers into the file RESULTS.OUT. Absentmindedly, you log out; this aborts all currently active tasks that you started from your terminal. When you log back in and try to read your answers, the file will be locked. A directory listing will show you that the size of the file RESULTS.OUT is zero blocks; you will also see the L for locked. To fix things, first unlock the file and then set the file size,

```
MCR: PIP RESULTS.OUT/UN
     PIP RESULTS.OUT/EOF
```

If you have the latest version of RSX, you may effect both of these commands from DCL,

```
DCL: UNLOCK RESULTS.OUT
     SET FILE/END RESULTS.OUT
```

Otherwise, you will need to issue the End_of_File command as an MCR command,

```
DCL: UNLOCK RESULTS.OUT
     MCR PIP RESULTS.OUT/EOF
```

In any case, once you have done all this, you should look at what is in the file and edit it as necessary.

14.5 Setting File Protection

The entire concept of file protection under RSX is unduly complicated. Depending on your work environment, the topic may be of no interest to you. Even if you are concerned about protection for your files, you will normally find the defaults chosen by RSX acceptable. Nonetheless, you may want to control access to your files, in which case you must first understand the protection system.

The file protection system controls the ability of users to access files. To do anything with a file, a user must be granted the corresponding access rights to that file. An attempt to access a file without the corresponding access rights results in a privilege violation. Files-11 defines four possible forms of file access

R = read
W = write
E = extend
D = delete

The distinction between write and extend is rather obscure and not worthy of clarification here.

Under Files-11, every file has an owner. Typically, the owner will be the user who created the file. The owner of a file is identified by his UIC. The concept of file ownership allows Files-11 to recognize four categories of user, based on UIC:

SY (system) —any UIC with a group number less than or equal to 10 (i.e., a privileged user)
OW (owner) —the UIC of the file owner
GR (group) —any UIC with the same group number as the owner
WO (world) —any other UIC

It is important to stress that these categories of user are defined relative to the owner of the file, not relative to the UFD of the file. With numbered directories, the UIC of the file owner will often be the same as the UFD, and this distinction can be easily overlooked. It is common to speak of another user as being in the same group as your directory. This is incorrect; with named directories, it is meaningless, since the UFD is not even of the same form as a UIC.

I must also be precise in defining what a UIC is. When you log in, you use a UIC to identify yourself. This log-in UIC is used to initially set a value known as your protection UIC. If you are nonprivileged, your

protection UIC can never be changed. Thus, the MCR command **SET /UIC** is a misnomer in that it does not affect your protection UIC. Instead, it changes your default directory, which in older systems was referred to as your default UIC. A privileged user, however, can change his protection UIC. When I speak of classifying a user according to UIC, I refer to the protection UIC.

For each file, Files-11 allows the four types of access to be selectively granted or denied to each of the four categories of user. This implies sixteen combinations of user and access type. Files-11 implements this by storing a 16-bit access code for each file. Default values are typically used, but you can override these. The access rights that are usually given to a file are:

```
SY  = RWED
OW  = RWED
GR  = RWED
WO  = R
```

If one of your files has these access rights, it means that any user with a system UIC, you (the owner), and anyone else in your group can do anything to the file; anyone else can read the file but not change or delete it. If you do not like these defaults, you choose your own protection codes.

Before discussing how to change the access rights to your files, we need to consider in a little more detail how a user accesses a file. Suppose, for example, you issue a command to print the file DR2:[265,1]TEST.FTN. This is what happens. Since there is a comma in the ufd portion of the file specifier, the ufd is interpreted as referring to a numbered directory. Files-11 then forms the file specifier DR2:[0,0]265001.DIR, which identifies the UFD corresponding to the user area containing TEST.FTN. This is a UFD, but it is still a file, and, as such, has access rights of its own. If you do not have read access to the UFD, you will get an error message citing a privilege violation and your command will be rejected. If you have read access to it, Files-11 will then read through the directory until it finds the entry for TEST.FTN. (If there are several entries, it will select the one with the highest version number.) Files-11 then examines the protection code for this file. If you have read access to it, your request to print it will be honored; otherwise, you will get a privilege violation at this point.

In general, to do anything with an existing file, you must pass two tests—one for access to the appropriate UFD, and one for access to the

file itself. For many actions, such as copying or printing a file, or running a task, you need read access to both the UFD and the file. To delete a file, you need delete access to the file, and read and write access to the UFD. To create a new file, you need read and write (and possibly extend) access to the UFD.

In the rest of this section I will show you how to set the access rights to your files. Since you own your files (more precisely, since you have write access to their UFD), you can set their access rights as you see fit. This involves two possible procedures: one to set the protection codes for existing files, and one to define a default protection code to be used for new files. You should not, however, overlook a basic point. If you want to keep other users away from your files, the easiest and safest way to do so is to deny them read access to your UFD. (If you do this, they will not even be able to get a listing of your directory to find out what files you have.) Since your UFD is a file in the Master File Directory [0,0], you cannot do this yourself. Changing the protection code for a UFD requires write access to the MFD, which is only available to a privileged user. If you want to deny other users read access to your UFD, ask your system manager.

After all these preliminaries, we can finally consider how to set protection codes for your files. Just as the topic is rather complicated, so too is the command structure rather awkward. If you are in MCR, you can change the protection for a file by using PIP with the Protection switch (**/PR**). If you are in DCL, you can use the command **SET PROTECTION**, which is commonly abbreviated to **SET PRO**. The command is of the form

```
MCR: PIP input/PR/xx:yyyy

DCL: SET PRO:xx:yyyy,xx:yyyy input
DCL: SET PRO input (xx:yyyy)
```

Here, we show two forms of the DCL command. Both are acceptable; the first is more recent and is now considered preferable. In all command forms, the input file specifier names the file(s) for which access rights are to be changed. The **xx:yyyy** is a particular protection code; up to four of these may be included in one command. In MCR, each **xx:yyyy** is preceded by a slash. In DCL, multiple protection codes are separated by commas. In the preferred form, these follow the **PRO:** and precede the file specifier. In the alternate form, the entire set of protection codes must be enclosed in parentheses and follows the file specifier. In all cases, the **xx** specifies the type of user for whom protection is to be changed—it can be either **SY**, **OW**, **GR**, or **WO**. The **:yyyy** is the pro-

tection code, which specifies the new access rights. It can include any of the letters **R**, **W**, **E**, or **D**—each letter included allows the corresponding form of access. A totally empty protection code is perfectly valid; it means that no access rights are to be granted to the corresponding class of user. If a particular xx user code is not included in the command, the access rights for that class of user are not changed. For example, if you want to change the file protection for all files of type FTN so that other users in your group will have only read access and nonsystem users in other groups will have no access, you can use the command

```
MCR: PIP *.FTN;*/PR/GR:R/WO
DCL: SET PRO:GR:R,WO *.FTN;*
DCL: SET PRO *.FTN;* (GR:R,WO)
```

Note the absence of a protection code field after **/WO**, which implies that no access rights are granted. Also note the absence of user codes **SY** and **OW**, which implies that you do not want to change the access rights for system users or for yourself.

The Set Protection command has a relatively awkward syntax, which is difficult to remember. If you are in DCL, this is a perfect example of a situation where the query command form is preferable. Using this style, the above example becomes

```
DCL>SET PROTECTION
File? *.FTN;*
Code? (GR:R,WO)
DCL>
```

It is important to note that the Set Protection command can be used to set access rights only for existing files. Any files subsequently created will assume certain default protection values. These defaults are formed as follows. First, in any RSX system, a set of file protection defaults is chosen by the system manager for each disk drive in the system. These will apply to every file in every user area on that disk and will typically be set as mentioned earlier. On older RSX systems, there is no way for you to override these default values—they will be used for any new files that you create. With version 3.0 of RSX-11M-PLUS and Micro/RSX, a new feature, known as user settable file protection, was added. (This capability is not available with RSX-11M.) This allows you to set your own file protection defaults, which override the general defaults. There are two ways you can set your own defaults. The easiest way is to have your desired protection defaults entered into the system account file by your system manager. Then, whenever you log in, your file protection

defaults will be set for you. Alternatively, you can set your protection defaults whenever you wish by using the Set Default Protection command,

```
MCR: SET /DPRO=[yyyy,yyyy,yyyy,yyyy]
DCL: SET PROTECTION=(xx:yyyy)/DEFAULT
```

The Set Default Protection command is roughly similar in form to the Set Protection command. In both the MCR and DCL forms, the **yyyy** protection code sets the file access rights as explained earlier. In the MCR form, the various classes of user are not explicitly identified, instead, they are expected to be in the order System, Owner, Group, and World. Thus, the command

```
MCR: SET /DPRO=[R,RWED,RWED,]
```

gives system users only read access to all future files that you will create, allows you and others in your group all forms of access, and denies all forms of access to other users. Note that the absence of a protection code for the World class in this example does not imply keeping the current access rights as it would in the Set Protection command—instead, it is interpreted as a protection code allowing no forms of access. To maintain the current rights, the MCR form of the Set Default Prodtection command allows an asterisk as a special protection code. For example, the command

```
MCR: SET /DPRO=[*,*,*,]
```

maintains the current access rights for system users, yourself, and users in your group, but directs that other users be denied any access to all future files. In the DCL form, the Set Default Protection command more closely resembles the Set Protection command. The various **xx:yyyy** have exactly the same meaning, are separated by commas, and can be omitted if no change to the current access rights are desired.

It is important to remember that the Set Protection command is used to change access rights for a file that already exists and that the Set Default Protection command is used to set access rights for any files that you will subsequently create. Also, the Set Default Protection command is part of the user settable file protection feature, which is only available with version 3.0 of RSX-11M-PLUS and Micro/RSX. If controlling access to your files is important to you, and if your system does not offer user settable file protection, you must remember to use the Set Protection command whenever you make new files. Since a new

version of an existing file is a new file, this means that you would need to do this after any editing, compiling, task building, etc. One way to automate this would be to include a Set Protection command in a Log-out command file (see Section 20.4). If you do this, you should realize that your files will not be protected until you log out and thus, while you are still working, another user can gain access to a newly created file of yours.

It might seem that file protection is at best clumsy and more likely annoying. This is especially true if your system does not allow you to set your own default file protection. Very often, the best approach to file protection is simply to ignore it. In a benign environment, there is no reason to assume that someone else will want to mess with your files. If you feel the need to worry about these things, then use the above techniques to reset the various access rights as you see fit. Just remember not to deny read access to yourself (use **WO** for world, not **OW** for owner), for, if you do, not only will you not be able to read your own files, but you will not be able to reset the access rights for them.

15

Using a Printer

Sooner or later you will want to get a printout of something. You may, for example, do all your work on a video terminal. This is very convenient for writing and editing your programs, but every now and then you will want to get a hard-copy listing of the latest version of your program. This is an example of taking data that is currently in a file (in your user area on the system disk) and printing it. As a different example, when you run your program, you might want to have it print its answers. You can always print your answers (and on RSX-11M you have to do it this way) by first writing them into a file and subsequently printing the file. From your point of view, this may be unnecessarily cumbersome—you simply want the answers to go from your program to a sheet of paper, without having to bother about any intermediate files. To do these things, you need to use a printer; in this section we discuss how that is done.

First, you need to have a printer. As silly as that sounds, you might not have one. On a very small PDP-11 system, your only resource for obtaining hard copy might be a printing terminal such as a DECwriter. In this case, you would use the printing terminal the same way as you would any other terminal. You may log in to that terminal and do your work on it. You would list a file by copying it to your terminal. (In MCR, you would use the command **PIP TI:=file**; in DCL you would use **TYPE file**.) Similarly, you would design your program to write its answers directly to TI:. Alternatively, if you are working on a video terminal and the hard-copy terminal is not being used, you could list a file on it by copying the file to it (**PIP TTnn:=file** or **COPY file TTnn:**). These are all very straightforward operations. In the remainder of this

chapter I will assume that your system has a real printer, and we will discuss the special commands that are available for using it.

15.1 Direct vs. Spooled Printing

Conceptually speaking, printing a file is easy. Your system has a printer; you have the file; you send the file to the printer and that is that. In practice, because RSX is a multiuser system, things are seldom that easy. Suppose you were to print a file by sending it directly to the printer. Suppose further that, while your file was being printed, some other user were to send a file to the printer. What would happen is this. Two separate tasks would each be sending data, one line at a time, to the same printer. The lines would come out in some intermingled order on the printer, resulting in garbage. Thus, if you do use a printer this way, you have to make sure that no one else will try to use it also. This is practical only on a very small system.

Most likely your system includes a feature known as print spooling, which allows several users to use the printer without getting things all mixed up. Only on a very old or very small system is it likely that you will use a printer by sending data directly to it. If this applies to you (if your system does not support print spooling), you use the printer as follows. A printer is a device, so you can use PIP to transfer a file to it. You do this using the Copy command. You can identify the printer by its physical device code. The device type for a printer is LP. If you have only one printer, it is printer 0, which is identified as **LP0:** or simply **LP:**. A second printer would be printer 1, **LP1:**, and so on. Alternatively, you can use the Console Listing device. This is a pseudo device that is assigned (by the system manager) to one of the printers, normally LP:, and is identified by the device code **CL:**. (The use of the pseudo device code CL: instead of an actual physical device code such as LP: or LP1: is similar to the use of SY: for the system disk rather than the actual disk name, DM1: or DR: etc.) You can print a file using a Copy command such as

```
MCR:PIP LPnn:=file
DCL:COPY file LPnn:
```

or

```
MCR:PIP CL:=file
DCL:COPY file CL:
```

This will print the file; all that you have to worry about is someone else also printing a file. There are two ways to take care of this. In a very small installation, you can simply ask the other users not to use the printer until you are done. If this is not feasible, you will have to make it impossible for anyone else to use the printer. You can do this by using the Allocate command. (This is not possible on RSX-11M systems that do not have multiuser protection. It is definitely possible that if your system is so small that it does not support print spooling, it also will not support multiuser protection—these are both system generation options.) The Allocate command gives you ownership of a device; the device is then said to be your "private" device. If you own a device, no one else is allowed to use it. (We discuss the Allocate command and the concept of private devices further in Section 23.1.) In MCR the command to allocate a device is **ALL**; in DCL it is **ALLOCATE**, commonly abbreviated **ALL**. The command syntax is the same whether you are in MCR or DCL (this is also true of the De-Allocate command discussed next),

MCR: **ALL ddnn:**

DCL: **ALL ddnn:**

where **ddnn:** is the physical device code. When you are done using the device, you should deallocate it so that it will be available to other users. The format of this command is

MCR: **DEA ddnn:**

DCL: **DEAL ddnn:**

(Note that in DCL you must include the **L** in **DEALLOCATE**. Normally you could use the three-character abbreviation, but this is a special case, since DEA is taken as the abbreviation of an entirely different command, Deassign.)

Suppose that you wish to print the file TEST.LST on printer 0. To avoid complications, you must first make yourself the owner of the printer. Similarly, after the file has been printed, you want to give up ownership so that some other user may have the printer. The command sequence to do this is:

```
MCR>ALL LP:
MCR>PIP LP:=TEST.LST
MCR>DEA LP:
```

or

```
DCL>ALL LP:
DCL>COPY TEST.LST LP:
DCL>DEAL LP:
```

We now turn our attention to the concept of print spooling. The word "SPOOL" is an acronym for Shared Peripheral Operations On-Line. With spooling, all requests for a particular device are funneled through one special task. This task accepts all requests for the device, puts them into a queue, and processes them one at a time. Spooling is a general concept, but we will consider it specifically for the control of printers. When you request that a file be printed, you are said to be spooling the file. Once the file has been spooled, it remains in the appropriate print queue until it has been printed.

Under RSX, the spooling process is accomplished by two or more tasks. One task, called the Queue Manager, keeps track of all the files that are waiting to be printed. This is the task you ask to print your files. There also is one task for each line printer in your PDP-11 configuration—each task is known as a Line Printer Processor. Each Line Printer Processor "owns" one printer. No other task, not even PIP, is allowed to use the printer. For simplicity, you should picture the Queue Manager as maintaining one queue for each printer. (Things may be more complicated than this.) Whenever one printer is available, the Queue Manager takes the name of the next file out of the appropriate queue and tells the appropriate Line Printer Processor to do the printing.

Since all requests for output to a printer go through the Queue Manager, it has complete control over what gets sent to the printer when. Thus, it can make sure that output from different users does not get mixed up. This is the major advantage of spooling. The other advantage lies in the manner whereby you spool a file. You enter a request to the Queue Manager. It reads this request, verifies that the file exists, and returns control to you. You can then go ahead and do other things without waiting for the file to be printed. You also basically have this capability without print spooling since, under RSX, you can run several tasks simultaneously. Thus, if you use PIP to print a file, you can also start other tasks. You will not, however, be able to use PIP for anything else (such as getting a directory listing or deleting a file) until the file has finished printing. There are other advantages inherent in using the printer via the Queue Manager. For instance, you can control when the file will be printed and you can direct that the file be deleted after printing.

15.2 Issuing a Print Request

The most common way to spool a file is by entering a request to the Queue Manager. Alternatively, certain system utilities may make a file and then automatically spool it for you. Also, you can write a program so that after it puts all its answers into a file, it will then spool the file for you. Finally, unique to RSX-11M-PLUS is a concept known as transparent spooling which we will discuss later.

Of these various methods, direct communication with the Queue Manager is the only one that allows you to specify all the various options that are generally available, such as making multiple copies and controlling flag pages. The others simply spool the file using default assumptions for most of the options. For example, a file that is automatically spooled will be printed as soon as possible—there is no way of requesting that the printing be delayed.

The most common examples of automatic spooling by a utility are the spooling of a listing file by a language processor and the spooling of a map file by the Task Builder. If you ask the FORTRAN compiler to make a listing file you may also request that it be spooled when the compiler finishes. We discuss how you can control this in the appropriate sections later in this book.

If you write a program in MACRO-11, you can spool a file from within the program by using the **PRINT$** macro. (This is described in the I/O Operations Reference Manual.) You may also spool a file from within a program written in FORTRAN. In this case, you do so when you close the file by using the **DISP='PRINT'** option in the Close statement. On some systems this will also cause the file to be deleted after it is printed—check with your system manager to determine what will happen on your system.

With the exception of the two special cases just discussed, you normally will spool a file by a direct request to the Queue Manager. You do this via the Print command. In MCR this is **PRI**; in DCL it is **PRINT**, which is normally shortened to **PRI** or even **P**. In the simplest form of this command, you only name the files to be printed. In this case, the command syntax is the same in both MCR and DCL:

```
MCR: PRI files
DCL: PRI files
```

You can specify as many files as you wish, and you can use wildcards in the file specifiers. For example, to print the latest version of TEST.FTN and all versions of TEST.DAT you could use the command

```
PRI TEST.FTN,TEST.DAT;*
```

Each time you issue a Print command, the Queue Manager creates what is called a print job. As in the example above, one job can consist of many files. You can modify the basic form of the Print command by including job switches or file switches. A job switch modifies certain aspects of the entire print job, whereas a file switch modifies the treatment of only some of the files in the job. With these switches, the format of the Print command becomes more complicated:

```
MCR: PRI /jobswitch(es)=files/fileswitch(es)
DCL: PRI /jobswitch(es) files/fileswitch(es)
```

These two command forms are very similar. The Queue Manager is a relatively recent addition to RSX, and the design of its user interface follows that of DCL. Thus, even if you are in MCR, Print commands will have a definite DCL flavor. The only syntax differences occur when job switches are included. In MCR an equals sign must be used before the file specifier, whereas DCL requires a blank; also, in DCL, the blank after the command word **PRINT** becomes optional. In the command forms above, note that each switch must be preceded by a slash. Also note, however, that although the command syntaxes are very similar, the names of the various switches may or may not be the same for MCR and DCL. As with many other RSX commands, many of the possible switches will be of no interest to you. The ones that you might find useful are described below.

The Queue Manager processes one complete job at a time. It takes the name of each file in the job (in the order given in the Print command) and directs the appropriate Line Printer Processor to do the printing. In addition, it may also direct that some flag pages be printed. There are two types of these—the job flag page and the file flag page. Each job can be preceded by one job flag page. This page has your UIC and the job name (which, by default, is the file name of the first file to be printed) on it in big letters. It serves to differentiate different jobs and is most useful in a large installation where a computer operator separates the output as it comes out of the printer. In addition, each file in a job may be preceded by a file flag page. This page has the complete file specifier (file name, type, and version) on it in big letters. It serves to separate the listings of different files in your job and is most useful if the contents of the file do not identify the file itself. (In your source code, you can put a comment at the top of each page that identifies the name of the file, the name of the program, etc. In general, this is a good thing to do.

You cannot do this so easily with data files; the file flag page is more useful in this case.) You can use two job switches to determine whether or not you will get these flag pages. By default, you will get the job flag page unless you specify otherwise. If you do not want the job flag page, include the No Job Flag Page job switch. In both MCR and DCL, this is **/NOJO**. (This switch was introduced with version 4.0 of RSX-11M.) The default for the file flag pages is system dependent. To ensure that you get them, include the File Flag Page job switch; to ensure that you do not get them, use the No File Flag Page job switch. In both MCR and DCL, these are **/FL** and **/NOFL**, respectively.

Certain types of files that you will print, such as listings and maps, are nicely formatted. Each page has a top and a bottom margin. Others, such as a long source file that you have made, may not be so nice. When printed, they may use every line on the page, leaving no margins. The line printed over the perforations is hard to read and, if you put your printouts into a binder, the top or bottom lines on each page will be almost impossible to read. Generally, if you put a form feed character (this is a special character, CTRL/L, with ASCII value 12 [decimal] or 14 [octal]) into your file, your printout will advance to the top of the next page at that point. By judicious use of form feeds (as well as comments and blank lines), you can convert a sloppy-looking source file into a professional looking product. (As simple as this is, many otherwise good programmers do not bother. Their programs look awful.) Sometimes it takes more effort to put form feeds into a file of answers being produced by your program, as you may literally have to count each line that is being written, which is a nuisance. As a convenience, the Queue Manager will, upon request, simulate form feeds. With this feature, you can ensure that, whether you have form feeds in your files or not, each page that is printed will have adequate top and bottom margins.

You can use the Length job switch to specify the length of a page. In both MCR and DCL, the form of this is **/LE:n**, where **n** is the maximum number of lines that you want to have printed on any page. As each line is printed, the Line Printer Processor increments a counter that keeps track of the total number of lines printed. Whenever a form feed is found in your file, the count is reset to zero. If the line count ever reaches **n**, a form feed is forced, and the line count is reset to zero. The default value of the page length **n** is zero. This value has the special meaning that the file should be printed as is—no extra form feeds should be inserted. To choose a page length, determine how big your paper is and what size margins you want. Most often printing is done at 6 lines to the inch. Normal size paper is 11 inches top to bottom,

so one page can hold 66 lines. If you want a 1-inch margin at the top and bottom, you should specify a page length of 54. This is typically a good choice.

The last job switch that may be of interest to you is the After switch. This switch specifies that the print job should not be started until after a certain time. If you have a big print job, you might use this out of courtesy to others—you would direct that your job be printed at night rather than during the day when other users might want to use the printer. This switch is rather awkward to specify. In MCR its form is

```
/AF:hh:mm:dd-mmm-yy
```

All the colons, hyphens, and various values must be included. In order, these values are: the hour (0 through 23) and the minute, the day, the month (the first three letters of the name in English), and the year (0 through 99). In DCL the form of the After switch is

```
/AF:(dd-mmm-yy hh:mm)
```

The various date and time components have the same meaning as in the MCR form, but they are in the somewhat more natural order of date first and then time. Note that in the DCL form the date and time must be enclosed in parentheses.

Let's now look at some examples. Suppose that TEST.DAT is a long data file that you would like printed. There are no form feeds in this file; you would like form feeds to be inserted after every 54th line to ensure adequate top and bottom margins. You would use this command:

```
MCR:PRI /LE:54=TEST.DAT
DCL:PRI /LE:54 TEST.DAT
```

As a second example, suppose that you want to print all your .LST files. To reduce the amount of paper, you will do without the file flag pages. Even so, you anticipate a few hundred pages, so you want the printing to be done late at night. If the date is March 23, 1986, you would use a command such as

```
MCR:PRI /NOFL/AF:0:0:24-MAR-86=*.LST
DCL:PRI /NOFL/AF:(24-MAR-86 0:0) *.LST
```

In addition to the job switches, two file switches might be of interest to you. One is the Delete switch and its counterpart, the No Delete switch. The Delete switch specifies that after it has been printed, a file should be deleted; the No Delete switch specifies the opposite. In MCR

these switches are /DE and /NODE, respectively; in DCL they are /DEL and /NODEL. You will often print files (such as listings, maps, or temporary output from a program that you are debugging) that you do not need to keep. Once you have a hard copy you can afford to throw away the files. The Delete switch lets you do this automatically. The default is No Delete.

You must be careful if you use the Delete switch. The files that you specify in the Print command are processed from left to right. Initially, the default condition is No Delete. Whenever you use either the Delete or the No Delete switch, the default is changed accordingly. Suppose that you use the command

```
MCR: PRI A.LST/DE,A.FTN
```

You are telling the Queue Manager to print A.LST, delete it, and then use /DE as the default for the remainder of the command line. Thus, the file A.FTN also will be deleted after being printed. Since you presumably do not want to delete your source file, you should either reset the default to No Delete,

```
MCR: PRI A.LST/DE,A.FTN/NODE
```

or reverse the order of the file specifiers

```
MCR: PRI A.FTN,A.LST/DE
```

In DCL, you need not be so careful. If you specify /DEL for one file, /NODEL is assumed for you for any further files. Thus, the command

```
DCL: PRINT A.LST/DEL,A.FTN
```

is translated into

```
MCR: PRI A.LST/DE,A.FTN/-DE
```

There is another reason to be cautious about using the Delete switch. A printer is a mechanical device and as such is subject to failure. If the sprocket holes in the page rip, the paper may jam. The printer will go on printing and eventually will finish, whereupon the Queue Manager will delete your file. All you will have is a sheet of paper with a lot of black ink and some holes in it. It is often better to print your files, get your output, and then use the conventional Delete command to get rid of the files you no longer need.

The other file switch that you might need to use is the Copies switch. This allows you to get multiple copies of the file being printed. In both

MCR and DCL, its form is **/CO:n**, where **n** is the (total) number of copies you want. The caution concerning the Delete switch applies to the Copy switch as well; if you specify it in a list of files, it applies to all successive files. DCL is again more forgiving than MCR if you are sloppy; it automatically resets the number of copies to 1 if another file is specified.

If yours is a large PDP-11 facility you might have more than one printer. For example, you might have one printer that normally is loaded with regular full-size (roughly 15-inches) paper and another that has narrow (8 1/2-inch) three-part, no-carbon-required forms. Similarly, even if you have only one printer, you might have various types of paper (forms) for it. In situations such as this, when you issue a Print request, you will also want to specify which printer and/or what type of paper should be used. You can do this by identifying the print queue into which you want your job to be put and/or the form type that you want to have in the printer. The Queue Manager can handle several different print queues and several Line Printer Processors. Each printer is defined as having a particular type of paper loaded into it which can change at any time; the interrelationship among all these factors can be complex and is entirely at the discretion of your system manager. To find out just how to do specialty printing at your installation, ask your system manager.

15.3 Working with the Print Queue

So far we have discussed the entering of jobs into a print queue via the Print command. The Queue Manager also allows you to do certain things once a job has been entered into a print queue. The Queue Manager commands that are likely to be of interest to you are those for finding out what jobs are in the print queue and for deleting a job from the queue. In MCR you do this by using the Queue command (**QUE**) with a switch to specify what you want to do; in DCL you use various special command words.

You can use the Queue List command to see what is currently in a queue. The simplest form of this command is

```
MCR: QUE /LI
DCL: SHOW QUEUE
```

If the Queue Manager on your system controls several queues, this command will give you information on every one of them. (There will always

be at least one print queue; this default queue has the name PRINT.) Also, if your system is RSX-11M-PLUS, this command will give you information on all the batch queues. (We discuss batch processing in Section 25.2.) In either case, you might want to restrict the displayed information to that for one particular print queue. In this case, the Queue List command looks like this:

```
MCR: QUE queue: /LI
DLC: SHOW QUEUE queue
```

where **queue** is the name of the print queue.

A sample queue list is shown below. This corresponds to an RSX-11M system, so there are no batch queues. Also, this system has only one printer, so there is only one print queue. I show the MCR command; the listing you get will be the same if you are in DCL.

```
MCR>QUE /LI
** PRINT QUEUES **
PRINT   => LP0
  [265,1]    ARCHIVE    ENTRY: 29                ACTIVE ON LP0
       >     1 DR2: [265,1]ARCHIVE.TXT;2           COP: 2
             2 DR2: [265,1]LOGIN.CMD;1             COP: 2
  [265,1]    MARCH86    ENTRY: 30
             1 DR2: [265,1]MARCH86.RPT;1
LP0     => LP0
```

For each queue, the names of the queue and the printer(s) associated with it are first displayed on your terminal. This is followed by a list of all jobs in the queue, in the order in which they will be processed, with the currently active job first. Each job is identified by a UIC, a job name, and an entry number. The job name is, by default, the same as the name of the first file in the job. The entry number is assigned by the Queue Manager. Following the job identifier is a list of files to be printed. These are also listed in order, and each is preceded with a number showing its place in the print job. A greater-than symbol (>) preceding a file entry is used as an arrow to indicate that it is currently active. Special requests, such as multiple copies or file deletion, are indicated after the file entry when appropriate.

If you issue a Print request and some time later you get a queue listing and you do not see your job in it, this means that your job has finished. Alternatively, if you see your job appearing with lots of other jobs ahead

of it, you know that it is likely to be a long time before your output is ready. In addition to checking the status of your print job, you can use the Queue List command to obtain information to be used subsequently with the Queue Delete command.

You can use the Queue Delete command to remove either an entire job or one file in a job from a print queue. You typically would use this command if, after requesting the printing, you decided that the contents of the file(s) were useless or no longer needed. To use this command, you must know how to identify the print job, and if you wish to delete only a particular file, you also must know the sequence number of that file in the job. You can obtain this information via the Queue List command. If you are not a privileged user, you can delete only those jobs that you requested.

There are two ways to identify a print job. One is via the job name, and the other is via the entry number for the job. To use the job name, you must also specify the queue name; if you use the entry number, that is sufficient. In the example above, the first print job can be identified as either job ARCHIVE in queue PRINT or as entry 29. The entry number form is more convenient to specify, but you must examine a queue listing to determine the number. You can, however, normally deduce the name of the print job from the name of the first file in it.

To delete an entire print job, you use the command form

```
MCR: QUE queuename:jobname/DEL
DCL: DELETE/JOB queuename jobname
```

or

```
MCR: QUE /EN:nn/DEL
DCL: DELETE/ENTRY:nn
```

To delete a single file from a job, you use the command form

```
MCR: QUE queuename:jobname/FI:mm/DEL
DCL: DELETE/JOB queuename jobname/FI:mm
```

or

```
MCR: QUE /EN:nn/FI:mm/DEL
DCL: DELETE/ENTRY:nn/FI:mm
```

where **mm** is the sequence number of the file in the job. Note that in

DCL the complete command name is either **DEL/JOB** or **DEL/ENTRY** (abbreviated to **DEL/J** or **DEL/E**) where the switch **/JOB** or **/ENTRY** distinguishes this command from the more familiar one used to delete a file.

For example, if your print job is the first in the list in the example above and you want to delete the second file in it, any of these commands will work:

```
MCR: QUE PRINT: ARCHIVE/FI: 2/DEL
MCR: QUE /EN: 29/FI: 2/DEL
DCL: DEL/JOB PRINT, ARCHIVE/FI: 2
DCL: DEL/EN: 29/FI: 2
```

15.4 Transparent Spooling

The last topic of interest regarding the use of printers is transparent spooling. This feature exists only on RSX-11M-PLUS and Micro/RSX. With this feature you can send data to a printer without explicitly putting it into a file and subsequently spooling it. Instead, you can pretend that you are sending it directly to the printer. What actually happens is that the operating system takes all your data and puts it into a temporary file, which is then spooled and deleted. You do not see this process—hence the term transparent spooling.

This feature imitates an old-fashioned single-user computer system. On a single-user system, when your program was running, if it wanted to write to the printer, it did so. Since no one else was running other programs, there never was any conflict. Old versions of FORTRAN developed the convention that logical unit number (LUN) 6 referred to the line printer—thus, to print a line, you wrote it to unit 6. This convention still influences the RSX operating system. (As we will see in our discussion of building a task, unit 6 is assumed by default to refer to the Console Listing device CL:.) You can use these same programming techniques with either RSX-11M-PLUS or Micro/RSX. You can write to unit 6, which will result in your program's output going to whichever printer has been declared to be the console listing device. Somewhere along the way, this output will be intercepted and put into a disk file instead, but it will get to the printer eventually.

With transparent spooling, you can also use PIP to copy a file to a printer. The command

MCR: **PIP LPnn:=file**

DCL: **COPY file LPnn:**

which is how you print a file if you do not have print spooling, will work if you have transparent spooling.

16

Language Processors

Several programming languages are available for use on the PDP-11 series under RSX. It is not my purpose in this book to discuss the actual writing of a program in any of these languages—however, several points concerning the use of the languages are worth mentioning. The languages we examine are:

MACRO-11 (assembly language)
FORTRAN-IV
FORTRAN-IV-PLUS
FORTRAN-77
BASIC
BASIC-PLUS-2
COBOL
COBOL-81

The system function that you use with a program written in one of these languages is known as a language processor. In general, a language processor is a translator, taking the user-written source code and producing instructions that the computer can directly understand; these are known as object code. Such a translator can be classified as either a compiler or an interpreter.

A compiler is the more common and less sophisticated of the two. It does not allow you to interact with it. A compiler takes as input a file containing source language statements and produces as output a file containing object code. The source file must have been previously created, e.g., via an editor. You must then use the Task Builder (Chapter 17) to create from this (and other) object code an executable task image. Finally, you must request RSX to run the task.

An interpreter is different from a compiler in that it interacts with

you and the operating system. Typically, an interpreter has an editing capability that allows you to enter (or change) source code directly. This source code is maintained in memory and is only optionally saved on disk. The interpreter also can link this object code and cause the resulting task to be executed. During task execution, the interpreter is, in a certain sense, still in charge. Thus, you have an on-line debugging capability.

An interpreter is probably easier for the beginning user to use than is a compiler, since the interpreter functions as somewhat of a miniature operating system. (Note that the language itself need not be easier to learn.) An interpreter is definitely easier to use for short programs meant to be run at most a few times; in fact, many interpreters can be used to make the computer act as a giant calculator. The defects of an interpreter are that it restricts you to a relatively inflexible means of using the computer system. This is not surprising, since an interpreter must offer, within one task, capabilities corresponding to those offered by several system functions. Thus, the editing capability offered by an interpreter typically is primitive compared to that of any of the editors discussed in Chapter 12. Although an interpreter produces both object code and a task image, neither of these is saved. For a program written in an interpreted language, the steps corresponding to compiling and task building must be repeated every time the program is run. Finally, the convenient debugging capability that is intrinsic to the manner whereby an interpreter executes a task is always present, whether it is needed or not, and results in an unavoidable decrease in efficiency.

As an operating system, RSX is primarily intended to support multiple task execution in real-time. Its multiprogramming capability, although powerful, is not a major design goal; rather, it is derived from the multitasking capability. Emphasis in RSX is placed on efficient loading and efficient execution of tasks. Thus, the concept of an interpreter is at odds with the basic philosophy of RSX. Accordingly, only one language, BASIC, is offered under RSX as a true interpreter. A more sophisticated version of this language is BASIC-PLUS-2, which is offered under RSX as a cross between an interpreter and a compiler. All the other languages available with RSX are compilers.

16.1 Using a Compiler

The language processors for MACRO-11, FORTRAN-IV, FORTRAN-IV-PLUS, FORTRAN-77, COBOL, and COBOL-81 all are compilers. One might argue that the MACRO-11 language processor is an assembler,

not a compiler. (The distinction is that an assembler works with a language that is specific to a type of computer whereas a compiler works with a higher-level language that can be used on many different computers.) I am concerned only with the distinction between a compiler and an interpreter, as detailed in the previous section. Under RSX, all compilers are used in a common manner. We first look at the general aspects of using a compiler and then discuss specific features of the various languages.

Although not necessarily a part of the basic RSX configuration (all language processors except MACRO-11 are separately purchased items; thus, many of the languages discussed in the next sections may not be present on your system), a compiler is nonetheless treated as a system function. As such, it can be invoked in any of the ways described in Section 11.2. The general form of a command is similar to that already seen for other functions:

MCR: **xxx output=input**

DCL: **xxx input**

Here, **xxx** is the name of the particular language processor. The **input** field defines the source program(s) to be compiled, and must always be included in the command. It will typically contain a single file specifier, although this is not always a restriction. When several input files are specified, they are separated by commas. Multiple input files have a special meaning in MACRO-11 which we will discuss later. For the higher level languages, multiple input files are concatenated to form the input to the compiler; this is legal but normally not advantageous. The input file specifier follows the general file specification rules detailed in Section 6.1. The file name must be included. The file type defaults to a standard type, which is dependent on the compiler being used. In some languages, an input file can contain more than one program unit (main program or subroutine). The compiler will always compile all program units specified by the input field. These program units are independently compiled in that errors in one do not affect another. If one program unit is contained in a file with other units and you wish to compile it separately, you must first (via an editor) move it to its own separate file.

In MCR, you explicitly name the output files in the command line. This is done via the **output** field, which may normally specify none, one, or two files. (With COBOL-81, a third file is possible.) The first output file is the object file, the second is the listing file; default file

types for these are OBJ and LST, respectively. The normal purpose of compiling a source file is to produce an object file that can be used as part of an executable task. If, however, no object file is specified, none is produced—you might do this if all you want is a list file. Similarly, if no listing file is specified, none is produced. The listing file, in addition to the source statements, contains other useful information such as sequence numbers, symbol tables and diagnostics (errors and warnings). Even when no listing file is specified, compiler diagnostics are printed on your terminal. The two possible output files are specified in the order **object,listing**.

In DCL, you normally do not name the output files. If they are created, they are given default names unless explicitly directed otherwise. These default names are formed by taking the name of the input file (if several input files are given, the name of the first is used) and using the appropriate default file type. The normal action is to produce an object file but no listing file. These choices can be overridden by switches appended to the name of the compiler itself. The form of these switches is the same for all the compilers available with RSX. To suppress generation of the object file, you use the No Object switch, **/NOOBJ**. To give a name to the object file other than the default, you use the Object switch followed by the desired file name, **/OBJ:objectfile**. To request a listing file, you use the Listing switch. In its simple form, this is **/LIST**, where the file name is formed using the defaults described above. If you want to give a different name to the listing file, you specify it after the Listing switch, **/LIST:listfile**.

When you generate a listing file, you will often want to read it, which implies that you will want to print it. Both FORTRAN and MACRO-11 allow you to request that the listing file be automatically printed. This process is known as spooling (see Section 15.1) and is controlled by the Spool and No Spool switches. The default action—whether a listing file is spooled or not—is system dependent. (It is often set for no spooling to avoid wasting paper.) In MCR these switches are **/SP** and **/NOSP**. DCL has no explicit switch to control spooling. Instead, an unduly contrived procedure involving the Listing switch is used. So far, I have shown the Listing switch placed after the compiler command. When you do this, DCL forces the Spool switch to be set for you. Alternatively, you can place the Listing switch after the source file name. When you do this, DCL forces the No Spool switch. To further complicate the issue, when you place the Listing switch after the source file specifier, you lose the ability to specify the name of the listing file. You are still able to control whether your listing file is spooled or not, but the mech-

anism is neither general nor obvious. This is essentially at odds with DCL's philosophy of having legible, syntactically consistent, self-documenting commands.

I will use the FORTRAN-77 compiler as an example for illustrating typical commands. The default input file type is FTN. In MCR the compiler name is **F77**. In DCL this compiler is invoked by using the command **FOR** with the **/F77** switch, that is, **FOR/F77**. Some typical commands to the FORTRAN-77 compiler are:

(1) MCR: **F77 TEST=TEST**

DCL: **FOR/F77 TEST**

This compiles TEST.FTN and produces object file TEST.OBJ. Default versions (highest existing version for input; highest existing version plus 1 for output) are assumed. No list file is made. Note that in MCR, when you omit the list file specifier, you also omit the preceding comma.

(2) MCR: **F77 TEST=TEST.FTN;3**

DCL: **FOR/F77 TEST.FTN;3**

This is similar to the above, except version 3 of the input file is used even if higher numbered versions exist. The same default version is used for the output.

(3) MCR: **F77 TEST,TEST=TEST**

DCL: **FOR/F77/LIST TEST**

This is the same as example 1 except that a listing file is also produced. The listing file is to have the name TEST.LST, with the standard output default version. In DCL the listing file will be automatically printed on the Console Listing device; in MCR the spooling of the listing file depends on how your system manager has set up the FORTRAN compiler.

(4) MCR: **F77 TEST,TEST/-SP=TEST**

DCL: **FOR/F77 TEST/LIST**

This is the same as example 3 except that you force the listing file to not be spooled.

(5) MCR: **F77 TEST,VER3=TEST**

DCL: **FOR/F77/LIST:VER3 TEST**

This is the same as example 1 except that the listing file is forced to have the name VER3.LST.

(6) MCR: **F77 TEST,TI:=TEST**

DCL: **FOR/F77/LIST:TI: TEST**

This is similar to example 5. The difference is that the listing file is specified to be your terminal. Because a terminal is not a file storage device, the effect of this is to produce a listing on the terminal immediately; no permanent listing file is created.

(7) MCR: **F77 VER3=TEST**

DCL: **FOR/F77/OBJ:VER3 TEST**

This is the same as example 1 except that the name of the object file is VER3.OBJ instead of TEST.OBJ. Note that in DCL, you must use the **/OBJ** switch to name the object file.

(8) MCR: **F77 ,TI:=TEST**

DCL: **FOR/F77/NOOBJ/LIST:TI: TEST**

In this example, no object file is produced. A listing is generated on your terminal but no list file is saved. You might use this command if you know that you have errors in your program and you want to see just what they are. Note that in DCL the switch **/NOOBJ** is required to suppress generation of an object file.

(9) MCR: **F77 =TEST**

DCL: **FOR/F77/NOOBJ TEST**

No output files are generated. Any diagnostics (compiler error messages) are printed on your terminal.

So far we have seen examples of giving single-line commands to a compiler. As discussed in Section 11.2, a system function may also be used in the multiple-line form. In MCR, this is entered by first typing the name of the compiler; in DCL, you must use the command **RUN** with the compiler name prefaced by a **$**:

MCR: **F77**

DCL: **RUN $F77**

When you do this, the compiler retains control of your terminal and will accept subsequent commands from you. In this case, it is ultimately necessary to exit from it. This is done via the command **CTRL/Z**, which is an end-of-file, indicating to the compiler the end of your commands. Note that whether you are initially in DCL or MCR, when you use a compiler in this manner, you will always use the MCR command forms.

In addition to the suppression or generation of object and listing files, it is possible to specify other options to the compiler. This is effected by including other switches in the command. In MCR, all switches are of the form **/xx**, which means that the compiler should do xx. If you wish the opposite action, the switch is specified as either **/NOxx** or **/-xx**, which instructs the compiler not to do xx. In DCL, switches have longer names but can be abbreviated. The location of these switches is dependent on the particular option. In MCR a switch is appended to either an input or output file specifier; in DCL all switches except the Listing switch are appended to the compiler name. The options controlled by these switches are compiler dependent and thus will be described in the separate sections for the various compilers.

16.2 FORTRAN

Digital Equipment Corporation offers two varieties of the familiar language FORTRAN. One of these is a rather simple and inexpensive implementation known as FORTRAN-IV. The other is considerably more sophisticated and expensive. It was originally called FORTRAN-IV-PLUS. Roughly concurrent with version 4.0 of RSX-11M, Digital introduced an upgrade to FORTRAN-IV-PLUS and changed the name to FORTRAN-77 to reflect the fact that the new version met the ANSI 1977 standards. Several enhancements to the language itself were made, most noticeably the addition of structured programming constructs. In terms of its interface to RSX, however, the language processor remained virtually unchanged. You can use the same command lines with either FORTRAN-77 or FORTRAN-IV-PLUS, with the possible exception that the name of the compiler itself will probably be different. The command for FORTRAN-IV-PLUS was as **F4P**; that for FORTRAN-77 is **F77**. Since FORTRAN-IV-PLUS is no longer a current product, we will consider FORTRAN-77 in our discussion. If you have the earlier version, you can do everything we say except you will have to use **F4P** instead of **F77**.

In MCR, the FORTRAN-IV compiler is invoked by the command **FOR**, and the command for the FORTRAN-77 compiler is **F77**. (If you still have FORTRAN-IV-PLUS on your system, you use the command **F4P**.) If you are using DCL, you use the generic command **FOR**. In this case, you get, by default, FORTRAN-IV. If you want FORTRAN-77, you must include the switch **/F77** after the command. (Similarly, to get FORTRAN-IV-PLUS, you include the switch **/F4P**.)

At the programming level, the two types of FORTRAN available are different in that FORTRAN-77 is a significantly enhanced superset of FORTRAN-IV. At the compiler level, however, the two languages are quite similar. Both compilers assume the input file type to be .FTN. FORTRAN-77 allows multiple input files; FORTRAN-IV allows only one input file. Loosely speaking, the compilers work by first translating the FORTRAN statements into MACRO-11 assembly language, which is itself then translated into the actual object code. Either FORTRAN compiler is used in the general manner described in Section 16.1. In addition, certain switches are available to specify various compiler options. Except where noted, these switches are common to both FORTRAN-IV and FORTRAN-77. The various switches assume default values when not explicitly set; the default values are intended to be the ones that you would normally elect. I indicate the normal default values; these may, however, have been changed by your system manager. The switches that you are most likely to use are described below.

We first consider the Debug switch. Both FORTRAN compilers allow source statements to have a D in the first column. When the Debug switch is not set, these lines are treated as comment lines. When the switch is set, the D is ignored and the remainder of the line is compiled as a normal statement line. The default value is No Debug. In MCR, the Debug switch is **/DE**. This is a poor choice of mnemonic because, to many other system functions, the switch **/DE** means "delete"; do not allow this inconsistency to confuse you. The Debug switch is included in the input file specifier. In DCL, the Debug switch is **/DLINES**, which can be shortened to **/DL**. Examples of compiling a file with Debug lines to be included are:

```
MCR: F77 TEST=TEST/DE
DCL: FOR/F77/DL TEST
```

The Spool switch controls the automatic printing of a listing file. When this switch is on, the listing file is spooled; when it is off, the listing file is produced but not spooled. In MCR, this switch is **/SP** and is included in the list file specifier, as in

```
MCR: F77 TEST,LIST/-SP=TEST
```

The default value is **/SP**, but this is often changed to **/-SP** to avoid printing a lot of listings. In DCL, this switch does not exist. Since setting the Spool switch makes sense only if a listing file is specified, DCL automatically sets it for you when you request a listing file. The logic here

is not obvious. In DCL, you can place the switch requesting a listing after either the compiler command name or the input file name. If you place the switch after the command, the Spool switch is forced on; if you place the listing switch after the input file name, the Spool switch is forced off.

The Listing Options switch specifies whether a listing file should be produced or not, and if so, the level of detail in the output listing. You have several choices concerning what is to be included in the listing file. In increasing order of detail, you can obtain: a summary of diagnostics (warnings or errors), a listing of the source statements (with sequence numbers), a storage map, and a listing of the generated assembly language code.

The syntax of the Listing switch differs totally between MCR and DCL. In MCR this switch is **/LI:n** and is placed after the listing file specifier,

```
MCR: F77 TEST,LIST/LI:n=TEST
```

The value of **n** determines the type of listing. The relationship between the choice of **n** and the form of the resulting listing depends on the choice of compiler. For FORTRAN-IV, you select the listing options desired and add the corresponding numbers given below to determine the value of **n**:

Value	List option
0	Diagnostics
1	Source program
2	Storage map
4	Assembly code

All values of **n** between 0 and 7 are valid. The default assumption is **/LI:3**—i.e., diagnostics, a source program listing, and a storage map. The FORTRAN-IV compiler also allows mnemonics to be used instead of the integer code for certain values of **n**. These are:

Mnemonic	Equiv. value	List option
SRC	1	Source and diagnostics
MAP	2	Map and diagnostics
COD	4	Assembly code and diagnostics
ALL	7	Everything

For FORTRAN-77, a different meaning is assigned to the value of **n**:

Value	List option
0	Diagnostics only
1	Diagnostics and source
2	Diagnostics, source, and storage map
3	Diagnostics, source, storage map, and assembly code

The default assumption is **/LI:2**. The FORTRAN-77 compiler does not allow mnemonics to be used to specify these choices.

In DCL, you can specify the Listing switch as simply **/LIS**. In this case, you will get the default listing form as indicated above. To obtain a different type of listing, you must use additional switches. Those that are available to you are **/SOURCE**, **/MAP**, and **/MACHINE**, along with their negative forms. These control, respectively, the listing of the source code, the storage map, and the assembly (machine language) code. The presence of any one of these switches in the command line causes the Listing switch to be automatically included. Also, the request for a listing option causes all less detailed listing options to be automatically requested. Thus, the following are equivalent:

```
DCL: FOR/LIS/SOU/MAP/NOMACH TEST
DCL: FOR/LIS/SOU/MAP TEST
DCL: FOR/LIS/MAP TEST
DCL: FOR/MAP TEST
```

The Array Subscript Checking switch exists only with FORTRAN-77. When present, it causes extra code to be generated so that every reference to an array is checked to verify that the array subscripts being used are within bounds. Any subscript that is either too high or too low (as determined by the dimension of the array) results in display of an error message when your program is run. Subscript checking can be an extremely useful tool for debugging a program as out-of-bounds array references can destroy other parts of the program, leading to very obscure malfunctions. It can, however, produce a significant increase in both program size and running time. The default is to not include subscript checking. In MCR the Subscript Checking switch is **/CK** and is included in the input file specifier. In DCL the switch is **/CHECK**

```
MCR: F77 TEST=TEST/CK
DCL: FOR/F77/CHECK TEST
```

The Trace and Sequence Numbers switches allow you to control the

level of traceback detail generated when an error occurs while your program is running. In general, whenever an error occurs during execution of a FORTRAN program, information pinpointing where the error occurred is displayed. This consists of where in the program unit the error was; which program unit (name of the main program or subroutine) the error occurred in; and how (via which sequence of subroutine calls, ultimately originating in the main program) that program unit was reached. Location within a program unit is identified by line (sequence) number; the line numbers are those given in the source listing.

The Trace switch is used with FORTRAN-77, and the Sequence Numbers switch is used with FORTRAN-IV. In MCR the Trace switch is **/TR:xxx** and the Sequence Numbers switch is **/SN**. In either case, the switch is included in the input file specifier. In DCL these switches are **/TRACEBACK:xxx** and **/LINE_NUMBERS**, which are commonly abbreviated to **/TR:xxx** and **/LINE**. In both the MCR and the DCL forms, the **xxx** in the Trace switch specifies the level of detail maintained by the traceback procedure.

FORTRAN-77 allows four choices of how the location within the program unit is specified. The keywords used to select these are the same for MCR and DCL. At the greatest level of detail (**ALL**), individual line numbers are identified. At the next level of detail (**BLOCKS**), lines are grouped into blocks and errors are identified as occurring "at or after" a certain line number. At the next level (**NAMES**), only the name of the program unit (no line numbers) is identified. At the least level of detail (**NONE**), neither the program unit nor the line number is identified. If the Trace switch is specified without the detail qualifier (simply **/TR**), then the default is **ALL**. It is important to note that if the Trace switch is not specified at all, then the default is to generate traceback information on a block basis. If you wish to suppress any traceback information, you must include the Trace switch and specify **NONE**.

FORTRAN-IV allows only two choices of traceback detail. If the Sequence Numbers switch is specified, internal sequence numbers are generated for every line. This corresponds to F77's **/TR:ALL**. If the No Sequence Numbers switch is specified, no sequence numbers are generated. This corresponds to F77's **/TR:NONE**. The default is to generate sequence numbers for every line of code. In MCR these switches are **/SN** and **/NOSN**. In DCL the switches are **/LIN** and **/NOLIN**.

With both compilers, the inclusion of traceback information simplifies debugging. Remember, however, that (similar to use of the Subscript Checking switch with F77) this increases both the program size and the running time.

16.3 COBOL

For use on the PDP-11 under RSX, Digital offers two versions of the COBOL language—COBOL, which is an implementation of the ANSI-1974 standard, and COBOL-81 which is an enhanced version. The manner whereby you use these two compilers is similar, but there are some differences. We discuss the two compilers together, identifying items unique to one or the other as appropriate.

The COBOL compiler processes your source file and produces as its primary output an object file. The default file type for the input file is CBL; the default for the object file is OBJ. In addition, a listing file (type LST) can be produced. COBOL-81 can produce a third output file known as a diagnostics file (default type DIA) which contains a summary listing of all diagnostic messages generated by the compiler. This can be useful if you need to locate only a few errors in a long program—it is easier than reading the entire listing file.

In MCR, the regular COBOL compiler is invoked by the command **CBL**. COBOL-81 is invoked by **C81**. The basic command forms are:

```
MCR: COBOL:     CBL obj,list=source
     COBOL-81:  C81 obj,list,diag=source
```

where **obj**, **list**, **diag**, and **source** are the names you supply for the various files. As an example, if you have written a COBOL program in file REP1.CBL, the command

```
MCR: CBL REP1=REP1
```

will compile it using ANSI-74 COBOL, producing an object file named REP1.OBJ. No listing file will be made. When you do not want a listing file, you simply omit the specifier for it; you can, as shown, also omit the preceding comma. With COBOL-81, the procedure is basically the same except that the comma before the listing file is needed to distinguish it from the diagnostics file, unless you wish to omit both. To compile REP1 using COBOL-81, producing a diagnostics file but no listing, you would use the command:

```
MCR: C81 REP1,,REP1=REP1
```

In DCL, the generic command **COBOL** is used for both compilers. By default, this will get you the COBOL-81 compiler; to get the regular COBOL compiler, you must place the switch **/C11** after the compiler

name. In DCL, the basic command forms are:

```
DCL: COBOL:      COBOL/C11 source
     COBOL-81:   COBOL source
```

As with other language processors, the generation of the various output files is controlled by the use of certain switches after the compiler name. By default, the object file is generated but not the listing (or, for COBOL-81, the diagnostics) file. The switches to override these defaults are **/NOOBJ**, **/LIST:listfile**, and **/DIAG:diagfile**. For example, to compile a COBOL program in file REP1.CBL, producing only the object file, you would use the command

```
DCL: COBOL/C11 REP1
```

To compile the same file with COBOL-81, producing only a diagnostics file, the command would be:

```
DCL: COBOL/NOOBJ/DIAG REP1
```

Note that in these examples I have chosen the names of the various output files to be the same as those of the input files, thereby distinguishing them on the basis of file type alone. This is not necessary (you could use any file names you wanted), but it is normally best to do things this way. The various files are logically interrelated, so they should all have the same name. Further, the procedure for building a task requires that the object file have the same name as the source file; we discuss this shortly.

You can modify the basic commands for compiling a COBOL program by including various switches in the command line. These switches can identify special properties of the source, object, or list files. In MCR, they are nonetheless all placed following the source file specifier—i.e. at the end of the command line. In DCL they are all placed after the compiler name. The complete command forms are:

```
MCR: CBL obj,list=source/switches
     C81 obj,list,diag=source/switches
DCL: COBOL/C11/switches source
     COBOL/switches source
```

You can specify several switches on the command line; each switch is preceded by a slash. Not all the switches that may be used will be of interest to you. We discuss the most useful ones below.

The ANSI standard for COBOL source statements specifies the 80-

column format left over from the days of punched cards. The Digital compilers allow you to use a shorter, more convenient format that is designed for use with a terminal instead of a deck of cards. Both compilers assume that you will use the terminal format for your source file. If your file is in ANSI (card reference) format, you must use the ANSI format switch. In MCR this is specified as **/CVF** (Convert Format); in DCL it is **/ANSI**. The switch names are the same for both compilers.

Both compilers allow you to include automatic checking of various operations when your program is run. This checking will discover many common errors, but it does so at the expense of making your task both larger and slower. With both compilers, you can check the values of subscripts and indices against their allowable bounds as defined by the corresponding Occurs clauses in your program. To include this checking you may use the Bounds switch; to specify that you do not want to check against bounds, you use the No Bounds switch. By default, bounds checking will be included. With COBOL-81, you may also check for improper nesting of Perform statements during program execution. The switches controlling this checking are the Perform and No Perform switches.Again, the default is to include this checking. In MCR, the Bounds and No Bounds switches are **/BOU** and **/-BOU**. The Perform and No Perform switches are **/PER** and **/-PER**. These switches are the same for both compilers (the (No) Perform switch does not pertain to regular COBOL). In DCL, the switches differ depending on whether you are using COBOL or COBOL-81. In both cases, you use the switch **/CHECK** to control checking. In COBOL, there is only bounds checking. Thus, the possible switches are **/CHECK** and **/NOCHECK**. In COBOL-81 there are more possibilities to control; for this, DCL uses subswitches to the switch **/CHECK**. Thus, for COBOL-81, you must use a compound switch of the form **/CHECK:xxx**. The allowable subswitches are **:BOU**, **:NOBOU**, **:PER**, and **:NOPER**. Use of **/CHECK** by itself causes both forms of checking to be included in your program. To suppress all checking, you can use either the No Check switch (**/NOCHECK**) or the None subswitch to the Check switch (**/CHECK:NONE**).

By including the appropriate file specifier in the command line, you request the generation of a listing file. This contains both the complete source code and any diagnostic messages. You can use various switches to control additional elements of this list file. With either compiler, you can request the addition of a cross-reference. This adds two cross-reference tables, one by data name and one by procedure name, to the basic listing file. In MCR, you use the switch **/CREF** with COBOL and the switch **/CRF** with COBOL-81. In DCL the switches are slightly dif-

ferent (**/CROSSREFERENCE** vs **CROSS_REFERENCE**) but there is a common abbreviation, **/CROSS**. With either compiler, no cross-reference is the default. For both compilers, you can also request maps of the Data Division and the Procedure Division. In MCR, with either compiler, the switch is **/MAP**. In DCL, with either compiler, the switch is **/SHOW:MAP**. With either compiler, the default is to not make the maps. With the regular COBOL compiler, you can control two other aspects of the listing. By specifying the No List switch, you can suppress the inclusion in the listing of source statements copied from a library file. When you do this, only the copy statement appears in your source listing. The default is to include these statements. By specifying the Object Location switch, you can include the object location (relative address) of the code produced for each program verb. The default is to not include this. The No List and Object Location switches do not have counterparts in COBOL-81. In MCR these switches are **/NL** and **/OBJ**; in DCL they are **/NOSHOW:COPY** and **SHOW:VERB**. Normally, when you generate a listing, the defaults for these various switches will be adequate. If you are doing detailed debugging of your program, however, you may need to use these switches to override the defaults.

The last switch that you may need to know is the Subprogram switch. This switch specifies that the source file being compiled contains a subprogram that does not use parameters from the main program—i.e., it does not contain the Procedure Division Using header. You can use this switch with either version of COBOL. In both MCR and DCL its form is **/SUB**. By default, it is not set.

In addition to producing an object file, the COBOL compiler (either one) produces what is known as a skeleton overlay descriptor file. You have no control over the name of this file; it has the same file name as your source file, and its file type is SKL. Correspondingly, you should always use the source file name for the name of the object file, or else subsequent operations will not work properly. Making a task from a COBOL program is a complicated process involving the use of what are known as overlays. (Overlays are outside the scope of casual use and hence are not discussed in this book.) To produce the overlay descriptor file that is needed for task building, you must combine the various skeleton overlay files from the various program units (main program and subprograms) that are needed for the task. Thus when you make an object file, the COBOL compiler automatically makes the skeleton file for you as well.

Under COBOL-81, the process of combining the skeleton files and

then building the task has been automated so that you do not have to understand what is really happening. All the necessary commands have been prepared and collected in what is known as an Indirect Command file. (We discuss these in Chapter 20.) To use this, you enter a special command:

```
MCR:@LB:[1,2]C81LNK files
DCL:LINK/C81 files
```

(The @ specifies that this is an indirect command). The **files** are the names of the files containing your various program units.

For example, if you have a main program in file REP1.CBL and subprograms in files IOSUBS.CBL and LIST.CBL, you could compile and then build under COBOL-81 with a sequence of commands similar to this:

```
MCR>C81 REP1,REP1=REP1/CRF
MCR>C81 IOSUBS=IOSUBS/SUB
MCR>C81 LIST=LIST/SUB
MCR>@LB:[1,2]C81LNK REP1,IOSUBS,LIST

DCL>COBOL/LIST/CROSS REP1
DCL>COBOL/SUB IOSUBS
DCL>COBOL/SUB LIST
DCL>LINK/C81 REP1,IOSUBS,LIST
```

Following all this (assuming that no errors are detected while compiling your programs), you could run the task by issuing the command

```
RUN REP1
```

Before doing all this, you should check with your system manager for exact details—some system dependent options may require you to use a slightly different form of the **@C81LNK** (**LINK/C81**) command above.

Under regular COBOL, you will not have a command file to do all the necessary things unless your system manager has made one especially for your installation. (The command file C81LNK is supplied as part of COBOL-81; no corresponding command file is supplied with COBOL.) In general, you will have to use a special utility called MERGE to combine the skeleton files and then use a complicated set of commands to build the task. The exact details of this process are system dependent; you should ask your system manager just how to do this.

16.4 MACRO-11

MACRO-11 is the assembly language for the PDP-11 series. The various features of the language itself do not depend on the operating system, as the manner in which you write a program in assembly language is the same whether you are running under RSX or any of the other operating systems available for the PDP-11. As with the other language processors that we discuss, I assume that you already know how to use the language itself.

One special interaction between your MACRO-11 program and RSX is worth mentioning. You can use the directive **.TITLE** in your program to define a title to be printed on each page if you request a listing. **.TITLE** is more important than this—it also defines the name of the object module produced by the Assembler. This name is used by the Librarian to identify modules within an object library (we discuss this in Chapter 24). The module name is taken as the first six nonblank characters in the title. If you do not specify a title, a default module name of .MAIN. will be used. It is good programming practice to always include a **.TITLE** directive in a MACRO-11 program.

Input to the MACRO Assembler is your source file; by default the file type is assumed to be MAC. Primary output is an object file (default type OBJ) and secondary output is a list file (default type LST). In MCR the basic MACRO-11 command line is of the form

```
MCR: MAC obj,list=source
```

For example, if your program is in REP2.MAC and you wish to compile it but you do not want a listing, you could use the command

```
MCR: MAC REP2=REP2
```

In DCL the basic MACRO-11 command is of the form

```
DCL: MACRO source
```

Generation of the various output files (object and/or listing) is controlled by the switches **/OBJ:objname**, **/NOOBJ**, **LIST:listname**, and **/NOLIST**. In DCL, the example above is

```
DCL: MACRO REP2
```

In most cases, a command in this very simple form will be all that you need.

With MACRO-11 it is possible to have more than one input file. Ap-

plications requiring this are normally limited to system level programming and are outside the scope of this book. I assume that all your applications will be limited to the straightforward assembly of individual files.

When MACRO-11 processes your source file, it may detect errors. If it does, and if you have specified a list file, the individual error messages will be put into that file; if you have not specified a list file, these error messages will instead be written to your terminal. In either case, a summary line stating the total number of errors will be written to your terminal. If, after invoking the MACRO Assembler, the next output to your terminal is a prompt (MCR> or DCL>), your assembly was successful—no errors were found.

The actions of the MACRO-11 Assembler can be modified by including various switches in the command line. Some of these may be of interest to you.

When you write your MACRO-11 program, you can include various directives. These control certain aspects of the assembly. You can also enter some of these directives via the command line—these will override any directives present in your source code and will remain in effect for the entire assembly.

The Assembler accepts certain function directives. These are assembly functions that you can enable or disable within your program via the directives **.ENABL arg** and **.DSABL arg**. Alternatively, you can specify these in the Assembler command by using the Enable and Disable switches. In MCR these are **/EN:arg** and **/DS:arg**. You can append these switches to either the object file or the source file specifier. In DCL these switches are **/ENABLE:arg** and **/DISABLE:arg**, which can be abbreviated **/EN:arg** and **/DI:arg**. In DCL, the switches are placed after the compiler name. In both MCR and DCL, **arg** identifies the particular function to be enabled or disabled. In MCR, you enter this in exactly the same form as you would in a directive inside your program. In DCL you use various keywords, which although closely related to the directive names, are not necessarily the same. Probably the only function that might be of interest to you is global mode. In MCR this is identified as **GBL**; in DCL it is **GLOBAL**. By default, this mode is enabled, in which case all symbols that are undefined in your program are assumed to be global symbols (i.e., they are defined in some other program module, which will be used when you eventually build a task). By disabling this mode, all undefined symbols are treated by the Assembler as errors and are marked as such in your listing with the code letter U. A common programming error is to mistype a name, which normally leads to an

undefined symbol. By disabling global mode, you can find these while you are writing and debugging your program. If, for example, you are writing program REP2 and you want to assemble it for the purpose of detecting errors (you do not yet want to produce an object file, but you will want a listing), you might use this command:

```
MCR: MAC ,REP2=REP2/DS:GBL
DCL: MAC/DIS:GLOBAL/NOOBJ/LIST REP2
```

MACRO-11 also offers a variety of Listing Control directives. These allow you to include or exclude various parts of the listing that the Assembler produces. You specify these in your source code as the directives **.LIST arg** and **.NLIST arg**. You can also specify them in the command line via the List and No List switches. In MCR, these are **/LI:arg** and **/NL:arg**, and they must be appended to the listing file specifier. In DCL the List and No List switches are **SHOW:arg** and **NOSHOW:arg**, and are placed after the command name. If you wish to include (or exclude) several different features, you must specify all of them in one switch, e.g., **/NL:arg1:arg2** or **/NOSHOW:arg1:arg2**. Normally, the default values for the various listing options should be adequate for your purposes, but you can change them (without editing your source program) in this manner. As with the Enable/Disable switches, if you are in MCR, the various **arg** codes that you might use are exactly the same as those you would use in a listing control directive; if you are in DCL, you use various keywords. For example, if you are done writing REP2 and you want to make an object file and also a final listing, this time without unsatisfied conditional statements (the default is to include these), you could use the command

```
MCR: MAC REP2,REP2/NL:CND=REP2
DCL: MACRO/LIST/NOSHOW:COND REP2
```

The other two switches that you might want to use also affect the list file. By default, every time you produce a list file, it is spooled. Often you will not want to print this file unless you need it to figure out errors. In MCR you can use the No Spool switch (**/-SP**) to suppress the automatic printing of the list file; you must append this to the listing file specifier in the command line. In DCL you control the spooling of a listing file by the placement of the switch **/LIST**; if the switch is placed after the command **MACRO**, the list file will be spooled; if the switch is placed after the source file name, the list file will not be spooled.

You might want a cross-reference as part of your listing. The default

is to not produce a cross-reference; to get one, you use the Cross Reference switch. In MCR this is **/CR**; in DCL it is **/CROSS**. To make the cross-reference, a special task has to run after the Assembler. You do not have to worry about this because it is automatically done; however, this task (CRF) must be installed for this to work. If you ask for a cross-reference and nothing happens, you will have to ask your system manager to install CRF for you.

16.5 Using BASIC and BASIC-PLUS-2

Under RSX, two versions of the language BASIC are available—these are known as BASIC (or BASIC-11) and BASIC-PLUS-2. BASIC-11 is essentially plain old BASIC. BASIC-PLUS-2 is a significantly enhanced version.

BASIC-11 is a true interpreter. Certain small features of the language may differ from what some might consider to be a "standard" BASIC, but the manner whereby it is used is unchanged. It contains its own editor and file handler. It interprets your program every time you want to run it. It may be used in the immediate mode of execution as a giant calculator. In short, it acts just the way you would expect. It does not depend upon the RSX operating system—in fact, once you are running BASIC-11, you can forget that your operating system is RSX. Thus, there is nothing further for us to discuss about BASIC-11 under RSX other than the command you use to invoke it:

MCR: **BAS**

DCL: **BASIC/B11**

BASIC-PLUS-2 is different. It offers many features not found in plain old BASIC and is significantly more powerful and flexible as a programming language. These features do not concern us here as it is not my intent to teach you how to write computer programs. What does concern us is that BASIC-PLUS-2 is not an interpreter. It does retain the editing and file handling capabilities of regular BASIC. Once you have your program written, however, you must compile it, producing an object file. You must then leave BASIC-PLUS-2 and use the Task Builder to build a task from this object file in much the same manner you would with an object file created by any other language processor. Finally, you must run the task using the MCR command **RUN**. Thus, BASIC-PLUS-2 does not shelter you from the RSX operating system the way that BASIC-11 does.

Whether you are in MCR or DCL, you first enter BASIC-PLUS-2. The command for this is:

MCR: **BP2**

DCL: **BASIC**

Once you are in BASIC-PLUS-2, you interact directly with it. At this point, it does not matter whether you were in MCR or DCL.

Once BASIC-PLUS-2 is running, you can edit and save source files just as you would with regular BASIC. If you want to run your program, however, things are different. Before you can do this, you must compile your program. This is done from within BASIC-PLUS-2 by using the Compile command, designated by the keyword **COM**. If your program is currently in memory (for instance, as the result of the command **OLD**) you can compile it directly with the command

```
COM
```

If your program is in a file, the command

```
COM file_specifier
```

will load the file into memory and then compile it. If you do not specify a file type, B2S is assumed by default. You can modify the Compile command to obtain double precision for all floating variables. In this case, the command is

```
COM/DOU
```

Note that all floating point variables and operations are either single or double precision—you cannot mix these types.

Once you have compiled your program, you must build a task from it. You use the RSX Task Builder to do this, with some help from BASIC-PLUS-2.While in BASIC-PLUS-2, you can then use the command **BUILD**. This constructs an indirect command file that contains all the commands required for task building. (We discuss task building in Chapter 17 and indirect command files in Chapter 20—for now, don't worry about them.) If all you need for your task is a main program, the Build command is of the form

```
BUILD main/switch
```

where **main** is the name of the file (default type .OBJ) previously made by using the Compile command. If your program uses subroutines (BASIC-PLUS-2 supports subroutines) the command is

```
BUILD main,sub1,sub2,.../switch
```

Here, the various **sub** entries are object files for the subroutines. If you have subroutines, they and the main program must have been compiled in the same precision—i.e., with or without the double precision option. Both the commands above include a switch following the object file specifiers. If you use any files in your program (if your source code contains an Open statement), you must use a switch here; otherwise, none is needed. Via what is known as the Record Management Services (RMS), BASIC-PLUS-2 supports a wide variety of file structures. For each type of file structure that your program uses, you must specify the appropriate switch. For simple applications (such as writing answers into a disk file for subsequent printing), you will use only sequential files, for which the switch is **/SEQ**. Thus, if TEST is a main program that writes to a disk file and does not use any subroutines, the Build command will be

```
BUILD TEST/SEQ
```

The Build command within BASIC-PLUS-2 creates an indirect command file that you can give directly to the Task Builder. This file has the same name as your program and has a file type of .CMD. To build your task, you must leave BASIC-PLUS-2 (via the Exit command), whereupon control returns to your CLI. You then invoke the Task Builder using the command form

```
MCR: TKB @prog
DCL: LINK @prog
```

Here, **prog** is the name of the program specified in the BASIC-PLUS-2 Build command. Finally, when the Task Builder is finished, you can run your program by using the Run command. This is the same for both MCR and DCL:

```
RUN prog
```

As an example of all this, suppose that you have a main program in the file TABLE.B2S which uses a subroutine in file AUX.B2S. For simplicity, assume that you have done all the necessary editing to these files already. To compile, build, and run, you would use the sequence of steps given below. Since our emphasis here is on your interaction with BASIC-PLUS-2 itself, I only show the example for MCR. Note that before you can enter a command to BASIC-PLUS-2, you must get the prompt "Basic2," as shown. This indicates that the previous operation has finished and that BASIC-PLUS-2 is ready to accept a new command.

In this example I also show the files that are made at the various steps. Until you get used to using BASIC-PLUS-2 under RSX, you will be amazed at the plethora of files that you will find in your user area.

```
MCR>BP2
Basic 2
OLD TABLE
Basic 2
COM                         TABLE.OBJ
Basic 2
OLD AUX
Basic 2
COM                         AUX.OBJ
Basic 2
BUILD TABLE,AUX             TABLE.CMD,  TABLE.ODL
Basic 2
EXIT
MCR>TKB @TABLE              TABLE.TSK
MCR>RUN TABLE
```

17

Building a Task

In the preceding chapters, we have discussed how to write a program and how to compile it. The next step is to create a task from it. You do this by using the system utility known as the Task Builder.

The Task Builder is used to link various object modules together to form a task. The Task Builder is a rather complicated program as it was designed to accommodate a large variety of tasks including those running in real-time or requiring very large amounts of memory. Most of the details concerning its possible use will be of no interest to you. (The official Task Builder manual fills an entire 2-inch binder.) In this section, we will consider only straightforward use of the Task Builder; even so, many things must be discussed. Be patient.

17.1 Introduction to the Task Builder

In general, a Task Builder command can consist of up to three parts: the basic command, which identifies the input and output files; various switches that specify special properties of these files; and options that modify the linking procedure. For simple task builds, only the basic command, with perhaps a few switches, is required. In this case, as long as you do not have a large number of input files, you can give your entire command to the Task Builder in a single line. When a more complicated command is required, you must use the multiple-line command form.

Before going any further, you should realize that the Task Builder in RSX corresponds to the system utility that in almost any other operating system is known as the Linker. For this reason, the DCL command for building a task is **LINK**. Under MCR, the command is taken from the official name of the utility; it is **TKB** for Task Builder.

The basic command to the Task Builder follows the standard rules for MCR and DCL:

```
MCR: TKB output=input
DCL: LINK input
```

Here, **output** and **input** are file specifiers. The generation of the various possible output files is controlled in the same manner as it is in the various language processors discussed earlier. MCR gives you complete control over each possible output file by producing only the ones that you name. DCL automatically makes certain output files for you; you can change these defaults by including switches in the command. In particular, DCL normally generates a task image file with a default name taken from that of the first input file.

As an example of this, suppose you have compiled a main program in a file called TEST.OBJ and that this main program calls a subroutine named F1, which is located in a file called MATH1.OBJ. To make a task of your program TEST, you must link the object modules in the files TEST.OBJ and MATH1.OBJ. If you want the name of the task image file to be TEST.TSK, the basic command to the Task Builder would be

```
MCR: TKB TEST=TEST,MATH1
DCL: LINK TEST,MATH1
```

Now, suppose you enter the same command but with the two input files specified in the reverse order:

```
MCR: TKB TEST=MATH1,TEST
DCL: LINK MATH1,TEST
```

In MCR, exactly the same task build will happen, since you explicitly name the output file TEST. In DCL, however, the task image file will now be called MATH1.TSK, since its name is taken, by default, from the first input file specified.

The example above represents a very simple task build in which a single-line command is sufficient. In many instances, this is the only means of using the Task Builder that you will need. Often, however, a more complicated task build, which does not fit this simple format, is required. There are two cases where this may happen to you. You may have too many input files to fit on one command line, or you may have to include certain options. The multiple-line Task Builder command provides for these cases.

In MCR, the multiple-line command is always started with the command

```
MCR>TKB
```

This invokes the Task Builder, which then accepts successive lines of input from you. The total number of these is variable. The multiple-line format always ends with the command

```
TKB>//
```

(two slashes followed by a carriage-return), which signifies the termination of input to TKB.

In DCL, the multiple-line form is started with the command

```
DCL>LINK
```

Successive lines of input are prompted for, processed, and accumulated by DCL until it determines that you have entered a complete command.

Because Task Builder commands may involve the multiple-line form, DCL cannot always translate your Link command into a single TKB command. DCL overcomes this problem by using indirect command files. We will discuss these at length in Chapter 20. For now, note that an indirect command file is a file of commands that may be given to a program so that they appear to have been typed in by you directly. Indirect command files are especially beneficial with the Task Builder because command sequences for the Task Builder are typically complex. Any time you enter a Link command, DCL forms an indirect command file from your inputs. It then gives this to the Task Builder. Finally, it deletes the indirect command file. Note that even if you enter a simple Link command—one that translates into a single-line TKB command—DCL will go through this process. Due to this extra overhead, linking under DCL is often noticeably slower than it is under MCR. Finally, note also that if the DCL Link process is somehow interrupted, the indirect command file that is temporarily used will not be deleted. If you find a file named ATLNK.TMP in your directory, this is how it got there.

17.2 Specifying Output Files

Strictly speaking, the Task Builder produces three possible output files—the task image file, the map file, and the symbol table file. Normally, you will want only one or two of these. In MCR you get only the files

for which you ask; you specify their names when you ask for them. In DCL, defaults apply to both the files you get and their names.

The primary output produced by the Task Builder is the task image file, which is assumed by default to be of type TSK. You will almost always want this. In addition to the task image file, you can produce map or symbol table files. Normally, your only use for a map file will be in debugging your program. Thus, it will be of occasional interest to you, but normally you will not want it. When you generate a map file, it is assumed by default to be of type MAP. You should never need a symbol table file for casual use.

In MCR the three output files are specified in the order

```
task_image,map,symbol_table
```

These are separated by commas as required. Any files that are not wanted are simply omitted. Note that if you do not want a certain output file but you do want the next one, you must include the comma separating them. If, for example, you have already built your task and you now want to generate a map, you would use a command of the form

```
MCR>TKB ,map=input
```

In the DCL command form, the default is to produce only the task image file and to give it the same name as the first input file. If you ask for a map file, its name is similarly formed. You can override these defaults by using the Task and Map switches. To suppress generation of the task image file, you use the No Task switch (**/NOTASK**). To change the name of the task image file, you use the Task switch with the desired file name as a parameter (**/TASK:name**). Similarly, you can get a map file with the default name by using the Map switch (**/MAP**). Finally, to get a map file with a different name, you include a file name with the Map switch (**/MAP:name**). You append these switches to the LINK command.

In the previous section, we examined an example with a main program in a file named TEST.OBJ that required subroutines in MATH1.OBJ. Continuing with this example, if you want to make a task image file TEST.TSK, the basic Task Builder command would be

```
MCR: TKB  TEST=TEST,MATH1
DCL: LINK  TEST,MATH1
```

Here, the default name (that of the first input file) supplied by DCL for the task image file is what you want. Now, if you want to link the same

input files but you want the task image file to be named TEST1.TSK, your commands would be

```
MCR: TKB  TEST1=TEST, MATH1
DCL: LINK/TASK: TEST1  TEST, MATH1
```

Note that in this case, in DCL, the switch **/TASK:name** is required to override the default file name. In MCR, the syntax is unchanged.

If, in addition to the task image file TEST.TSK, you also wanted a map file named TEST.MAP, the command would be

```
MCR: TKB  TEST, TEST=TEST, MATH1
DCL: LINK/MAP  TEST, MATH1
```

In the DCL command form, the Map switch is required to generate a map file; the name is again chosen by default. Similarly, you can change the name of the map file by using the Map switch in the form **/MAP:name.**

17.3 Specifying Input Files

As obvious as it may be, it is worth mentioning that you cannot successfully build a task unless you make available to the Task Builder all the required object modules. If your program is at all complex, it will probably be split into several program units or modules. References between these will involve function calls and/or global data items. The names of these functions or data items are known as symbols. The linking process requires that a definition be found for every symbol that is referred to in your program. These definitions may be located in various files, and if you do not name all of these as input files, the Task Builder will not be able to link your program.

Forgetting to name a required input file is a rather common user error. When it happens, the Task Builder will quit, giving you the following error message:

```
nn Undefined symbols segment xxx
```

Here, nn is the total number of undefined symbols and xxx is the segment name, which, for our purposes, can be ignored. If you have not generated a map file, then the names of the missing symbols will be listed on your terminal. If you have made a map file, they will be listed in it, and the Task Builder will not bother repeating them on your terminal. (They

will be listed at the end of the map file; the easiest way to see them is to use an editor to read the map file and jump to its end.) Note that this error message will only identify the missing symbol(s); it is your responsibility to determine what file (if any) each is in.

In general, the object modules used to form a task consist of one main program and an arbitrary number of subroutines. The simplest example is one in which there are no subroutines. In that case, only the file containing the main program need be specified. Even when several subroutines are used, the corresponding object modules may be included in the object file containing the main program so that only one input file is required. (This is not necessarily good programming practice.) Often, however, you will have to specify several input files. Although not necessary, it is recommended practice that the first input file specified be the one containing the main program unit. If there are not too many input files, you can include them on one line. It is then possible to use the single-line form of the Task Builder command. If you cannot do this, you must use the multiple-line command. (A second reason for using the multiple-line form is to enter options, as discussed in Section 17.6).

In the multiple-line command form, you can use as many lines as you wish to specify all your input files. Each line can specify one or more files. As with other system utilities in RSX, when several input files are specified on one line, they are separated by commas. This is true for both MCR and DCL. When you need to use more than one line of input files, however, things are not so straightforward; the syntax depends on the CLI you are using. In MCR, the last file specification in a line is never followed by a comma. If you do end the line with a comma, the Task Builder will expect another file specifier to follow on that line and will give you an error message when it does not find one. In DCL the rules are reversed. If you need to use several lines to name all the input files, you must end each line except the last with a comma to signify to DCL that you have further lines of input to enter. If you do not end a line with a comma, DCL assumes that you have entered all the input specifiers that are required for the linking process and will not give you a chance to enter any others.

In MCR, after the initial **TKB** command, successive lines of input file specifiers are not prefaced by any special words or symbols. The Task Builder simply assumes that all further lines contain input specifiers until it finds a line beginning with a slash. For example, to build task ABCD from object modules contained in the files A, B, C, and D, you could use the single-line command

```
MCR: TKB  ABCD=A,B,C,D
```

Alternatively, you could use multiple-line commands such as

```
MCR>TKB
TKB>ABCD=A
TKB>B,C,D
TKB>//
MCR>
```

or

```
MCR>TKB
TKB>ABCD=
TKB>A,B
TKB>C,D
TKB>//
MCR>
```

When using the multiple-line command form in MCR, note that once the Task Builder has been invoked, the output specifier and the equals sign must appear on the first line you enter, although (as shown in the second example) nothing else need appear on that line.

In DCL, the command to do the same task build could be given in single-line form

```
DCL: LINK/TASK:ABCD  A,B,C,D
```

Multiple-line commands also are possible, for example:

```
DCL>LINK/TASK:ABCD  A,
File(s)?  B,C,D
DCL>
```

or

```
DCL>LINK/TASK:ABCD
File(s)?  A,B,
File(s)?  C,D
DCL>
```

Note carefully that these last two examples do not correspond to the MCR examples above, even though the input files are split across the various lines of input in the same way. Instead, DCL puts all the input files that you specify into one line in the MCR command. (This action can have some subtle ramifications.) Thus the two DCL commands printed above both translate into this MCR command:

```
MCR>TKB
TKB>ABCD=
TKB>A,B,C,D
TKB>//
MCR>
```

Individual input files (and the output file[s] as well) follow all the normal rules for file specification. Thus, in the above examples, I have omitted the device, ufd, file type, and version specifications, thereby using the default values for these. Sometimes, input files must be taken from several user areas. In this case, some care is needed in specifying the device and ufd due to the manner whereby the Task Builder forms defaults for these. To examine this, let's suppose you have a main program in file A that calls subroutines in files BESSEL and CUBIC, all of which are in your user area [123,1], and that these call subroutines in file TRIG, which is in a general user area [100,1].

Let's first consider what happens in MCR. You might try to use the command

```
MCR: TKB A=A,BESSEL,CUBIC,[100,1]TRIG
```

to build task A from all this. This command will work. Another command that you might try is

```
MCR: TKB A=A,[100,1]TRIG,BESSEL,CUBIC
```

This command will fail. The reason for this is as follows. The Task Builder initially sets the default for the ufd portion of an input file specificier to your default directory. In this example, this is [123,1], which is used to complete the file specifier for the first input file, A.OBJ. The appearance of the explicit ufd specification [100,1] resets the default, and this new value is used for any further input file specifiers.Thus, the Task Builder looks for files [100,1]BESSEL.OBJ and [100,1]CUBIC.OBJ, which (presumably) do not exist. It is important to note that when the multiple-line Task Builder command is used, the default for the ufd is reset (to your default directory) at the beginning of each line of input specification. Thus, the command sequence

```
MCR>TKB
TKB>A=A
TKB>[100,1]TRIG
TKB>BESSEL,CUBIC
TKB>//
MCR>
```

will work. (The UFD of [100,1] is used as the default only on the line in which it appears; for the next line, the default reverts to [123,1].) Although we have used numbered directories in this example, the same rules apply for named directories. Defaults for the device specifier are handled in the same manner.

Now, in DCL, multiple lines of input specifiers are packed into one line as part of the command translation into MCR. Thus, the command sequence

```
DCL>LINK
File(s)? A,
File(s)? [100,1]TRIG,
File(s)? BESSEL,CUBIC
DCL>
```

will fail, since it is equivalent to the MCR command

```
MCR: TKB A=A,[100,1]TRIG,BESSEL,CUBIC
```

which, as noted above, fails.

Similar considerations apply to the output file specifiers. If no device or ufd specifiers are present, then the usual defaults apply to all output files; an explicitly specified device or ufd applies to all successive output files. Note, however, that the defaults for the output files do not affect those for the input files, even if a one-line Task Builder command is being used.

So far we have considered only the use of object files as input (default type OBJ). A second type of input file, known as an object library (default type OLB) also is allowed. (The use of object libraries is discussed in Chapter 24) The syntax for including these in the input specifier is the same as for object files except that object library files must be denoted as such by using the Library switch (Section 17.5). You can specify files of types OBJ and OLB on the same command line. The specification of object libraries is especially important for accessing system modules, as is discussed in the next section.

17.4 Accessing System Object Modules

Before discussing how to access object modules that are part of the system, let's briefly discuss why it is necessary for you to do so. We will use an example from FORTRAN; similar arguments hold for all other languages. Suppose you have a program that includes the statement

```
A = SQRT(B)
```

What you intend by this statement is that the square root of B be taken and that A be assigned this value. Presumably, it is of no concern to you what mechanism is employed to determine the square root of B. In actuality, a subroutine is required to calculate the square root function—this subroutine is supplied as part of the FORTRAN system. (This is what you should expect, since if FORTRAN allows you to use **SQRT** to designate the square root, it should correspondingly provide the means of determining the square root.) The fact that you do not have to write the subroutine for calculating the square root function does not necessarily mean that you can ignore this subroutine when you build the task. You must include it as part of your task, and depending on how your RSX system is structured, you may accordingly have to tell the Task Builder where it is.

In our example above I used the square root function—I could equally well have chosen many others, for instance the exponential, absolute value or modulo functions. All of these are functions that you explicitly name in a program. Perhaps it is therefore reasonable to expect you to remember which functions are used in a program and to correspondingly identify each subroutine at task build time, although doing so would be clumsy. Continuing with an example of a FORTRAN program, many other subroutines are typically needed but are never explicitly called—notable among these are the many subroutines for input and output. As you probably will not even know the names of these subroutines, you clearly cannot be expected to identify them at task build time.

The use of system object libraries solves this problem. An object library is a file (default file type OLB) containing many object modules. The file is organized so that these modules are readily accessible on an individual basis. When the Task Builder encounters an object library as an input file, it selects from it those object modules required by the task it is building and ignores the others. (The manner whereby the Task Builder processes object libraries is discussed in greater detail in Section 24.1.) Note that this differs from normal Task Builder operation. When regular object files are specified as input, the Task Builder takes all modules, whether required or not, and includes them in the task being built. The principle behind the use of system object libraries is as follows. All object modules that are in some broad functional sense interrelated are organized into an object library. To access any of these system object modules, you need merely specify the object library as a Task Builder input file. You need not worry about identifying the object

modules that are required by your program or about making your task unduly large by including modules that are not required.

As an example of this, let's consider FORTRAN-77. The aggregate of subroutines used to implement various features of this language is referred to as the Object Time System (OTS). These are supplied in a single object library. The name and location of this file can vary, but it is normally known as F77OTS.OLB and is commonly on the system library device (LB:) in user area [1,1]. A typical way to access the FORTRAN-77 OTS is to include the file specifier (note the Library switch) in the Task Builder input,.

```
MCR: LB: [1,1]F77OTS/LB
DCL: LB: [1,1]F77OTS/LIB
```

Check with your system manager for the exact details for your system.

Although this is much simpler than specifying individual subroutines, it is still somewhat of an inconvenience. From your viewpoint, it should not be necessary to do anything special to obtain required system object modules when building a task. (This is in accord with the philosophy that a good operating system should bother you with as few details as possible.) The Task Builder provides a means of maintaining this "invisibility" regarding system modules. The extent to which this applies to you depends on the details of your RSX configuration. To every list of input files, the Task Builder automatically adds the file LB:[1,1]SYSLIB.OLB. SYSLIB is the system library; by using this file, the Task Builder has access to any system object modules that it contains. On many systems, F77OTS and other similar libraries are included in SYSLIB so that you do not have to specify them.

So far, we have discussed the use of object library files for system object modules. Under RSX, another possibility exists: the memory resident system library. Under this scheme, the most commonly used system modules are permanently located in memory. The advantage of this is that they need not be included in a task, thereby allowing larger tasks. (The task, however, must know where in memory they are.) The disadvantage is that a certain portion of memory is permanently dedicated to the resident library. If your system has a resident library, it must be specified via a Task Builder option (see Section 17.6).

In conclusion, there are three basic ways of obtaining access to the system object modules required by your task. These are not mutually exclusive, and you may have to use more than one method. First, you can specify an object library file containing the required modules as an

input file to the Task Builder. Second, the default system library file, LB:[1,1]SYSLIB.OLB, may contain the required modules. Third, you may have to specify a memory-resident system library. To be sure of what to do for your particular system, check with your system manager.

17.5 Task Builder Switches

You can modify or control the actions of the Task Builder by switches and options. Switches are appended to file specifiers and modify the interpretation or characteristics of these files. Switches typically can be set to one of two possible values (a yes/no dichotomy). If not specified, a switch assumes a default value. Options are used to supply additional information about certain aspects of the task to be built. Options typically allow specification of a wider range of values and do not always have defaults. Switches can be used without departing from the one-line command format. Options, however, are always entered as separate lines and thus can only be used in the multiple-line format. We discuss switches for the Task Builder in this section and options in the next section.

Task Builder switches are specified in standard syntax, as discussed in Chapter 7. All switches assume default values; thus, it is common to specify a switch only if you wish the nondefault value. The default values for the various Task Builder switches are defined as part of RSX but some of them can be changed during System Generation. You should either verify the default values for your system or specify switches rather than relying on the defaults. Many switches are available. Most of these will be of little interest to you, so we need to discuss only a few.

The Floating Point switch states that the task will use the Floating Point Processor (FPP). The FPP is a special piece of hardware that allows floating point instructions to be executed directly in hardware rather than by subroutine. On large PDP-11 systems, the FPP is standard equipment; on smaller systems, it is optional. Most scientific applications (e.g., FORTRAN programs using real variables) will use the FPP if it is present. Since RSX can be used on a PDP-11 that does not have an FPP, the standard RSX distribution sets the default value for the Floating Point switch to NO. (If your installation includes the FPP, your system manager has probably changed this default.) The FPP contains a set of registers; the Floating Point switch specifies that space is to be reserved in the task image file to save the contents of these registers. Even if it is built with the Floating Point switch turned off, a task can

use the FPP—what will happen, however, is that sometimes the FPP registers will be loaded with seemingly random values. Thus, unless you set the Floating Point switch, or unless the default is redefined, your task might not work properly. In MCR, the Floating Point switch is **/FP** and is appended to the task image file specifier. In DCL, the Floating Point switch is **/CODE:FPP** and is appended to the Link command.

The second switch worth noting is the Checkpoint switch, which signifies that the task is "checkpointable." The concept of checkpointing refers to the temporary copying of a task from memory to disk so that some other task can use the memory space. RSX does this as part of its procedure for running several tasks "simultaneously." Normally, you will not care whether your task is checkpointable or not, and you should accordingly set the Checkpoint switch to enable checkpointing. The standard default assumption is NO, which is based on the concept of critical tasks running in real-time. Thus, unless this default has been changed for your system, you might be asked to build tasks using the Checkpoint switch. In MCR, this switch is **/CP** and is appended to the task image file. In DCL, it is **/CHECKPOINT**, typically shortened to **/CH**, which is appended to the Link command.

Using the Floating Point and Checkpoint switches, the Task Builder command for our earlier example is

```
MCR: TKB  TEST1/FP/CP=TEST,MATH1
DCL: LINK/CODE:FPP/CH/TASK:TEST1  TEST,MATH1
```

In RSX-11M-PLUS, you can take advantage of a hardware feature of the PDP-11 known as separate I and D space. You can also use this in version 3.0 of Micro/RSX if your Micro/PDP-11 has the J11 processor chip; if it has the F11 processor chip, this capability will not be available. The separate I and D space feature is not supported in RSX-11M. Without this feature, a normal task is limited to a total address space of 64 kilobytes (65,536 bytes or 32,768 words). With this feature, a normal task can use up to 64 kB for instructions (I space) and another 64 kB for data (D space). You can use the I and D switch to cause your task to be built in this manner. In MCR, this switch is **/ID**; it is appended to the task image file. In DCL, it is **/CODE:DATA SPACE** or **/CODE:DATA**, which is appended to the Link command. In DCL, both the Floating Point and the I and D switches are variants of the **/CODE** switch. When you wish to use both, you can specify each individually or combine them in the form **/CODE:(DATA,FPP)**.

In MCR and in the new versions of DCL, you can use the Spool switch

to control whether or not a map file will be automatically printed. The Spool switch specifies that the map file is to be spooled; the No Spool switch similarly forces it to be not spooled. As RSX is distributed, the default is to spool the map file, but often this is changed to avoid wasting paper. Sometimes you might make a map file but not print it unless you need it for debugging; other times you may always want to print it. Thus, you may need to use either the Spool or the No Spool switch. In MCR, the Spool switch is **/SP** and is appended to the map file specifier. With the new versions of DCL (version 4.2 of RSX-11M, version 3.0 of RSX-11M-PLUS, and version 3.0 of Micro/RSX), you can use the switch **/PRINT**, which is appended to the Link command.

We have already met the Task and Map switches in our discussion of the basic Task Builder command. These exist only in DCL. When you use the Map switch, the Task Builder produces a file which you may want to have printed. In older versions of DCL, there is no direct counterpart of the Spool switch found in MCR. Instead, spooling is controlled via a rather contrived technique based on how and where the Map switch is used. If you append the Map switch to the Link command and you accept the default map file name, the map file will be printed automatically (the Task Builder command in MCR format that DCL generates will have the Spool switch set). If you name the map file, or if you append the Map switch to an input file (in this case, the default name for the map file is that of this particular input file), the MCR command will have the No Spool switch set. These same rules apply to the new versions of DCL, but as noted above, you can use the Spool switch **/PRINT** or **/NOPRINT** to directly control the spooling of the map file.

The Library switch is always appended to an input file specifier; it declares the input file to be an object library file rather than a simple object file. When the Library switch is present, the default file type is OLB. The default setting of this switch is off so that input files are normally assumed to be object files (type OBJ). In MCR the Library switch is **/LB**; in DCL it is **/LIBRARY** or **/LIB**. You can modify the Library switch to specify that only certain object modules be taken from the particular object library. In MCR the syntax for this is **/LB:module**. In DCL the Library switch is changed to the Include switch to allow this; the syntax is **/INCLUDE:module**. The subtleties of including only certain modules from a library are discussed in Section 24.1.

The last switch that you might need is the Options switch. This is used only in DCL; it signifies that you wish to include some options in your command to the Task Builder. If you do not use this switch,

you will not be given a chance to include any options. The Options switch is **/OPTION**, commonly shortened to **/OPT**, and is appended to the Link command. We discuss the use of options in the next section.

17.6 Task Builder Options

In addition to switches, the Task Builder allows you to modify certain features of the task build by entering options. To utilize this capability, you must always use the multiple-line form of the Task Builder command. The multiple-line command form does not require you to enter options; it merely allows you to. If you want to enter options, you must first tell the Task Builder that you want to do so. The manner whereby you do this depends on which CLI you are using.

In MCR you first invoke TKB and enter the basic Task Builder command. After specification of all input files, you must enter a line consisting of a single slash (terminated, as usual, with a carriage return). The purpose of this line is to inform TKB that you wish to enter options. The Task Builder then prompts you for these with "ENTER OPTIONS." You enter as many options as you wish, one per line. To signify that you are done, you enter a line consisting of two slashes. This marks the end of the option input as well as the end of all input to the Task Builder.

In DCL you must include the Options switch in the basic Task Builder command. You can append this to the Link command itself or to any of the input file specifiers. When you have entered all the input files, DCL will then prompt you for the first line of options with "Option?" You enter as many options as you wish, one per line. DCL repeats the prompt at the start of each line. To signify that you are done, you enter an empty line (one consisting of only the terminating carriage return).

Although the manner whereby you declare your intention to enter options differs between MCR and DCL, the options themselves are entered in exactly the same form. This is somewhat unusual for DCL. As we have seen, command names, switches, and the entire command syntax are often quite different from their MCR counterparts. Perhaps this is because the options are very specific to RSX; Digital Equipment Corporation may have seen no reason to rewrite them in a more "standard" format. Whatever the reason, if you are in DCL, you will enter Task Builder options using MCR syntax. Thus, in the remainder of this section, I have the rare luxury of presenting commands in only one syntax.

Options are specified in the form

keyword=value

where **keyword** identifies the particular option being set and **value** is the choice for said option. The RSX Task Builder provides a wide variety of options, most of which will be of little interest to you.

The three options that are most commonly used are interrelated; they refer to the definition and use of logical units. Under RSX, all I/O operations refer to a logical unit; each unit is identified by a Logical Unit Number (LUN). The system must associate each logical unit with a particular device for I/O to occur. There are defaults for these; the Units, Assign, and Active Files options allow you to override these defaults. The default assignments made by the Task Builder are:

Logical unit	Device
1	User disk (SY:)
2	User disk (SY:)
3	User disk (SY:)
4	User disk (SY:)
5	User Terminal (TI:)
6	Console Listing (CL:)
7 or larger	Invalid

Thus, an I/O operation to logical unit 1 will, by default, refer to a file on your disk. Logical units 2 through 4 similarly refer to other files on your disk, thereby allowing you to simultaneously refer to several different data files. Logical units 5 and 6 are used for "normal" input and output, as discussed in Section 15.1; however, you may not be able to use unit 6 in this way.

The Units and Assign options allow you to change these default assignments during a task build. The new assignments affect only the one task being built. From the default table, we see that 6 is assumed to be the largest legal LUN. If you wish to use LUNs greater than 6 (you can use LUNs from 1 to 250), you must first use the Units option to reset the number of LUNs used by the task. The form of this is

```
UNITS=value
```

where **value** is the largest LUN. Each allowable LUN for the task requires extra storage, so you can make a task needlessly large by careless use of this feature.

The Assign option is used to assign a device to a LUN. The form of the Assign option is

```
ASG=device:LUN:LUN——etc.
```

Each **ASG** statement names one device, which may be a peripheral or pseudo device. More than one LUN may be assigned to that device in the command. To assign LUNs 7 through 10 to disk drive DR1:, you would use

```
ASG=DR1:7:8:9:10
```

Any actual I/O operation requires some storage area (known as a buffer) to hold the characters being transmitted. Under RSX, each task has a certain number of buffers, which are assigned to the various logical units as required while the task is running. Each logical unit that is being used (i.e., each unit that has been opened and not yet closed) requires a buffer. The greatest number of units that might simultaneously be open during execution determines the buffer requirements for the task. If an insufficient number of buffers is available, the task will exit due to an error; if too many buffers are included in the task, the task will be unnecessarily large. The number of buffers that are included in the task image by the Task Builder is controlled by the Active Files option. The form of this is

```
ACTFIL=value
```

The default value is 4.

As an example of the use of these three options, suppose you wish to build a task that needs to access many different files. Specifically, assume that it will access files on LUNs 3 and 4 from SY: and will access files on LUNs 5 through 8 from DR0:. Further, LUN 1 is to be used for interactive I/O. At any point in the program, however, only three or fewer LUNs will simultaneously be open. The following set of Task Builder commands will work:

```
MCR>TKB
TKB>output=input
TKB>/
TKB>ASG=SY:3:4
TKB>UNITS=8
TKB>ASG=DR0:5:6:7:8
TKB>ASG=TI:1
TKB>ACTFIL=3
TKB>//
MCR>
```

Note that you can enter these options in any order, with the exception that the Units command must precede any Assign command that attempts to define a LUN greater than the default maximum of 6.

You may require one other Task Builder option. This is the Library option, which is used to specify a memory-resident system library. As discussed in Section 17.4, system object modules may be located in a memory resident library. If they are, it is necessary to use this option. A typical use of the Library option might be

```
LIBR=SYSRES:RO
```

Here, **SYSRES** (system resident) is the name of the library, and **:RO** specifies that the library has read-only protection. The **LIBR** option has no default—that is, the default is that the task is not to refer to any resident library. If your task is to use a resident library, you must use the multiple-line form of the Task Builder command to be able to include this option. Check with your system manager to determine whether or not the **LIBR** option is required for your system and if it is, just how you should use it.

17.7 The Fast Task Builder

The Task Builder is designed to provide a great degree of flexibility in the building of a task and is correspondingly large and clumsy. For the casual user, a simple subset of its capabilities is all that is normally required. This capability is provided by the Fast Task Builder. For simple task builds, the Fast Task Builder is reputedly at least four times as fast as the regular Task Builder. The Fast Task Builder is available with RSX-11M and RSX-11M-PLUS. It is not available with Micro/RSX.

The Fast Task Builder is used in exactly the same manner as the regular Task Builder except for the manner whereby it is invoked. In MCR, you use the name **FTB** rather than **TKB**. In DCL, you append the switch **/FAST** to the Link command.

You can use the Fast Task Builder with both the single-line and multiple-line command. Unlike the regular Task Builder, however, the Fast Task Builder supports only a few of the more common switches and options. It allows you to use the Floating Point, Checkpoint, Spool, and Library switches discussed in Section 17.5, but it does not support the I and D switch, or the module selection usage of the Library switch discussed in Section 24.1. The Fast Task Builder allows the Assign, Units and Active Files options; it does not support the Library option.

In several of the following sections, I will give examples that include the building of a task. In these, I will indicate the use of the Fast Task Builder. If your system is Micro/RSX, you will not have this and you

should substitute the regular Task Builder. If your system is either RSX-11M or RSX-11M-PLUS, you will have both available. When you have the chance, try building the same task with the regular and with the Fast Task Builder. This simple experiment should convince you of the difference in running time, after which you will use the regular Task Builder only when the Fast Task Builder does not suffice.

18

Using a Task

So far, we have discussed how to write, compile and make a task from a program. If you have gone this far with your program, you presumably will next want to run it. As with many other aspects of RSX, the procedure for running a task offers a high degree of flexibility with a correspondingly high degree of complexity. Fortunately, RSX offers an extremely simplified special case of the more general procedure. In most cases, this is all you need to know to run your task. We will look at this first, then we will examine the entire procedure in more detail. If you are impatient, read only Section 18.1—otherwise, here we go!

18.1 The Simplified Method

Suppose that you have used the Task Builder to build a task image file. You now wish to execute this. The appropriate command is the same whether you are in MCR or DCL—it is simply

```
RUN task_image_file
```

The default file type is TSK, so you typically need only the file name portion of the specifier for the task image. Execution of the task should (assuming that you wrote the program correctly) proceed as you would expect. If any error messages are generated, they will appear at your terminal. If everything goes properly, your task will eventually terminate, after which you will get another prompt from your CLI.

If you wish to stop task execution prematurely (for instance, if you

discover a mistake), you can do so via the Abort command. In this simplified form of running a task, a correspondingly simple form of this command will suffice. This command also has the same form for both MCR and DCL,

```
ABO
```

Note that you may need to use a CTRL/C to get your CLI's attention before it will accept the Abort command.

In Section 11.2 we discussed how to run a system function. Just as you might want to abort your task once it has started running, so too might you want to abort a system utility. You can use the Abort command to prematurely terminate task execution in this case as well. Things are a little more complicated here—we discuss this in Section 18.5.

18.2 Tasks vs. Task Images

Up to now, I have been purposely sloppy in referring to tasks and task images. At this point, we must distinguish between these two concepts. Although you normally will be able to ignore the distinctions, it is nonetheless useful for you to understand them.

Under RSX, a task is something to which control of the CPU may be given. When a task is executing, it must be in the main memory of the CPU. When it is not executing, it need not be in memory and, due to limited memory resources, is normally removed from memory. Because a task is not always in memory, a copy of it must be kept somewhere. This copy is known as the task image and is stored in a task image file, which must be on a disk. (Actually, our emphasis here is backward, as the task image file is created by the Task Builder first and the task is then made by copying this image into memory.)

This distinction between a task in memory and a copy of it on a mass storage device is typical of most operating systems although the terminology may not be the same. On any system, to run a program, the corresponding task image file must be loaded into main CPU memory, and control must then be passed to it. In some operating systems, this would require searching through directory files to locate the task image file. RSX introduces an intermediate step, motivated by the desire to expedite this procedure for real-time processing applications. This is known as installing a task.

18.3 Installing and Removing a Task

RSX maintains a special list known as the System Task Directory (STD). When a task is installed, it is added to the list; when a task is removed, it is deleted from the list. The processes of installation and removal have no effect on the task image files. Each entry in the STD consists of the name of the installed task, the location (on disk) of the corresponding task image file, and several other pertinent parameters. Only when a task has been installed can it be run. When you make a request to run a task, RSX searches the STD until it finds the entry for that task. RSX obtains the location of the task image file, copies it into memory, and then declares it to be active (i.e., eligible to compete with other active tasks for control of the CPU).

Note that the process of installation creates a one-way association between a task and a task image. That is, each installed task is associated with a unique task image, but each task image is not associated with a unique task. Clearly, it is possible to use the Task Builder to create a task image without ever installing a task corresponding to it. Perhaps not so obviously, it is also possible to install several tasks, all corresponding to the same task image. The only restriction here is that the various tasks must all have different names. From this, we see that a task name need not be the same as that of the corresponding task image. In fact, the name of the task image (as with any other file) can be up to nine characters long, whereas a task name can be no more than six characters long.

When RSX processes a request to run a task, it checks whether that task is already active. If it is, the request is denied. It is often useful, however, to run the same program several times at once (for example, you and some other user might want to use the FORTRAN compiler simultaneously). This may be accomplished by installing two different tasks for one task image. When these two tasks are run, two different copies of the same original task image file will be brought into memory. The operating system does this with the system utilities, but the process is transparent to the user.

Note that when a system utility is running, its task name is often not the same as its name in the STD. In Section 11.2 we saw that the name used in the STD was the first three characters of the MCR command used to invoke the utility, preceded by three periods (e.g., the FORTRAN-77 compiler is invoked by the command **F77** and is installed as ...F77). In RSX the name used in the STD is known as the prototype

task name. System utilities can, in general, be run by more than one user at a time—that is, several copies of the same basic task may be simultaneously active. This requires that they all have distinct names—they cannot all be given the prototype name.

The way task names are chosen for system utilities depends on the type of RSX you have. For RSX-11M, the task name will, if possible, be the prototype name. If this name is already being used, the task name will be the three-character utility name followed by Tnn, where nn is the number of the user's terminal. Thus, if you are on terminal 2 and you are running the utility PIP, your task might be either PIPT2 or ...PIP. Under RSX-11M-PLUS and Micro/RSX, this ambiguity is avoided. The prototype name is never used for an active task, the task name is always the three character utility name followed by Tnn.

To install a task, you must use the Install command. This is a privileged command. The general form of this command is

MCR: **INS task_image_file/options**

DCL: **INS/options task_image_file**

If no options are used, the MCR and DCL commands reduce to the same form. All the usual defaults apply to specification of the task image file with the file type assumed to be TSK. The only option likely to be of interest to you is the task name option. In MCR this is entered as **/TASK=task_name**; in DCL it is entered as **/TASK:task_name**. If this is omitted, the task name is taken by default to be the same as the name of the task image file. If this is longer than six characters, only the first six are used. For example, the command

```
INS TEST2
```

creates an entry in the System Task Directory for a task called TEST2, which refers to the task image file TEST2.TSK. As another example, the command

```
INS TESTPROG2
```

installs a task known as TESTPR. If this were followed by the command

```
INS TESTPROG3
```

an error would result due to the attempt to install two tasks with the same name. To avoid this, you could specify unique task names with commands such as

```
MCR: INS TESTPROG2/TASK=TEST2
DCL: INS/TASK:TEST2 TESTPROG2
```

As a final example, suppose that you wished to simultaneously execute two copies of a program called CALCULATE. To create two separate tasks, you could use the commands

```
MCR: INS CALCULATE/TASK=CALC1
     INS CALCULATE/TASK=CALC2
DCL: INS/TASK:CALC1 CALCULATE
     INS/TASK:CALC2 CALCULATE
```

After a task has been installed, you can delete the corresponding task image file yet the task name will remain in the STD. A subsequent attempt to run the task will lead to unpredictable results, as the operating system will attempt to copy the task into memory from the same disk address. To remove a task name from the STD, you use the Remove command. The form of this command is the same for both MCR and DCL:

```
REM task
```

This also is a privileged command.

The Install and Remove commands are, as noted, both privileged. As a casual user, you probably will not be able to use them. There may be situations where it will be desirable for you to be able to install a task; in these cases, you will have to ask your system manager to do so for you. An example of this is running a long task overnight, which we discuss in Section 25.1.

18.4 Running a Task

Once a task has been installed, it can be made active by using the Run command. There are several forms of this command. In the simplest form, the command is the same in MCR and DCL:

```
RUN task
```

This command can be used by nonprivileged or privileged users and results in immediate (subject to the existence of other active tasks in the system) execution of the task.

Because the Install command is available to privileged users only, RSX provides another means to execute a task. This is the simplified form

of the Run command described in Section 18.1. In addition to running, it automatically installs and removes the task. The command is the same for both MCR and DCL:

```
RUN task_image_file
```

In terms of system action, this command is equivalent to the following sequence of MCR commands

```
INS task_image_file/TASK=TTnn
RUN TTnn
REM TTnn
```

First, the task is installed under the default task name TTnn where TTnn: is the peripheral device name of the terminal in use. (This task name is chosen because it is unique to the user. Names such as TT, TT0, TT1, etc., should accordingly not be used as task names for permanently installed tasks.) The task TTnn is then run and, following completion, removed from the STD. This command may also be used for system utilities that are not installed. In this case, the utility name is preceded by a dollar sign. For example, if PIP were not installed, the nonprivileged user could run it via the command

```
RUN $PIP
```

In this case, although it will be only temporarily installed, the utility will retain its traditional name (either ...PIP or PIPTnn); it will not be called TTnn.

So far, we have discussed two forms of the Run command—the normal form for an installed task and the Install-Run-Remove form. Syntactically, these two forms are identical. In either MCR or DCL, the command

```
RUN SIMUL
```

could be intended to refer to either an installed task named SIMUL or a task image in a file named SIMUL.TSK. Under RSX, this ambiguity is resolved as follows. When a Run command of this form is entered, the STD is searched for a task with the given name. If one is found, it is executed. If not, a search is made for a task image file (file type = TSK) with the given name. If the first character of the name is not a dollar sign, your directory is searched; if the name begins with a dollar sign, the UFD containing system tasks is searched instead. If the appropriate file is found, it is executed using the Install-Run-Remove command. If not, an error message is displayed on your terminal.

This ambiguity can lead to rather obscure behavior. Suppose some other user has created a task image file in his user area called [147,1]SIMULATE.TSK and has had it installed under the task name SIMUL. Now suppose you write a program to solve simultaneous equations and from it you make a task image file [123,1]SIMUL.TSK. In all innocence, you enter the command

```
RUN SIMUL
```

What happens is that SIMULATE.TSK is copied from user area [147,1] into memory, and it is this program, not yours, that is executed.

A similar problem can occur if it is your own task that is installed. Suppose you write a program, build a task image, and have it installed for you by a privileged user. After running the task for a while, you decide to make some changes to the program. You edit the source, recompile and rebuild. Before arranging to have the new version installed, you decide to test it using the Install-Run-Remove command. If, as is often the case, the task name is the same as the task image name, you will execute the old, not the new, version. This can lead to anguished confusion concerning why your changes did not work. To avoid this, you should have the old task removed once the new one has been built. Alternatively, you can specify the file type in the Run command rather than relying on the default, as in

```
RUN REPORT.TSK
```

The period here forces the name to be a file name, thereby avoiding any possible confusion with the name of an installed task.

In addition to the commands for the immediate running of an installed task and for installing, running, and removing an uninstalled task, other forms of the Run command exist. These are limited to use by privileged users. They allow a task (it must be installed) to start execution in the future and thus may be of value to you for scheduling lengthy runs to occur during relatively slow hours. (Presumably, you will be able to get a privileged user to install the task and enter this type of command for you.) These forms of the Run command are discussed in Section 25.1.

18.5 What Does DCL Really Do?

I have consistently stated that DCL translates your commands into MCR commands and that it is these MCR commands that are executed. This is essentially true and is normally a sufficient explanation of how DCL

works. In this section, however, we need to discuss somewhat more exactly just what it is that DCL does.

Let's suppose you are in DCL and you wish to get a directory listing. To do this, you enter the command **DIR**. As we have seen, the system utility that makes the listing for you is PIP. Thus, the DCL command **DIR** is translated into an MCR command that invokes PIP. Although this is essentially true, it is not the full story. In somewhat more exact detail, this is what happens. Since MCR is the only user interface that really exists, it receives the command that you enter. This is given to the primary MCR command dispatcher, which is a task with the name MCR.... Since your terminal is identified as being set to DCL, this dispatcher gives your command to the DCL command parser, which is a task with the name ...DCL. This activates a special DCL command task, which is named from the first three characters of your original command, followed by the letter T and your two-digit terminal number. (In this example, if you are on terminal 7, the intermediate task would be called DIRT07.) It is this task that actually translates (the remainder of) your command into the equivalent MCR command (e.g., **PIP /LI**) and then gives this command to the secondary MCR command dispatcher, which is known as ...MCR. If required (as in this example), the secondary dispatcher gives your command to the appropriate system utility; otherwise, an internal part of RSX performs the processing. The DCL command task remains active while the MCR task runs. When the MCR task finishes, the DCL command task may translate and display its output for you, give you error messages, or issue another MCR command. When everything is done, it exits.

I hinted above at the fact that a single DCL command could result in several MCR commands being generated. This is normally not true, but it is possible. For example, as we saw in Section 10.2, the DCL command **SET DEFAULT** can be equivalent to the pair of MCR commands **ASN** and **SET /UIC**. When this happens, the intermediate DCL command task gives one command to MCR, then the next, and so on, until all requisite MCR commands have finished, or until one of them fails. To simplify our discussion, let's assume that only a single MCR command results from a DCL command.

The process used to implement the DCL user interface may seem to be incredibly convoluted. After all, the command **DIR** translates into a simple PIP command; why is an intermediate task needed? Why not just have the DCL command parser do the translation directly? The reason is threefold. First, remember the basic reason for the existence of DCL—to provide a common user interface and to shelter you from

the vagaries of MCR. Thus, when you type in **DIR** to get a directory listing, you probably do not want to know that something called PIP is doing the work for you. From your viewpoint, it is appropriate that a task named DIRTnn appears to be doing it all. This reason is not as frivolous as it might sound—we return to it in the next section.

Second, not all functions are as simple as that of getting a directory listing. In Chapter 19 we will meet various DCL commands that all begin with the word **SHOW**; depending on what is to be shown, various MCR commands or system utilities may be used. All the DCL Show commands activate an intermediate DCL command task called SHOTnn. This task, in turn, determines just which MCR command is appropriate. As an even more complicated example, the Link command activates the intermediate task LINTnn, which collects all your additional input, translates it into the appropriate set of input lines for the Task Builder (either normal or fast), puts these lines into a temporary file on disk, and finally calls the Task Builder, naming this file as an indirect command file. Of course, even though these translations are more complicated than the simple directory example, the DCL command parser could do them all. This brings us to the third and most important reason for having the intermediate tasks. Without such a division of labor, the DCL command parser would be enormous and would occupy an excessive amount of memory.

Whatever the reasons behind the manner whereby DCL functions, you must remember that when you enter a DCL command, a task with a corresponding name appears to do the work for you. This is important in the next section when we consider how to abort a system function, for in order to abort a system function, you need to know how to identify it. It is also important to understand all this when you use the Show Active Tasks command (Section 19.5); otherwise you will wonder what that task called SHOTnn is all about.

18.6 Aborting A Task

Any time that you run a task, there is a possibility that it may be desirable for you to prematurely terminate it. For example, you may write a program to perform the same calculation on many different inputs and to print a corresponding table of answers. Inspection of the first few lines of output may show an error in the calculation. In this case, it would be useless to continue execution and you would like to be able to stop execution of the task.

The Abort command is used to terminate a task. This is a privileged command with the exception that a nonprivileged user can abort a task that he started. In MCR the form of this command is

```
MCR: ABO task
```

where **task** is the name of the installed task. It does not matter whether the task is a system utility or your own task. In DCL, however, there are two different forms of the Abort command—one for a system utility and one for a "regular" task.

Let's first consider the aborting of a user task. It is important to note once again the distinction between the name of a task and the name of its task image file. Suppose that you, a nonprivileged user, are logged in to a video display known to the system as TT2: and that you have built a task called TEST1.TSK, which you wish to execute. You may do this, as explained, in either MCR or DCL, via the command

```
RUN TEST1
```

If you subsequently decide to abort the program, it might quite naturally seem sensible to enter a command that names TEST1 as the task to be aborted, e.g.,

```
ABO TEST1
```

In both MCR and DCL, this will be rejected since, due to your use of the Install-Run-Remove command, the task name actually is TT2.

In MCR, the correct command is one that names the task as TT2,

```
MCR: ABO TT2
```

This distinction is is easy to forget. To simplify this for you, RSX assumes a default task name of TTnn (where nn is the number of your terminal) for the Abort command. Thus, in the above example, you could simply enter

```
MCR: ABO
```

and the desired effect would be achieved.

In DCL, the same arguments concerning naming the task to be aborted apply as well. The same simplified command also exists—that is, you can enter

```
DCL: ABO
```

This is translated into an MCR Abort command with no task name, which, by the above default, causes the desired effect.

In DCL, the more general form of the Abort command for a user task is

```
DCL: ABO/TASK task_name
```

Here, the **task_name** is the name of the installed task and the switch **/TASK** is necessary to tell DCL that the task is a user and not a system task.

To sum up, suppose you wish to run and then abort a task of yours called TEST1. If this task is not installed (i.e., you will use the Install-Run-Remove form of the Run command), the command sequences are

```
MCR: RUN TEST1
     ABO

DCL: RUN TEST1
     ABO
```

If, however, you have arranged to have your task installed, the correct command sequences are

```
MCR: RUN TEST1
     ABO TEST1

DCL: RUN TEST1
     ABO/TASK TEST1
```

The Abort command may also be used to abort system functions. Suppose you have started a lengthy FORTRAN compilation via the command

```
MCR: F77 TEST=TEST

DCL: FOR/F77 TEST
```

You then remember that you did not make a necessary change to the source code. Rather than waiting for the compilation to be finished, you can abort the FORTRAN compiler. (This does not affect the ability of any other user on the system to use the FORTRAN compiler.) In MCR the command to do this is

```
MCR: ABO F77
```

In keeping with the philosophy of DCL, you do not need to know that the command **FOR** is translated into an MCR command invoking the task F77. Instead, you can abort the DCL command itself. In this example, the DCL command would be

```
DCL: ABO FOR
```

In the previous section, I was rather pedantic about the exact sequence

of operations that result when you issue a DCL command. From that, you can see that the DCL command to compile a FORTRAN program results in a task named FORTnn, which, in turn, activates the F77 task. The mechanism used in RSX that enables one task to start another is called spawning. Without going into details, if one task spawns another and is then aborted, the task that it spawned will also be aborted. Thus, by aborting the FOR task, you indirectly (but just as effectively) abort the F77 task. This does not mean that you cannot abort the F77 task directly from DCL. You can if you want to; the same command that you would use in MCR will also work in DCL. In this example, once the DCL task FORTnn spawns F77, it waits until F77 finishes. When you abort F77, FORTnn is reactivated, much the same as if F77 had terminated normally, whereupon it exits as well. Normally, if you are in DCL, it will be easier (and probably more meaningful) for you to abort the DCL command, and I will assume this in our examples.

In Section 18.3 we discussed how a task name is formed for a system utility. Just as RSX normally hides that relation from you when you run the task, so too does it hide it from you when you abort the task. For example, in MCR you need not wonder whether your copy of the FORTRAN compiler is FORTnn or the prototype ...F77. You can simply request that F77 be aborted, and RSX will take care of the details. Similarly, in DCL you can simply abort FOR without remembering what your terminal number is so that you can properly identify the task as FORTnn. Note these simplifications in the examples above.

When a task is aborted, any user output files that are open will not be properly closed. This may result in a locked file or a file of length zero (see Section 14.4). After aborting a task, you should use PIP (or its DCL counterparts) to detect (via a directory listing) and correct (via the Unlock, End Of File or Delete commands) such conditions.

Note that log out (via the command **BYE** in MCR or **LOG** in DCL) may cause active tasks to be aborted. Specifically, if a nonprivileged user runs a task immediately and then logs out prior to normal task completion, the task will be automatically aborted. To avoid this, you must use a delayed form of the Run command. (The delay can be arbitrarily brief; a one-second delay effectively produces immediate execution but nonetheless causes the operating system to recognize the task as being run in delayed rather than immediate mode.) This is discussed in Section 25.1.

19

Other Useful Commands

There are many other commands that we have not yet discussed. Some of these are an intrinsic part of RSX itself. Others are tasks that should (but might not) exist on your system. Most of these will not be of any interest to you, the casual user. You will, however, find some to be useful, although they are not as important as the ones that we have already examined. We discuss these remaining commands in this section.

19.1 GETTING HELP

The Help command allows you to obtain information on various facets of the RSX operating system. The general form of the command is the same for both MCR and DCL

```
HELP topic(s)
```

where **topic** is a word or group of words specifying the topic about which you would like information.

In MCR, all four letters of the word **HELP** must be used to distinguish it from the Hello command (**HEL**)—this is the only exception to the rule that all MCR commands may be three characters long. In DCL, the command can be shortened to **HEL** or even **H**, since HELLO is not a command in DCL. You can use the **HELP** command even if you are not logged in to the system. In this manner, you can obtain information about the log-in procedure.

The various messages that may be displayed in response to a Help command are stored in a file. If the particular topic is found in the file, the corresponding message is displayed. A rather extensive Help file is

provided as part of the RSX system. In general, as more recent versions of RSX have been released, the Help file has been made increasingly comprehensive. Due to the size of this file, it is sometimes removed from the system. Alternatively, the system manager may edit the Help file to include more information. The choice of possible Help messages will depend on the particular installation.

If your RSX system supports both MCR and DCL (version 4.0 or later of RSX-11M, all versions of RSX-11M-PLUS), it should have separate Help files for MCR and DCL. The one that you get is automatically selected for you depending on which CLI you are currently using. Thus, if you are in DCL and you type

```
HELP LINK
```

you will get Help messages for the Link command. If you enter this same command while in MCR, you will get an "Unknown HELP Qualifier" error message, since MCR does not use **LINK** as a command name. Similarly, if you type

```
HELP RUN
```

you will, in either MCR or DCL, get information on how to use the Run command, since this command has the same name in both CLIs. The information that you get will, however, depend on whether you are in MCR or DCL.

The topics for the Help command are organized in a tree-like structure. The manner whereby topics are broken into levels is the same for both MCR and DCL (the topics themselves differ, but not the tree structure) so in the discussion below we consider MCR only.

At the most basic level, the topic is blank—i.e., the command is simply

```
HELP
```

This will produce the listing of a group of very basic topics with a brief description of each. Suppose one of these is MORE which, in one version of the main MCR Help screen, is described as offering information on system utilities. The command

```
HELP MORE
```

will give a brief description of various utilities. One of these is PIP. The command

```
HELP PIP
```

will give more detailed information on PIP. This might be a list of switches, followed by the instruction to enter

```
HELP PIP switch
```

for detailed information on a particular switch. In this case, the command

```
HELP PIP DE
```

would display information on the use of the Delete switch for PIP. Note that this last example specifies two topics. In general, you may specify several topics in a Help command. These must be separated by spaces, with the first being the primary topic, the next the secondary topic, and so on.

The Help command allows the Help file to include many levels of topic specification. Normally, no more than two such levels will be used. The Help file sometimes is broken into levels and sublevels so that no individual response is more than 23 lines long. This allows the entire response and the subsequent prompt to fit on a standard 24 line video display. This is not always the case, so you should be ready to use CTRL/S and CTRL/Q (Scroll On/Off) if you are using a video terminal.

19.2 Setting and Showing Terminal Characteristics

A PDP-11 installation under RSX can accommodate a large variety of terminal types. Since these do not all have the same capabilities, each terminal in an RSX system has associated with it a list of characteristics that tell the terminal driver how to work with it. The system manager initially sets these values. His choices will normally be correct for your use; you may nonetheless need to change them. The Set and Show commands allow you to alter or display these terminal characteristics.

We will first examine setting terminal characteristics. In its most general form, the command to do this is

```
MCR: SET /keyword=TTnn:value
DCL: SET TERMINAL:TTnn:/keyword:value
```

This form allows you to set characteristics for any terminal (as specified by the terminal number **nn**) in the system and is accordingly limited to privileged users only. The nonprivileged user may always set char-

acteristics for his own terminal. In this case, rather than remembering the terminal number, you can use the pseudo device identifier TI: in the MCR command form. In DCL a further simplification is possible; you need not enter any terminal code at all, and TI: will be assumed. Thus, the commands to set characteristics for your own terminal are

```
MCR: SET /keyword=TI:value
DCL: SET TERMINAL/keyword:value
```

The **keyword** specifies the particular terminal characteristic to be set. Note that DCL does not always use the same keywords for the various terminal characteristics that MCR does. The **:value** may or may not be needed depending on the particular characteristic.

MCR has been criticized as having an awkward and unnatural command structure. The Set command for terminal characteristics may well represent its syntactical nadir. Do not be surprised if you cannot remember the exact form of this command. On most systems, you will be able to use the command

```
HELP SET
```

to refresh your memory.

Several terminal characteristics might be of use to you. Before presenting these, we should discuss some reasons why it might be necessary to set the characteristics of your terminal. As noted, default values are established for all terminal characteristics. Assuming that these have been properly chosen, why should it be necessary to change them? One important reason is the use of remote terminals. If your computer installation includes a dial-up capability (a phone line and a modem) then you can access it via the telephone network if you have your own terminal and modem. In this case, the modem at the computer end is identified as being a remote terminal and is assigned a number just as is a terminal that is directly connected to the computer. In actuality, the terminal corresponding to this number is whatever device is at the other end of the telephone link and may arbitrarily vary from one use of the link to another. To accommodate this, RSX assumes that remote terminals are of the most primitive type possible. Specifically, it is assumed that a remote terminal does not have lower case, is not a video terminal and can only print 72 characters per line. If you call into the system from a more sophisticated terminal, you will want to redefine these characteristics.

Many characteristics are associated with a terminal. You will want

to change at most a few of them. You will probably be able to change all of these as described below, but this is not guaranteed, since some of them are system generation options.

The Lowercase keyword specifies that the terminal is capable of transmitting and receiving lowercase alphabetic characters. In both MCR and DCL, this keyword is **/LOWER**. Similarly, the keyword **/NOLOWER** specifies that the terminal does not have this capability. Essentially all modern terminals offer lowercase; vintage teletypes (such as the type ASR-33) from the 1960s are examples of terminals that do not. The keywords for controlling lowercase do not require a value; the Set commands are

```
MCR: SET /LOWER=TI:
     SET /NOLOWER=TI:
DCL: SET TERM/LOWER
     SET TERM/NOLOWER
```

By setting your terminal to NO Lowercase, you can force all terminal input to be in uppercase. When you do so, the terminal driver will convert any lowercase letters entered into uppercase before being passed on to the rest of the system. Most terminals have a Caps Lock key that gives the same effect; if your terminal does not, you can use this command instead.

The Video keyword specifies that the terminal is a video, not a hard-copy terminal; the Hard-copy keyword specifies the reverse. The significance of this distinction lies in the treatment of the Delete character, as discussed in Section 9.1. In MCR the Video keyword is **/CRT** (for Cathode Ray Tube); in DCL it is either **/SCOPE** or **/NOHARD**. In MCR the Hard-copy keyword is **/NOCRT**; in DCL it is either **/NOSCOPE** or **/HARD**. These keywords do not require a value.

The Width keyword is used to specify the buffer size of the terminal, which is equivalent to the number of characters that can be printed per line. In MCR this keyword is **/BUF**; in DCL it is **/WIDTH**. The Width keyword requires a value to specify the buffer size. In MCR this value is interpreted as being octal unless it is terminated with a period to signify that it is decimal. In DCL you do not have to worry about this; DCL assumes that you are entering a decimal value and appends the required period for you. A video terminal can normally display up to 80 characters per line; a DECwriter can print up to 132. To set the buffer size corresponding to a DECwriter, you would use the command

```
MCR: SET /BUF=TI:132.
DCL: SET TERM/WIDTH:132
```

Note that in MCR, the command

```
MCR: SET /BUF=TI:132
```

sets the terminal width to 90 since the 132 is not followed by a period; this represents a common error.

If the number of characters sent to a terminal in one line is greater than the defined buffer size for that terminal, one of two things can occur. Either the excess characters will be put on a second line (i.e., the terminal driver will generate a Carriage Return and Line Feed after filling the terminal's buffer) or the excess characters will be lost. This is controlled by the wraparound characteristic. In both MCR and DCL the keyword for this is **/WRAP** which directs the terminal driver to wrap excessively long lines around to the next line position on the terminal. The **/NOWRAP** keyword is used to disable this option, resulting instead in the loss of excess characters. These keywords do not require a value.

Another keyword that you might find useful for setting terminal characteristics is the Terminal Type keyword. In both MCR and DCL, the general form of this keyword is **/type**, where **type** is the particular terminal type, such as **VT100**, **VT52**, or **LA36**. In a sense, this is a "super" keyword because it can cause several characteristics to be set at once. Specifically, it will cause the Video and Width keywords (and others) to be set corresponding to the capabilities of the specified terminal type. Thus, the Type keyword can decrease the number of commands you must enter to reconfigure your terminal and is particularly useful if you are on a remote terminal. Note, however, that it will not affect the lowercase characteristic of your terminal, since even if your terminal supports lowercase characters, you may wish to force it into an uppercase only mode as discussed above. If you have a VT100 terminal and you log in on a remote line, you can reset all the significant terminal characteristics with the two commands

```
MCR: SET /VT100=TI:
     SET /LOWER=TI:

DCL: SET TERM/VT100
     SET TERM/LOWER
```

Setting the terminal type characteristic might be useful if you are using a VT100 type terminal, as these terminals offer two possible modes of operation. (In the mid-1970s, the VT52 was Digital's standard video terminal. In the late 1970s it was replaced by the VT100, which offers many enhancements. To offer compatibility with software specifically written for the VT52, the VT100 was designed with a user-selectable

mode of operation wherein it acts as though it were a VT52.) If you are using a VT100 and you want it to imitate a VT52, you must do two things. First, you must change what the terminal thinks it is (this is a hardware function, accomplished by using the Setup feature of the VT100). Second, you must change what the operating system thinks the terminal is. This is done using the keyword **/VT52**, as in the command

```
MCR: SET /VT52=TI:
DCL: SET TERM/VT52
```

In the early 1980s, the VT200 terminal series was released as an upwards compatible enhancement of the VT100 series. With version 4.2 of RSX-11M, the Set Terminal Type command was expanded to include keywords for the various models in this series. Currently, there are three models in the VT200 series: the VT200, VT240, and VT241. The VT240 and VT241 offer graphics capabilities as well all the normal features of the VT200. As RSX does not use any of these graphics capabilities (a specific application might, but the operating system itself does not), it does not need to distinguish among these three models. Thus, RSX simply identifies a generic VT200 series terminal; this is done with the keyword **/VT2XX**. To declare that you are using one of the VT200 series terminals, you use the command

```
MCR: SET /VT2XX=TI:
DCL: SET TERM/VT2XX
```

Note that the keyword is **/VT2XX**. The actual terminal type, VT200, VT240, or VT241, is not a valid keyword and will not be recognized.

One last terminal characteristic of interest is the (No) Broadcast option. Normally, any message broadcast to you by another user will appear on your terminal, no matter what you are doing at the time. (The Broadcast command is discussed in Section 19.4.) If you wish to prevent this, you may set your terminal to No Broadcast. The keyword for this is the same for both MCR and DCL; it is **/NOBRO**. If you later wish to reenable the broadcast capability, you use the keyword **/BRO**. These commands were first made available in version 4.0 of RSX-11M.

Besides setting characteristics for your own terminal, you may want to examine the characteristics of other terminals. The commands to do this are generically known as Show commands although their underlying MCR implementations are effected in a variety of ways. There are two basic forms of the Show command.

The first shows all terminals that have a certain characteristic. This command is

```
MCR: SET /keyword
DCL: SHOW TERMINAL/keyword
```

Note that in MCR the syntax is particularly strange. By not including a terminal identifier in the Set command, you are directing MCR to show rather than set characteristics. The **keyword** may be any that may be set with the Set command. As an example, the command

```
MCR: SET /CRT
DCL: SHOW TERM/SCOPE
```

will list all terminals in the PDP-11 configuration that are video terminals. Normally, this is not very useful. There is only one keyword that you may want to use with this command. This is the Privileged keyword, which is **/PRIV** in both MCR and DCL. (If you are a privileged user, you can also use this keyword to turn the privileged status of a terminal on or off.) There may be times when you need the assistance of a privileged user; by using the command

```
MCR: SET /PRIV
DCL: SHOW TERM/PRIV
```

you can determine which (if any) terminals are currently privileged.

The other useful Show command is the one that shows all the characteristics of your terminal. This command was made available in version 4.0 of RSX-11M. It produces a display of the status of every characteristic of your terminal. These include all those just discussed under the Set command as well as many others. As all of the terminal characteristics are controllable and individually displayable via the Set command in MCR, you might expect that this master display would also be obtainable via a form of the Set command. It is not. In MCR, the command is

```
MCR: DEV TI:
```

This is a special form of the Devices command, which we discuss in Section 19.8. In DCL the command syntax is more natural. It is the Show command without any keywords,

```
DCL: SHOW TERMINAL
```

19.3 Who's on the System?

At times (such as we discuss in the next section) it is useful to know who else is on the system—that is, who else is currently logged in to a terminal. The Show Users command enables you to find this out. It is

```
MCR: DEV /LOG
DCL: SHOW USERS
```

In MCR, this command is a special form of the Devices command which provides information about a specified class of devices in the system—here, the class is specified to be logged-in terminals. In DCL, the more appropriate command name **SHOW USERS** is used. The Show Users command requests a listing (on your terminal) of all terminals that are currently being used, along with the UIC of each user. You can use this to get information about another user or terminal. If, for example, you need to consult with a co-worker whose UIC is [150,1], you can use this command, examine the listing, and see if he is logged in to any of the terminals. If you are working on a remote terminal, this ability can be invaluable. Alternatively, if you are on one type of terminal (for instance, video) and wish to switch to another type (hard-copy), you can use this command to check whether any of the appropriate terminals (if you know their numbers) are not being used. If your system has terminals that are not located close together, this is easier than walking all over the place.

The Show Users command is somewhat of a misnomer. It does show which terminals are being used, but it does not always display the log-in UIC, which is what really identifies a user. Rather, in RSX-11M, it displays only the protection UIC of the user. (We discussed the concepts of protection UIC and default UIC in Section 14.5.)For a nonprivileged user, the protection UIC is the same as the log-in UIC. For a privileged user, it is whatever his default UIC happens to be. In RSX-11M-PLUS, for any user, the Show Users command shows both the default and the log-in UIC. Suppose that your system manager logs in to the system on TT6: using UIC [2,1] and then changes his UIC to [351,11]. In RSX-11M, he will appear in the list of system users as

```
TT6: [351,11]
```

In RSX-11M-PLUS he will be listed as

```
TT6: [351,11]  [2,1]
```

If the RSX-11M-PLUS system supports Resource Accounting (this is a system generation option), some additional information will be given.

If your system does not support multiuser protection (Chapter 4), there are no separate log-in, protection, and default UICs. Instead, each user simply has a current UIC, which corresponds to his default directory. (This is only possible on RSX-11M, since both RSX-11M-PLUS and Micro/RSX always include multiuser protection. Thus, named directories are not possible and the default directory will always be in the same form as a UIC.) This current UIC is all that will be shown by the Show Users command.

The utility **WHO** offers another way to find out who is on your system. This is not a part of RSX, thus it might not exist on your system. **WHO** is an enhancement to RSX that is available from DECUS. It combines the functions of the Show Users and the Active Tasks commands. It shows you each user who is logged in as well as which (if any) tasks are currently active on his terminal. **WHO** is not an intrinsic part of the operating system. Assuming that it has been installed as a system function (under the task name ...WHO), you can simply enter its name as a command to run it, if you are in MCR. If you are in DCL, you can run it by prefacing this MCR command form with the command **MCR**,

```
MCR: WHO
DCL: MCR WHO
```

Alternatively, if the task image file for **WHO** is in the standard user area for RSX system tasks, you can use a Run command and identify **WHO** as a system function by prefacing its name with a dollar sign,

```
DCL: RUN $WHO
```

If **WHO** exists on your system, you will find it useful. Check with your system manager concerning its availability.

19.4 Talking to Another User

The Broadcast command can be used to send a message to another user. The form of this command is the same for both MCR and DCL:

```
BRO TTnn: message
```

This command causes the specified message to be printed on terminal number **nn**, preceded by a header stating that the message was sent via the Broadcast command and from which terminal it was sent. The mes-

sage itself cannot be more than 80 characters long and cannot continue past the end of the line—the carriage return acts as a message terminator.

When using the Broadcast command, it is important to understand just how it works. To this end, we must discuss the terminal driver a little more fully. Normally, when a program wishes to write something to a terminal, a conventional Write command is issued to the terminal driver. The driver first checks as to whether the specified terminal is busy or not. If the terminal is not busy, the write is performed; if the terminal is busy, the Write command is rejected.

Another type of write, known as a break-through write, exists. This is a system generation option; it is typically found on all except very small RSX systems. The break-through write does just what its name implies: it breaks through whatever the recipient terminal is doing. Essentially, when a program issues a Break-through Write command, the terminal driver does the write immediately, whether the specified terminal is busy or not.

If your system supports break-through writes, Broadcast will always use them. In this case, whenever you use the Broadcast command, even if you are not privileged, your message will appear on the recipient's terminal, regardless of what the other user is doing, unless he has specifically disabled broadcasts for his terminal. This could be very annoying because it might affect hard-copy output or cause confusion during a video edit. It will not, however, be catastrophic, as it will not interfere with the task running on the other terminal, even if that task is doing I/O.

On a system without the Break-through Write capability, Broadcast must use a conventional Write command. In this case, if the terminal to be written to is busy, BRO will tell you that it is unable to send the message. If this happens, you will have to wait until the terminal is no longer busy before you can send your message. By busy, I mean that some task has "attached" the terminal. I do not want to define this precisely. A task can use a terminal without attaching it; if so, the terminal driver will not consider the terminal to be busy. For example, if MCR is waiting for a command from a user, that terminal is not busy.

Due to its interaction with other users, it is best to use the Broadcast command in conjunction with the Show Users and Active Tasks commands. The Show Users command (previous section) can be used to find out who is currently using which terminal. This tells you to whom you can broadcast. The Active Tasks command (next section) can be used to find out what each user is doing. This tells you whether it is a good time to send a message to a particular terminal or not. Of course, if

someone uses Broadcast to send you a message, you should feel free to use Broadcast to send a reply without first checking.

Just as a broadcast from you can interfere with someone else, so too can a broadcast from someone else interfere with you. You can prevent this by using the Set No Broadcast command, as described in Section 19.2. This command was introduced with version 4.0 of RSX-11M.

Broadcast offers a rather primitive capability. The limitation of the message to one line means that long messages must be sent via multiple broadcasts, each of which prints a heading on the recipient's terminal. On older versions of RSX, lowercase letters are forced into uppercase. Nonetheless, Broadcast can be very useful for short communications and is often invaluable if you are working on a remote terminal.

19.5 Listing Active Tasks

The Active Tasks command requests a listing of currently active tasks. This consists simply of the name of each active task. Any further information about a particular task must be obtained by another means, such as the Task Status command (discussed in the next section).

The basic form of the Active Tasks command, which is

```
MCR: ACT
DCL: SHOW TASKS/ACTIVE
```

lists only those tasks associated with your terminal. If you are not doing anything, only two tasks will be shown as being active; these are part of the operating system itelf. In MCR these will be called MCR... and ...MCR, which are the primary and secondary command processors. In DCL you will see MCR... and a task called SHOTnn. The first of these illustrates that even though you are working in DCL, MCR is waiting there, behind the scenes, and doing the actual work. The second of these is the DCL task that processes the Show command. As explained in Section 18.5, it is one example of the formation of special tasks to implement the DCL interface. In this example, the SHO task forms the corresponding **ACT** command and passes it to MCR. It then waits until the **ACT** command finishes successfully (it may need to give you an error message) and is thus active along with MCR... when the Active Tasks command actually runs.

The Active Tasks command is often used out of impatience. Suppose, for example, you have started a task build and a minute later you have

not yet received a prompt from your CLI indicating completion of this command. Perhaps the system is unusually slow, or perhaps something has gone wrong. By using the Active Tasks command, you can determine whether the Task Builder is still active. Remember that if you are in DCL, you should look for the MCR task name, which in this example would be TKB or FTB.

Two variants of the Active Tasks command also are of interest. The command

MCR: **ACT /ALL**

DCL: **SHOW TASKS/ACT/ALL**

lists all active tasks in the system. This may be used to list tasks running in background (see Section 25.1) or to get a feeling for how busy the system is as a whole.

The other variant is

MCR: **ACT /TERM=TTnn:**

DCL: **SHOW TASKS/ACT:TTnn:**

where **nn** is the number of a particular terminal. This command lists those active tasks associated with the specified terminal. You can use it to find out what another user is doing. If the user is at command level (i.e., his CLI has given him a prompt and is awaiting further input), the list of active tasks for his terminal will be empty. This command is particularly useful before broadcasting a message (see the previous section) to another user.

19.6 Displaying Task Status

You can use the Active Tasks command (see the previous section) to obtain a listing of the names of active tasks. To obtain detailed information about the status of a particular active task, you must use another command. This information is most likely to be of interest if you have a task running in background.

You can use two virtually identical commands for this purpose. You use the Active Task List command only for a task that is currently active. The form of this command is

MCR: **ATL task_name**

DCL: **SHOW TASK:task_name**

If the specified task is not active, an error message will be displayed stating that the task is not in the system. You can use the Installed Task List command for any installed task. Its form is

MCR: **TAL task_name**

DCL: **SHOW TASK: task_name/INS/FULL**

If the specified task is installed but not active, its list of status codes will include -EXE for "not in execution." For an active task, the response to an Installed Task List command will be the same as that to an Active Task List command. Note that you must specify a system task by its exact name (see Section 18.3). For example, to examine the status of PIP, you must use either the task name ...PIP or PIPTnn.

The status information listed by either command appears on four lines. Some of this will not be of interest to you and is not discussed here. The first line contains the task name and priority. The second line contains the status flags. The last two lines contain the event flags, the processor status word, and the registers, all of which are in octal format.

There are many possible status flags. These are all three- or four-letter codes, some preceded by a minus sign. The ones most likely to be of interest are those explaining why the task is not running. The flag CKP indicates that the task has been checkpointed, while the flag OUT indicates that the task is out of memory, that is, it has been removed from memory. These two conditions are virtually synonymous; rarely will one flag be shown without the other. Through the process of checkpointing, tasks are moved between memory and disk so that limited memory resources can be shared among many different tasks. This occurs routinely as part of multiple-task scheduling under RSX and should not be a source of alarm. If the task is OUT, then the register contents will not be displayed.

The flag WFR indicates that the task is in a wait-for condition. This means that the task cannot continue execution until the condition for which it is waiting has been satisfied. The most common condition here is the completion of an I/O request. The TIO flag represents a specific wait-for condition; it specifies that the task is waiting for input from the user's terminal (TI:). Since a task's attempt to get input from TI: need not be preceded by a prompt to the user, it is possible to have a task hang in this state. The TIO flag is useful for diagnosing this condition.

The flags STP and BLK indicate that the task is stopped or blocked. These two conditions are identical; older versions of RSX use the ter-

minology stop/unstop. With version 4.0 of RSX-11M, these names were changed to block/unblock. When a task is blocked (stopped) it cannot continue to execute until it is unblocked (unstopped). Blocking a task is not the same as aborting it; as long as it is not time-sensitive, an active task can be blocked and subsequently unblocked with no impact on its results. Various system tasks will often be blocked, but your own tasks normally should never be in this state. If you find your task unexpectedly blocked or stopped, ask your system manager to figure out why.

The event flags (EFLG in the third line of status information) are available to you and normally are used for advanced applications involving multiple-task coordination.They may be also used by your task as a means of displaying up to 24 bits of status information, such as the number of times a main loop in your program has been executed. To do this, you must know how to use Executive References, which are outside the scope of this book.

The remainder of the task status information displayed is a snapshot of the contents of the processor registers. This is unlikely to be of interest unless you have to do some complicated debugging.

19.7 Displaying System Status

Most RSX installations include a system task that dynamically portrays the status of the entire system. Although of greatest value to the system manager, it is also available to you. This task is officially known as RMDEMO and is installed under the name RMD. RMD is the Resource Monitor Demonstration program. RMD is intended for use from a video terminal, although you can use it from a hardcopy terminal. From a video terminal, it offers a continuously updated picture of some aspect of the overall system status. From a hard-copy terminal, it offers a single snapshot of the same system parameters.

Under earlier versions of RSX, RMD offered only one mode of operation, a memory display. In this, a display of the total available memory space along with the tasks currently occupying memory is shown. This not only depicts which tasks are currently in memory, but also shows how much memory each occupies and how much (if any) memory is available for other tasks. Other status information, such as the currently executing task, the time of day, the amount of free disk space, and the number and total size of active tasks in and out of memory, also is

given. By watching RMD on a video terminal, you can "see" tasks being loaded into or swapped out of memory.

With version 4.0 of RSX-11M, two other capabilities were added to RMD. An active task display offers a continuously updated list of all currently active tasks, along with a brief status display for each. An individual task display offers a continuously updated display of detailed status information for one particular task.

To run the RMDEMO program, you need merely enter the command

```
MCR: RMD
DCL: SHOW MEMORY
```

The default mode (memory map) will be assumed. If your version of RSX supports the additional RMD capabilities, you can switch from one display to another by entering a single character via your terminal. The choices are:

H = Help: Display options available
M = Memory Map
A = Active Task List
T = Individual Task Display
CTRL/Z = Exit

Note that once RMD is running, it does not matter whether you started in MCR or DCL. Thus, the above commands are the same regardless of your CLI.

You may also start RMD in one of the other modes. To go immediately to the Active Task List display, you use the command

```
MCR: RMD A
DCL: SHOW TASKS/DYNAMIC
```

To go immediately to the Individual Task Display, you use this command

```
MCR: RMD T,T=task_name
DCL: SHOW TASK: task_name/DYNAMIC
```

RMDEMO is perhaps most useful as a system diagnosis tool. It is nonetheless extremely useful as a means of monitoring a single task, which might be of special value to you for debugging a program. A picture is worth a thousand words—the easiest way to understand RMD's value is to use it.

19.8 Obtaining Device Information

The Show Devices command is used to get information concerning the various devices in the computer system. The most basic form of this command is simply

MCR: **DEV**

DCL: **SHOW DEVICES**

This results in a listing of every device along with an extremely brief status report on each. This command is normally not very useful unless you forget the physical device code for a particular peripheral (is the tape drive a type MM: or a type MT: device?), as it reports on every device in the system.

You can also use the Show Devices command to get information on all devices of a specified type

MCR: **DEV dd:**

DCL: **SHOW DEVICES dd:**

or on just one particular device

MCR: **DEV ddnn:**

DCL: **SHOW DEVICES ddnn:**

Earlier versions of RSX allow an alternate form of the DCL command,

DCL: **SHOW DEVICES/ddnn:**

but the form without the / is now preferred. In these commands, **dd:** is the device type, and **ddnn:** is the complete device specifier. Note that when you ask for all devices of a given type, the **dd** device type is followed by a colon. Thus, **DB:** means all disk drives of type DB. In other RSX commands, this would be an abbreviation for DB0:. For example, suppose you wish to copy some files onto a floppy diskette. For this operation, you need a type DY drive. To find out if one is available, you could walk to the computer room and look. Being lazy, you instead enter the command

MCR: **DEV DY:**

DCL: **SHOW DEV DY:**

and let the system tell you.

We have already met a special case of the Show Devices command in Section 19.2. The command

```
MCR: DEV TI:
```

displays the status of the device TI:, which is your terminal. In DCL you could enter this command as

```
DCL: SHOW DEVICES/TI:
```

but the special command **SHOW TERMINAL** is more convenient.

19.9 What Time Is It?

An RSX system measures the time of day. (This is not surprising, since RSX originally was intended for real-time processing applications.) The Time command requests that the current time of day and date be displayed on your terminal. The form of this command is simply

```
MCR: TIM
DCL: SHOW TIME
```

Part III Advanced Techniques

In this, the third and final part of our book, we discuss some of the more advanced techniques available with RSX. The distinction between the topics discussed here and those discussed in parts I and II cannot be sharp. By "advanced" I mean that you do not have to know how to use these techniques to do your work. In parts I and II my intent was to present all that you need to know to create, compile, and run a program. In this part, we discuss techniques that, although not necessary for casual use, will make your work easier. There is bound to be some overlap between these categories, and you may feel that I have put certain topics in one part of this book when they should have been in the other. So be it.

Given my rationale for distinguishing basic from advanced techniques, it must be noted that advanced does not mean more complicated. Some of the topics in part II, such as the use of the Task Builder and the distinction between tasks and task images, are likely to strike you as being more confusing than some of the techniques we consider here. Thus, I urge you not to be afraid to explore these topics, although it is probably best that you become relatively comfortable with what you have already studied before going further. Too much at once is no fun, and it is my hope that mastering RSX will be fun for you.

20

Indirect Command Files

The ability to process indirect commands is one of the more powerful features of the RSX operating system. Once you have become familiar with the basics of using RSX, the use of indirect command files will probably be the first advanced technique you will use. Once you realize how much it can simplify routine operations, you will consider it to be part of your repertoire of basic techniques. Nonetheless, it will undoubtedly be a while before you can fully exploit the capabilities offered by indirect command files.

In general, an indirect command file is a file containing a sequence of commands. You can direct a task to read commands from this file rather than from your terminal. In essence, a command to process an indirect command file causes the source of further command input to be switched from your terminal to the file. When the end of the command file is reached, command input is returned to its normal source, i.e., your terminal. The commands taken from the file have the same effect as if you entered them directly; the only difference lies in the indirect (via the command file) means by which they are entered.

You can use indirect command files at two different levels. At the lower level, you enter the command to take further input from the file to an individual task, typically a system utility. Thus, the commands that can be in the file are limited to those that are acceptable to the particular task. This form of usage is available only from MCR. At the higher level, you enter the command to take further input from the file to your CLI. In this case, the contents of the file are processed by a special system utility known as the Indirect Command Processor. At the lower level of usage, the file is known as an indirect task command file; at the higher level, it is known as an indirect CLI command file.

20.1 Indirect Task Command Files

The use of an indirect task command file represents the simplest way to use command files. In it, all the commands in the indirect command file are given to one task. This type of command file is most commonly used when a long command (or sequence of commands) must be given to a task several times. Note that this does not eliminate the need to enter the command. Instead, it offers a shortened means of doing so. If the full command is itself very short, little is saved by putting it in a command file.

Before going any further, note that you can use indirect task command files only if you are in MCR. This is not a limitation imposed by the structure of RSX; it simply is a limitation in the implementation of DCL. Due to the basic structure of DCL—a shell between you and the actual RSX tasks—you will have fewer opportunities to benefit from using an indirect command file at the task level than you will in MCR. Thus, this feature has not been provided in DCL. If you are using DCL, you must use an indirect command file at the CLI level, where your commands are given directly to DCL. (We discuss this in the next section.) Since the material in this section applies only to MCR usage, I will consider only MCR commands in our examples. If you are a DCL user, you should read this section anyway. It will provide useful background for the remainder of this chapter on indirect command files, and it will illustrate some capabilities that you may wish to exploit, even if you have to use MCR to do so.

The basic form of an indirect command to a task in MCR is

```
tsk @file_specifier
```

Here, **tsk** is the name of the task. It must be installed and will typically be a system utility. The at-sign (@) signifies to the task that whatever follows is not a command but should be used as the name of a command file. The file specifier follows all the normal rules for specifying a file; if it does not include a file type, a default type of CMD is used.

Your most common use of an indirect task command file will probably be with the Task Builder. This is simply because the Task Builder typically requires more complicated commands than do the other system utilities you are likely to use. Let's use it for a few examples of how command files are used at the task level.

Let's consider a very simple example first. Suppose you have compiled a main program in a file named TEST, which requires subroutines in a

file named MATH1. From these you wish to make a task image file named TEST1. (Assume that these files all have standard file types so that the file type need not be specified.) You can build the task via a one-line command to the Fast Task Builder:

```
FTB TEST1=TEST,MATH1
```

Now, suppose you make a file named T.CMD that contains the single line

```
TEST1=TEST,MATH1
```

You can then build the task via the command

```
FTB @T
```

You might argue that, in this example, the savings you obtain by typing @T rather than the actual command line are not significant and do not justify the creation of the file T.CMD. This may be so. Let's move on to a slightly more complicated example.

Suppose now that your task uses the tape drive (MM:), which should be assigned to unit 7. To build the task, you have to use a multiple-line command to specify these options. You could do use this in the conventional manner—that is, after invoking the Fast Task Builder via the command

```
FTB
```

you could type the following set of command lines to FTB:

```
TEST1=TEST,MATH1
/
UNITS=7
ASG=MM:
//
```

Alternatively, if you put these five lines into the file T.CMD, you could again effect the task build via the simple command

```
FTB @T
```

In this case, it is clear that entering the Task Builder command via a command file offers a significant savings.

Continuing with this example, it is of course true that there is no reason to use the command file if you are going to build the task TEST1

only once. After all, by the time you enter the command lines into the command file (via an editor), you could have entered them directly to FTB. The decision here must be based on how many times you are going to build the task, along with the complexity of the build procedure itself.

It is often the case that you will underestimate the number of times you will repeat a task build. A typical example of how things do not work properly the first time might go like this. You have just finished writing the main program TEST; the subroutines in MATH1 have been previously written and tested. You want to build the task TEST1 and run it a few times to get some answers. The first time you run it, you get a divide-by-zero error. You track this down to a mistyped variable name in TEST; you fix it, recompile, and rebuild. The program now runs but gives ridiculous answers. You again figure out why, edit, recompile, and rebuild. This time the answers seem correct, but since you want to put them into a report, you decide to print them out in a slightly different manner. This is yet another change and requires yet another build. By the time you are done, you will have built many versions of TEST1.TSK, each requiring exactly the same set of commands. This is in no way an exaggeration—it is a normal part of the program development process and happens even with short and simple programs.

So far, I have confined our attention to the Task Builder. As noted earlier, indirect command files may be used with many tasks. (Conceptually, any task could take its command input from a command file. In actuality, some additional program logic must be included to offer this capability. Under RSX, most system functions have been designed to accept indirect command files. You should assume that any system function that you are likely to use offers this capability.) As another example, we will consider the use of PIP for cleaning up your directory.

When you are developing a program, the recurrent process of editing, compiling, building, and running can lead to an accumulation of many versions of source, object, task image, and possibly list, map, and data files. It is good practice to clean these up periodically, especially if your disk is relatively full. One common approach is to purge all object and task image files; delete all list and map files; and subsequently purge or delete source and data files on a selective basis. Since all of these are file maintenance activities offered by PIP, you can do this entire sequence via a single indirect task command file. First, you make the command file (call it CLEANUP.CMD) containing the following commands:

```
*.OBJ,*.TSK/PU
*.LST;*,*.MAP;*/DE
*.FTN;*,*.DAT;*/LI
```

Then, when you feel that you have accumulated too many old versions of files in your directory, you can clean the directory up via the command

```
PIP @CLEANUP
```

In Section 11.2 we discussed the various means of invoking a system function. Briefly, if **tsk** is an installed system function, it may be invoked in two ways. The single-line command form is:

```
tsk command
```

In this case, MCR starts the task. The task takes and processes the command, then returns control to MCR. The multiple-line command form is

```
tsk
```

In this case, MCR similarly starts the task, but when the task fails to find a command, it continues to accept commands from your terminal. In this case, the task does not return control to MCR until specifically commanded to do so.

When command files are used, a similar set of rules is followed. Invoking a system function via a command of the form

```
tsk @command file
```

represents a cross between the single- and multiple-line commands discussed above. It is a multiple-line command in that, as in our examples, more than one line of actual command can be given to the task simply by having multiple lines in the command file. It is, however, a single-line command in that, upon detecting the end of the command file, the task returns control to MCR. Note that the commands taken from the file are not echoed on your terminal.

To illustrate this, we consider again our example of cleaning up a directory. If you use the command file CLEANUP, this is what will appear on your terminal:

```
MCR>PIP @CLEANUP
    .
    .
  directory listing for
  *.FTN;* and *.DAT;*
    .
    .
MCR>
```

If you effect exactly the same commands by typing them directly into PIP, this is what will appear on your terminal:

```
MCR>PIP
PIP>*.OBJ,*.TSK/PU
PIP>*.LST;*,*.MAP;*/DE
PIP>*.FTN;*,*.DAT;*/LI
    .
    .
  directory listing
    .
    .
PIP>CTRL/Z
MCR>
```

A command file may also be used as part of a multiple-line command. You do this via a sequence of commands of the form:

```
MCR>tsk
tsk>...individual command lines (optional)
tsk>@command file
tsk>...individual command lines (optional)
tsk>command terminator (optional)
MCR>
```

The invocation of the task puts it into the multiple-line command mode. You can then enter individual command lines. When you enter the line **@command file**, the task takes the contents of the file as a continuation of command input. The significant point is that upon detection of the end of the command file, control does not automatically return to MCR, and you can continue to enter command lines. Concomitantly, it is your responsibility to enter the appropriate command terminator. (If the command file includes a command terminator, control will revert directly to MCR.)

This form of using a command file is particularly useful with the Task Builder. For example, suppose you are writing a set of interrelated main programs that are to be made into separate tasks. All these tasks will require the use of tape drive 2 (MM2:), which must be assigned to unit 3. Each program uses special subroutines for reading or writing to the tape, which are in an object library named TAPEIO.OLB. To support the various task builds required, you would make a file containing a partial set of Task Builder commands. Specifically, you might call this file TAPE.CMD; it would contain the following commands

```
TAPEIO/LB
/
ASG=MM2:3
//
```

To make a task image called PROC1 from the main program PROCESS1, you could use the command file as follows:

```
MCR>FTB
FTB>PROC1=PROCESS1
FTB>@TAPE
MCR>
```

Similarly, to make a task image called PROC2 with a map called PROC2 from the main program PROCESS2, your command sequence would be:

```
MCR>FTB
FTB>PROC2,PROC2=PROCESS2
FTB>@TAPE
MCR>
```

In these examples, the command file contains the special command //, which terminates input to the Task Builder. If it did not, you would be able to follow the command **@TAPE** with the specification of further options, but you would eventually have to enter the termination command yourself.

A special feature of the Task Builder is useful for forming complicated command files. When you are entering commands to the Task Builder, be it directly or indirectly, you can enter comments. Any line beginning with a semicolon will be treated by both TKB and FTB as a comment—that is, it will be ignored. This is useful for documenting task build command files. For example, you could rewrite the file TAPE.CMD as

```
; this is the .OLB file containing IO routines for mag-tape:
TAPEIO/LB
/
; tape drive assumed to be logical unit 3!
ASG=MM2:3
//
```

The last important point concerning indirect task command files is that they can be nested—that is, one command file can refer to another. Consider again our example of building PROC1. The use of the command file TAPE.CMD simplified the specification of the tape subroutines and

options. The whole procedure can be simplified further by creating another command file containing the commands

```
PROC1=PROCESS1
@TAPE
```

If this file is called PROC1.CMD, the entire task build can be done with the single-line command

```
FTB @PROC1
```

In this example, PROC1 is an indirect command file at the first level and TAPE is at the second level. The degree to which nesting is permitted depends on the particular system utility. The Task Builder (both FTB and TKB) and PIP accept two levels of command files. In general, it is unlikely that you will want to nest command files to a depth greater than that allowed.

20.2 Indirect CLI Command Files

The use of indirect command files at the CLI level offers you an even more powerful capability than the task level usage just discussed. This is partially because your Command Line Interpreter (MCR or DCL) itself offers more capabilities than does an individual task, but primarily because this form of usage offers the additional capability of passing directives to the Indirect Command Processor. We will discuss this latter capability in the next section; here, we will consider simpler means of using CLI command files.

If the special directives are not used, the use of a command file at the CLI level is conceptually identical to that at the task level. The CLI is itself a task. Thus, the giving of commands to the CLI via a command file may be thought of as a special example of the general concept of giving commands to any task via an indirect command file. What is special is that the commands given to the CLI typically request other tasks to be run. Thus, these commands to activate a task are at a higher level than the commands that actually tell the task what to do.

The concepts of using an indirect CLI command file (with or without the special directives discussed in the next section) are the same whether you are in MCR or DCL. If you are using MCR, it is probably more natural to refer to these files as indirect MCR command files; similarly, in DCL you would prefer to call them indirect DCL command files.

When we discuss a specific example, I may use one of these alternate designations to stress which CLI is being used.

Let's consider an example. Suppose you are writing a program in MACRO-11 to copy data files. The source file for the main program is COPY.MAC; special subroutines are in an object library UTILITIES.OLB. After each major change to the source code, you want to assemble (making a listing file), task build, and then purge all associated files. A repetitive sequence of operations such as this is an obvious candidate for the use of an indirect CLI command file.

Let's first examine how to do this in MCR. If you interact directly with MCR, your commands will be

```
MCR>MAC COPY,COPY/SP=COPY
MCR>FTB COPY=COPY,UTILITIES/LB
MCR>PIP COPY.*/PU
```

This sequence of commands can also be entered via an indirect MCR command file. First, you make a file (call it COPY.CMD) containing the following three lines:

```
MAC COPY,COPY/SP=COPY
FTB COPY=COPY,UTILITIES/LB
PIP COPY.*/PU
```

Then, when you are at MCR level, the single command

```
MCR>@COPY
```

will suffice. When you enter this command, this is what you will see on your terminal:

```
MCR>@COPY
>MAC COPY,COPY/SP=COPY
>FTB COPY=COPY,UTILITIES/LB
>PIP COPY.*/PU
@ <EOF>
>
```

If you are in DCL, the process is conceptually identical. If you interact directly with DCL, your commands will be

```
DCL>MACRO/LIST COPY
DCL>LINK/FAST COPY,UTILITIES/LIB
DCL>PURGE COPY.*
```

You can make a file (you can again call it COPY.CMD) containing the following three lines

```
MACRO/LIST COPY
LINK/FAST COPY,UTILITIES/LIB
PURGE COPY.*
```

If you make such a file, you can perform the entire process by entering the command

```
DCL>@COPY
```

when you are at DCL level. The only difference between using an indirect MCR command file and an indirect DCL command file is that one contains MCR commands and the other contains DCL commands.

As implied by this example, the syntax for using an indirect command file at the CLI level is the same as that at the task level. The command that you give to the CLI consists of an at-sign (@) followed by the name of a command file. If a file type is not specified, it is taken by default to be CMD.

It is useful to briefly discuss just what happens when you use an indirect CLI command file. Continuing with our example above, in response to the CLI prompt, you enter the command

```
@COPY
```

The fact that the first character is an at-sign (@) signifies that the remainder of the line specifies a command file. A special system utility is activated by MCR (even if you are in DCL) to process the contents of the command file; this utility is known as the Indirect Command Processor. As with other system utilities, its task name is three periods followed by a three-character abbreviation. This abbreviation is AT.—that is, the utility is named after the at-sign used to invoke it. The Indirect Command Processor then processes the file COPY.CMD, one line at a time.

The Indirect Command Processor expects each line of an indirect CLI command file to be either a complete CLI command or a special directive (see the next section). By a complete CLI command, I mean that if you are in MCR, an MCR command is expected, and if you are in DCL, a DCL command is expected. As we discussed in Section 7.3, when you are interacting directly with your CLI, you can use a command from another CLI by putting the name of that CLI in front of the command. You can also do this at the indirect command level. For example, in an

indirect DCL command file, you could have a line beginning with **MCR** followed by any valid MCR command.

Returning to our example above, in the MCR form, the first line is

```
MAC COPY,COPY/SP=COPY
```

The Indirect Command Processor determines that this is a command requesting the MACRO-Assembler. It activates MAC, giving it the rest of the command line. It then waits until MAC is finished, whereupon it processes the second line of the file. This causes control to be passed to FTB. The Indirect Command Processor follows this procedure until there are no more lines remaining in the indirect CLI command file, whereupon it quits, returning control to your CLI.

Two important points related to this process concern the means of aborting the sequence of commands started by an indirect command and the restriction to single-line commands in a command file.

To understand the subtleties associated with aborting a sequence of indirect commands, you must first realize that when an indirect CLI command file is being processed, you will typically have several tasks active at your terminal. In MCR there will normally be two tasks. One will be the Indirect Command Processor (AT.) and the other the MCR task appropriate to the particular command line. In our example above, during the assembly of file COPY.MAC, an active task listing would show both ...AT. and ...MAC. Similarly, during the task build both ...AT. and ...FTB would be active. (I have shown the prototype task names here; as noted in Section 18.3, the names might be AT.Tnn, MACTnn, and so on.) In DCL there will often be three active tasks, since (as described in Section 18.5) DCL introduces an intermediate task as part of the process of translating your commands. The Indirect Command Processor will appear as AT.Tnn. The intermediate task will be as described in Section 18.5 and the actual MCR task will have its normal MCR name.

Suppose now that during the assembly, you realize that you forgot to make an important correction to the source file so that it would be a waste of time to finish the assembly, build the task, etc. Let's assume that you are in MCR. If you get MCR's attention and abort the assembly via the command

```
MCR>ABO MAC
```

the assembly will, as desired, indeed be aborted. The Indirect Command Processor, however, will still be active. It will note that MAC has ter-

minated and will then process the next command, causing FTB to run. The proper way to terminate the processing of an indirect command file is to first abort the Indirect Command Processor and then abort the task currently executing. Thus, in our example, you would enter the two commands

```
MCR>ABO AT.
MCR>ABO MAC
```

When MAC terminates, there will be no AT. for control to return to; control will thus return to MCR. Similar arguments and procedures hold for DCL except that after aborting AT., you next abort the appropriate intermediate DCL task.

The second point is that the Indirect Command Processor expects each task command to be complete within one line. You cannot have multiple-line commands for utilities such as the Task Builder within an indirect CLI command file. Suppose that in our example the program COPY that you are writing is intended to take data that was recorded during an experiment on magtape and copy it into a disk file. The task COPY will now need to use a tape drive, which must be assigned during the task build. If you were entering commands directly into MCR, the procedure could be modified to be

```
MCR>MAC COPY,COPY/SP=COPY
MCR>FTB
FTB>COPY=COPY,UTILITIES/LB
FTB>/
FTB>ASG=MM1:3
FTB>//
MCR>PIP COPY.*/PU
```

You cannot use this procedure in this form in an MCR command file. Clearly, not all the commands above are MCR commands; some of them are FTB commands. If you tried to do this via an MCR command file, the Indirect Command Processor would attempt to interpret the lines of FTB commands as individual MCR commands. Similarly, in DCL you could not use a multiple-line Link command.

To execute this sequence of commands indirectly, you must separate the Task Builder commands and put them into their own command file. You would then have a single-line command to the Task Builder that referred to this indirect task command file. Since indirect task command files are not allowed in DCL, this procedure is, in general, possible only in MCR. [There is one exception to this. In a DCL Link command, you can specify an indirect command file. This is allowed since **LINK** always

forms an indirect command file for the Task Builder anyway. When, however, you use the construct **LINK @file**; the indirect command file must contain Task Builder commands; that is, it must be in MCR form. If you are going to do this, you might as well do everything using MCR commands. In describing how to do this, I will consider MCR only.)

In our modified example, the command file for the Task Builder (you might call it COPYBLD.CMD) would contain the lines

```
COPY=COPY,UTILITIES/LB
/
ASG=MM1:3
//
```

The file COPY.CMD would then be this:

```
MAC COPY,COPY/SP=COPY
FTB @COPYBLD
PIP COPY.*/PU
```

With these two files, you can do everything by entering one simple command

```
MCR>@COPY
```

The procedure just described is somewhat clumsy because it requires two separate command files. Nonetheless, it is still much easier than typing in the complete set of commands, especially if you are making numerous changes to the source code.

In general, you will need two (or more) command files whenever you have to pass a multiple-line command to a task from an MCR command file. This might happen in a variety of ways; you will probably encounter it only with the Task Builder. I will restrict our attention to that case, but my comments apply directly to other examples.

When you need two command files, a slight problem arises regarding the choice of file names. In cases where only one command file is used—be it the MCR command file or the Task Builder command file—it is a natural choice to choose the file name to be the same as that of the main program and the file type to be CMD. Then, all files related to the project can be isolated from the other files in your user area by a wildcard construct of the form **name.***. This is useful for getting directory listings, making backups, etc. When both command files are needed, you cannot use the natural name for each, and you have to give one a different name. In this case, it is common to use the natural name for

the MCR command file and a modified name for the other command file. I did this in our example above by using the name COPYBLD.CMD, which parallels the constructions used by Digital in the RSX software. Another choice would be BLDCOPY.CMD. This is not quite as good, since the file name does not begin with the letters COPY. As we discuss in Chapter 23, if several related files have names that are not identical, but that all begin with the same characters, you can use the special PIP wildcards to access all of them in a single command. An alternate approach for naming the second command file is to leave the file name the same but change the type—for instance, COPY.FTB. This approach has the advantage that the normal wildcard construct COPY.* can be used; it has the disadvantage that you must specify the file type when you use the command file. There are no strong reasons to choose one method over another; it is a matter of personal preference.

Finally, it is important to realize that CLI command files can be nested. (This is true for DCL as well as MCR.) The basic rule noted earlier is that each line in a command file (if not a special directive) must be a complete command. This can be satisfied by a reference to another CLI command file. As an example of nesting, suppose you are writing a series of programs that will be used to create three interrelated tasks. Let's call these TSK1, TSK2 and TSK3. You might have a command file BUILDTSK1.CMD which does the compilations and task build for TSK1. Command files BUILDTSK2.CMD and BUILDTSK3.CMD would similarly be for the other tasks. If you make changes only to programs in one task, you need to execute only one of these command files. If, however, your changes affect all the tasks, then you must execute all three of these command files. You can do this with another command file (call it BUILDALL.CMD) which functions as a sort of super command file. This file would contain three lines:

```
@BUILDTSK1
@BUILDTSK2
@BUILDTSK3
```

To perform all the compilations and task builds for the three tasks, your only input would be

```
@BUILDALL
```

20.3 The Indirect Command Processor

In the last section we discussed the means by which the Indirect Command Processor (AT.) processes a simple indirect CLI command file. In

this section we will examine more sophisticated command files—that is, those files containing special directives to the Indirect Command Processor.

The Indirect Command Processor directives offer a set of commands that forms a high level programming language. Using these directives, you can direct the Indirect Command Processor to interact with you the user, read and create files, test conditions, control the processing of the directives, and construct commands to be given to the CLI. We will not discuss every possible directive that can be used with the Indirect Command Processor; many of these are intended for complicated system level use such as in the command files used for RSX system generation. Instead, we will consider only those directives that are likely to be of the most interest to you. Nonetheless, our discussion here will be rather lengthy. This is simply because it must be, in effect, a description of an entire programming language.

In our discussion of the use of Indirect Command Processor directives, I will not deal with the various possible directives one at a time, explaining in full detail the syntax and special features of each. Rather, I will present realistic examples, proceeding from the very basic to those at a medium level of sophistication. As I present solutions for each example, I will introduce new directives or new features of ones introduced earlier. In this way, I hope you will be able to obtain a feeling for why each directive exists.

An important point to bear in mind is that when you create a complicated indirect command file, you are, in effect, programming. You should use the same techniques of choosing meaningful names and providing informative comments that you would use when writing a computer program. Similarly, in long command lines, you should use blanks to improve legibility. Just as you might have to look at an old program and remember what it does, so too might you have to reuse or revise old command files.

Much of the material in this section is dependent upon the version of RSX being run. The Indirect Command Processor is one of the features of RSX that has been significantly enhanced in recent releases of the operating system. Unfortunately, some of the topics we discuss do not apply to older versions. These topics are important enough to discuss anyway; if they do not apply to your system, bear with me.

Let's first discuss very briefly just what a directive is. In general, each line in an indirect CLI command file is either a command acceptable to your CLI or a special directive. AT. reads a command file one line at a time. If a line is determined to be a CLI command, it is (after possible modification) given to the CLI; if a line is determined to be a directive,

it is processed by AT. itself. Directives are characterized by a period as the first non-blank character in the line. The nature of the directive is specified by a keyword, which must immediately follow the period. Depending on what the keyword is, one or more parameters or values may follow.

Before I can explain the use of particular directives, we need to discuss the concept of symbols. The Indirect Command Processor offers the means to define and subsequently use symbols. (The use of symbols in a command file is conceptually equivalent to the use of variables in a computer program; I use the term "symbol" because that is the term used in the RSX manuals.) The name of a symbol can be up to six characters (a letter, digit or dollar sign) long and must begin with a letter or dollar sign. There are three possible types of symbols—string (S), logical (L) and numeric (N). A string symbol can have as its value a string consisting of 0 through 132 characters. (A string of 0 characters is the null, or empty, string.) A logical symbol can have the value either true or false. A numeric symbol can have as its value a 16-bit nonnegative integer (0 through 65,535).

You can use various directives to define the value of a symbol. The particular directive used also specifies the type of the symbol. (Note that the concept of declaring the existence of a variable without specifying its value, typical of most programming languages, does not apply to the Indirect Command Processor.) Once a symbol has been defined, you can subsequently change its value but not its type. There are three ways to define a symbol: by a Set or an Ask directive for user-defined symbols, or automatically for system-defined symbols.

The Set directives define a symbol to have a predetermined value. The Set directives for string, logical, and numeric symbols, respectively, are:

```
.SETS symbol string_expression
.SETL symbol logical_expression
.SETN symbol numeric_expression
```

In each case, the expression specifies the value to be assigned to the symbol; the rules for writing expressions will be discussed later. In addition, the following directives set a logical symbol to the values true and false, respectively:

```
.SETT symbol
.SETF symbol
```

The Ask directives define a symbol to have the value typed in by the

user in response to a displayed message. The Ask directives for string, logical, and numeric symbols, respectively, are:

```
.ASKS symbol text_string
.ASK symbol text_string
.ASKN symbol text_string
```

(Note that the Ask directive for a logical symbol is **.ASK**, not **.ASKL**.) In each case, the text string is displayed on your terminal; the value that you enter is then assigned to the symbol. The text string itself is not enclosed in quotes; it is delimited by the mandatory blank following the symbol name and by the end of the directive line. Most typically, you will write command files that you will use yourself. In this instance, you will be "the user," and you will respond to your own queries. Nonetheless, as in using interactive I/O statements in a normal program, you should construct your Ask directives so that they can be readily understood by someone else.

Special symbols have values assigned to them by the operating system. These symbols are distinguished from user-defined symbols by their names being enclosed in angle brackets—e.g., <EXSTAT> is the name of a special symbol. These symbols also are categorized as being string, logical, or numeric and may be used in the same manner as user-defined symbols. In addition, some special symbols are designated as being reserved symbols. Reserved symbols are similar to special symbols in the sense that their values are automatically defined for you; unlike special symbols, however, their names are not enclosed in angle brackets.

As a casual user, your most common use of Indirect Command Processor directives will be to manipulate strings, and your most common use of strings will be to fill in file names for various commands to your CLI. Intrinsic to all this is the concept of symbol value substitution. Whenever a symbol name enclosed in single quotes (apostrophes) is encountered in a line in a command file, symbol value substitution causes the value of the symbol to be substituted for its name. For example, if the string symbol A has been assigned the value TEST, the line

```
F77 'A'='A'
```

will be transformed into the line

```
F77 TEST=TEST
```

Substitution can be used for string, logical, or numeric symbols; we will consider its use only for string symbols.

It is important to note that substitution does not happen automatically. In general, AT. has several modes of operation, each of which can be selectively turned on or off via the directives **.ENABLE** and **.DISABLE**. One of these modes controls substitution. By default, this mode is turned off. When it is off, the Indirect Command Processor will not perform the substitution illustrated above. Thus, before any substitutions can be done, you must turn on this mode using the directive

```
.ENABLE SUBSTITUTION
```

Let's illustrate all of this so far by a simple yet useful example. Suppose you commonly write simple FORTRAN programs (by "simple" I mean that these programs do not utilize subroutines in other files). If **filename** is the actual name of a particular source file, a typical series of MCR commands that you would use might be:

```
F77 filename,filename=filename
FTB filename=filename
PIP filename.*/PU
```

In the previous section we discussed how to make a command file that would do this series of operations for one particular file name. Using this technique, every time you write a new program, you will have to write a command file to go with it. This is clearly undesirable. Using directives, you can instead write one general purpose command file that can be used to perform this series of operations for any source file. You can do this by writing the MCR commands above in terms of symbols. After assigning a particular file name to the symbol, symbol value substitution will transform these commands from the general to the specific. Note that a simple, repetitious use of the same file name such as we have here is a perfect choice for symbol value substitution in a command file.

I will use this same general set of operations (compiling, linking, and purging) for most of our examples in this section. (Note that if you are in DCL, the commands will be different, but the concepts concerning symbol value substitution, etc., and the directives used to effect them will be identical. For simplicity, I will restrict our examples to the MCR command forms.)

We now reconsider the above sequence of operations, this time using directives to allow the substitution of an arbitrary file name. A command file that you might use is as follows:

```
.ENABLE SUBSTITUTION
.ASKS FILE Enter file name:
```

```
F77 'FILE', 'FILE' = 'FILE'
FTB 'FILE' = 'FILE'
PIP 'FILE'.*/PU
```

In this example, the first two lines begin with periods and are thus interpreted as directives to be processed by AT. itself. The Ask directive (**.ASKS**) takes whatever you type in and assigns it as the value of the symbol FILE; because it is an **.ASKS** and not an **.ASK** or **.ASKN**, FILE is defined to be a string symbol. Since the third line in the file does not begin with a period, AT. assumes it to be an MCR command. Since substitution mode has been enabled, AT. first scans the line to see whether it contains any single quotes. The first pair found encloses the characters **FILE**; this matches the name of a defined symbol, and the value of that symbol is substituted for the six characters **'FILE'**. The other two occurrences of **'FILE'** in the line are similarly changed. Only after substitution has been completed is the line passed as a command to MCR. The Fast Task Builder and PIP commands are similarly changed before being passed to MCR.

In the example above, we have put additional spaces in the F77 and FTB commands. These are not necessary, but they do improve the legibility of the commands, especially with all the apostrophes and other punctuation. The key concept here is that you will type the command into the indirect command file only once; the extra effort to insert spaces for legibility is minimal. We will use this technique throughout this section. Note that after symbol value substitution, these additional spaces will remain, and we accordingly include them when we show what appears on your terminal.

Suppose the command file above is called COMPILE.CMD. A typical use of this file might look like this:

```
>@COMPILE
>*Enter File name: [S]: RWTEST
>F77 RWTEST, RWTEST = RWTEST
>FTB RWTEST = RWTEST
>PIP RWTEST.*/PU
>@ <EOF>
>
```

In this example, the first line is the command to MCR to run the Indirect Command Processor with file COMPILE.CMD as input. The next line is produced by AT. in response to the Ask directive. The specified text string is preceded by an asterisk as a prompt and is followed by an S in brackets to signify that a string value is required. Your response, which in this case is the file name **RWTEST**, is entered on the same line. The

next three lines are the commands for compiling, building, and purging, all of which have the file name properly substituted. The last line is generated by AT. and specifies that the end of the command file has been reached; AT. then returns control to MCR.

By using an indirect CLI command file, your required input in this example is reduced to only two things: the command to process the command file and the name of the source file. The simplification here is twofold. First, you need to enter the file name only once rather than several (in this particular example, six) times. This reduces the chances of error and eliminates monotonous work. In addition, you do not have to enter the individual CLI commands directly. This too reduces the amount that you must type in. What is sometimes more important is that this gives you the ability to do things without being there. In the example above, after typing in the file name, you can walk away from your terminal. AT. will automatically generate the commands for compiling, building, and purging; there is no need for you to sit and watch them. You may instead get a cup of coffee. When you return, the processing will probably be complete, in which case your CLI will be waiting for your next command.

As shown by this simple example, the technique of symbol value substitution for file names can be extremely useful. The basic concept is not limited to the example I have given but can be extended to more complicated examples with multiple symbols. Because the compile and build sequence is so common, I will continue with this example, using it to illustrate some other things you can do with directives and symbols.

I earlier mentioned the existence of reserved symbols. These are symbols that are predefined. Unlike the special symbols, their names are not enclosed in angle brackets and they are defined by AT. itself, not by RSX. There are eleven such symbols; their names are COMMAN and P0 through P9. These are all string symbols. (These symbols are officially implemented on version 4.0 of RSX-11M; they exist as unsupported and undocumented features on version 3.2.) When AT. is activated by MCR in response to an @ command, it assigns values to each of these symbols based on the exact contents of the command line. The symbol COMMAN contains the entire line, and the symbols P0 through P9 contain individual parameters in the command line, where parameters are identified as being separated by spaces. The zeroeth parameter, P0, is assigned a string containing an at-sign and the name of the file. If other parameters are in the command line, they are assigned to the symbols P1, P2, etc. Otherwise these symbols are assigned the empty string. To illustrate this, suppose you enter the following command:

```
@SYMBOLS XYZ X Y Z
```

Before starting to process the file SYMBOLS.CMD, the Indirect Command Processor will define the reserved symbols as follows:

COMMAN	@SYMBOLS XYZ X Y Z
P0	@SYMBOLS
P1	XYZ
P2	X
P3	Y
P4	Z
P5–P9	empty

Within the command file, you can use these reserved symbols as you wish.

We now modify our command file COMPILE to exploit this capability. Suppose you rewrite it as follows:

```
.ENABLE SUBSTITUTION
F77 'P1', 'P1' = 'P1'
FTB 'P1' = 'P1'
PIP 'P1'.*/PU
```

To avoid confusion, call this file COMPILE2.CMD. To use it, you enter a command such as

```
@COMPILE2 filename
```

where **filename** is the file to be compiled, built, etc. A typical use of this would look like this:

```
>@COMPILE2 RWTEST
>F77 RWTEST, RWTEST = RWTEST
>FTB RWTEST = RWTEST
>PIP RWTEST.*/PU
>@ <EOF>
>
```

The net effect is exactly the same as with the earlier example of COMPILE.CMD. The difference in usage is that there is no Ask directive and correspondingly no need for user input after the command line. This may seem to be a rather minor improvement, and that is true if all you are doing is compiling and building one file. The importance of this capability is that by eliminatng the need for further user input, it allows the command to be issued from within another command file. As an

example of this nesting of command files, suppose you have three separate programs named PROG1, PROG2, and PROG3. You can use a single command file (call it COMPALL.CMD) to compile and build all three of these programs by issuing three successive commands to process the command file COMPILE2. The command file COMPALL would contain the three lines

```
@COMPILE2 PROG1
@COMPILE2 PROG2
@COMPILE2 PROG3
```

This is what happens when you use this command file:

```
>@COMPALL
>F77 PROG1, PROG1 = PROG1
>FTB PROG1 = PROG1
>PIP PROG1.*/PU
>F77 PROG2, PROG2 = PROG2
>FTB PROG2 = PROG2
>PIP PROG2.*/PU
>F77 PROG3, PROG3 = PROG3
>FTB PROG3 = PROG3
>PIP PROG3.*/PU
>@ <EOF>
>
```

In essence, we have created something (let's call it a procedure) that offers capabilities greater than those of a single utility but that is nonetheless used as though it were a single utility. In this example, the procedure @COMPILE2 is syntactically equivalent to any other single MCR command, yet it results in the succession of three separate utilities (F77, FTB, and PIP). This is a very powerful and useful capability.

Now, suppose you would like to have a command file similar to COMPILE2 but that you do not want to be required to purge your source files. (Automatic purging of your source files is not necessarily a good idea. We discuss this topic in Section 21.1.) You nonetheless would like to automatically purge the other files made during compiling and building. One way to do this is to list all the file types that you wish to purge—that is, you could change the last line of the command file COMPILE2 to

```
PIP 'P1'.OBJ, 'P1'.LST, 'P1'.TSK/PU
```

There is a certain inelegance about this compared to the use of the wild-

card file type. Another technique is to use one file name for the source file and another for all the other files. In this case, you would probably use a long, meaningful name for the source file and a short, convenient name for the other files. You can modify COMPILE2.CMD to accommodate two distinct file names simply by using both the P1 and P2 parameters from the command line. Choose P1 for the source file name and P2 for the name of the other files. The command file now looks like this:

```
.ENABLE SUBSTITUTION
F77 'P2', 'P2' = 'P1'
FTB 'P2' = 'P2'
PIP 'P2'.*/PU
```

Let's call this file COMPILE3.CMD. You might use it with a source file named ASTROLOGY, using the simple name A for the other files. If you do, this is what happens:

```
>@COMPILE3 ASTROLOGY A
>F77 A, A = ASTROLOGY
>FTB A = A
>PIP A.*/PU
>@ <EOF>
```

As the next complication, suppose that at times you would like to be able to have a different name for the source file but that at other times you would like to use just one name for everything. In other words, you would like to have a command file that was sometimes like COMPILE3 and sometimes like COMPILE2. You can do this by using the conditional capability of the Indirect Command Processor. The logic is very simply this. If there are two parameters on the command line you will use them as separate file names, but if there is only one, you will use it as a name for all the files. Thus, you need to be able to test the second parameter (P2) and see if it contains anything. You can do this by comparing it against the empty string.

The comparison is effected by the If directive, the general form of which is

```
.IF condition statement
```

This directive is very similar to the Logical-IF of FORTRAN—if the condition is true, then the statement is executed; if the condition is not true, then the statement is not executed. The statement can be either

a CLI command or another directive. You must follow special rules for specifying the condition to be tested. The condition must compare a symbol to an expression; the symbol must be given first. The condition is written as

```
symbol relation expression
```

The possible relations are:

EQ	Equal
NE	Not equal
GT	Greater than
GE	Greater than or equal
LT	Less than
LE	Less than or equal

You can use these relations with either numeric or string symbols and expressions. (If you are comparing string quantities, you will probably be interested in using only the EQ and NE relational operators.) We have not yet discussed the rules for forming expressions; for now, it is sufficient to note that any constant, any symbol, or any combination thereof constitutes a valid expression. In general, a string constant is delimited by quotation marks ("). In particular, two consecutive quotation marks ("") represent the empty string. Thus, the condition you wish to test can be written as

```
P2 NE ""
```

This form of the If directive is used with string or numeric symbols. For logical symbols, the only conditions that make sense are whether the symbol has the value true or false. To test a logical symbol, two special forms of the If directive are provided. These are

```
.IFT symbol statement
.IFF symbol statement
```

In the first case, the statement is executed if the symbol is true; in the second case it is executed if the symbol is false.

We now utilize the conditional directives to determine how to set the output file names. This is really no different from writing a simple program, and as is often the case, there are several equally valid ways to design the basic program logic. I will use one string symbol (OUT) for the name of all the nonsource files and will set it equal to either the second or the first parameter in the command line. For the sake of

generality, I will use another string symbol (IN) for the name of the source file, although, since this will always be set equal to the first parameter, it is redundant. The complete command file may then be written as:

```
.ENABLE SUBSTITUTION
.SETS IN P1
.IF P2 EQ "" .SETS OUT P1
.IF P2 NE "" .SETS OUT P2
F77 'OUT', 'OUT' = 'IN'
FTB 'OUT' = 'OUT'
PIP 'OUT'.*/PU
```

Let's call this file COMPILE4.CMD. If you use it with distinct input and output file names, you will get an interaction that looks like this:

```
>@COMPILE4 RWTEST RW
>F77 RW, RW = RWTEST
>FTB RW = RW
>PIP RW.*/PU
>@ <EOF>
>
```

Alternatively, if you specify only one file name, the sequence of commands that results looks like this:

```
>@COMPILE4 RWTEST
>F77 RWTEST, RWTEST = RWTEST
>FTB RWTEST = RWTEST
>PIP RWTEST.*/PU
>@ <EOF>
>
```

Let's study the command file COMPILE4 a bit further. In it, I have used the Set directive,

```
.SETS IN P1
```

Here, IN is a string symbol that is assigned a value determined by the string expression P1. This is one of the simpler forms of a string expression as it consists of a single string symbol. It is important to note that in this case, we simply use the symbol name (P1), not the name enclosed in apostrophes ('P1'). This may seem confusing because when you substitute the value of a symbol into a CLI command you need to use the apostrophes.

As I mentioned earlier, lines in an indirect CLI command file are either directives or CLI commands. If you remember this distinction, you will

know when to enclose a symbol name in apostrophes and when not to. Directives are processed by AT., the Indirect Command Processor; CLI commands are processed by your CLI. AT. understands symbols; the CLI does not. Thus, when AT. processes a directive, it uses symbol values when symbol names are encountered. AT. does not need apostrophes to signify this substitution in a directive; in fact, this action takes place whether substitution mode is enabled or not. It might seem that apostrophes in a directive would be merely redundant. This is not so—they are actually wrong. If substitution mode is enabled, AT. looks for apostrophes as it reads each line from the command file. For each pair found, it substitutes the value of the enclosed symbol name. Only after this is done does AT. interpret the line as being either a directive or a CLI command. Thus, apostrophes in a directive actually result in double substitution. For example, suppose you have the directive

```
.SETS T1 'A'
```

in a command file. If substitution mode has not been enabled, the three-character string **'A'** will be rejected as being an invalid symbol name. (The apostrophe is not a valid character for forming a symbol name.) If substitution mode has been enabled, the value of the string symbol A will be substituted for 'A' and will itself subsequently be interpreted as the name of a symbol. In contradistinction to this, when the CLI is given a line from a command file, it does not (it cannot) do any substitution. Thus, in CLI command lines, symbol names must be enclosed in apostrophes to force substitution prior to AT.'s giving the command line being given to the CLI. All of this may be confusing at first, but the logic behind it is consistent, and once you understand how it works, you will be able to remember how to use it.

In our examples, I have sometimes used simple expressions. Now I will define just what an expression is—and when you can use one. An expression may be used only as the second operand in either an **.IF** or in a **.SETS**, **.SETN**, or **.SETL** directive. An expression is any combination of constants and symbols, written as one or more such terms with adjacent terms connected by operators. No spaces or tabs are allowed between terms and connecting operators, since a space or tab is interpreted as marking the end of the expression. Unless parentheses are used, terms are combined from left to right.

We are primarily interested in string expressions. These are made from string symbols and string constants. The only operator allowed in a string expression is the concatenation operator, which is denoted by a plus sign. This operator simply combines two strings into one, putting the

second after the first. Thus, the string expression

```
"A"+"B"+"C"
```

is equivalent to the string constant "ABC". Similarly, if the string symbol NAME has been assigned the value "TEST" and the string symbol TYPE has been assigned the value "MAC", then the string expression

```
NAME+"."+TYPE
```

is equivalent to the string constant "TEST.MAC".

You will probably not need to use numeric or logical expressions that are more complicated than a single constant or symbol. More complicated numeric or logical expressions can, however, be formed by using operators to combine constants and/or symbols. For the sake of completeness, there are four allowable numeric operators: +, −, *, and /, which follow the normal rules of integer arithmetic. There are three allowable logical operators: inclusive OR (!), AND (&), and NOT (#), which follow the normal rules of Boolean algebra.

Let's return to our ongoing example and introduce yet another improvement. The basic sequence of operations we have been considering is compiling followed by task building. If you want your command file to do exactly what you would do interactively at a terminal, you must include an intermediate step—checking for errors in the compilation. You will want to do the task build only if the compilation was successful. Generalizing, you will often want to proceed with the rest of the steps in a command file only if the previous step was successful. This capability is available via the special symbol <EXSTAT>.

<EXSTAT> is a special numeric symbol that is assigned a value whenever a task exits. In general, a task can give RSX a status value when it exits. This is the value that is assigned to <EXSTAT>. Most RSX utilities support this feature. The standard values used are:

```
 1 = Success
 0 = Warning
 2 = Error
 4 = Severe error
17 = No status information available
```

The last value (17) is used by the Indirect Command Processor if the task does not supply a status value when it exits. On version 3.2 of RSX-11M, the return value 17 is not implemented; if a task does not return a status value, the value 1 (success) is assumed.

To clarify the use of <EXSTAT>, let's use a simple command file as

an example—we repeat our earlier example of COMPILE2:

```
.ENABLE SUBSTITUTION
F77 'P1', 'P1' = 'P1'
FTB 'P1' = 'P1'
PIP 'P1'.*/PU
```

We first want to check whether the compilation was successful or not. You can do this by checking the value of <EXSTAT> between the F77 and FTB commands. Since F77 returns a status value when it exits, it should be impossible to obtain a value of 17 at this point. Thus, the only value that we will consider acceptable is 1 (Success). If <EXSTAT> has any other value, we should terminate processing of the indirect command file. The test is done with an If directive. There are two directives available for terminating processing. These are **.STOP** and / (a single slash). These are identical in their action; **.STOP** is more readily understood if you are reading the command file. The complete test may be written either as

```
.IF <EXSTAT> NE 1 .STOP
```

or as

```
.IF <EXSTAT> NE 1 /
```

Note that this directive does not depend upon its coming after an F77 command, it merely expects the exit status to be set and could be used equally well after other utilities. In particular, we can also use it after the FTB command. The complete command file then becomes

```
.ENABLE SUBSTITUTION
F77 'P1', 'P1' = 'P1'
.IF <EXSTAT> NE 1 .STOP
FTB 'P1' = 'P1'
.IF <EXSTAT> NE 1 .STOP
PIP 'P1'.*/PU
```

Let's call this command file COMPILE2X.CMD. If you use it with a source file that has no errors, the sequence of operations that results looks like this:

```
>@COMPILE2X RWTEST
>F77 RWTEST, RWTEST = RWTEST
>FTB RWTEST = RWTEST
>PIP RWTEST.*/PU
>@ <EOF>
>
```

If you use it with a source file that has errors, you get something like this:

```
>@COMPILE2X RWERROR
>F77 RWERROR, RWERROR = RWERROR
F77---ERROR 48-F Missing keyword
  [n (unit=1, file] in module.MAIN. at line 11
F77---1 Error RWERROR.FTN;2

>@ <EOF>
>
```

More complicated decisions can be made based on the value of the exit status. For example, if there is a compilation error, you might want to issue a Print command for the listing file before terminating the processing of the command file. In this case, you must be able to do two things when a bad exit status is detected. Since the Indirect Command Processor language does not support compound statements (such as those found in languages supporting structured programming), you must be able to branch or jump within the command file. This is done with the **.GOTO** directive, the form of which is simply

```
.GOTO label
```

In function, this is equivalent to the dreaded GO TO statement of FORTRAN. To define the point to which control is to be transferred, a label directive is used. This is of the form

```
.label:
```

where **label** may be up to six characters (alphabetic, numeric, or dollar sign). The label directive must be at the beginning of a line. It can be followed on that line by a directive or command or it can be on a line by itself. If you do not indent every line of the command file, you may wish to put the label on its own separate line, as it is more noticeable that way. Also, this allows the test and resulting action to be written as a block of complete lines, which is a convenience if you are modifying a command file with a line-oriented text editor such as EDI. If you have already made the file COMPILE2X.CMD and you wish to change it so that the list file will be printed before stopping when an error is detected, you could replace the first If directive with the block of lines

```
.IF <EXSTAT> EQ 1 .GOTO GOOD
PRI 'P1'.LST
.STOP
.GOOD:
```

You can construct more sophisticated control flows based on examining the value of <EXSTAT> at various points in a command file; I leave these to your imagination.

The list of available special symbols is rather large. (This is something that is highly dependent on the version of RSX being run.) Almost all of these, however, will probably never be of interest to you. In addition to <EXSTAT>, you will probably find only one other special symbol useful. This is the logical symbol <LOCAL>, which we will discuss in the next section.

At the beginning of this section, I stated that using the Indirect Command Processor directives is equivalent to writing a program in a high level language. To highlight each feature that we have learned, I have kept our examples relatively simple. Nonetheless, it should be easy to see that you may find yourself creating rather complex command files as you become more and more familiar with the capabilities of the Indirect Command Processor. You may reach a point at which you are modifying your command files in a manner analogous to maintaining "regular" computer programs. At this point, if not earlier, it will be very useful for you to put comments into your command files.

You can put comments into an indirect CLI command file either as separate lines or at the end of other lines. The Indirect Command Processor treats a line that begins with a semicolon as a comment. It is echoed to your terminal but otherwise ignored. You can suppress the display of the comment on your terminal by putting a period in front of the semicolon. (This must be the first character in the line.) The two forms of comment look like this:

```
; comment that is echoed to terminal
.; comment that is not echoed
```

In many cases, a directive may be followed by a comment in the same line. The comment is preceded by a semicolon. An example of this form of comment is

```
.SETS IN P1 ; input the file name
```

In general, in-line comments cannot be used with directives that end with a text string (for example **.ASK** or **.DATA**) or with a file specifier (**.OPEN**). If a directive ends with a text string, any comment (including the introductory semicolon) will be taken to be part of the text string. When a directive ends with a file specifier, you cannot use the in line comment because the semicolon for the comment could be confused with the semicolon used for specifying the version number of the file.

Any CLI command may be followed by a comment in the same line. In this case, an exclamation point is used instead of a semicolon to introduce the comment—this choice of syntax is again due to possibly ambiguous interpretation of the semicolon. (This syntax is a function of the CLI, not of the Indirect Command Processor; both MCR and DCL allow it.) An example of this form of comment is

```
PIP 'OUT'.*/PU ! Purge only the output files.
```

When we were discussing the technique used for symbol value substitution, I stated that substitution (if enabled) occurs before any interpretation of the contents of the line is made. This is also true of comments. Thus, assuming substitution mode to be enabled, the line

```
;A = 'A'
```

will be displayed on your terminal with the 'A' replaced by the current value of symbol A. This is the easiest way to display the value of a symbol and can be quite useful for debugging a command file. (Yes, writing a command file is like writing a program, including the possible need for debugging.) As a simple example, suppose that you have a command file and you would like to add some extra statements based on testing the value of the exit status. Before you do this, however, you need to know just how the exit status is set under various conditions. You could try to determine this from documentation (the theoretical approach), or you could simply run a test case and see what happens (the empirical approach). Suppose specifically that you want to know what status is returned by PIP when you ask it to delete a file that does not exist. The easiest way to find out is to write and execute a very short command file such as

```
.ENABLE SUBSTITUTION
PIP ARENTANY.JNK;*/DE
;<EXSTAT> = '<EXSTAT>'
```

If you call this file TESTEX.CMD and run it, you will see that the status value in this case is 0 (Warning):

```
>@TESTEX
>PIP ARENTANY.JNK;*/DE
PIP --- No such file(s)
SYO:[265,10]ARENTANY.JNK;*
>;<EXSTAT> = 0
>@ <EOF>
```

This trick of displaying symbol values depends on the fact that comments preceded by a semicolon are echoed to your terminal. More specifically, all CLI commands (after substitution, if enabled) are echoed to your terminal; all directives are not echoed. (An exception to this is the @ command used to nest command files. This is not echoed nor is anything displayed when control returns from the lower level to the high level command file.) If you think of ; as being a CLI comment and .; as being a directive comment, it will be easier to remember that the first one is echoed and the second is not.

It is sometimes tiresome to see the CLI commands being echoed every time the command file is used. This echoing is controlled by a mode of Indirect Command Processor operation known as quiet mode. Normally, the quiet mode is disabled — that is, CLI commands and comments are echoed. On some RSX systems (this is a system generation option) it is possible to enable the quiet mode, in which case CLI commands and comments will no longer be echoed. (Even when quiet mode is in effect, output such as error messages, produced as a result of executing a CLI command, will be displayed on the terminal.) You can enable and disable the quiet mode via the directives

```
.ENABLE QUIET
.DISABLE QUIET
```

You can alternate these so that echoing will be done for only certain portions of the command file.

So far, I have concentrated our attention on the use of directives and symbol value substitution for the formation of commands to be given to your CLI. In RSX-11M commands are limited to one line (79 characters), which is not always long enough. When a longer command line is needed for a utility, you can use an indirect task command file. As explained in the previous section, this is most typically the case with the Task Builder, where you might need to name many input files. The final topic we discuss in this section is how to use directives and symbol value substitution to make entire command files rather than single line commands.

The Indirect Command Processor includes directives for opening and closing files and for writing into and reading from files. There are several directives for opening a file. **.OPEN** opens a file for output. This file is always a new file; if a file with the same name already exists, the file with the next higher version number is made; if the version is specified and that file already exists, then it is written over. Two variants of **.OPEN** also exist. **.OPENA** is similar to **.OPEN** except that the specified file is

assumed to already exist and any output will be appended to (written at the end of) the file. If no file matching the given specifier exists, then **.OPENA** is handled in the same manner as **.OPEN**. The other variant is **.OPENR** (version 4.0 of RSX-11M and later) which opens a file for reading—the specified file must already exist. For those familiar with FORTRAN-77, these three directives are equivalent to Open statements with the following choices of keywords:

```
.OPEN       type = 'new'
.OPENA      type = 'unknown', access = 'append'
.OPENR      type = 'old', readonly
```

In general, several files may be simultaneously open during execution of a command file. There should be no reason why you will need to have more than one file open, in which case you can use the simple form of **.OPEN**. This is

```
.OPEN file_specifier
```

If no file type is specified, DAT is assumed by default. The form of **.OPENA** and **.OPENR** is identical. When only one file has been opened, the corresponding directive to close the file is simply

```
.CLOSE
```

This is used no matter how the file was opened.

Once a file has been opened, data may be written into or read from it during execution of the indirect command file. We will not discuss reading, as you probably will not need to use it. You can write text to a file in one of two manners. The Data directive (**.DATA**) may be used to write a single line (record) to the file. The form of this is

```
.DATA text
```

The text string is written to the output file. A blank is normally used to separate the text string from the directive word **.DATA**, but this is not mandatory. If the first character after **.DATA** is a blank, it is discarded; otherwise, the text string is written as it is. Successive Data directives can be used to write additional lines to the output file. If several lines are to be written, it is more effective to use the data mode of Indirect Command Processor operation. As with other modes of operation, you can enable or disable this via directives. These are

```
.ENABLE DATA
.DISABLE DATA
```

Once the Indirect Command Processor has been put into data mode, all successive lines encountered in the command file are written to the output file. (If enabled, substitution is performed first.) This continues until either a **.DISABLE DATA** or a **.CLOSE** directive is found. This terminating statement must begin in the first column of its line to be recognized; otherwise, it too will be written to the output file.

Let's now put all this to practical use. The procedure we have considered thoughout this section is that of compiling, task building, and purging. Implicit in this has been the assumption that you can do the task building with the single-line form of the Task Builder command. If you must use the multiple-line command form, then the procedure we have been using will no longer suffice. Unfortunately, there is no way to put a multiple-line command into an indirect CLI command file. As we discussed in the previous section, the Indirect Command Processor expects each line to be a complete command by itself. The only way to get around this is to put the complete multiple-line command into an indirect task command file (remember, these are not available in DCL) and then run the task via the single-line command

```
task @file
```

Following this, you may delete the command file just created if you wish.

In Section 20.1 I gave an example of programs that need to use a tape drive. There, I put the tape specific commands into a file TAPE.CMD so that the task build could be done with two command lines:

```
file = file
@TAPE
```

Suppose we wish to incorporate this into an MCR command file that also compiles and purges. (Other than the use of a multiple-line Task Builder command, this will be conceptually identical to our earlier example COMPILE2.CMD.) We may do this via a command file such as:

```
.ENABLE SUBSTITUTION
.F77 'P1', 'P1' = 'P1'
.OPEN TEMP.CMD
.ENABLE DATA
'P1' = 'P1'
@TAPE
.CLOSE
FTB @TEMP
PIP TEMP.CMD;*/DE
PIP 'P1'.*/PU
```

If you call this file TAPECOMP.CMD and you use it on a source file called PROCESS1,

```
@TAPECOMP PROCESS1
```

the file TEMP.CMD will contain the two lines

```
PROCESS1 = PROCESS1
@TAPE
```

The FTB command will then cause the Fast Task Builder to take these two command lines from the file. Although the mechanism is clumsy, we have effectively passed a multiple-line command to the Task Builder from our command file. On the assumption that you do not need the temporary command file following the task build, it is then deleted. Note that **.OPEN** must precede **.ENABLE DATA** and that **.CLOSE** must precede any attempt to use the file.

In this section we have examined some of the more advanced features of the Indirect Command Processor. The directives we have discussed offer you the ability to write programs at the CLI level. As our examples have shown, this can take the monotony out of your more common interactions with RSX. The Indirect Command Processor, especially via its directives, offers you some truly powerful capabilities—you will have fun exploring them.

20.4 Log-In and Log-Out Command Files

In the previous sections, we have examined some of the more common uses of indirect command files, both at the individual task and at the CLI levels. Here, we will discuss two specialized uses of CLI command files. These are features of the log-in and log-out procedures and as such, apply only to RSX systems with multiuser protection.

As discussed in Section 10.1, you use the command **HELLO** in MCR or **LOG** in DCL to log in to your RSX system if it has multiuser protection. With this command you enter a UIC. After you have been logged in, RSX may display a log-in message on your terminal. Following this, you normally receive a prompt from your CLI. At this point, however, it is possible that a special indirect command file will be executed. Specifically, RSX searches your default directory (as determined by your log-in account) for a file with name LOGIN.CMD. If one is found, it is passed to the Indirect Command Processor. This mechanism enables you to have a CLI command file that is automatically executed whenever you log in to your system.

The purpose of such a file is to automate certain commands that you might wish to use when you log in. You would do this if you used the system in a nonstandard manner and needed to configure it in some special way. The most common example of this is the use of remote terminals. In Section 19.2 we saw that when you log in on a remote terminal the system has no way of knowing what type of terminal you are using and therefore assumes a default set of terminal characteristics. After logging in, you can use the Set commands discussed in Section 19.2 to change these characteristics so that they match the terminal you are using. Putting these Set commands into a LOGIN.CMD file will result in their being automatically done for you.

Let's assume that your remote terminal is a conventional video terminal. The terminal characteristics that you will probably want to set are your terminal type (e.g., VT100); the fact that the terminal is video, not hard-copy; a line length of 80 characters; and lowercase capability. You can do this with a command file consisting of the following four lines:

```
MCR: SET /VT100=TI:
     SET /CRT=TI:
     SET /BUF=TI:80.
     SET /LOWER=TI:

DCL: SET TERM/VT100
     SET TERM/SCOPE
     SET TERM/WIDTH:80
     SET TERM/LOWER
```

If you call this file LOGIN.CMD, it will automatically perform this reconfiguration whenever you log in to the system.

Use of this as a log-in command file would be appropriate if you always used the RSX system as a remote user (e.g., via a telephone link to a commercial timesharing organization). With the continued decline in hardware prices, it is becoming more and more common for individuals to own their own terminals and modems and to work at home. If you are fortunate enough to be in this position (it is by no means unrealistic) you will find yourself logging in at home as a remote user and at the office as a local user. When you log in as a local user, you should not need to change the default terminal characteristics because they should already be properly set. In other words, you might not want to execute this command file whenever you log in. As one solution, you could give the command file a name other than LOGIN.CMD and enter the command to execute it yourself only when you are on a remote terminal. A more elegant technique is to keep it as a log-in command file but to

include in it a test to determine whether your terminal is remote or local. You can do this by using the special logical symbol <LOCAL> (available on versions 4.0 and later of RSX-11M) which is true if your terminal is local and false if it is remote. Using the directives presented in the last section, you can write a log-in command file that executes the various Set commands only when you are on a remote terminal. A typical MCR example looks like this:

```
.IFT <LOCAL> .STOP
SET /VT100=TI:
SET /CRT=TI:
SET /BUF=TI:80.
SET /LOWER=TI:
```

If you make this your LOGIN.CMD file, this is what will happen when you log in on a remote terminal:

```
>@LOGIN.CMD
>SET /VT100=TI:
>SET /CRT=TI:
>SET /BUF=TI:80.
>SET /LOWER=TI:
>@ <EOF>
>
```

When you log in on a local terminal, however, this is what will happen:

```
>@LOGIN.CMD
>@ <EOF>
>
```

If you are using a remote terminal, you will find it useful to have a LOGIN.CMD file similar to the ones presented here. You may also wish to add some embellishments. For example, it is often nice to know who else is on the system, so you might add the Show Users command (Section 19.3) after the Set commands. If, however, you never log in to your system as a remote user, it is not likely that a LOGIN.CMD file will be of much use to you.

Corresponding to the execution of a LOGIN.CMD file when you log in to an RSX system is the execution of a LOGOUT.CMD file when you log out from the system. This feature is available only on versions 4.0 and later of RSX-11M and is a system generation option. If you wish to use a log-out command file, you should check to see whether this capability is available on your system or not.

The log-out command capability is generally less useful than the log-

in command capability. There are two reasons that you might have for using a log-out command file. One is to restore terminal characteristics after setting them to unusual values. When you use Set commands to set characteristics for your terminal, some of them stay set, even after you log out and even if you are not privileged. In that case, the terminal characteristics may not be properly set for the next user. An example of this is using APL on a DECwriter that is equipped with the special APL character set—this requires that you set the terminal as not having lower case capability. In all likelihood, the next person to use that terminal will expect it to support lowercase. You should restore this capability before you log out. Do this with a LOGOUT.CMD file consisting of the single line

```
MCR: SET /LOWER=TI:
DCL: SET TERM/LOWER
```

Similar examples involve the setting of a VT100 to be a VT52 emulator and the setting of a VT100 to display 132 characters per line.

The other reason you might have for using a log-out command file is to set up a procedure for performing various file maintenance activities. You might want to do certain things every time you use the system. Doing these via a log-out command file is convenient in that they will always be done every time you log out, whether you remember them or not. Activities of this sort include purging your files (we discuss this, along with the associated dangers, in Section 21.1) and changing the protection codes for newly created files (we discuss this in Section 14.5).

20.5 Portability Considerations

So far, our discussion of the various uses of indirect command files has been motivated solely by the consideration of making things easier for you. A second set of reasons centers on issues of portability. These might not seem to be as important to you, but in some cases, they can distinguish a truly professional product from a well-done but nonetheless amateurish effort.

By portability, I refer to the taking of computer programs written and tested on one machine and running them on another. Portability is normally thought of as an issue only when the two computers are of different types—for example, a FORTRAN program written for a PDP-11 might not work on an IBM-370. Portability can be an important issue in other cases, however, and should be considered even when both computers

involved are PDP-11s running under RSX. Our discussion of portability is based on two considerations: the documentation of required procedures and the possible variation in standard procedures between different sites.

First, let's briefly consider why you might want to run your programs on another computer. This depends on the type of work you do. Perhaps you use the computer purely as a tool for scientific analysis—you formulate a solution to a problem, write a simple FORTRAN program to evaluate it, and run it for a few cases. In this case, you would be unlikely to go beyond your own computer system, and our discussion here will not have much relevance for you. As another possibility, you may write programs that, after testing, are to be delivered to a customer so that he can run them on his computer. In this case, your responsibility includes not only making your programs work on your computer, but also ensuring that they will work on his. We will use this scenario as an example here.

Suppose now that your programs require a relatively complicated task build—e.g., as in our earlier examples, a certain logical unit number has to be assigned to the tape drive and certain subroutines in an object library are required. You cannot simply give your customer the source code and expect him to figure out how to make it work—at least not if you expect to get further work from him. The easiest and often the best way to document the required task build procedure is via a command file. This contains the required information in as succinct a form as is possible. The use of a command file offers a further advantage over a separate document. Since you will probably have already made the command file (for use during program development and testing), no further writing will be required on your part. Similarly, if you supply the command file on the same medium (tape or disk) as the source code, your customer will not have to re-create it, as it will already be there.

When you use a command file as a means of documenting procedures, it is natural to use the command file in the form you have been using on your system. Without realizing it, you may be relying on details that are specific to your system. Portability requires that you take a more general approach. Suppose, for example, that a lot of scientific processing is done at your facility so that most tasks use the floating point processor. Your system manager may then have installed a special version of the Task Builder in which the Floating Point switch is, by default, on. In your Task Builder command, you will probably not bother to specify that your task uses the FPP. If, however, your customer's facility uses the normal RSX default (Floating Point off), a Task Builder command without the Floating Point switch will not produce the correct task image

on his system. To avoid this problem, you should specify the Floating Point switch (**/FP** in MCR; **/CODE:FPP** in DCL) in your command file. Similar arguments may apply to other switches or options, during both compilation and task building, and also to device and ufd specifiers.

It may strike you as being redundant to specify parameters in your commands that you know are correctly specified by default. Although this is indeed true, you should remember two points. The most important is that these defaults may not apply on another system, so you should not rely on them. Further, although the effort is (to you) redundant, it is small, as you only need to type the switches and other specifications when you make the command file. The general principle here is that since it costs very little to put extra details into command files, you might as well do so.

21

USER AREA MANAGEMENT

When you work on your system, your immediate attention will probably be directed at only a few files. This is natural, as you will typically work on at most a few programs at a time. Nonetheless, you will most likely have many files that are still important to you, along with a slew of other files that you no longer need. In terms of working on the files that are of immediate interest to you, all these others can only get in the way. This leads to the need for user area management. Under this heading, we consider both the cleaning up and sorting of an individual user area as well as the use of multiple user areas.

21.1 Cleaning Up Your User Area

A basic fact of life (at least insofar as computer programming is concerned) is that things are never perfect the first time. Even for what is conceptually a very simple program, you will probably make several changes to your source file before you are done. These changes typically result in corresponding versions of object and task image files. You can quickly develop a large number of old versions of these files. This leads to what is probably the most common need for periodic cleaning up of your user area.

There are two reasons for cleaning up your area. The first is for your own direct benefit. In general, the more files you have in one area, the clumsier it is to do anything in that area. You have probably already seen this in terms of directory listings. The greater the number of files in your user area, the more difficult it is to read through a directory listing. You can partially alleviate this by getting a listing of only the latest versions of your files,

MCR: **PIP *.*/LI**

DCL: **DIR *.***

however, this merely masks the problem. Whenever you ask the operating system to do anything with a file, it has to search through your directory until it locates that file. The greater the number of files, the longer this search takes. Under RSX, especially when the system is heavily loaded, this difference can be surprisingly noticeable. By keeping your user area relatively clean, you not only get easier-to-read directory listings, but you also improve the response time for many system interactions.

The second reason for cleaning up your area is that all those files that you no longer need still occupy disk space, thereby preventing any other use of that space. The significance of this depends on how crowded your system is. If your computer system has more disk storage than it really needs, then you can afford to be sloppy. This is seldom the case. It is more likely that disk space will be at a premium, in which case it is your duty to try to use only what you really need. On systems where lack of disk space has become a serious problem, the decision will be taken out of your hands, as your system manager will conduct periodic disk cleanups. It is also possible that you (or, more specifically, the project you are working on) will be billed for total disk space utilized. This is yet another reason for you to get rid of files you do not need.

So much for motivations. How should you clean up your directory? Probably the most common method is via the Purge command (Section 14.2). Generally speaking, purging the files in your directory is a good habit to develop. In Chapter 20 we saw various examples of how to include purging in your command files. One warning is important: automatic purging of your object, task, list, and map (.OBJ, .TSK, .LST, and .MAP) files is virtually harmless. You should not, however, purge your source files until you are sure that your latest edits are good. Sooner or later you will mess up an edit. (It is not all that hard to do and you do not always realize it right away.) When this happens, it will be easier to delete the newest version (which you can always do by specifying **0** as the version number in the Delete command) and start again. If, however, you have already purged your source files, you will no longer be able to do this—deleting the latest version will be equivalent to deleting the only version, and you will be left with nothing. Thus, you should not purge after editing until you have checked the new version. Although I speak of this in terms of source files, it also applies to TXT files (used for documentation), CMD files, and to any other type of file that you

would make with an editor. As a final related caution, the editor EDI has two commands for exiting. **EX** offers a normal exit, but **ED** first deletes the previous version of the file that was just edited. You may be tempted to use **ED** because it saves you the effort of subsequently purging your source files. Sooner or later this will catch up with you—don't do it.

Although purging is the most common and normally the most useful means of cleaning up your directory, you cannot use it for everything. Purging merely deletes old versions of your files. If you decide that you no longer need (any version of) a particular file, you will have to use the Delete command (Section 14.2) to get rid of it. Suppose, for example, that you write a simple FORTRAN program to evaluate some complicated expression. (In essence, you are using the computer as a giant calculator.) After you have gotten your answer, you will have no further need of the TSK or OBJ files, although you may wish to retain the source file. Purging will leave you with a single copy of the FTN, OBJ, and TSK files; to get rid of the OBJ and TSK files, you must use the Delete command. Note that of these three files, the TSK file will often be considerably bigger than the others put together. (For a simple FORTRAN program, the source and object files might each be from 1 to 10 blocks long whereas the task image file will typically be at least 40 blocks long.)

There are other situations in which you can delete rather than just purge. Consider again a simple task made from only a main program. Even if you wish to keep the task for a while, there is no reason to keep the object file. If you change the main program, you will have to recompile it, producing a new version of the object file. All you really need to keep are the source and task image files. (If the task also includes subroutines from other files, you may wish to keep those object files because they might be useful for building other tasks. It is, however, very unlikely that the object file corresponding to the main program would ever be useful for more than one task.) In a similar vein, you can produce listing files when compiling or map files when task building. These are useful for debugging a program; once your program is running properly, they are often of no further use. At that point you should delete them. If for some reason you need them later, it is relatively painless to recreate them. The general principle here is that if you are unlikely to need a file in the immediate future and if it is not hard to recreate it, then it will most often be advantageous to delete it. The decision depends on the size of the file, the effort to make it, and how crowded the disk is. At one extreme, you may allow all your unnecessary files

to accumulate over a period of weeks or months; at the other extreme you may delete all your LST, MAP, OBJ and TSK files every day.

In Section 14.2 we discussed the use of the Purge and Delete commands. These are basic PIP commands (even if you access them via the DCL commands **PURGE** and **DELETE**). Several advanced features of PIP often simplify the procedure of deleting files. We discuss these in Chapter 22.

Another way to clean up your user area is not as dramatic as deleting files. This is making a backup copy of various files and then deleting them. Often you will have files that you are not using and that you probably will not use again but that you do not want to get rid of, just in case you do need them one day. By making a backup copy, you can save these files, but not at the expense of valuable disk space in your probably already crowded directory. Instead, you save the files on another medium—magtape, DECtape, or floppy disk. Once you have backed-up a file, you can delete it from your user area without really losing it (a situation somewhat analogous to the proverbial having your cake and eating it too). Various utilities are available for backup; we discuss these in Chapter 23.

Finally, another technique is worth noting. Although this does not reduce the number of files you have, it does reduce the amount of disk space they occupy. This is the Truncate command. Before discussing this command, it is necessary to explain a bit about how the Files-11 system works in RSX. When you create a file (via an edit, by writing data from a program, by compiling, or by any other means) an entry is made in the directory for your user area and an initial number of disk blocks are allocated for the file. As required, these blocks are used; when they are all full, another allocation is made, etc. It is quite likely that not all the disk blocks allocated to the file will be needed. When the file is closed, the excess blocks are not returned; they remain as an unused part of your file. If you get a full directory listing for the file, you will see two sizes shown: the number of blocks used and the number of blocks allocated. The difference between these two is the number of blocks that are being wasted. The Truncate command allows you to return these blocks to the system.

In the older versions of RSX, the Truncate command is available only in MCR. With the latest versions (version 4.2 of RSX-11M, version 3.0 of RSX-11M-PLUS, version 3.0 of Micro/RSX), it is also available in DCL. The form of the Truncate command is

MCR: **PIP file_specifier/TR**

DCL: **SET FILE/TRUNCATE file_specifier**

The truncation reduces the allocation for each file to the number of blocks actually used. Except in very unusual circumstances, there is never any danger in doing this. The simplest and most common way to use the Truncate command is on all your files,

MCR: **PIP *.*;*/TR**

DCL: **SET FILE/TRUNCATE *.*;***

The number of blocks given per allocation is a system generation option; five is a commonly used value. In this case, each file in your directory could have as many as four unused blocks in it. (Command files are particularly wasteful, as they normally are short enough to need only one block.) The summary line that appears at the end of the standard directory listing (this is also what you get in the summary form of directory listing) shows the total number of blocks allocated for all the files specified. If you look at this prior to truncating, you will know how many blocks you are returning to the system. Here is an example:

```
PIP>/TB
Storage used/allocated for Directory DR2:[351,7]
3-JAN-84 10:08
Total of 304./375. blocks in 29. files
PIP>*.*;*/TR
PIP>/TB
Storage used/allocated for Directory DR2:[351,7]
3-JAN-84 10:08
Total of 304./304. blocks in 29. files
```

By truncating all your files, you have made 71 blocks available. You might argue that this is a small number. Depending on how big your disk is, it may indeed appear to be small. Even big disks fill up eventually, and those few unused blocks may be more important than you think. You do not need them, so why keep them? Give them back—every little bit helps.

21.2 Sorting a Directory

To manage your user area, it is necessary to know what is in it. You can determine this by getting a directory listing. As you probably know by now, in RSX a directory listing has the files in no particular order. If you could sort the files in your directory first, you would then get a much easier to read list. This capability is offered by SRD (Sorted Directory Utility). SRD is not an official part of RSX—it is a utility supplied

by DECUS. If SRD is not available on your system, you should ask your system manager to get it, as it can be quite useful.

SRD offers several capabilities that both overlap and extend those of PIP for file management. For example, you can use both SRD and PIP to list or delete files. SRD allows the selection of files by a portion of the file name or by the date of file creation which, with earlier versions of RSX, was not possible with PIP. Improvements to PIP have made these features of SRD unimportant. I will limit our discussion of SRD to using it for sorting the files in a directory.

SRD commands follow the MCR style. Since SRD is not an official part of RSX, DCL does not include commands that translate into SRD commands. If you want to use the services provided by SRD, you will have to do so from the MCR environment. In all our examples, I assume that you are in MCR.

For sorting files, the basic SRD command is

```
SRD file_specifier/switches
```

The **file_specifier** determines the files that will be sorted. It can specify anything from a single file to a set of directories. For example,

```
*.MAC
```

specifies that you wish to sort all MACRO-11 source files in your area, and

```
DR2:[265,11]
```

specifies that you wish to sort all files in user area [265,11] on disk DR2:. Omitting the file specifier is equivalent to specifying all files in your default directory. Normally, you will want to sort your entire user area, and I will assume this in our examples.

The sorting done by SRD is alphabetical. The exact nature of the sorting is controlled by three switches. The alphabetizing is done for both file name and file type. The Name switch (**/NA**) determines which of these predominates. If you specify **/NA**, all files with the same file name will be grouped together. Different groups will be alphabetized by file name; within any group, the files will then be alphabetized by file type. If you specify **/-NA** (not by name), the reverse ordering will occur.

It is not always the case that one ordering will be preferable to another, hence the choice. If you omit the Name switch, the default will probably be **/-NA**, but this can vary with the installation. For purposes of cleaning up your area, sorting by file type (**/-NA**) is normally most useful. With

this ordering, you can easily determine how many files of each type you have in your area. This helps you decide whether you should delete all your LST files, all your OBJ files, etc. On the other hand, if you want to see how many files you have that are associated with a particular topic, it is more meaningful to sort your directory by file name.

The Writeback switch (**/WB**) determines whether the sorting is to be permanent or not. If you specify **/-WB** (no writeback), the sorting is temporary. You will be given a listing of your files in alphabetical order, but your directory itself will remain unchanged. To make the sorting permanent, you must ask for writeback. This means that SRD should write the alphabetized list of files back into your directory file. Following this, any reference to your directory (for instance, making a conventional directory listing) will access the files in the sorted order. In addition, the SRD writeback feature compresses your directory by removing gaps caused by file deletions. This increases the efficiency of operations that refer to the directory to find a file. If you do not specify this switch at all, the default is to not write back.

Finally, when you sort, you can also obtain a listing. This is controlled by the List switch (**/LI**). Since the major purpose of sorting your directory is to obtain an alphabetized directory listing, it might seem that you would always want to get a listing. This is not so. The listing produced by SRD is essentially the same as the brief form produced by PIP. SRD does not offer a listing equivalent to the normal PIP listing format. The difference here is that the PIP directory listing includes the date and time of file creation, which is often useful. Thus, you may want to use SRD (with writeback) to order your directory and then use PIP to get a directory listing. In this case, you would use the No List switch (**/-LI**) to suppress the SRD listing.

Let's now consider a simple example of how SRD works. Suppose the directory listing of your user area, as produced by PIP, is

```
ASSEMBLE.OBJ;3
SYMTABLE.MAC;2
COMMAND.MAC;2
ASSEMBLE.MAC;2
ASSEMBLE.TSK;3
COMMAND.OBJ;2
ASSEMBLE.MAC;3
SYMTABLE.OBJ;2
```

This seemingly haphazard order is caused by the fact that when you create a new file, its entry does not always go at the end of your directory. If there is an empty space in your directory due to the deletion of some

other file, the new entry will go there instead. Suppose you wish to sort this directory by file type and then get a listing. Depending on how much detail you wish to get in the listing, either the command

```
MCR: SRD /WB/-NA/LI
```

or the set of commands

```
MCR: SRD /WB/-NA/-LI
     PIP /LI
```

would be appropriate. In either case, the listing (as well as the directory itself due to the writeback) would show the files in this order:

```
ASSEMBLE.MAC;3
ASSEMBLE.MAC;2
COMMAND.MAC;2
SYMTABLE.MAC;2
ASSEMBLE.OBJ;3
COMMAND.OBJ;2
SYMTABLE.OBJ;2
ASSEMBLE.TSK;3
```

If you want to get a sorted listing such as this, but you do not want to change your directory, then you should use the command

```
MCR: SRD /-WB/-NA/LI
```

If you want to sort your directory by file name instead of file type, you should use either the command

```
MCR: SRD /WB/NA/LI
```

or the set of commands

```
MCR: SRD /WB/NA/-LI
     PIP /LI
```

The files will then be listed in this order:

```
ASSEMBLE.MAC;3
ASSEMBLE.MAC;2
ASSEMBLE.OBJ;3
ASSEMBLE.TSK;3
COMMAND.MAC;2
COMMAND.OBJ;2
SYMTABLE.MAC;2
SYMTABLE.OBJ;2
```

Note that whether the files are sorted by name or by type, multiple versions (as in ASSEMBLE.MAC above) will always appear together with the highest version number first.

As noted, SRD offers capabilities other than sorting a directory. These have been incorporated into PIP in the more recent versions of RSX. Rather than discussing them as part of SRD, which is not an official part of RSX, we will discuss their use via PIP in Chapter 22. If your system has an old version of RSX that does not offer these capabilities but does have SRD, then you should ask your system manager for a summary of the available SRD commands.

21.3 The Use of Multiple User Areas

So far, I have limited our discussion of user area management primarily to the cleaning up of an area by the permanent removal of unnecessary files. Sometimes you encounter a different problem. You may have many files in one area, all of which are still useful, but not all of which are interrelated. When the total number of files involved is large, this makes things hard to manage. The best way to avoid this problem is to have several user areas.

For example, suppose you are given a lengthy data analysis assignment. You write several programs and command files and make object and task image files by compiling and building. You have various input data files and output data files as well as some text files for documentation. Even after purging, you may still have a lot of different files. As this is a long-range assignment, you wish to keep all these files active. Once most of the work is completed, you are given an assignment to write a precompiler. The number of files involved in this effort soon becomes sizable also. Whenever you get a directory listing, files from the two projects are mixed together. Using SRD does not help this problem very much, as it can only sort alphabetically. To make life easier for yourself, you should have two separate user areas—one for the data analysis project and the other for the precompiler. In a sense, you are treating yourself as two different people; your identity as a user depends on what you are working on at a given time.

Although you are a single user, RSX allows you to have many user areas. Additional user areas are made by the Create Directory command, which is available only to privileged users. In our example, you should ask your system manager to give you two more user areas. You move all your data analysis files into one of your new areas and all your pre-

compiler files into the other. Any other files you may have had you leave in your original area. Now, things are nicely organized into separate areas and file maintenance is easier for you.

On a system with multiuser protection, when you log in, your entry in the system account file specifies the initial value of your default directory. This does not mean that you need a separate log-in account (with UIC) for each user area you have. On the contrary, this is wasteful and is normally avoided. You should need only one log-in account; after you use it to log in, you simply change your default directory as explained in Section 10.2.

This last point needs some clarification. As we discussed in Section 14.5, a ufd is a file and accordingly has an owner. For you to be able to work in a user area, you must have full access to the ufd for that area. When your system manager creates a numbered directory, Files-11 defines the owner of the new ufd to be the ufd itself. For example, suppose your UIC is [110,1] and you ask your system manager to create [110,2] as a new user area for you. The owner of this new ufd will be [110,2]. Note that there need not be an actual user whose UIC is [110,2]; this is merely a fiction used to determine the access rights to the ufd. Since your log-in account identifies you as UIC [110,1], you will not be the owner of the ufd for [110,2]. You will, however, be in the same group as the owner. Thus, you will be granted group access to the directory file. Since the file protection defaults for group users allow full access rights, you can use the new user area as though it were your own. Normally, you will be totally unaware of all this—as far as you are concerned, both user areas, [110,1] and [110,2], are yours, and when you use them, it all works. Underlying this is the fact that both areas have the same group number, which is what makes it work. Generalizing, you can have as many numbered directories as you want, as long as they are in the same group, and still need only one log-in account.

Things are not so obvious when you use named directories. When a named directory is created, there is no obvious way for Files-11 to define an owner for it. Thus, it uses the protection UIC of the user issuing the Create Directory command. Unless your system manager takes special precautions, he will be the owner of the new ufd, and you will be unable to use it. To avoid this, he should set his protection UIC to be your log-in UIC and then issue the Create Directory command. Your UIC will then be used to define the ownership of the named directory, and you will have no problems.

In addition to using the Set Default command (Section 10.2) to change your default directory, there is one other way to maneuver among a

variety of user areas. This is via the Default switch (**/DF**) in PIP. This change is temporary—it remains in effect only while you are in PIP. Nonetheless, it can be quite convenient.

You cannot (it does not make sense to) use the Default switch in PIP's single-line command form. Instead, you must first get into PIP and then use it in an interactive manner (see Section 11.2). Thus, you can use this technique only in MCR. To change your default directory within PIP, you can use a command of the form

```
PIP>[ufd]/DF
```

Any subsequent PIP commands will use **ufd** as the default for file specifiers, regardless of what your actual default directory is.

For example, suppose you are working in [110,1]. You have edited and tested a file of general purpose subroutines, creating an object file MATHSUBS.OBJ. You now wish to put a copy of this into [110,2], first deleting any previous versions that might be there. You intend to continue working in [110,1], hence you do not want to make more than a temporary change to your default directory. You can do all this with PIP as follows:

```
MCR>PIP
PIP>[110,2]/DF
PIP>MATHSUBS.OBJ;*/DE
PIP>=[110,1]MATHSUBS.OBJ
PIP>CTRL/Z
```

When you have several user areas, the Rename command (Section 14.3) is a convenient way to move files from one area to another. Functionally, it is equivalent to making copies in the second area and then deleting the originals in the first area, but it is much faster, since no actual copying is done. Following with our example, suppose you have just been given [110,2] and [110,3] as extra user areas. You wish to move all your MACRO-11 source files for the precompiler project from [110,1] into [110,3], and you also wish to move all your FORTRAN source files for the data analysis project into [110,2]. To do this, you use the following commands:

```
MCR:PIP  [110,3]/RE=[110,1]*.MAC;*
    PIP  [110,2]/RE=[110,1]*.FTN;*

DCL:REN  [110,1]*.MAC;*  [110,3]
    REN  [110,1]*.FTN;*  [110,2]
```

If all your MACRO-11 files are for the precompiler and all your FOR-

TRAN files are for data analysis, you are all set. Suppose that as part of the data analysis effort, you have a random number generator written in MACRO-11 contained in file RANDOM.MAC. This will have been moved, along with the other MACRO-11 source files, into the precompiler area. This poses no real problem, as you can move (rename) a file as many times as you like. Move it again, this time to where it belongs:

```
MCR: PIP  [110,2]/RE=[110,3]RANDOM.MAC;*
DCL: REN  [110,3]RANDOM.MAC;*  [110,2]
```

In the examples above, I have specified the ufds for both input and output. Although not always necessary, this is recommended. Suppose you are in user area [110,2] and you wish to move all the FTN files from [110,1]. If you try this command, it will not work:

```
MCR: PIP  /RE=[110,1]*.FTN;*
DCL: REN  [110,1]*.FTN;*  *.*
```

It is, however, a perfectly legal command, so no error message will appear, making you think that it did work. In the MCR example, the output file specifier is void. In DCL this cannot be done, since it is taken as a signal to prompt you for the output specifier. Thus, in the DCL example, I use a wildcard output file specifier. The PIP Rename command is special in the way it handles output specifiers. The output ufd, file name, file type, and/or version, if not explicitly given, or if a wildcard, will be set to the corresponding portion of the input specifier. This allows you to rename certain portions of the file specifier without affecting the others. When renaming a file, you must specify at least one portion of the output specifier for the command to make any sense. If the entire output specifier is void, then the entire input file specifier is simply used for the output. Thus, the command shown above indeed renames all the FORTRAN files in [110,1], but it renames them to their current names. This is perfectly valid but also perfectly useless. On the other hand, you can rely on your default directory being used for the input file specifier. If you are in user area [110,1] and you wish to move the FTN files into [110,2], the following command will work:

```
MCR: PIP  [110,2]/RE=*.FTN;*
DCL: REN  *.FTN;*  [110,2]
```

Rather than remembering that you can get away with omitting the ufd from the input but not the output specifier, it is safer to always include it in both.

So far we have discussed why you might want to have more than one user area and also how to work with files in different areas. One point remains to be considered. Given that you decide to use several areas, how should you choose the names of the various ufds?

If your system supports named directories, you can pick any name you want. It might be tempting to use your own name (for example, PIERCE), but this does not tell you anything about the contents of the user area, other than who you are, which you probably already know. Further, when you get another area, calling it something like PIERCE2 is even less meaningful. Instead, you should choose a name that relates to the files that will be in the area. This might be a project name or code, a customer name, or a generic term such as MATHSUBS. Subject to the nine character limitation, try to pick a useful name.

If you are limited to numbered directories, you will not have this flexibility. In defining a ufd, both the group and member numbers must be from 000 to 377 octal. You may have no control over your group number, but within that group, there is probably no reason why you cannot pick your own member numbers. Member numbers do not have to be in the order 1, 2, 3 and so on. Instead, you can (and should) pick numbers that are meaningful to you. For example, if you want another user area to segregate work for a particular project, and if your company has assigned that project a code number such as A265, you might as well use 265 as the member number for the new user area. (If the project code number is not a valid member number, such as 268 or 600, this will not work so nicely.) Alternatively, you might prefer member numbers of 10, 20, etc. Pick whatever is meaningful to you; you will be the one using it.

If you are using numbered directories, it is perfectly valid to have zero as a member number. This does, however, raise some interesting points. You cannot set your default directory to have zero as the member number. That is, RSX will reject a command such as

```
MCR: SET /UIC=[110,0]
DCL: SET DEFAULT [110,0]
```

You will be told that the UIC is invalid. This is not true; the UIC is perfectly valid, but RSX will not let you do this as it reserves group and member numbers of zero. If, however, you are in PIP, you can set your default directory to have a member number of zero:

```
PIP>[110,0]/DF
```

Unless you are in PIP and have used the Default switch, any use of

a file in a user area with member number zero can and must be effected by explicit specification of the ufd. This is clumsy enough to make zero a bad choice for a member number of a frequently used area. It does, however, offer an interesting option for protection of certain files. Since you cannot access files in an area with member number zero without specifying the ufd, the chances of accidental deletion are minimized. You may lose track of where you are when you switch your default directory around several times. In this case, a Delete command, especially with wildcards, can be disastrous. The restriction on setting the default directory protects files in a member number zero area from this type of accident, which is the main reason RSX processes group and member numbers of zero specially. If you have files that are often used but seldom changed, the choice of zero as the member number may be a good one. If your use of these files is typically via a command file, the requirement to specify the ufd is not onerous. Files that might fall into this category are object libraries of general purpose subroutines (used in task builds in other user areas) and master copies of files that are used often enough to warrant separate copies in other areas (e.g., command or data files). In general, you will probably not want a zero member number area until you have quite a few user areas.

In conclusion, when you first start working on an RSX system, one user area will probably be sufficient. In fact, the idea of having several user areas will strike you as unnecessarily confusing and inconvenient. As your use of the system grows and you accumulate more and more files, you will reach a point where this will no longer be the case. Do yourself a favor. Go to your system manager and get some extra user areas. Divide your files among them. You will soon get used to switching from one user area to another, and once you master this, you will find user area maintenance significantly easier.

22

Advanced Features of PIP

In Chapter 14, we discussed some commands that are useful for file maintenance. As noted, almost all of these involved using the utility PIP. In this section we discuss some additional features of PIP. Referring to these as being advanced features is somewhat misleading. They are not advanced in the sense that they are more difficult to learn. They are advanced in that they extend the basic capabilities of PIP discussed in Chapter 14 or they are implemented only under the later versions of RSX. These features are not necessary, but they will make life easier for you. If your system is very old, you may not be able to take advantage of them all, but the basic features that were presented earlier will still be applicable.

22.1 Overview

In general, a PIP command (whether entered directly in the MCR PIP format or via the DCL equivalent) specifies one or more files as well as what should be done with them. The advanced features we consider here offer a more flexible means of specifying files, or they offer modifications to the basic commands considered earlier that can be applied to the specified files. In general, these advanced features are most useful for identifying or deleting files and are thus directly related to the topic of user area maintenance, which we discussed in the previous chapter.

Let's consider what happens when you enter a typical file maintenance command. This might, for instance, be a command to delete, list, or copy files. Assume for simplicity that the files you want to work with are in your default directory. PIP first reads through your directory to

determine which, if any, files match the specifier that you entered in the command. By now, you should be relatively familiar with this concept. If, for example, the file specifier is **TEST.FTN**, PIP locates all files with file name TEST, file type FTN, and an arbitrary version number; it then selects the one with the highest version number. This is an example of a unique match—exactly one (unless there are none, in which case an error message is generated) file is selected as a match for your specifier. As another example, if the file specifier is ***.OBJ;***, all files with file type OBJ, regardless of file name or version number, are taken as matching your specifier. This is an example of a multiple match. Finally, if no specifier is given, `*.*;*` is assumed by default, in which case all files in your area are taken as matches.

Many of the advanced features of PIP that we discuss in this section modify the manner in which PIP looks through your directory for matches to your file specifier. Since these features do not depend on what is to be done with the files that are selected, you can use them in combination with any of the PIP action commands, including the basic ones previously discussed. In MCR, these features are effected by including additional switches in the PIP command. Similarly, in DCL they are effected by adding a command switch. In DCL, although the basic command names themselves vary, the switches used for these features do not.

In most of the rest of this book, I present topics by their function, not by their underlying implementation. For example, all of the activities studied in Chapter 14 relate to file maintenance. The fact that almost all of them are effected by the utility PIP is somewhat of a side issue and, if you are using DCL, is something that you might never realize. In the remainder of this section, we discuss special techniques that can be used only with PIP. This has an important interpretation for DCL—if the command you are using does not translate into a function performed by PIP, then you will not be able to use these advanced techniques with it. Jumping ahead a bit, one of these techniques allows you to restrict directory searches to those files created on the current day. By using this, you will be able to get a directory listing of only the files made today, since the listing is generated by PIP. You will not, however, be able to get a printout of all files made today, since the printing of files (via the Print command) does not use PIP. If you consider only the DCL command forms, you might wonder why some commands allow you more flexibility than others. Now you know.

To emphasize the restriction of these advanced features to the utility PIP, most of our discussion in the rest of this section will be in terms

of the MCR command form. For the benefit of the DCL user, here are the DCL commands that are effected by PIP:

COPY
APPEND
TYPE
DIRECTORY
DELETE
PURGE
RENAME
UNLOCK
SET PROTECTION

The techniques for modifying directory searches will apply to any of these commands; I will give an example or two in DCL, but there is no need to give an example for each of these commands. The other advanced techniques, those that modify a basic command action, will apply to only the corresponding command and will be noted as such.

22.2 Special PIP Wildcards

In Section 6.1, I discussed the use of wildcards in file specifiers. This technique applies not only to PIP but to many other utilities as well. With version 4.0 of RSX-11M, a powerful extension of the basic wildcard technique was added to PIP. This has not (at least yet) been implemented in other utilities. (This capability was already available in SRD, but as noted in Section 21.2, SRD is not an official part of RSX.) The special PIP wildcards are probably the most powerful and useful of all the advanced features of PIP.

You can use the special PIP wildcards in either the file name or file type portion of the file specifier; you cannot use them for the device, ufd, or version number. To explain how these work, we will first consider only the file name. The basic wildcard consists of a single asterisk (*) in place of an actual file name. When your directory is searched, any file name is taken as a match. More specifically, any string of characters (including none) is taken as a match for the asterisk. With special PIP wildcards, asterisks may be mixed with other characters; the above interpretation still applies to each asterisk. Thus, the file name specifier T* is equivalent to specifying "T followed by an arbitrary string (including none) of characters." If you want a directory listing of all files with a name that begins with T, you would use the command

```
MCR:PIP T*.*/LI
DCL:DIR T*.*
```

File names such as T, TT, and TRANSLATE are acceptable matches for this. Similarly, the file name specifier ***356*** is matched by any file name that has 356 somewhere in it, such as TEST356, TEST356xx, 356WORK, or even just 356. As a more thorough example, in Table 2, we show several possible file names and specifiers and indicate by the letter Y (for Yes) all matches.

Version 4.0 of RSX-11M also introduced a second wildcard character for PIP. This is the percent sign (%). Any single character is taken as a match for a %. This is in distinction to the asterisk just discussed, which is matched by any string of characters. Thus, if you specify T%, the file name T1 will be selected as a match but T11 will not, whereas both names will match T*. As another example, %%% specifies any file name that is exactly three characters long. In general, the percent sign will be less useful than the asterisk wildcard, but you may have occasion to use it.

You can also use the asterisk and percent sign wildcards in specifying the file type, although this is much less likely to be useful. Generally, file types are chosen to match defaults for certain utilities, not to follow naming patterns that you have set up. It is more often coincidence than intent that enables the use of these special wildcards with file types.

Although the special wildcards of PIP are not implemented for other utilities, you can effectively use them with other utilities if you are willing to go through a few extra steps. For example, in Chapter 23 we will examine techniques for making backups of your files. Without going into details at this time, you specify a set of files and a device on which you want a copy made. Following the backup, you may also want to

Table 2
Matches between File Names and Specifiers

File Name	*File Name Specifier*				
	*T**	**T**	**T*11*	**11*	**11**
TEST	Y	Y			
TEST11	Y	Y	Y	Y	Y
TEST12	Y	Y			
BEST11		Y	Y	Y	Y
BEST12		Y			
WORK11				Y	Y
TEST11A	Y	Y			Y
WORK11A					Y

delete the files from disk. Suppose that you wish to make a backup copy of all files (source, object, listing, etc.) whose names begin with the project title RED. Since this includes names such as RED and REDPART1, you cannot use a simple file specifier such as **RED.*;***. Instead, the more sophisticated file specifier **RED*.*;*** would be ideal; however, most backup utilities will not accept this. Suppose that all these files are in user area [217,2] and that you have another user area, [217,3], which currently has no files in it. (If you do not have another area available, get one! See Section 21.3.)You can then use the Rename command with special wildcards to move the desired files into the spare area. Because there will be no other files in the spare area, you may effect the backup from that area, using a conventional ***.*;*** file specifier. Following the backup, you can either move the files back or delete them. By using the Rename command, you do not really have to move the files, so almost no system overhead is involved. This is how the backup goes:

```
MCR: PIP [217,3]/RE=[217,2]RED*.*;*
     backup [217,3]*.*;*
DCL: REN [217,2]RED*.*;* [217,3]
     backup [217,3]*.*;*
```

(I use **backup** to denote an arbitrary backup utility.) To clean up afterwards, use either

```
MCR: PIP [217,2]/RE=[217,3]*.*;*
DCL: REN [217,3]*.*;* [217,2]
```

or

```
MCR: PIP [217,3]*.*;*/DE
DCL: DEL [217,3]*.*;*
```

You can use similar tricks with the directory search modification commands discussed in the following sections.

22.3 Directory Search Modifiers

The special PIP wildcards allow you to create special file specifiers. Several other PIP commands modify the way that PIP searches your directory for files that match a specifier. Unlike the wildcards, these are not used within the file specifier. Instead, they are separate commands that force a modified interpretation of the file specifier. You can use these directory

search modification commands with the regular PIP action commands, in either the single- or multiple-line command form.

Three commands are available for modifying the means whereby PIP searches your directory for file specifier matches. These are the Today, Default Date, and Exclude commands, which we discuss in the next three sections. These commands all work in the same basic manner. Each specifies a modification (i.e., a restriction) on file matching. If you use a single-line command, the modification affects only that command. If you enter PIP and use a multiple-line command, the modification remains in effect from the time it is entered either until it is overridden by another similar command or until you exit from PIP.

There are some important syntactical points concerning the use of the directory search modification commands. In the next section we will discuss the Today command. We use this as an example here; the concepts that we discuss concerning its use apply to the other directory search modification commands as well. As you might guess, the Today command directs PIP to consider only files created on the current date. In MCR the Today command is effected by the switch **/TD**; in DCL the command switch **/TODAY** is used.

If you are in MCR, the easiest way to use the directory search modification commands is in the multiple-line command form. To do this, you enter PIP, give the desired directory search modification command(s), and then give the appropriate action command. For example, if you enter PIP, give the Today command, and then ask for a directory listing of all files, the resulting listing will show only those files created on the current day. The proper command sequence is

```
MCR>PIP
PIP>/TD
PIP>/LI
```

Since PIP processes command input one line at a time when you use it in this interactive manner, you must enter the Today command before the Directory command. Note that I do not show your leaving PIP (via a **CTRL/Z**). If you remain in PIP, the Today command will remain in effect for subsequent commands. Depending on what you are doing, this may or may not be desired.

In MCR, you can also do the above in a single-line command. The syntax for doing this is perhaps tied with that of the Set command (Section 19.2) for ungainliness. First, you must realize that the directory search modification commands are separate, self-standing commands. They are not subcommands to the basic action commands that we discussed in Chapter 14. You enter a directory search modification com-

mand directly after the PIP command itself. (This is similar to the distinction that DCL makes between a command switch and a file switch and is one of the few instances in MCR syntax where a switch must be appended to the command itself.) You follow the command with an ampersand (&) which is recognized by PIP as a command separator. This allows you to enter more than one command in a single command line. Finally, you enter the basic PIP command (which is, in general, a file specifier followed by the command switch). Thus, the above example, in a single-line command to PIP, is

```
MCR>PIP /TD&/LI
```

Note that if you omit the ampersand

```
PIP /TD/LI
```

you will get the error message

```
PIP -- Too many command switches - ambiguous
```

(The PIP manual for RSX-11M version 4.0/4.1 shows examples in this combined form but gives the wrong syntax; it does not include the ampersand.) Now, if you interchange the two switches,

```
PIP /LI&/TD
```

you will get no error message, but the Today switch will be ignored. What happens here is that the Directory command is processed first (it accordingly considers every file in your user area), and the Today command is processed only after the directory listing is complete.

As bad as it is, this example is deceptively simple, since there is no file specifier. Suppose you want a directory of only the MACRO-11 source files created today. The file specifier (***.MAC;***) is part of the Directory command, so it (and the switch **/LI**) must be entered after the ampersand,

```
MCR>PIP /TD&*.MAC;*/LI
```

It is much more natural to enter the command as

```
PIP *.MAC;*/TD/LI
```

This will not work—it is a more general case of the mistake noted above. You will again get the error message

```
PIP -- Too many command switches - ambiguous
```

Even if you put an ampersand after the Today switch (**/TD**), this form with both switches at the end of the command is still wrong. The command

```
PIP *.MAC;*/TD&/LI
```

results in the error message

```
PIP -- Command syntax error
```

followed by a directory listing that ignores the Today switch. As you can see, the single-line command form is quite a bit more difficult to remember than the multiple-line form. Normally, a single-line command, once you know it, is faster to use. Here, you will probably find the multiple-line command faster to use because you will probably be able to at least get it correct. Perhaps the only time that the single-line form will be advantageous is in an indirect MCR command file where you cannot use multiple-line command forms.

If you are in DCL, you are restricted to a single-line command form. Here, however, things are easy and sensible. You can put the directory search modification command switch after the DCL command name, the way that you would expect. Thus, in DCL, our last example is

```
DCL>DIR/TODAY *.MAC;*
```

As a final point, you can combine several directory search modification commands. (It does not make sense to combine the Today and the Default Date commands, since one overrides the other. It does make sense to combine either of these with the Exclude command.) In the multiple-line MCR command form, each of these is a separate command and is entered on a separate line. In the single-line MCR command form, each is entered as a switch appended to the command **PIP** and must be followed by an ampersand. In DCL each is entered as a command switch appended to the appropriate DCL command.

22.4. The Today Command

The first directory search modification command that we consider is the Today command. It is the easiest of the three to use; the basic mechanisms illustrated in our examples apply to the other commands as well. The Today command is available with version 4.0 of RSX-11M. There are no parameters or values associated with this command. In MCR it is simply entered in the PIP command as the switch **/TD**. In DCL it is entered as the command switch **/TODAY**.

The Today command specifies that only those files created on the current date will be eligible for consideration in matching your file specifier. For example, suppose that you first get a normal (without any constraints) directory listing:

```
Directory DR2:[265,5]
16-DEC-85 11:27

JUNK.FTN;5              2.      15-SEP-85 10:00
RANDOMBIT.MAC;4         5.      02-DEC-85 16:38
RANDOM.MAC;3            2.      16-DEC-85 09:39
RANDOM.OBJ;3            2.      16-DEC-85 09:43
TRAND.MAC;5             2.      16-DEC-85 10:22
TRAND.OBJ;5             2.      16-DEC-85 10:24
RANDOM.MAC;2            2.      15-DEC-85 17:44
RANDOMBIT.OBJ;2         1.      02-DEC-85 10:07
A.OBJ;1                 1.      16-NOV-85 10:03
RANDOM.OBJ;2            1.      15-DEC-85 17:44

Total of 20./20. blocks in 10. files
```

Suppose that you have made several changes today and you would like a listing of only those files made today. You can do this with the command

MCR: **PIP /TD&/LI**

DCL: **DIR/TODAY**

This is what you get:

```
Directory DR2:[265,5]
16-DEC-85 11:27
Day of 16-DEC-85

RANDOM.MAC;3        2.      16-DEC-85 09:39
RANDOM.OBJ;3        2.      16-DEC-85 09:43
TRAND.MAC;5         2.      16-DEC-85 10:22
TRAND.OBJ;5         2.      16-DEC-85 10:24

Total of 8./8. blocks in 4. files
```

Note that the heading in the directory listing (normally two lines long) now contains a third line ("Day of..."). This is to document the fact that the directory listing was prepared with the Today command in effect.

Continuing with this example, suppose you wish to delete all object files created today. You can follow the Directory command with a second command to delete these files,

MCR: **PIP /TD&*.OBJ;*/DE**

DCL: **DELETE/TODAY *.OBJ;***

Alternatively, if you are in MCR, you can enter PIP and interact with it. In this case, you should first issue the Today command by itself, since it will affect all subsequent commands while you remain in PIP. Your sequence of commands will be

```
MCR>PIP
PIP>/TD
PIP>/LI
PIP>*.OBJ;*/DE
```

Note that the file A.OBJ;1 was created on a day other than today and thus will not be affected by the Delete command. If you are using PIP interactively and you want to assure yourself that this file is still there, you cannot do so by entering the command

```
PIP>/LI
```

Since the Today command is still in effect, and since A.OBJ was not created today, it will not be taken as a match. (This is a common source of confusion.) At this point, a directory listing will show this:

```
Directory DR2:[265,5]
16-DEC-85 11:28
Day of 16-DEC-85

RANDOM.MAC;3       2.      16-DEC-85 09:39
TRAND.MAC;5        2.      16-DEC-85 10:22

Total of 4./4. blocks in 2. files
```

To see that the file A.OBJ;1 is still in your directory, you have to cancel the effect of the Today switch. You can do this by leaving PIP (**CTRL/Z**), returning to MCR, and then reentering PIP, or by staying in PIP but cancelling the Today command via the switch **/DD** as explained in the next section. If you have been entering individual single line commands (which is what you have to do from DCL), you do not have to worry about this, since the Today command does not have a lasting effect.

Finally, suppose you decide to delete any FTN files created today. You can do this with the command

```
MCR:PIP /TD&*.FTN;*/DE
DCL:DELETE/TODAY *.FTN;*
```

In this case, you will get the response

```
PIP -- No such file(s)
SYO:[265,5]*.FTN;*
```

which simply means that although you may have some FTN files in your user area, you have none that were created today. (Note that, unlike the directory listing, no reminder is typed stating that the Today command is in effect. Thus, this message is by itself misleading.)

The Today command is often valuable when you have a working version of a program and make a series of changes to it. Suppose, for example, that you have an old program called PRE.MAC. You decide to add some extra features to it. You edit the file, which (inevitably) results in some errors; these are corrected via another edit, etc. After several edits, a directory of all your PRE.MAC files looks like this:

```
MCR>PIP PRE.MAC;*/LI


Directory DR2:[265,6]
16-DEC-85 11:29

PRE.MAC;3       2.      16-DEC-85 09:43
PRE.MAC;4       2.      16-DEC-85 09:53
PRE.MAC;2       2.      16-DEC-85 09:39
PRE.MAC;5       2       16-DEC-85 10:22
PRE.MAC;1       1.      27-MAY-85 15:19

Total of 9./9. blocks in 5. files
```

You would like to keep only the most recent of the new versions, but since you have not finished testing the enhancements, you do not wish to get rid of the old version. A simple purge will not do what you want, but you can purge today's files:

```
MCR:PIP /TD&PRE.MAC/PU
DCL:PURGE/TODAY PRE.MAC
```

The Today command excludes PRE.MAC;1 from consideration for file matching, so it will not be affected by the Purge command. Note that by using a single line command, the Today command does not affect subsequent PIP commands. If you then get a directory listing, it will show only the two files that you wanted to keep:

```
MCR>PIP PRE.MAC;*/LI


Directory DR2:[265,6]

PRE.MAC;5       2.      16-DEC-85 10:22
PRE.MAC;1       1.      27-MAY-85 15:19

Total of 3./3. blocks in 2. files
```

22.5 The Default Date Command

The second directory search modification command is the Default Date command. (This is available with version 4.0 of RSX-11M.) Conceptually this is very similar to the Today command, but it offers a more generalized capability. The Default Date command specifies a beginning and an end date. When the Default Date command has been specified, only files created between these two dates are eligible for matching file specifiers. The dates are inclusive; a file must have been created on or after the beginning date and on or before the end date. By setting the beginning and end dates the same, you can specify one particular day. By setting both dates equal to the current date, you obtain exactly the same action as you would with the Today command. (The Today command, however, is easier to use.) When you are in an interactive session with PIP, you can specify the Default Date command as often as you like. Each time you do, you change the start and end dates.

In MCR the Default Date command is entered by using the switch **/DD**. The form of this is

```
/DD:startdate:enddate
```

The colons preceding **startdate** and **enddate** are required. The dates are both entered in the standard RSX date format:

```
dd-mmm-yy
```

The hyphens also are required. **dd** is a two-digit day number (you can omit the leading zero if the day is between 01 and 09); **mmm** is a three-letter month abbreviation (the first three letters of the month name in English); and **yy** is a two-digit year value. As an alternative, you can use a wildcard date; this is entered as an asterisk (*) and specifies that the corresponding limit (start or end) is to be ignored.

In DCL, two separate commands correspond to the two date limits allowed in the Default Date command. These are the Since and Through commands, which are entered as command switches in the form

```
/SINCE:startdate
/THROUGH:enddate
```

where **startdate** and **enddate** have the same meaning and are specified in exactly the same manner as in the MCR command. You can specify both these commands; the switches are simply appended one after the other to the basic DCL command. When you wish to specify just one day, you can use the special form

rather than entering both the Since and Through commands with the same date.

For example, suppose you want a directory listing of those files created during the months of January, February, or March of 1985. If you are in MCR, you can enter PIP and then restrict all further directory searches to this time period by entering the command

```
PIP>/DD:01-JAN-85:31-MAR-85
```

Following this, a conventional Directory command to PIP will give you what you want. Alternatively, you can use a single line command

```
MCR:PIP /DD:01-JAN-85:31-MAR-85&/LI
DCL:DIR/SIN:01-JAN-85/THR:31-MAR-85
```

If you want all files created prior to 1985, you specify December 31, 1984, as the end date. In MCR you use a wildcard for the start date; in DCL you simply omit the Since command:

```
MCR:PIP /DD:*:31-DEC-84& /LI
DCL:DIR/THR:31-DEC-84
```

Similarly, to copy only those files created on or after November 2, 1985, from your user area to the floppy disk drive DY1: (this is a form of backup that we will examine below in Chapter 23), you would use the command

```
MCR:PIP /DD:02-NOV-85:*&DY1:=*.*
DCL:COPY/SINCE:02-NOV-85 *.* DY1:
```

If you use PIP interactively, you can enter the Default Date and the Today commands as often as you wish. (Remember that the Today command is merely a special case of the Default Date command.) Entering one of these commands cancels any previous ones; only the last one entered is significant. In particular, you may use the command **/DD:*:*** to declare that you do not care at all about the creation date of the file. This condition is in effect when you first enter PIP, so there is no need to use the command **/DD:*:*** unless you specify date restrictions and then wish to cancel them.

The way in which the Default Date command modifies further PIP commands is very similar to that already discussed for the Today command. Other than the extra flexibility, the only point worth noting is

the identification of the default dates on directory listings. If you specify actual values (no wildcards) for both the start and end dates, the directory heading will have a third line of the form

```
Dates from startdate through enddate
```

where startdate and enddate are the values you specified. If you use a wildcard for the startdate (or in DCL omit the Since switch), this line will instead be of the form

```
Dates before enddate
```

Similarly, if you use a wildcard for the enddate (or in DCL omit the Through switch), the line will be of the form

```
Dates after startdate
```

Finally, if you restrict the directory search to a single day, either by setting the startdate equal to the enddate or, in DCL, by using the Date switch, the third line of the directory heading will be of the form

```
Day of date
```

We have already seen a special case of this form with the Today command.

22.6 The Exclude Command

The third command available for modifying directory searches is the Exclude command. (This is available with version 4.0 of RSX-11M.) This command identifies certain file specifiers that are to be excluded from directory searches. In MCR the form of the Exclude command is

```
file_specifier/EX
```

In DCL it is

```
/EX:file_specifier
```

The **file_specifier** determines which files are to be excluded; it cannot include device or ufd specifiers, but it can include any combination of file name, file type and version.

The file specifier is interpreted exactly as in other PIP commands, with the exception of the version number. The version number cannot

be omitted; if it is, the PIP command will be rejected, resulting in the error message

```
PIP -- Version must be explicit or "*"
```

Furthermore, you cannot use a version number of zero to exclude the latest version of a file. To exclude the latest version of a file, you must determine (via a directory listing) what that version number is and then specify it in the Exclude command. Finally, you can use a wildcard for the version number, which causes all versions of the file to be excluded.

The treatment of the version number in the Exclude command is somewhat nonstandard but the file name and file type portions of the file specifier are treated normally. For example, if you are using PIP interactively, the command

```
PIP>*.FTN;*/EX
```

specifies that all files of type .FTN be excluded, regardless of file name or version. Similarly, you can use the command

```
PIP>TEST.*;*/EX
```

to exclude all files with name TEST. The Exclude command and the special PIP wildcards were both introduced in version 4.0 of RSX-11M. Thus, if your system has one of these features, it has both. With these features, you can use commands such as

```
PIP>*TEST*.*;*/EX
```

which directs that all files with TEST appearing anywhere in the file name are to be excluded.

Let's now consider a simple example. In our discussion of user area maintenance, I explained that you might not want to purge source files when cleaning up your area. The Exclude command can help you here. Suppose your directory looks like this:

```
MCR>PIP/LI

Directory DR2:[265,6]
30-DEC-85 14:57

MAKEFILES.FTN;4      4.          14-DEC-85 16:37
MAKEFILES.FTN;2      2.          09-NOV-85 17:14
MAKEFILES.LST;4      7.          14-DEC-85 17:03
MAKEFILES.LST;3      6.          13-DEC-85 15:52
```

```
MAKEFILES.OBJ;4      3.          14-DEC-85 17:04
MAKEFILES.OBJ;3      2.          13-DEC-85 16:08
MAKEFILES.TSK;2      42.      C 30-DEC-85 09:44

Total of 66./66. blocks in 7. files
```

You would like to purge everything except your FORTRAN source files. You can do so by excluding these files and then purging:

```
MCR:PIP *.FTN;*/EX&*.*/PU
DCL:PURGE/EX:*.FTN;* *.*
```

You can also do this interactively:

```
MCR>PIP
PIP>*.FTN;*/EX
PIP>*.*/PU
PIP>/LI


Directory DR2:[265,6]
30-DEC-85 14:58
*.FTN;* excluded

MAKEFILES.LST;4      7.          14-DEC-85 17:03
MAKEFILES.OBJ;4      3.          14-DEC-85 17:04
MAKEFILES.TSK;2      42.      C 30-DEC-85 09:44

Total of 52./52. blocks in 3. files
```

In this example, after purging, we get a directory listing. Because the Exclude command is still in effect, no .FTN files appear in the listing. Note also that a third line appears in the heading stating that *.FTN;* is excluded.

When you are using PIP interactively, you can use the Exclude command repeatedly. Each time you use it, the previous exclusion is dropped, and the new file specifier exclusion becomes effective. A special form of the Exclude command in which no file specifier is given,

```
PIP>/EX
```

directs that no files are to be excluded. You can use this to cancel any previous exclusions, returning to the basic PIP directory search condition. In our example above, if you want to convince yourself that all original versions of your FORTRAN source files remain after the purge, you would first cancel the current exclusion and then get a directory listing:

```
PIP>/EX
PIP>/LI

Directory DR2:[265,6]
30-DEC-85 14:59

MAKEFILES.FTN;4      4.          14-DEC-85 16:37
MAKEFILES.FTN;2      2.          09-NOV-85 17:14
MAKEFILES.LST;4      7.          14-DEC-85 17:03
MAKEFILES.OBJ;4      3.          14-DEC-85 17:04
MAKEFILES.TSK;2      42.      C  30-DEC-85 09:44

Total of 58./58. blocks in 5. files
```

Because the Default Date (or Today) command and the Exclude command restrict directory searches in different ways, they can be used together. Let's return to our example above. Suppose you want to know which files, other than FORTRAN source files, are more than a week old. You can do this by excluding .FTN files, setting the end date, and getting a directory listing. Let's first look at the interactive form of doing this:

```
MCR>PIP
PIP>*.FTN;*/EX
PIP>/DD:*:23-DEC-85
PIP>/LI

Directory DR2:[265,6]
30-DEC-85 15:03
*.FTN;* excluded
Dates before 23-DEC-85

MAKEFILES.LST;4      7.          14-DEC-85 17:03
MAKEFILES.OBJ;4      3.          14-DEC-85 17:04

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

You can enter the Exclude and the Default Date (or Today) commands in any order. When you use both and get a directory listing, the file specifier exclusion is indicated in the third line of the header and the date restriction appears as the fourth line.

You could also use a single-line command in the above example. In MCR, the command line becomes incredibly awkward; in DCL, it is not too bad:

```
MCR: PIP *.FTN;*/EX&/DD:*:23-DEC-85&/LI
DCL: DIR/EX:*.FTN;*/THR:23-DEC-85
```

From these examples, you might conclude that the general forms of MCR Exclude and Default Date commands differ. It is natural to think of either command as specifying an action, with the remainder of the command line supplying details of said action. If you think of them this way, the Exclude command looks like "what files" followed by the switch **/EX**, whereas the Default Date command looks like the switch **/DD** followed by "what dates." This lack of parallel structure is an unfortunate source of confusion. Actually, both commands follow the same basic syntax that is used for all PIP commands:

```
PIP>file_specifier/switch:parameter(s)
```

With the Exclude switch, the "what files" is the file specifier and comes first, whereas with the Default Date switch the "what dates" is a parameter value and comes last. Try to not let it confuse you.

The Exclude command is useful, but it suffers from an important limitation. Exclusions cannot be combined; only a single file specifier can be excluded. For example, you cannot exclude both FORTRAN and MACRO-11 source files. If you try this

```
PIP>*.FTN;*/EX
PIP>*.MAC;*/EX
```

the second exclusion will override the first, and only the MACRO-11 files will be excluded. Similarly, there is no way that you can simultaneously exclude all files with TEST in the file name (*TEST*.*;*) and all files of type MAC (*.MAC;*). The command

```
MCR: PIP *TEST*.MAC;*/EX&*.*/PU
DCL: PURGE/EX:*TEST*.MAC;* *.*
```

excludes only those files that have TEST in the file name and that also are of type MAC. Thus, files such as TEST2.FTN and SCREEN.MAC will not be excluded from the purge. In many instances where excluding files would be useful, you will find that you need to exclude combinations that cannot be cited in a single file specifier; the Exclude command will then be of no use. Don't dismiss it entirely, however, as there will still be times when the Exclude command will be helpful to you.

22.7 The Selective Delete Command

The basic means of deleting files with PIP is via the Delete command. There is also a variant of this known as the Selective Delete command. The difference between these two commands is this. When you use the conventional Delete command, any file that matches the file specifier is immediately deleted, and that is that. When you use the Selective Delete command, each file that matches the file specifier results in a query; based on your response, the file is either deleted or left alone. There is no sense in using the Selective Delete command when you explicitly name files, but it is very useful when you use wildcards. By doing a wildcard selective deletion, you get PIP to list each file that is eligible for deletion and to then give you a chance to not delete it.

After naming a file, PIP gives you four choices:

Y = Yes
N = No
G = Go
Q = Quit

Your response must be one of these four. A **Y** means "Yes, delete the file just named." Similarly, an **N** means "No, do not delete it." In either case, after the appropriate action, the name of the next file eligible for deletion will be listed followed by a similar prompt. The other two responses that you might enter are different in that they terminate the interactive process. A **G** means "Go; delete this file and also all remaining ones without listing their names." A **Q** means "Quit; do not delete this file and do not delete or ask me about any others."

In MCR, the Selective Delete command is a syntactically identical functional replacement for the conventional Delete command. It is effected by using the switch **/SD** rather than the switch **/DE**; otherwise, the commands are identical. In DCL, the Selective Delete command is effected by adding the switch **/QUERY** (this is typically shortened to just **/Q**) to the normal Delete command. Thus, the basic form of the Selective Delete command is

MCR: **PIP file_specifier/SD**

DCL: **DEL/QUERY file_specifier**

With the new versions of RSX (version 4.2 of RSX-11M, version 3.0 of RSX-11M-PLUS and version 3.0 of Micro/RSX), the DCL has been mod-

ified to allow the switch **/CONFIRM** as a synonym for the Query switch. This is now the preferred usage as it is the same as that used in DCL on the VAX.

As an example of using the Selective Delete command, suppose you decide that it is time to get rid of some of the object files in your user area. You first get a brief directory listing via the command

```
MCR:PIP *.OBJ;*/BR
DCL:DIR/BR *.OBJ;*
```

so that you can see what there is. The listing that you get looks like this:

```
Directory DR2:[265,6]

A.OBJ;1
MAIN.OBJ;1
SUBS1.OBJ;1
TAPESUBS.OBJ;1
SUBS2.OBJ;1
MAINVERS3.OBJ;1
TEST.OBJ;1
QFUNCTION.OBJ;1
MAINVERS7.OBJ;1
```

You decide that you no longer need the object files named A, TEST, MAIN, and MAINVERS3. You could delete these by individually typing their names and using the conventional Delete command, but this seems to be too laborious. (After all, computers are supposed to do the work, not you.) Instead, you decide to do use the Selective Delete command, which avoids the need for typing the individual file names. In MCR this is what happens:

```
MCR>PIP *.OBJ;*/SD
Delete file     DR2:[265,6]A.OBJ;1              [Y/N/G/Q]? Y
Delete file     DR2:[265,6]MAIN.OBJ;1           [Y/N/G/Q]? Y
Delete file     DR2:[265,6]SUBS1.OBJ;1          [Y/N/G/Q]? N
Delete file     DR2:[265,6]TAPESUBS.OBJ;1       [Y/N/G/Q]? N
Delete file     DR2:[265,6]SUBS2.OBJ;1          [Y/N/G/Q]? N
Delete file     DR2:[265,6]MAINVERS3.OBJ;1      [Y/N/G/Q]? Y
Delete file     DR2:[265,6]TEST.OBJ;1           [Y/N/G/Q]? Y
Delete file     DR2:[265,6]QFUNCTION.OBJ;1      [Y/N/G/Q]? Q
MCR>
```

(In DCL exactly the same thing happens except that the command you

enter is **DEL/Q *.OBJ;*** or **DEL/CON *.OBJ;***.) A brief directory listing now shows that only the other object files remain:

```
MCR>PIP *.OBJ;*/BR


Directory DR2:[265,6]

SUBS1.OBJ;1
TAPESUBS.OBJ;1
SUBS2.OBJ;1
QFUNCTION.OBJ;1
MAINVERS7.OBJ;1
```

Of course, you do not have to examine your directory before using the Selective Delete command. Even if you do not remember in advance each file that you have, you will normally know by looking at the file name whether you want to keep it or not. The one danger here is that if you have multiple versions, you may inadvertently delete the latest version if it is listed first. If you have multiple versions, you should first purge and then selectively delete, or you should get a directory listing so that you know exactly which files to delete.

You can use the Selective Delete command in conjunction with the other advanced features that we have already examined. For example, suppose you have been editing, compiling, and building several portions of a new video game called Wombat. You do not wish to disturb any of the MACRO-11 source files, but you do wish to clean up some of the others that have accumulated. You do this by combining the advanced PIP wildcards, the Exclude switch and the Selective Delete command:

```
MCR>PIP
PIP>*.MAC;*/EX
PIP>*WMB*.*/PU
PIP>*WMB*.*;*/SD
Delete file        DR1:[107,3]WMBSUBS.LST;3        [Y/N/G/Q]? Y
Delete file        DR1:[107,3]WMBMAIN.OBJ;6        [Y/N/G/Q]? N
Delete file        DR1:[107,3]WMBSUBS.OBJ;3        [Y/N/G/Q]? Y
Delete file        DR1:[107,3]WMB.CMD;2            [Y/N/G/Q]? N
Delete file        DR1:[107,3]WMBMAIN.LST;6        [Y/N/G/Q]? N
Delete file        DR1:[107,3]WMB.TSK;3            [Y/N/G/Q]? Y
Delete file        DR1:[107,3]WMB.MAP;3            [Y/N/G/Q]? Y
PIP>^Z
```

Note that in this example it is preferable to use the multiple-line PIP command form since you want the file exclusion to apply to both the Purge and the Selective Delete commands. It is, of course, possible to

do this with two single-line commands, but you will have to include the Exclude command in each:

```
MCR: PIP *.MAC;*/EX&*WMB*.*/PU
     PIP *.MAC;*/EX&*WMB*.*;*/SD
DCL: PURGE/EX:*.MAC;* *WMB*.*
     DEL/Q/EX:*.MAC;* *WMB*.*;*
```

As a final point, on older versions of DCL, if you omit the version portion of the file specifier in the Delete command, a Selective Delete command will automatically be generated. Thus, the two commands

```
DCL>DEL/Q *.FTN;*
DCL>DEL *.FTN
```

are identical, and both translate into the MCR command

```
MCR>PIP *.FTN;*/SD
```

This is not a good feature, since if you get used to it and then inadvertently include the ;* in the file specifier,

```
DCL>DEL *.FTN;*
```

you will delete everything, without any queries. With the new versions of RSX (version 4.2 of RSX-11M, version 3.0 of RSX-11M-PLUS and version 3.0 of Micro/RSX), this special case has been removed. Now, a Delete command in DCL without an explicit version will be rejected.

22.8 The List Deletions Switch

It is possible with either the conventional Delete or the Purge command to delete many files without actually naming them. It is sometimes useful to know which files were deleted. The awkward way of determining this is to compare directory listings made before and after the deletions. The List Deletions switch eliminates this clumsiness by directing PIP to tell you which files it has deleted. (Note that this does not give you any choices, as does the Selective Delete command. It merely tells you what happened after it is all over.)

You can use the List Deletions switch as a modifier for either the Delete or the Purge command. In MCR it is indicated by the switch **/LD**, which is a subswitch of the switches **/DE** and **/PU**. By a subswitch I mean that it is appended directly to the main command switch; there is no ampersand as in the directory search modifiers. In DCL the List

Deletions switch is effected by the switch **/LOG** which is appended to either the Delete or Purge command. When you use this switch, PIP lists on your terminal the name of each file it has deleted. For example, if you want to delete all the files made today and you want a record of what the files were, you can do this:

```
MCR>PIP
PIP>/TD
PIP>*.*;*/DE/LD


The following files have been deleted:
DR2:[351,4]MAKEFILES.TSK;1
DR2:[351,4]MAKEFILES.OBJ;5
PIP>^Z
MCR>
```

The same command in single-line form would be

```
MCR:PIP /TD&*.*;*/DE/LD
DCL:DEL/TODAY/LOG *.*;*
```

22.9 The Creation Date Switch

In Chapter 13, I introduced the basic PIP copy function. When you copy a file, not only is a new file made, but a new entry in your directory is made as well. Part of this entry is the creation date of the file. The default choice for the creation date of the new file is typically the date of the copy. (Prior to version 4.0 of RSX-11M this was always the default; with version 4.0, a system generation option was introduced whereby the default could be changed to be the creation date of the old file.) The Creation Date switch allows you to override the default choice. In older versions of RSX, the Creation Date switch is available only if you are in MCR.

In many contexts, it is more meaningful to keep the creation date of the original file than it is to use the date of the copy. By keeping the original date, a certain historical perspective can be maintained. Whether this is useful or not is up to you. It is sometimes important to consider the possible subsequent action of the Default Date command when you decide whether to use the creation date or not.

In MCR, the Creation Date switch is specified as either **/CD** or **/-CD**. Regardless of the default, **/CD** specifies that you want to save the creation date of the original file. Similarly, **/-CD** specifies that you want to use

the date that the copy was made. In either case, you can use the switch with either the output or the input file specifier. (If you do not include the switch at all, you will get the default, which is system dependent.) For example, to make a copy of a master dictionary file from user area [100,1] on disk DR0: into your user area, with the creation date preserved, you could use the command

```
MCR: PIP /CD=DR0:[100,1]DICT.TXT
```

With the new versions of RSX (version 4.2 of RSX-11M, version 3.0 of RSX-11M-PLUS, and version 3.0 of Micro/RSX), this capability has been added to DCL. It is effected by the switch **/PRESERVE_DATE**. In DCL, the above command is now

```
DCL: COPY/PRESERVE DR0:[100,1]DICT.TXT SY:
```

23

File Backup Techniques

In this section, we discuss various techniques for making backup copies of your disk files. In general, when you back up a file, you make a copy of that file onto some device other than the one containing the original copy of the file. When you restore a file, you reverse this procedure. If you could guarantee that nothing would ever happen to the original copy of the file, there would be no reason to worry about backing up or restoring. This is not possible, hence the need for backup techniques.

Two basic things can happen to your files, which lead to two distinct motivations for backing them up. First, your files may be inadvertently destroyed. This may be due to some sort of disaster, such as a disk crash, but more commonly results from a mistake, such as accidental deletion. Second, as discussed in Section 21.1, you may deliberately remove from disk some of your files that, for the moment at least, you no longer need. The only way to guard against accidental destruction is to periodically make backup copies of your files. This might involve backing up either all your files or only those that have been created since the last backup. In a well-managed computer system, it should not be necessary for you to worry about protecting your files. Your system manager should periodically back up not only your files but those of all the other users as well. Even so, you may occasionally wish to make a backup copy of all your files, especially when you have reached some milestone, such as the completion of a major project. The second reason is more likely to be of interest to you. By making a backup copy of selected files, you can then delete those files without really losing them. You are merely transferring the files from disk to the backup device so as to relieve some of the clutter in your user area on the disk.

The important feature of any backup operation is that the backup

copy should not be on the same physical device as the original copy. If you were to put the backup copy on the same device, it would not be effective. For example, a head crash can destroy an entire disk, resulting in the loss of both the original and backup copies. Presumably, the device containing the original will be your pseudo device (SY:), which will be a large disk drive. If your system has several large disk drives, it is possible to use another for backup. Your system manager would normally do this when making a backup for many (or all) users. For your individual backups, we will consider only the use of smaller devices. Backup copies are most typically made to magtape. On older systems, DECtape is common, while on newer systems, floppy disks (diskettes) are likely to be found. In all these cases, the object on which data is stored is removable from the device and is small enough to keep in your desk drawer or file cabinet. Thus, you would literally make and keep your own backup tapes or disks.

Removability leads to another important use of the techniques that we will discuss. By making a backup copy of files from your computer system and restoring the files onto another system you have effected a physical transfer of files from one computer system to another. This transfer is made possible by the transportability of the backup medium. Prior to local area networking, this was the only way to transfer files from one system to another; it is still an important technique. Examples of this range from receiving data to be processed on a magtape to delivering programs on a diskette. As long as the other computer system has the appropriate device and backup programs, you can use any of the backup/restore techniques that we discuss for transferring files between computer installations.

23.1 Using Backup Volumes

Most of the time, your use of your PDP-11 computer facility will be limited to only a few devices (your terminal TI:, your default disk SY:, and possibly the printer LP: or the system library disk LB:). The system may have a variety of other devices, such as magtape drives, DECtape drives, and floppy disk drives, but you will normally not use these. Backing up and restoring files (or possibly transferring files to or from another computer facility) are probably the only operations you will do that require use of one of these other devices. With these devices, it is sometimes necessary to perform some special steps, which we discuss in this section.

Let's first get some terminology straight. We will be talking about devices and volumes. A device is something that can read or write data; a volume is something in the device on which data can be stored. For example, a magtape drive is a device; a reel of magtape is a volume. In most cases, volumes are interchangeable (you can remove one reel of tape and put on another one), although this is not true of some disk drives. As part of the system generation procedure, the RSX operating system is told about all the physical devices in the computer configuration. It does not, however, know which volumes are on the various devices. Thus, when you physically mount a volume (your own personal backup tape or disk) on a device, it also may be necessary to tell the system certain things about the volume before you can use it.

I say it "may be necessary" because it is not always necessary to do this. This is where things get confusing. The exact details of what you have to do depend on several things:

1. Whether your operating system is RSX-11M, RSX-11M-PLUS, or Micro/RSX
2. Whether your system includes Multiuser Protection or not
3. The backup/restore technique that you are using
4. Whether the backup volume is magtape or not
5. Whether the files on your backup volume are (to be) in standard RSX format or not
6. Whether the backup volume has ever been used before or not

I will attempt to discuss all possibilities, but if you have any doubts about what you should do on your particular system, ask your system manager. All of this might strike you as being unnecessarily complicated. Perhaps it is—so be it.

In the other sections in this chapter we will discuss several utilities that you can use for backing up and restoring files. In this section, for brevity, I will refer to these by only their MCR names, which are FLX, PIP, RMS, and BRU. For now, note that the ways in which these utilities use backup volumes vary considerably. In turn, the preparatory steps that you may have to take before you can use a volume with one of these utilities will vary considerably. For example, if you use FLX, almost none of what we discuss here will be necessary. On the other hand, if you use PIP, all of it will be relevant. As we go along, I will try to indicate the distinctions, but I will defer complete summaries of just what has to be done for each backup technique until the appropriate sections.

In general, the various RSX commands that we will discuss offer a

large number of options and possibilities. Almost all of these are totally irrelevant insofar as we are concerned. The entire set of procedures, in all their combinations, accomplishes only two things. First, these procedures prepare the volume so that it can be used by the various backup utilities. In general, this need be done only once per volume. Second, these procedures provide a protection mechanism whereby, once you have initialized it, no one else is allowed to use your backup volume. More specifically, no one else can read the files already on it or add new files to it. Another user, even if he is nonprivileged, can, however, erase everything on your volume. Thus, this protection mechanism does not really do you any good unless you are paranoid and would rather have your files destroyed than read by someone else. The entire protection mechanism may be unnecessary anyway, since in many installations you have actual physical custody of your backup volume and can lock it up or otherwise protect it as you see fit.

Loosely speaking, the sequence of things you have to do goes like this. A volume must be initially formatted via the Initialize Volume command. As part of this procedure, a label is written onto the volume. This label is the key to the protection system—another user cannot use the volume unless he knows the label. To use the volume for any sort of operation, you must first mount it via the Mount command. As part of this command, you tell the system what the volume label is; this is checked against the label that is on the volume. Only if they agree will the Mount command succeed. Next, you may have to use the Create Directory command to create user areas on the volume. At this point, you can use the volume for backup and restore operations. When you are done using it, you dismount the volume.

In some cases, before you can do any of this, you have to allocate the device on which the volume will be mounted and, after everything else is done, you have to correspondingly deallocate it. Under RSX-11M-PLUS and Micro/RSX, the initializing procedure is more complicated.

As a side comment, suppose you succeed in going through all this and make a backup of your files as desired. You then put the volume, be it tape, floppy disk or whatever, in your desk. Months later you decide to restore some files from it, but by then you have forgotten what the label is. You cannot mount the volume, since the Mount command requires that you specify the volume label. This means that you cannot read your own files. There is, fortunately, a way out of this. A privileged user (such as your system manager) can mount a volume without knowing its label, find out what the label is, and tell you. To avoid this problem, many users write the label on a piece of paper and tape it to the

outside of the volume. This defeats the protection concept as it makes the label known to anyone who has the volume, but it certainly is more convenient than not being able to read your own files.

Before discussing in detail the various procedures required for using backup volumes, I must introduce a few concepts. The first is the concept of volume formats. RSX has a standard format for files known as Files-11. The Files-11 format is used for all large disk devices and may be used for backup devices as well. To be acceptable to the Files-11 system, a volume must be specially formatted. (It may be helpful to think of this as being analogous to putting a set of hanging folders into an empty file cabinet drawer. This need be done only once, but until it has been done, the drawer cannot be used for storing papers.) Files-11 is not the only possible volume format; many others exist, corresponding to different operating systems or computer types. RSX considers any volume that is not in Files-11 format to be a foreign volume. Note that a foreign volume is not necessarily one that has come from another computer installation (although the term might imply this). A brand new volume is in no particular format and is considered to be foreign.

The Files-11 format is typically associated with disk volumes but can also be used with magtapes. In our discussion of backup techniques I will frequently need to distinguish a disk volume from a magtape volume. By "disk" I mean "disklike." Disklike devices include large disk drives (such as your system disk), cartridge disks (such as type RL01 or RL02 drives), floppy disks (such as type RX01 or RX02 drives), and DECtape and DECtape II. Although these offer a wide range of storage capacities and data transfer rates, and although DECtape physically resembles conventional magtape (except that it is 1 inch wide), all these storage media are handled in the same manner by the Files-11 software. For simplicity's sake, I will refer to all these devices as disks. On the other hand, conventional (1/2-inch-wide) magtapes are handled differently. You can put a magtape into Files-11 format, or, more exactly, you can make it sort of compatible with the Files-11 format. Such a tape volume is known as an ANSI tape because the Files-11 format for tapes conforms to ANSI (American National Standards Institute) specifications. Nonetheless, an ANSI tape is not a true Files-11 volume. In certain ways, you can pretend that it is, but you can do several things with a Files-11 disk volume (the most important is deleting a file) that you cannot do with an ANSI tape.

It is important to understand the interrelationship between the volume format (Files-11 or foreign) and the backup utility. FLX is used only with foreign volumes. BRU uses foreign volumes for magtape, but it uses

Files-11 volumes for disk. Both PIP and RMS require Files-11 format for both magtape and disk.

There is, by the way, a third category of backup volume, which is much less important but may be of interest to you. It includes paper tape and the TU60 cassette, neither of which is a Files-11 device. Backup and restore operations to paper tape or cassette are only possible with the utility FLX.

The second concept that you must understand is the status of a device. This discussion applies only to systems with multiuser protection. Most of the time, you can use whatever devices you need in the system without knowing anything about their status. This is not so for the devices that you are likely to use for backup. Any device that can store information can have a status of public, private, or unowned. A public device is one that can be used by any user. The system disk (SY:) is, for example, almost always public. You cannot use a public device in any way that would interfere with another user. If your system has several large disk drives, all of which are public, you may be able to perform a backup by copying files from one disk to another. You could not, however, then remove the disk. Thus, devices with removable volumes suitable for backup (magtapes, DECtapes, floppy disks, and small cartridge disks) are typically not public. A private device is one that can be used by only one user. This user is known as the owner of the device; you do not need to be privileged to own a private device. A device that is unowned is neither public nor private. As long as a device is unowned, it is similar to a public device in that anyone can use it. Unlike a public device, however, an unowned device can be made private by a nonprivileged user and can subsequently be returned to the unowned status.

Let's now turn to the various commands that you may need to use for backup volumes. For simplicity, I will (at first) ignore the special restrictions that RSX-11M-PLUS and Micro/RSX impose on foreign volumes. In all of these descriptions, I will use **ddnn:** to denote the name of a particular physical device, where **dd** is the two-character device-type mnemonic and **nn** is the device number.

To make a device private, you use the Allocate command. (This applies only to systems with multiuser protection.) The basic format of this command is

MCR: **ALL ddnn:**

DCL: **ALLOCATE ddnn:**

In DCL, the command **ALLOCATE** is commonly abbreviated to **ALL**.

The Allocate command is then the same for both MCR and DCL. For example, to allocate tape drive number 1 of type MM, you would use the command

```
ALL MM1:
```

You can also use a modification of this command form in which you do not have to include the device number. In this case, the command form is

```
ALL dd
```

Note that the colon is omitted. If you include the colon, it is equivalent to specifying a default device number of zero. By omitting the device number and colon, you are requesting that any available (currently unowned) device of the specified type be allocated to you. If all devices in the system of the specified type are in use, an error message is returned. Suppose, for example, that you want to restore some files from a backup that was made onto an RX02 floppy disk. Your system has four RX02 drives (device code DY), which are heavily used. These drives are in the main computer room, which is some distance away from where your terminal is. You could walk to the computer room, find a free drive, load your floppy disk, walk back to your terminal, and then allocate that particular drive. If, during this time, someone else allocates it, you have to find another drive. Instead of going through all that, you first enter the command

```
ALL DY
```

Assuming that there is an available drive, you get a response such as

```
ALL -- DY2: NOW ALLOCATED
```

You then walk to the computer room and load your floppy disk onto drive 2.

There are several reasons why you might want to allocate a device. Conceptually, if you have your own personal backup volume on a device, it makes sense that the device should be private to you. Practically speaking, this does not matter, since once you have started using the volume, it is very hard for another nonprivileged user to interfere with you, even if the device remains unowned. There are only two compelling reasons why you will want to allocate a device—it is necessary to do so to initialize a volume on that device and to create user areas on the volume.

To put a volume into Files-11 format, you must use the Initialize Volume command. The initialization procedure does three things: it formats the volume, erases any data that may have previously been on the volume, and writes a volume label. Once you have initialized a volume, you should not use the Initialize Volume command unless you wish to erase the volume and then reuse it.

If your system has multiuser protection, you must allocate a device before you can initialize a volume on it. If your system does not have multiuser protection, only a privileged user can use the Initialize Volume command. In this case, you will have to have someone such as your system manager initialize your backup volume.

In its basic form, the Initialize Volume command is

MCR: **INI ddnn: label**

DCL: **INITIALIZE ddnn: label**

In DCL, the command is commonly abbreviated to **INI**. The basic Initialize Volume command is then the same for both MCR and DCL. The **label** is a string of alphanumeric characters that will be written onto the volume as a label for subsequent identification and verification. If the volume is an ANSI magtape, **label** may be up to 6 characters long; otherwise, it may be up to 12 characters long. In the label, there is no distinction between upper- and lowercase alphabetic characters; you can also use digits and some special (punctuation) characters. For example, you could use the command

```
INI DY2: JAN84BACKUP
```

to initialize the floppy disk volume on drive 2 of type DY. Any previous data on the disk will be effectively overwritten. The label will be checked whenever you subsequently attempt to use the volume. If your backup volume is a magtape, you might use the following command instead:

```
INI MT: JAN84
```

Here, you use only the date as a volume label to comply with the six-character limitation.

When you use the Initialize Volume command, it is often because you have a disk or tape that has never been formatted. It is also possible to use this command with a volume that is already in Files-11 format. The initialization procedure does not check for a volume label from a previous initialization—it simply writes your new label over the old one and erases any existing files. Suppose that you, as a nonprivileged

user, have gotten access to someone else's tape or disk volume. Without knowing the volume label, you will not be allowed to read any of the files on the volume. Nonetheless, you can use the Initialize Volume command to initialize the volume, thereby destroying all the files that are currently on it.

For magtape volumes, this is all that you need to know about initializing. Disk volumes are a little more complicated. Let's consider the basic nature of a Files-11 disk volume such as a floppy disk or a reel of DECtape. As far as you are concerned, it is just a backup volume—something onto which you occasionally will want to write or from which you will want to retrieve files. As far as Files-11 is concerned, it is a disk volume, and it is handled in exactly the same manner as a large disk. The disk volume can have many user areas on it; each user area can have many files in it. In essence, your backup volume is your own private (albeit little) disk. Just as your system manager assigns areas to various users on the system disk and has to worry about how much room is left on it (sooner or later they tend to fill up), so too must you manage your backup volume. The management aspect that is of interest to us is how many files you want to have on the volume.

When you initialize a disk volume, you can specify the maximum number of files that will be allowed on the volume. If you do not specify this, a default value will be used. This value, regardless of the actual storage capacity of the disk, limits how many files you can have at any time. Of these, five are required for the Files-11 structure on the disk. (You may never see this structure, but the files are there.) For your convenience, you probably will want to make user areas on the disk. (We discuss this shortly; you do not have to, but it is clumsy not to.) Each user area requires one file for its directory and accordingly decreases the number of files available to you even further. Countering this somewhat is the fact that with Files-11 disk volumes, it is possible to delete files. Thus, you can decrease the number of files in use on the volume, thereby increasing the number of available files.

For example, an RX01 floppy disk has a total storage capacity of 494 blocks. (In Files-11, regardless of device type, a block holds 512 bytes.) If you accept the default, you will be allowed a maximum of 29 files on the floppy. As noted, 5 of these are used by Files-11 itself. Even if you do not make any user areas, only 24 files will be left over for your use. With user areas, this number will be less. If your files are moderately sized source files averaging between 8 and 9 blocks, the 24 files that are available to you by default will fill only half the capacity of the disk. Thus, if you accept the default value for the maximum number of files,

you probably will wind up underutilizing the capacity of your backup volume. This, in turn, often leads to using more volumes than is necessary. The expense of buying more volumes than you need is not necessarily a problem (floppy disks, for example, are cheap). It is, however, a nuisance to have to handle several volumes at once. Continuing with our example of an RX01 floppy, if you wanted to back up 30 files, you would not be able to do so easily. You would have to put up to 24 of them onto one floppy and put the other files onto another floppy. You would then have to keep track of which files were on which floppy.

When you use a disk for backup, the default value of the maximum file count will typically be too small. This is because it is chosen for typical active use, not backup. When you actively use a volume (as in your user area on the system disk) you will have a mixture of file types—source files, object files, task image files, etc. Task image files are typically much larger than the others, and they tend to raise the average file size. When you back up your user area, you will often select only source (along with text, command, and other similar) files. The average size of these selected files will be significantly smaller than that on which the default maximum file count is based. Due to this smaller average size, your backup volume can hold more files than would be possible with the typical mixture of file types.

From all this it should be clear that you will often want to allow more than the default number of files on your backup volume. You can do this when you initialize the volume by specifying the Maximum Files keyword. Although the simplest form of the Initialize Volume command is the same for both MCR and DCL, the syntax for keywords depends on which CLI you are using. With the Maximum Files keyword, the format of the Initialize Volume command is

MCR: **INI ddnn: label/MXF=value.**

DCL: **INI/MAX:value ddnn: label**

Depending on your CLI, either **/MXF=** or **/MAX:** is required syntax for specifying the Maximum Files keyword. In either case, **value** is the number of files you want. Note carefully, however, that as shown above, in MCR the value should be followed by a decimal point (.) or else it will be interpreted as an octal value. In DCL you do not have to worry about this, since the conversion to MCR supplies the required decimal point for you.

If your system is version 3.2 of RSX-11M or earlier, once you have specified the maximum file count for a volume, it is fixed. There is no

way to change it other than by reinitializing the volume, which will destroy everything on it. With version 4.0 of RSX-11M, a modification of the Initialize Volume command was added to MCR. This is the Home command, which allows you to change certain characteristics of a volume that has already been put into Files-11 format. (The Initialize Volume command creates a Home block on the volume, which contains certain key parameter values for the volume. The Home command allows you to change parameters in this block without disturbing the contents of the volume.) The requirements for using the Home command are the same as for the Initialize Volume command: you must allocate the device and, as we will see later, in RSX-11M, you need not mount the volume, but in RSX-11M-PLUS or Micro/RSX, you must mount it as a foreign volume. The most common use of the Home command is to increase the maximum file count allowed on the volume. The format of the Home command is identical to that of the Initialize command except for the command name itself. In MCR the command changes from **INI** to **HOM**; in DCL the switch **/UPDATE** is appended to the **INI** command. For the specific purpose of increasing the maximum number of files on the volume, the form of the Home command is

```
MCR: HOM ddnn: label/MXF=value.
DCL: INI/UPDATE/MAX:value ddnn: label
```

Before using the Home command, make sure that your version of RSX-11M is 4.0 or later (or, for RSX-11M-PLUS, 2.0 or later). If you do not have the proper version of RSX, do not try to use this command—it will do nasty things to your volume.

You now have the means of specifying how many files should be allowed on your backup volume. How do you determine what value to use? To answer this, you must know a little bit more about Files-11 volumes. As part of the Files-11 format, each file has a header associated with it. The file header contains information about the file, such as the name of the file and exactly where on the disk it is. The size of a file header is one block. Thus, for example, 100 files require a total of 100 blocks for the headers, independent of the actual content of the files. When a disk volume is initialized, a certain amount of space is reserved for file headers; the size of this space limits how many files you can simultaneously have on the volume. The determination of how much space to reserve for headers is based on the maximum file count value. If, for example, you initialize an RX01 floppy and specify a maximum file count of 100, 100 of the total 494 blocks, or just over 20 percent of the entire capacity of the floppy, will be reserved for file headers. Note

that the Home command only allows you to increase the file count; once the header space has been set aside, you can never use it for any other purpose. Thus, you do not want to specify an unnecessarily large maximum file count, as this also limits how many files you can have.

Table 3 lists various possible backup devices. For each, I show the capacity in blocks of a volume and the default value for the maximum file count. You can assess your particular backup requirements from this. Suppose, for example, that you want to back up all the files in two of your user areas. From standard directory listings, you determine that this involves 40 files totaling 200 blocks. If your backup medium is DECtape, you will have more than ample capacity on one reel of tape; you will, however, have to change the maximum file count. If the 40 files total 500 blocks, it is unlikely that (considering the file headers and other overhead) they will all fit on one reel of DECtape. They will fit on two reels, in which case the default maximum file count will probably be sufficient for each reel.

Suppose now that you want to get a feeling for how much space you will have left over (space that you can actually use) on a volume after you initialize it. You can do this via the following procedure. First, choose the type of backup volume and ascertain its capacity from the table. When you use the volume, this total capacity will be divided among three functions: the five files required by the Files-11 structure, directory files for any user areas you may have created, and your files. The size in blocks of the five Files-11 files and the directory files will be some small number plus the maximum file count. If you have only a couple of user areas, you may approximate this small number with the following equation: (20 + the maximum file count/32). This may be slightly pessimistic, but it is not worth trying to be more exact. This represents

Table 3
Possible Backup Devices

Device Code	*Volume Type*	*Capacity (blocks)*	*Max Files (default)*
DT	TU56 DECtape	578	34
DD	TU58 DECtape	512	30
DX	RX01 8-inch floppy	494	29
DY	RX02 8-inch floppy	988	60
DU	RX50 5.25-inch floppy	800	48
DK	RK05 cartridge disk	4800	294
DL	RL01 cartridge disk	10240	629
DL	RL02 cartridge disk	20480	1259
DM	RK06 cartridge disk	27126	1668
DM	RK07 cartridge disk	53790	3308

the overhead imposed by Files-11. Subtract it from the total capacity—this gives a close approximation to the number of blocks that will be left over for your files. The number of files that you can use is the maximum file count minus five for the Files-11 structure minus one for each user area. If you divide these two numbers, you will get the average file size available. For example, suppose you initialize an RX02 floppy and specify a maximum file count of 100. From the table, you see that the disk capacity is 988 blocks. You will want to have two user areas on the disk. The overhead taken by Files-11 will be roughly 124 blocks (20 + (100/32) rounded up + 100). The number of usable blocks will be roughly 864. The number of files available to you will be 93 (100 minus 5 minus 2 directories). This implies an average file size of between 9 and 10 blocks. If after initializing the floppy you intend to immediately copy 50 files totaling 420 blocks, you will have the capability of later adding up to 43 files totaling 444 blocks.

Now suppose you have initialized a volume to be used for backing up your files. It is now (be it a tape or a disk) in Files-11 format and ready to be used. Before you can actually use it, however, you must tell RSX certain things about it. You do this via the Mount command, which also verifies that you are allowed to use the volume. There are two forms of the Mount command that you may need to know—the form to mount a Files-11 volume and the form to mount a foreign volume. If your system is RSX-11M, you need only mount Files-11 volumes; if your system is RSX-11M-PLUS or Micro/RSX, you must mount any volume. We discuss mounting foreign volumes later.

If your volume has been put into Files-11 format (via the Initialize Volume command), you use the following form of the Mount command:

```
MCR: MOU ddnn: label
DCL: MOUNT ddnn: label
```

In DCL, **MOUNT** is typically abbreviated to **MOU** so that, in this simple case, the Mount command is the same for both MCR and DCL. Further, except for the command word **MOU**, this is identical to the simple form of the Initialize Volume command. When you enter a Mount command, RSX reads the label off the volume on the specified device and compares it against the label that you give in the command line. If the labels do not match, you will get an error message (the exact form of which depends on your CLI). If the volume is a magtape, you will get this message:

```
MCR: MOU -- incorrect file set identifier
DCL: MOU -- wrong volume label
```

If the volume is a disk, you will get this message:

```
MCR:MOU -- incorrect volume label
DCL:MOU -- wrong volume label
```

The Mount command will then be rejected, and you will not be able to do anything with the volume. If the label you specify is correct, the Mount command will succeed. No particular message will be given; you will simply get a prompt from your CLI, indicating that you may proceed. For example, to put an RX01 floppy disk into Files-11 format and then mount it, in either MCR or DCL, you might use the commands

```
INI DX1: JOB427
MOU DX1: JOB427
```

If you subsequently want to use the same volume (perhaps to restore some files from it or to back up more files onto it) you would use only the Mount command.

Once a Files-11 volume has been mounted you can use it in various ways. An ANSI tape may be used immediately by the various backup utilities. A disk volume requires an additional step. When we discussed the maximum number of files on a disk volume, I alluded to having several user areas on the disk. We now consider this issue in detail. When you initialize a disk, one user area is created on it. This is identified by the UFD [0,0]. In general, [0,0] is a special user area used by the operating system itself. It contains the five files required by the Files-11 system as well as the directory file for every other user area on the disk. (The directory for the area [0,0] is one of the five Files-11 system files.) On a public disk (such as your system disk) the area [0,0] typically is not used for anything else.

The UFD [0,0] identifies an area that, despite the special importance of some of the files in it, is nonetheless conceptually the same as any other user area. On your own private disk volume, you can use area [0,0] not only for the system files but also for your own files. Thus, if you are willing to put all of your backup files into UFD [0,0], your volume is ready to use after you have initialized it.

Although it is possible to do things this way, it is poor practice for several reasons. When you make a backup copy of your files, you typically want to preserve all attributes of these files. This includes not only the file name and file type, but also the ufd. By copying all your files into user area [0,0], you lose each file's ufd. This makes it difficult to restore the files onto your system disk if your files originated from several different user areas. A related problem is that during restore op-

erations, you may want to copy all your files from the backup volume. If you do this by using wildcards (*.*;*), you will get not only your files but the Files-11 files as well. By putting your files into their own user area, you are isolating the system files in area [0,0]. Finally, unless you are extremely short of file headers, it costs very little to put user areas on your backup volume, so you might as well do so.

You use the Create Directory command to put user areas onto a volume. We examined this command earlier in the context of having several user areas for yourself on the system disk. I noted that this was a privileged command. The Create Directory command is privileged for a public device, but it is not so for your own private device. Thus, if you have allocated the device, you can use this command to establish as many user areas as you wish on whatever volume is mounted on that device. The form of this command is

```
MCR: UFD ddnn: [ufd]
DCL: CREATE/DIR ddnn: [ufd]
```

where **ufd** is a normal ufd specifier. If you have either version 3.0 of RSX-11M-PLUS or any version of Micro/RSX, you may use a named directory if you wish.

The Create Directory command creates a User File Directory. This directory is itself a file, and is located in the system area [0,0]. If you are using numbered directories, the UFD is specified as **[ggg,mmm]**, where **ggg** and **mmm** are the three-digit (with initial zeros, if necessary) octal group and member numbers, and the name of the directory file is gggmmm.DIR. If you are using named directories, the UFD is specified as **[name]** where **name** is a string of up to nine alphanumeric characters, and the name of the directory file is name.DIR. Thus, for example, if you issue the command

```
MCR: UFD DY1: [123,1]
DCL: CRE/DIR DY1: [123,1]
```

you are actually creating the directory file DY1:[0,0]123001.DIR. This is all that you are doing. Once you have created the directory, you can then put files on the volume and associate them with the corresponding UFD.

Once you have created a user area on a Files-11 disk volume, you can use PIP to perform file maintenance functions in exactly the same manner as you would in your area on the system disk. In particular, you can delete files. You can delete all the files in an area, but the area itself

will remain. If you want to remove a user area and all the files in it from your backup volume, you must delete not only your files but the directory file as well. (You might do this, for instance, to remove outdated files to make more space available on your volume.) These deletions must be done in the proper order. The correct sequence is

```
MCR: PIP ddnn: [g,m]*.*;*/DE
     PIP ddnn: [0,0]gggmmm.DIR;*/DE

DCL: DEL ddnn: [g,m]*.*;*
     DEL ddnn: [0,0]gggmmm.DIR;*
```

(I have shown the command forms for numbered directories; those for named directories are analogous.)

Putting user areas onto a backup volume is similar to initializing a volume. Your only reason for doing it is to prepare the volume for later use. Unlike the Initialize Volume command, the Create Directory command is nondestructive. You can use it as often as you like, when you like. If you have put a bunch of files onto a backup volume and you subsequently decide that you want to create another user area for backing up other files, you can do so. Also, although the Initialize Volume and Create Directory commands both serve to prepare a volume, one must be done before the volume is mounted and the other afterward. As a final note, I repeat that the creation of user areas on a Files-11 volume relates only to disk volumes. The concept of different UFDs does not exist for ANSI tapes.

If you mount a volume, you should correspondingly dismount it when you are done using it. When you issue a Dismount command, you are telling RSX to forget the relationship between device and volume that was established by the Mount command. The form of this command is simply

```
MCR: DMO ddnn:

DCL: DISMOUNT ddnn:
```

Note that it is not necessary to specify the volume label. You can specify it by using this form of the Dismount command:

```
MCR: DMO ddnn: label

DCL: DISMOUNT ddnn: label
```

In this case, the label must be correct or the command will be rejected.

In some cases, the Dismount command will also physically dismount the volume. For example, if you are using a tape drive, the operating

system will issue commands to the drive that will cause the tape to be rewound (and with some drive types unloaded) and the drive will then be put into off-line status. Following completion of the Dismount command, you can physically remove your volume from the device.

If you have allocated the device, you should deallocate it after dismounting the volume. In MCR the name of this command is **DEA**. In DCL the official name of this command is **DEALLOCATE**. Some versions of the DCL manual (e.g., version 4.1 of RSX-11M) state that you can shorten this to DEA. This is not true. The command **DEA** is recognized by DCL as being an abbreviation of the command **DEASSIGN**. The correct abbreviation is **DEAL**. Thus, the command forms are

MCR: **DEA ddnn:**

DCL: **DEAL ddnn:**

Note that to deallocate a device, you must first dismount the volume on it. When you issue a Deallocate command, you are telling RSX that you no longer need the device. This allows someone else to use it. If you allocate a device and mount a volume, then subsequently dismount the volume but do not deallocate the device, you still own the device, and no one else can use it. If you log out and have forgotten to dismount or deallocate, RSX will do so for you. You should not, however, rely on the log-out procedure to clean things up. It is common courtesy for you to free up system resources as soon as you no longer need them.

It is now time to consider the extra complications introduced by RSX-11M-PLUS. These all apply to Micro/RSX as well; for brevity here, we will refer to only RSX-11M-PLUS. Under RSX-11M-PLUS, any volume must be mounted before you can do anything with it. If the volume is foreign, it must be mounted as such; if the volume is in Files-11 format, it must be mounted as such. The command form for mounting a foreign volume is

MCR: **MOU ddnn: /FOR**

DCL: **MOU/FOR ddnn:**

The keyword **/FOR** specifies that the volume is foreign; in this case, there is no volume label for you to specify. Before you can use the Initialize Volume command to put a foreign volume into Files-11 format, you must first mount it as a foreign volume. It is important to note that once you mount a volume as foreign, RSX-11M-PLUS remembers that it is foreign. When you then initialize the volume, you physically put it into Files-11 format, but RSX-11M-PLUS does not remember this and

continues to regard the volume as foreign. Thus, you must next dismount the volume, as this is the only way to make the operating system forget that the volume was foreign. (The Dismount command is the same for a foreign volume as it is for a Files-11 volume; you use the command form discussed earlier.) Of course, you must then mount the volume again, this time as a Files-11 volume.

It is also possible to mount a foreign volume under RSX-11M. This is, however, not necessary and normally is not done. Under RSX-11M, certain utilities can work with foreign volumes whether they are mounted or not. These include the Initialize Volume command and the backup utilities FLX and BRU. Thus, if your operating system is RSX-11M and you use the backup/restore techniques that we discuss, you will never have to mount a foreign volume. Under RSX-11M-PLUS and Micro/RSX, however, you have no choice—you must always mount a foreign volume.

By way of summary, I will list the sequence of operations required for using a backup volume under various conditions. I will not indicate the Create Directory command for putting user areas onto a disk volume, since, although you should use it, it is not required. Nor will I worry about changing the maximum number of files on a disklike device from its default value (although you probably should). Rather than giving specific examples, I will show the generic command forms—that is, I will refer to **ddnn:** and **label** as being arbitrary device identifiers and volume labels. Similarly, I will use **uti** to denote the backup/restore utility. For simplicity, I will show only the MCR command forms. In these examples, the only commands that are different for DCL are the Mount Foreign, Dismount, and Deallocate commands.

The various conditions I will consider are:

a. Does the backup/restore utility require the volume to be in Files-11 format?
b. Does your system have multiuser protection?
c. Has the volume been used previously?
d. Is the operating system RSX-11M, or RSX-11M-PLUS or Micro/RSX?

The various command sequences are summarized below. In the next sections we discuss several utilities that can be used for backup and restore operations. If necessary, the command sequences are explained in greater detail.

1. FLX with any volume type, or BRU with magtape

   a. Files-11 volume: NO
   b. Multiuser protection: YES or NO
   c. Volume previously used: YES or NO

RSX-11M

```
uti
... commands
```

RSX-11M-PLUS or Micro/RSX

```
MOU ddnn:/FOR
uti
... commands
DMO ddnn:
```

2a. BRU with disk; PIP or RMS with any volume type

   a. Files-11 volume: YES
   b. Multiuser protection: NO
   c. Volume previously used: NO

RSX-11M or RSX-11M-PLUS or Micro/RSX

```
MOU ddnn:label
uti
... commands
DMO ddnn:
```

2b. BRU with disk; PIP or RMS with any volume type

   a. Files-11 volume: YES
   b. Multiuser protection: YES
   c. Volume previously used: NO

RSX-11M

```
ALL ddnn:
INI ddnn: label
MOU ddnn: label
uti
... commands
DMO ddnn:
DEA ddnn:
```

RSX-11M-PLUS or Micro/RSX

```
ALL ddnn:
MOU ddnn:/FOR
INI ddnn: label
DMO ddnn:
MOU ddnn: label
uti
... commands
DMO ddnn:
DEA ddnn:
```

2c. BRU with disk; PIP or RMS with any volume type

   a. Files-11 volume: YES
   b. Multiuser protection: YES or NO
   c. Volume previously used: YES

```
MOU ddnn: label
uti
... commands
DMO ddnn: label
```

23.2 The File Exchange Utility

The first backup technique that we will consider is the file exchange utility, FLX. Historically, this is the oldest of the methods that we will consider; it should exist on any RSX installation. FLX originally was written as a utility to support the transfer of files between various PDP-11 installations. Its use as a backup utility is an incidental outgrowth of this capability. Nonetheless, FLX is often both the simplest and quickest technique available for backing up or restoring files. This is especially true if your backup medium is magtape. This is the most likely possibility, except on relatively small systems where a less expensive device might be found. The use of FLX is not limited to magtape, but most of our discussion will assume that this is what you are using.

RSX is but one of several possible operating systems offered by Digital for its PDP-11 series of computers. Under RSX, files are stored in the Files-11 format. Other operating systems have their own file storage formats. Two of these are DOS-11 and RT-11. (The DOS operating system no longer exists for the PDP-11, but its file storage system lives on and is still widely used as a universal means of transferring files on magtape.) The utility FLX supports file transfers by performing conversion between Files-11 and these two other formats. For example, you can use FLX to take a file in your user area (which is on disk in Files-11 format), convert it into DOS-11 format, and write it out to magtape. You could then deliver this tape to a PDP-11 facility that could read a tape in DOS-11 format. Similarly, this facility could deliver files to you by putting them on magtape in DOS-11 format. You would use FLX to read these files from the tape, convert them into Files-11 format, and store them in your user area. File transfer operations such as these were the original purpose for using FLX. As noted earlier, file transfer and backup/restore are, in a sense, indistinguishable. Backing up a file and subsequently restoring it is equivalent to transferring the file from yourself now to yourself some time in the future. Thus, you can use FLX to copy files onto a tape in DOS-11 format, which you can then save as a backup. If

you ever need to restore the files, you can use FLX to read them in, converting them from DOS-11 back to Files-11 format.

It may seem strange to you that we discuss using FLX for backup in the context of converting between Files-11 and DOS-11 format. Why not simply copy the files from disk to tape in Files-11 format? For one reason, FLX does not support this capability. FLX was designed for transferring files from one computer installation to another, and the DOS-11 format historically has been used for this transfer. The second reason is that it is actually faster to do the format conversion than it is to copy the files in Files-11 format. This is admittedly counterintuitive. The Files-11 format is designed for disks; it becomes very inefficient when used with magtape. On the other hand, the DOS-11 format is quite efficient for use with magtape. The writing to or reading from magtape is the slowest step in the process of backing up or restoring files. By using DOS-11 format on the tape, the increased efficiency in the tape I/O outweighs the extra work required for format conversions, and the entire procedure can be done in less time.

FLX can also be used to back up files in RT-11 format. The DOS format is limited to an older generation of devices: magtape, cassettes, DECtape, RK05 disk cartridges, and paper tape. The RT format is not supported for magtape. It can be used only with small disk devices: DECtape, RK05, RK06, RK07 and RL01 or RL02 disk cartridges, and RX01 or RX02 floppy disks. Based on these restrictions, you may not have any choice of format to use with FLX—you may have to use either the DOS-11 or RT-11 format according to the devices available to you.

In DCL there is no direct way to use FLX—that is, there are no DCL commands that translate into MCR commands to FLX. Instead, you must use the MCR command forms. As with other RSX utilities, you can use FLX either in a single-line command or interactively. You enter a single-line FLX command as follows:

```
MCR>FLX command
DCL>MCR FLX command
```

To use FLX interactively, you invoke it via the command

```
MCR>FLX
DCL>RUN $FLX
```

Once in FLX, you enter individual commands, eventually terminating the sequence with a **CTRL/Z** which causes FLX to exit, returning control

to your CLI. Unless you use FLX frequently, it is easy to forget an exact command form, in which case you may need several tries before you get it right. For this reason, it is probably best to use FLX interactively, even for single commands.

The basic FLX command is the File Transfer or Copy command, which is of the general form

```
output_device/format=input_files/format
```

In addition, other commands exist for performing special functions to the backup device.

Let's first consider the basic file transfer command shown above. The Format switches (**/format**) specify the format of the input and output files. We will discuss these later. Ignoring these switches, the basic command form is

```
output_device=input_files
```

which, syntactically, is identical to the basic PIP Copy command. The input side may specify several files, either explicitly or via wildcards. This is, however, subject to restrictions as noted below. The output side of the command differs from what we have seen before in utilities such as PIP as it can specify only the output device or ufd. Output file names cannot be specified. They are automatically set equal to those of the input files, to the extent that this is possible. You have no choice in this, but for purposes of backing up and restoring, you should have no reason to change file names. A more significant limitation arises from the fact that neither DOS-11 nor RT-11 supports the full Files-11 file specification. Version numbers do not exist in either DOS-11 or RT-11. In addition, UFDs do not exist in RT-11. The greatest difference is that file names in RT-11 can be only six characters long (the file type is the normal three characters). These limitations can lead to problems when you are backing up your files.

As a very simple example, suppose you wish to back up a group of files that includes the two files NOTES.TXT;3 and NOTES.TXT;2. With either the DOS-11 or the RT-11 output format, both of these input files would lead to an output file specifier of NOTES.TXT. Since there can be only one file associated with any particular specifier, a naming conflict results. In this case, one of the versions of NOTES.TXT (whichever is specified or found first) will be copied as requested. The other will not be copied; instead, you will get a "File Already Exists" error message. Similarly, the two files [100,1]MATH.FTN;1 and [100,2]MATH.FTN;1 will lead to a naming conflict if the output format is RT-11. (Under

DOS-11, the different UFDs will result in distinct file specifiers so that there will be no conflict.) Finally, you must be careful with files that have long names. If you have two files called CHAPTER1.TXT;12 and CHAPTER2.TXT;33, you will get a naming conflict if you use the RT-11 output format, since both file specifiers will collapse to the name CHAPTE.TXT. (You will not have a problem with DOS-11, since it preserves all nine characters in the Files-11 file name.) Even when this truncation does not produce naming conflicts, it can take all the meaning out of carefully chosen file names.

To avoid the most common source of naming conflicts, you are not allowed to use a wildcard for the version number in the input specifier. You can omit the version number (in which case the latest version will be used) or you can specify any version number you wish. When you are backing up files, you probably will want to save only the latest version of each file. This will not pose a problem as you can simply use an input file specifier such as `*.*`. If your backup medium is in DOS-11 format, this will eliminate naming conflicts, at least for a single backup operation. If your backup medium is in RT-11 format, you will still have to worry about naming conflicts due to the lack of UFDs and also due to the truncation of file names.

The problem of naming conflicts can arise in other situations in which it is not so easily dismissed. FLX acts in an append mode. That means that you can use FLX to make a backup of several files onto your volume. If you later use FLX to back up some other files, they will be written after the first batch of files. The composite then forms one set of files, and naming conflicts are not allowed within this set. Suppose that you back up a set of files including MATHSUBS.FTN;2. Over the next several weeks you write some new programs that require modifications to your mathematical subroutines. These changes lead to the file MATHSUBS.FTN;3. Your original backup used only a small fraction of the storage capacity of the volume, so you try to back up these new files onto the same volume. A naming conflict results, since the file MATHSUBS.FTN already exists on the volume. What is particularly annoying about this situation is that you are left with only the old version of the file on your backup.

In general, you can do several things when naming conflicts arise with FLX. If the conflict occurs during a single backup session, you can avoid it by specifically not copying certain files. If the conflict occurs between a file that was copied previously and one that you wish to copy now, you can sometimes delete the earlier copy. If this is not possible, you can use a different volume (e.g., a second reel of tape). Regardless of how the conflict arises, you can always get around it by renaming certain

files prior to backing them up. If none of these choices is acceptable, you must use one of the other, more sophisticated backup techniques that we discuss later.

Restoring files with FLX is easier than backing them up because naming conflicts almost never arise. Instead, the opposite problem arises. Since each input file is in either DOS-11 or RT-11 format, there is no version number in the input specifier for FLX to copy into the Files-11 specifier. Since some version number has to be used, FLX acts like PIP with the New Version switch set. If you back up a group of files and subsequently restore them into an empty user area (or any area where there are no files with the same name), they will all appear as version 1. This may be confusing and represents a certain loss of historical perspective, but it will not cause any real problems. If, however, you have other versions of the same files already in your user area, the files brought in via FLX will be given higher version numbers, which may or may not be what you want. If the input is in RT-11 format, there also is no ufd to use. In this case, your default directory is used.

Let's return to the basic FLX File Transfer command. Either side of this command may end in a switch that specifies the file format. The possible choices are:

```
/RS     Files-11
/DO     DOS-11
/RT     RT-11
```

If you do not use any file format switches, these defaults are assumed:

```
/RS     for output
/DO     for input
```

Thus, the default configuration for FLX is to read DOS files into an RSX system. This corresponds to restoring files that were backed up in DOS format. If you are backing up, you can reverse these defaults via the special command

```
FLX>/RS
```

Following this command, the default assumptions are

```
/DO     for output
/RS     for input
```

You can return to the original defaults via the command

```
FLX>/DO
```

These are not easy to remember, and it is probably better to specify the format switches all the time. Also, if you are using the RT-11 format for your backup volume, you will always have to specify the formats.

Let's look at some simple examples. The command

```
FLX>MM:/DO=A.DAT/RS
```

takes the latest version of file A.DAT from your user area, converts it to DOS-11 format, and puts it onto the reel of magtape currently on tape drive MM:. The command

```
FLX>DT1:/RT=*.FTN,*.MAC/RS
```

takes the latest versions of all FORTRAN and MACRO-11 source files in your area, converts them to RT-11 format, and writes them to DEC-tape drive 1. All file names are truncated to six characters.

Commands for restoring files are just as simple. The command

```
FLX>/RS=MT2:A.DAT/DO
```

reads the tape on MT2: until the single file A.DAT is found. The file is then converted from DOS-11 back to Files-11 format and put into your user area. The command

```
FLX>/RS=DX:*.*/RT
```

restores all files from floppy disk drive DX: into your area. Note that due to the file name truncation during backup, the file names after this restore operation may not be the same as those you originally had.

You will need to know about a few auxiliary commands in FLX. Before you can back up any files with FLX, you have to initialize the output volume. {This is conceptually identical to initializing a Files-11 volume as discussed in the previous section.} Initializing is done with the Zero command, which is denoted by the switch **/ZE**. The form of this command is

```
FLX>device:/ZE/DO
or
FLX>device:/ZE/RT
```

When you use the Zero command, you must remember to specify whether the device format is to be DOS-11 or RT-11. If you do not specify the format, DOS is assumed. Thus, the command

```
FLX>DY1:/ZE
```

will not work, since the assumed DOS format is not applicable to floppy disk devices. The Zero command is potentially dangerous. When you use it, you effectively erase whatever is already on the volume. Suppose you are given a DECtape and you wish to back up some of your files onto it in DOS format. The first time you use FLX with that particular DECtape, you must initialize it with a command such as **DT1:/ZE/DO**. You can then back up files onto it. If, several months later, you decide to put more files on the DECtape, you must not use the Zero command. If for some reason you no longer need any of the files on the DECtape, you can use **/ZE** to reinitialize it so that you can use it for something else.

The next useful FLX command is the Directory List command. This command is identical in function to the conventional PIP Directory command, but you can use it only on volumes in DOS or RT format. In FLX, you request a directory lisiting by using the List switch (**/LI**). The simple form of this command is

```
device:file_specifier/LI/format
```

This causes the directory listing to be displayed on your terminal. For example,

```
FLX>DT:/LI/RT
```

specifies that the volume currently on DECtape drive 0 is in RT-11 format and requests a directory listing of all files on it. Similarly,

```
FLX>MM:[*,*]*.MAC/LI
```

produces a directory of all MACRO source files in all directories for MM:. (As with **/ZE** if no format is specified, DOS-11 is assumed for **/LI**.)

The last auxiliary command that we consider is the Delete command. You can use this to delete files that have been previously copied via FLX. You can use it only with disks or DECtapes—you cannot use it with other devices such as magtape. The Delete command is identified by the Delete switch (**/DE**)

```
FLX>device:file_specifier/DE/format
```

Suppose that you have previously made a backup copy of version 2 of MATH.FTN onto DECtape in DOS format. Several months later you modify the file and decide to back it up. The command

```
FLX>DT:/DO=MATH.FTN;3/RS
```

will fail because the file already exists on DT:. In this case, you can overcome the naming conflict by first deleting the old copy via the command

```
FLX>DT:MATH.FTN/DE/DO
```

Let's summarize the various FLX commands with an example of backing up and subsequently restoring a group of files. First, get a directory of your area:

```
MCR>PIP/LI

Directory DR2:[351,11]
13-JAN-85 14:14

MAKEFILEB.MAC;3        4.      14-DEC-84 16:37
TESTFILEA.MAC;16.      6.      13-DEC-84 15:52
NOTES.TXT;2            2.      09-NOV-84 17:14
NOTES.TXT;3            2.      13-DEC-84 16:08

Total of 14./14. blocks in 4. files
```

Next, put a new reel of tape on tape drive MM:, invoke FLX, and initialize the backup volume (DOS-11 format):

```
MCR>FLX
FLX>MM:/ZE
```

Now you can use the tape. Make a backup copy of (the latest version of) all files and then get a directory listing of the tape, just to double-check:

```
FLX>MM:/DO=*.*/RS
FLX>MM:/LI


Directory MM:[351,11]
13-Jan-85

MAKEFILEB.MAC      4.      13-Jan-85 <233>
TESTFILEA.MAC      5.      13-Jan-85 <233>
NOTES.TXT          2.      13-Jan-85 <233>

Total of 11. blocks in 3. files
```

In this directory listing of MM:, note that there are no version numbers. Also note that all the dates reflect the creation of the file on MM:, not the original file creation on SY:. Type CTRL/Z to get out of FLX, take the tape off the drive, and put it in a safe place. Now suppose something

happens to the originals of these files on disk. Perhaps you enter this command:

```
PIP *.*;*/DE
```

Whether you did so deliberately to remove unneeded files or accidentally is irrelevant. At this point, the files are gone. That is why you have a backup tape. To restore the files, dig out your backup tape and read it with the command

```
MCR>FLX /RS=MM:*.*/DO
```

A directory listing of your user area will look like this:

```
MCR>PIP /LI


Directory DR2:[351,11]
22-Feb-85 14:16

MAKEFILEB.MAC;1      4.      22-FEB-85 14:16
TESTFILEA.MAC;1      6.      22-FEB-85 14:16
NOTES.TXT;1          2.      22-FEB-85 14:16

Total of 12./12. blocks in 3. files
```

Note that following the restore, all files are version 1 and that the creation dates are all the day of the restore operation.

If you want to back up the files in the example above but you wish to use an RX02 floppy, you will have to go to RT-11 format. Assume that you load your floppy onto drive DY:. In this case, your sequence of commands will be

```
MCR>FLX
FLX>DY:/ZE/RT
FLX>DY:/RT=*.*/RS
FLX>DY:/LI
```

Similarly, to restore these files later, you would use the command

```
MCR>FLX /RS=DY:*.*/RT
```

A directory listing of your user area will now look like this:

```
MCR>PIP /LI


Directory DR2:[351,11]
22-Feb-85 14:16

MAKEFI.MAC;1      4.      22-FEB-85 14:16
```

```
TESTFI.MAC;1      6.      22-FEB-85 14:16
NOTES.TXT;1       2.      22-FEB-85 14:16

Total of 12./12. blocks in 3. files
```

Note that the file names have all been truncated to six characters. Although this did not lead to naming conflicts when you made the backup, it led to the loss of all the information distinguishing File A from File B that you had so carefully put into your original file names. This is probably the greatest weakness of using the RT-11 format for FLX backups.

23.3 Using PIP for Backup

When I first introduced the utility PIP, I explained that its name was an acronym for Peripheral Interchange Program. Since the essence of backing up and restoring files is moving them from one peripheral device to another, it seems that PIP should be well suited for these operations. In general this is true.

With the possible exception of magtapes, PIP is capable of writing files to or from any device that you might use for backup. PIP can also support magtapes (if they are in Files-11 format—that is, if they are ANSI tapes); however, this is a system generation option. This feature is known as ANSI support. ANSI support is offered as an option because you may not use it often enough to justify the increased size of the PIP task. There are three possibilities for your system. First, PIP may include ANSI support. Second, the installed version of PIP may not, but another version of PIP may be available somewhere that does have ANSI support. Third, ANSI support may not be available at all, in which case you will not be able to use PIP for magtape volumes.

You can easily determine whether PIP, as installed on your system, has ANSI support or not. To do this, you get into PIP (if you are in DCL, you use the command **RUN $PIP**) and enter the Identify command,

```
PIP>/ID
```

In response to this, PIP displays its version number followed by "(ANSI)" if it has ANSI support. Typical responses look like

```
PIP VERSION M1340 (ANSI)
```

or

```
PIP VERSION M1340
```

If the installed PIP does not have ANSI support, check with your system manager to find out whether another version does. In the remainder of this section I will assume that ANSI PIP is available to you.

To use PIP, your backup volume must be in Files-11 format and must be mounted. (This is accomplished as explained in Section 23.1.) If you are using a disk volume, you should already have created user areas on it via the Create Directory command (Section 23.1). It is possible to effect this from within PIP, but it is easier to do so from your CLI.

A backup or restore operation is accomplished using the basic PIP Copy command. This is conceptually identical to other uses of the PIP Copy command with the important exception that now your backup volume is mounted on either the output device (for file backup) or the input device (for file restoring). For backup, the basic command form is

```
MCR: PIP ddnn:=file_specifier(s)
DCL: COPY file_specifier(s) ddnn:
```

and for restoring it is

```
MCR: PIP =ddnn:file_specifier(s)
DCL: COPY ddnn:file_specifier(s) SY:
```

where **ddnn:** is the backup device. These simple commands are all you need if your backup volume is magtape. They will also work with disk volumes, but if you are using a disk volume, you should take advantage of all PIP's capabilities.

Normally, you will not want to change file names, so I defaulted these in the output specifiers above. Typically, for each file being backed up or restored, you will want the output file specifier to be identical to the input specifier except for the device code. If you are working with only one user area, the commands above are sufficient. When you are working within several user areas, you must be careful with the output specifier. If you leave the ufd portion of the output file specifier blank, it will be set to your default directory. If you are copying files from several UFDs on one device, they will all end up in one UFD on the output device. Presumably, you will want to preserve the ufd along with the other parts of the file specifier when you copy files for backing up or restoring. You can force the output ufd to be set equal to the input ufd by using wild cards for the ufd portion of the output specifier. If you have numbered directories, these wildcards are **[*,*]**; if you have named directories, they are **[*]**. In the examples below we will use the form for numbered directories, since your system may not support named directories.

With ufd specifiers included, the more generalized command forms for backing up and restoring, respectively, are

```
MCR: PIP ddnn: [*,*]=file_specifier(s)
DCL: COPY file_specifier(s) ddnn: [*,*]
```

```
MCR: PIP [*,*]=ddnn: file_specifier(s)
DCL: COPY ddnn: file_specifier(s) [*,*]
```

Finally, you should preserve the creation dates of your files. As explained in Section 22.9, you can do this by using the Creation Date switch. Although we include the DCL forms below, this feature is available to you from DCL only with the new versions of RSX (version 4.2 of RSX-11M, version 3.0 of RSX-11M-PLUS and version 3.0 of Micro/RSX). With the preservation of the creation date included, the PIP commands for backing up and restoring, respectively, are:

```
MCR: PIP ddnn: [*,*]/CD=file specifiers
DCL: COPY/PRESERVE file_specifier(s) ddnn: [*,*]
```

```
MCR: PIP [*,*]/CD=ddnn: file specifiers
DCL: COPY/PRESERVE ddnn: file_specifier(s) [*,*]
```

The features for preserving the ufd and the creation date apply only to the use of disk volumes. The ANSI standard for magtapes does not support UFDs. The ANSI standard does include the file creation date (but not the time of day), but the PIP Creation Date switch specifically does not work with ANSI volumes. You can specify a ufd or the Creation Date switch for a magtape, but they will be ignored.

Let's now look at some examples. These will be several lines long; for brevity, I will show only the MCR command forms. If you wish to perform these operations from DCL, you should by now be able to translate my examples into the appropriate DCL Copy commands with little difficulty. Remember, if you are using DCL, you must have a new version of RSX in order to use the Creation Date switch. If you have an older version, you may want to effect these commands from MCR to avoid losing the historical perspective afforded by this capability.

Let's suppose you have group number 212 on your system and in this group you have several user areas. The areas defined by member numbers

4 and 5 are for projects that you have recently finished and do not anticipate using again. You decide to make a backup copy of all these files so that you can then delete them from the system disk.

As a first example, suppose that your backup is to be made onto DECtape. You get a reel of DECtape and mount it on drive DT1:. (If necessary, you put the volume into Files-11 format and establish UFDs for areas [212,4] and [212,5] on it.) You then enter the following commands

```
MCR>PIP
PIP>DT1:[*,*]/CD=[212,4]*.*;*,[212,5]*.*;*
PIP>[212,4]*.*;*,[212,5]*.*;*/DE
PIP>^Z
MCR>
```

When your backup is complete, you can dismount the DECtape and put it in a safe place. Since it is a disk volume, you can use PIP to work with the files just as you do on the system disk. You can add more files whenever you want to by using the same procedure. You can get a directory listing, either of all files or of a subset, which may be specified using wildcards. You can store multiple versions of a file on the volume; delete files, which is useful if the volume fills up; and purge to get rid of old versions. You can also restore files from the DECtape back into your regular user area on the system disk simply by reversing the copy command used for backup. For example, suppose you decide that you want to restore all FORTRAN source files from area [212,4] and all MACRO-11 source files from area [212,5] back onto your system disk. You get your DECtape and mount it, but this time you do not initialize it or create user areas on it. If you use drive 2, the PIP command to restore your files would simply be

```
MCR>PIP [*,*]=DT2:[212,4]*.FTN;*,[212,5]*.MAC;*
```

As a somewhat different example, suppose your backup is to be to magtape. If necessary, you initialize the tape to put it into ANSI format. You do not (cannot) put user areas on it. You can then back up your files in much the same way as just discussed:

```
MCR>PIP
PIP>MM:=[212,4]*.*;*,[212,5]*.*;*
PIP>[212,4]*.*;*,[212,5]*.*;*/DE
PIP>^Z
MCR>
```

Since you cannot distinguish files on a tape volume by user area, you should make sure that you do not have any files in [212,4] with the same

file name, type, and version (you should avoid even the same name and type with different version number) as any in [212,5]. Once you have made your backup tape, you can obtain directory listings with PIP in the same manner that you would for any other device. For a directory listing from magtape, as opposed to one from disk, you will notice the following differences: there is no user area specification; the format for printing the name, type, and version is slightly different; the creation date is always the date of the file copy; and the creation time is always 00:00. A further restriction is that you cannot delete files from an ANSI tape volume. Since files on tape are not identified by user area, the restoration of files that were backed up from different user areas is not as simple as the backing up operation. To restore all FORTRAN files from [212,4] and all MACRO-11 files from [212,5], you would have to use something like this:

```
MCR>PIP
PIP>[212,4]=MM:*.FTN;*
PIP>[212,5]=MM:*.MAC;*
PIP>
```

If, when you make the backup, you have no MACRO-11 files in [212,4] and no FORTRAN files in [212,5], you will be all set at this point. If, however, you have FORTRAN files in both areas, this restore operation will put all of these back into [212,4]. Presumably, you would then want to get a directory listing of all of these to find out just what you have and then use PIP to delete (from the system disk) those that originally came from [212,5].

In conclusion, for disk volumes, PIP is a wonderful utility to use for backup and restore operations. For magtape volumes, the lack of UFDs and the inability to delete files limit its usefulness.

23.4 The Record Management Services

As part of your RSX system, you have a collection of programs known collectively as the Record Management Services (RMS-11 or simply RMS). RMS is designed to provide a means of storing and retrieving large amounts of data. It allows you to define a wide variety of file structures—for example, you can use files that are designed specifically to store personnel information, stockroom inventory data, etc. RMS is intended to be used for applications involving what is commonly known as data base management.

The ways of organizing files allowed by RMS-11 are more versatile

than those offered by Files-11. When you use RMS files, you cannot use the Files-11 system. RMS-11 provides all the file management services that you need. As part of this, RMS provides its own utilities for backing up and restoring RMS-11 files. The RMS-11 file structure is a superset of the Files-11 structure, which means that any Files-11 file is (a relatively simple form of) an RMS-11 file. Thus, you can use the RMS backup and restore utilities for your normal Files-11 files. You do not have to know anything about the RMS-11 file structure to do this. As strange as it may seem, these RMS utilities are sometimes more convenient than the more traditional Files-11 backup routines.

Unlike the other backup and restore utilities that we consider in this section, RMS offers the various functions that you will require from several interrelated but distinct utilities. To back up your files, you will use one RMS utility; to restore your files, you will use another. This may seem somewhat strange, but it should pose no problems for you.

The RMS utilities are not available to you directly from DCL. To use them, you will have to use the MCR command forms. For single-line commands, you can do this by prefacing the appropriate MCR command with the command name **MCR**. For multiple-line commands, you will have to enter the utility, which you can do from DCL with the command **RUN $rms**, where **rms** is the particular RMS utility you wish to run. In our examples in this section, I will consider only MCR usage.

RMS offers three interrelated utilities for manipulating backup volumes. These are RMSBCK, RMSRST, and RMSDSP, which are the RMS-11 File Backup, Restore, and Display utilities, respectively. These are typically installed as **BCK**, **RST**, and **DSP**. If they are not installed on your system, ask your system manager whether they are available. You use these in the same manner as any other RSX utility. You can enter a single-line command if the utility is installed, or you can run it (either by invoking it by its installed name or by using a Run command if it is not installed) and then interact with it.

You can use the RMS utilities with both disk and tape volumes. The backup volume must be in Files-11 format, and it must be mounted. This highlights an important distinction. The volume itself is in Files-11 format (it has normal directory structures) but the individual files are not normal Files-11 files. The RMS backup utility writes files onto a backup volume in a special format that can be read by only the RMS restore utility. (If you try to look at a file written by BCK with a command such as **PIP TI:=file**, you will get a lot of garbage on your screen.)

On a disk volume, this special format imposes an overhead of 4 blocks per file. For example, if you use BCK to back up 10 files totaling 40

blocks from your user area onto a disk volume, the copies made will be in 10 files totaling 80 blocks. For very large files (such as a data file with several hundred blocks) these extra 4 blocks are not very significant. For small files, especially if your backup volume has limited storage capacity, these extra blocks are intolerable. Thus, unless you specifically need to use the RMS backup/restore utilities because you have RMS-11 structured files, you will not want to use the RMS utilities for backup onto small disk volumes.

If you are using a magtape volume, things are different. Magtapes have much greater storage capacity than floppy disks or DECtapes. Further, RMS backup to tape offers features not available with either FLX or PIP. In the remainder of this section I will limit our discussion to using the RMS utilities with magtape.

RMS backup to an ANSI tape volume is effected via what is known as a container file. Each backup operation (each command to BCK to copy some of your files) puts one container file on the tape. The basic command form is

```
BCK ddnn:container_file=input_files
```

All the input files that you specify are combined into one large data structure; this is the container file. It is organized in such a way that only the other RMS utilities (DSP or RST) can identify or retrieve the individual files contained therein.

The container file concept may strike you as being confusing, but it is actually very clever. We have discussed the limitations of the ANSI tape format. With RMS, these limitations apply only to the container files, not to their contents. Thus, you will not be able to identify container files by user area, and the creation date of each will be the date on which it was copied (which, in this case, makes sense). Your files do not appear on the magtape as files; they appear as data inside the container file. Thus, they are not affected by the ANSI tape volume format. BCK takes advantage of this to store, along with the contents of your files, complete information concerning your files. Correspondingly, RST extracts this information and uses it when it copies your files back from tape. With the RMS utilities, you can back up files onto magtape and subsequently restore them. All identifying information—user area, file name, file type, version number, creation date, etc.—will be preserved.

When you create a container file, you can obtain a summary listing of which files were backed up into it. You do this as part of the Backup

command. This listing identifies each input file that was put into the container file and thus provides a permanent record of the contents of your backup. If you want to get a summary listing, you should use **BCK** with the Summary Listing switch. The form of this command is

```
BCK ddnn:container_file=input_files/SL:list_file
```

Here, **list_file** is the name of a file (in your default directory) into which the summary listing will be written. If a file of this name already exists in your directory, the summary will be appended to its contents.

A container file is identified by a file name, type, and version according to normal Files-11 conventions. You must specify the file name and type. If you do not specify the version, a default value of zero will be used. To avoid overwriting a previously made container file of the same name, you must specify a version number each time you make a new container file.

Let's consider some examples. Suppose you have mounted a freshly initialized tape onto drive MM0:. The commands

```
MCR>BCK
BCK>MM:PROJECT22.BAK=[351,22]*.*;*/SL:22.B
BCK>MM:SOURCE17.BAK;2=[351,17]*.FTN,*.MAC
BCK>^Z
MCR>
```

will make two container files on your tape. The first will contain copies of all the files in your user area [351,22]. A summary listing of this operation will be written into file 22.B in your default user area. The second command will add a container file to the tape. (BCK writes each new container file after all other container files already on your tape.) This second container file will contain copies of the latest versions of the source files in your area [351,17]. No summary listing will be generated for this.

Despite its special form, a container file is nonetheless a file on an ANSI tape volume. You can determine which container files are on an RMS backup tape by obtaining a simple PIP directory listing. In our example, if you had started with a blank tape, a PIP directory would show a total of two files on the tape:

```
PROJECT22.BAK;0
SOURCE17.BAK;2
```

This is a useful capability, since over the course of months or years,

you might occasionally use BCK to put another container file on the tape. It is easy to lose track of what you have there; this is an easy way to find out.

Of course, PIP can display only the name of each container file—it cannot tell you what is in a file. If all you need is a general idea of the contents, a well-chosen file name will be sufficient. To find out what you actually have in a particular container file (if you do not have a summary listing from when you made it) you can use the RMS Display utility, DSP. (You can also use DSP to find out the names of all the container files, but DSP shows you other information that will not be of interest to you.) To examine the contents of a container file, the general command form is

```
DSP ddnn:container_file/BP
```

Here, the Backup switch (**/BP**) identifies the specified file as being a container file. DSP will list on your terminal the contents of the container file in a style very similar to that used by PIP for directory listings. Referring to our earlier example, the command

```
DSP MM:SOURCE17.BAK;2/BP
```

will produce a listing that might look like this:

```
Contents of container file MMO:SOURCE17.BAK;2

[351,017]PART1.FTN;3          22./22.      1-JAN-1984 14:22
[351,017]BIGSUBS.MAC;11       17./17.      31-DEC-1983 17:57
          etc.
```

This listing gives you more information than the summary listing obtained with the Backup command. You can put this listing of the contents of the container file into a disk file. This is useful if you are running on a video terminal and you want to keep a hard-copy listing of the file contents. You do this by including the name of an output file:

```
DSP outfile=ddnn:container_file/BP
```

For example, if you are on a video terminal, a typical command sequence might look like this:

```
MOU MM:label
BCK MM:container_file=input_files
DSP list_file=MM:container_file/BP
DMO MM:
PRI list_file/DEL
```

This would give you a listing that you could keep with your backup tape so that you would know just what you have on it.

The third RMS utility is RST. This allows you to restore files from one or more container files on tape. You can either restore every file in a container file or specify individual files to be selected from the container file. There are several forms of the Restore command. The basic form is

```
RST output_files=ddnn:container_file/switches
```

The file name and type in **output_files** must be *.*—that is, you cannot rename files when you restore them. You can, however, change user areas. If you specify a ufd in the output file specifier, all the files will be restored into that user area. (As a special case of this, if you omit the ufd, your default directory will be used.) If you use wildcards to specify the ufd, as each file is restored from the container file, it will be put back into its original user area. You will probably want to preserve the user area in all cases, so the Restore command will normally look like this:

```
RST [*,*]*.*=ddnn:container_file/switches
```

(If you are using named directories, the wildcard ufd specifier is [*].)

The switches determine which files are restored from the container file. If you use no switches, the entire contents will be put back onto your system disk. You can use two switches to control this action. The Original Account switch restricts the restoration to files that were originally in a specified user area. The format of this switch is

```
/OA:[ufd]
```

The Select switch allows you to specify up to 10 file specifiers. Here, a file specifier consists only of the file name, type, and version. With this switch, a file will be restored only if it matches one of the specifiers. If you use only one specifier, the format of the switch is

```
/SE:file_spec
```

If you want to have several file specifiers, you enclose them in parentheses and separate them with commas,

```
/SE:(spec1,spec2,..)
```

Let's consider some examples. Suppose your backup tape is mounted on drive MM2. The command

```
RST [*,*]*.*=MM2:SOURCE17.BAK
```

will take all files out of the container file SOURCE17.BAK and restore them into their original user areas on your system disk. The command

```
RST [*,*]*.*=MM2:*.*/OA:[352,21]
```

will go through all container files on the tape and restore all files that were backed up from user area [351,21]. The command

```
RST [*,*]*.*=MM2:SOURCE17.BAK/SE:*.FTN
```

will restore all FORTRAN source files in the container file SOURCE17.BAK. Finally, the command

```
RST [*,*]*.*=MM2:*.BAK/OA:[351,21]/SE:*.MAC
```

will restore, from all container files of type BAK, all MACRO-11 source files that were originally in area [351,21].

23.5 The Backup and Restore Utility

The last utility that we examine for backing up your files is the Backup and Restore Utility, which I will refer to by its MCR name of BRU. BRU is available to you from DCL via the command **BACKUP**. In both MCR and DCL, its name implies that is the utility to use for backup and restore operations. Do not let its somewhat grandiose name fool you. It may be the utility for some backup and restore operations, but not necessarily for yours. BRU was designed to provide an efficient means of backing up large disks. For instance, your system manager probably uses BRU to periodically back up the major disks in your system. Such a backup would include not only all the files in your user area, but also those in everyone else's user area as well. BRU contains many features that are designed around this type of operation. For your purposes, BRU is a much more powerful utility than you need. Unfortunately, when you want to use BRU to do what is comparatively a trivial backup operation, the means of using it are not simplified. You will probably find BRU difficult and awkward to use. If your backup medium is a disk volume, you almost definitely should not use BRU. Anything you can do with BRU, you can also do with PIP. For large file transfers (such as an entire disk) BRU will be significantly more efficient than PIP, but for your purposes, the more important consideration is that PIP will be easier to use. If your backup medium is magtape, however, you might

want to use BRU. In the remainder of this section, I will limit our discussion of BRU to the use of magtape.

BRU has its own format for backup tapes. The tape volume should not be in Files-11 format. If your system is RSX-11M, the tape should not be mounted; if your system is RSX-11M-PLUS or Micro/RSX the tape must be mounted foreign.

When you use BRU with magtapes, backup is accomplished via what is known as a backup set. This is conceptually similar to the container file that we discussed for RMS. A backup set is defined as the entire collection of data that is transferred from disk to tape during one backup operation. On your tape the data is organized as a collection of backup sets, each of which contains as many of your files as you wish. A backup set is identified by a name which can be up to 12 characters long. Since BRU has its own format for tapes, you cannot use PIP to determine which (if any) backup sets are already on a tape. Instead, you must use BRU to do this, as I describe later.

You can use BRU from MCR (via the command **BRU**) or from DCL (via the command **BACKUP**). Although you can access BRU from MCR, it uses DCL style commands. (The common command style notwithstanding, the actual commands used in MCR are not the same as those for DCL.) If you are in MCR, the command style alone will make BRU difficult for you to master. If you are in DCL, this will not bother you. Regardless of which CLI you are in, the generality of BRU will force you to specify things in the command that will, from your viewpoint, be unnecessary.

As noted, whether you are in MCR or DCL, you enter commands in the DCL form. Also, the same basic command form is used whether you are doing a backup or a restore operation. Although the general form is the same, the details do depend on your CLI. The greatest distinction here is that the keywords for some of the command qualifiers change from MCR to DCL.

You can enter a BRU command in any of several manners. Regardless of how you enter it, a command will typically consist of three parts: the command qualifiers (or switches), the input specifier, and the output specifier. These three parts are separated by blanks; these are the only blanks that are allowed in the command line. Thus, if you type a blank in the middle of your qualifiers, the remainder of the qualifiers will be taken as being the input specifier.

In the single-line form, a BRU command looks like this:

```
MCR>BRU/qualifiers input output
DCL>BACKUP/qualifiers input output
```

It is, however, more common to invoke the BRU utility and then enter your command interactively. If you are not familiar with using BRU, this will be the easiest form to use. The multiple-line command form differs somewhat between MCR and DCL. In MCR you first enter the command **BRU**. When you do this, you will get a prompt, at which point you can enter a command. (This may be the entire command, but I will assume that you want to enter it in pieces.) In response to the prompt, you might first enter only the qualifiers. BRU will then prompt you to enter the input specifier, and on the next line it will prompt you to enter the output specifier. Thus, you can enter the command in this manner:

```
MCR: MCR>BRU
     BRU>switches
     From:   input
     To:   output
```

In DCL you use the command **BACKUP**. If you enter the word **BACKUP** by itself, you will have lost your chance to enter any command switches. Instead, you must either enter them on the same line as the command **BACKUP** or end that line with a slash (/) to signify that you want to enter switches. In the latter case, you will be prompted with the word "Qualifier?" At this point, you enter the appropriate switch without its initial slash (the slash ending the previous line is used by DCL for this). If you want multiple switches, they are separated by slashes as usual. Following the switches, the entry of the input and output specifiers is the same as for MCR. Thus, in DCL, you can enter a BRU command in this manner:

```
DCL: DCL>BACKUP/
     Qualifier? switches
     From?  input
     To?  output
```

A BRU command can be rather long. When you use BRU interactively, you can continue a command across several lines. This is independent of whether you enter the command in three separate parts or all at once. To continue a command line being entered to BRU, end the current line with a hyphen (-). BRU will then give you another prompt, at which point you continue. (The use of a hyphen to continue a command line is general DCL syntax; for BRU, you can also use it in MCR.) Again, you must not enter any extraneous spaces. For your purposes, you probably will not need to continue a BRU command (especially if you split it into three pieces) unless you have a long input specifier.

The first part of a BRU command consists of a list of command switches or qualifiers. The qualifiers control various aspects of the data transfer from the input to the output. Each qualifier may specify something about either the input or the output, but the qualifiers are all entered together, regardless of function. Each qualifier is entered in the normal form for entering a command switch:

```
/switch
```

or

```
/switch:value
```

Qualifiers are entered one after the other with no intervening spaces. Each qualifier may be shortened to a three-letter abbreviation.

There are many possible qualifiers to the BRU command. We will discuss only the ones that you will need to use. First, whether you are backing up or restoring your files, you must identify the backup set on the tape. This is done with the Backup Set qualifier. In MCR this is **BACKUP_SET**, and in DCL it is **SAVE_SET**. These are typically abbreviated to **/BAC** and **/SAV**, respectively:

MCR: **/BAC:name**

DCL: **/SAV:name**

The **name** of the backup set may be up to 12 characters long. For a backup operation, it specifies the name to be given to the backup set that will be written to tape. For a restore operation, it identifies the backup set from which files will be copied back to disk.

When you use BRU for backup, you must use the Mounted qualifier. For both MCR and DCL, this switch may be entered as **/MOU**, which informs BRU that the input device (your system disk) is mounted. If you forget this switch when you enter a BRU command to back up your files, you will get an error message for a "Privilege violation" citing I/O error code 16. This is certainly obscure, and you will probably not be able to decipher it as meaning that BRU is trying to treat the system disk as an unmounted device.

Although you must use the Mounted switch for backing up your files, you do not want to use this qualifier when you are restoring files because in that case, the input comes from an unmounted tape volume. Similarly, when you restore files, you must use the No Initialize qualifier. The form of this (**/NOI**) is the same for both MCR and DCL. This specifies that the output device (your system disk) should not be initialized. (Since

you are not privileged, you will not be allowed to initialize the system disk, but you still have to tell BRU that you do not want it to try.) This qualifier is not necessary for backup operations to magtape because the concept of initializing the output device applies only to a disk.

With BRU you can put several backup sets on one reel of magtape. Unlike the other utilities we have examined, BRU does not assume that you want to do this. (Note again BRU's origin as a tool for backing up an entire disk; it is not designed for consecutive small backups such as you will want.) During backup you can use the Append qualifier to specify that you want the new backup set to be written after any others already on the tape. If you forget to specify this and you are at the beginning of the tape, the new backup set will be written over whatever might already be on the tape. Presumably, this is not what you want. Although you can avoid this by using the Append switch, things are not as easy as they might seem. If you are at the beginning of the tape (you have just loaded it onto the tape drive) the Append qualifier will do what you expect: the tape will be moved past all existing backup sets before the new backup set is written. Similarly, if you are at the logical end of the tape (you have just finished writing the last backup set), you can again use the Append qualifier. In this case, BRU will remember that it already is at the end of the tape, and it will simply write out the next backup set. It is when you are somewhere in the middle of the tape that problems arise. If you use the Append switch, BRU will not advance to the end of the tape. Instead, it will give you an error message for a "Tape label error," citing I/O error code 13. (As with the problem that happens if you forget to declare the system disk to be mounted, this is likely to send you screaming to your system manager.) In this situation, you must specify the Rewind qualifier, which will force BRU to rewind the tape, following which the Append switch will be correctly interpreted. In both MCR and DCL, the Append and Rewind switches are entered as **/APP** and **/REW**. You may specify both of these in the same command.

The Append switch has no meaning for a restore operation. You may have to use the Rewind qualifier when you are restoring files. During restore operations, BRU will advance the tape from its current position until it finds the specified backup set or until it reaches the logical end of tape. If you are already past the part of the tape that contains the backup set you want to use (perhaps as the result of previous BRU commands), you will have to rewind the tape. Rewinding a tape typically takes a long time. By not automatically rewinding, BRU can be significantly more efficient for operations such as the creation of several con-

secutive backup sets. In return for this, you have to keep track of where you are on the tape and rewind it as required.

In summary, for backup operations, you will need the Backup Set and the Mounted qualifiers; you will probably want the Append qualifier; and you may also need the Rewind qualifier. The various qualifier combinations that you are likely to use when you back up your files from disk to tape are:

```
MCR: /MOU/BAC:name
     /MOU/BAC:name/APP
     /MOU/BAC:name/APP/REW
     /MOU/BAC:name/REW

DCL: /MOU/SAV:name
     /MOU/SAV:name/APP
     /MOU/SAV:name/APP/REW
     /MOU/SAV:name/REW
```

Note that the last command (Rewind without Append) will erase anything already on the tape when the backup is made. The same effect is achieved with the first command if the tape is already at its beginning.

For restore operations, you will need the Backup Set and the No Initialize qualifiers; you may also need the Rewind qualifier. Possible qualifier combinations for file restore operations are:

```
MCR: /NOI/BAC:name
     /NOI/BAC:name/REW

DCL: /NOI/SAV:name
     /NOI/SAV:name/REW
```

After the qualifiers, the next part of the BRU command is the input specifier. This includes both the input device and also various file specifiers.

When you use BRU for backup, the input specifier names those files that you wish to copy from disk to your backup volume. You must specify the input device as **SY:**, as there is no default. You also must include a file specifier. If you do not, BRU will try to copy every file (not just yours!) from the system disk to your backup volume. When this happens, you will notice that BRU runs for a very, very long time. What is especially frustrating is that you will not be able to abort BRU once it has started—only a privileged user can do this.

In the input specifier you may have as many as 16 file specifiers, which are separated by commas. If necessary, you can continue the input specifier across several lines. For example, if you wanted to back up all

command and text files from your areas [300,1] and [300,2], you could use this input specifier:

```
From: SY:[300,1]*.CMD;*,[300,1]*.TXT;*,-
From: [300,2]*.CMD;*,[300,2]*.TXT;*
```

In this example, I assume that you are using BRU interactively from MCR and that you have already entered the qualifiers, hence the prompt "From:" on the first line above. Note that the first line of input ends with a hyphen. In response to this continuation indicator, BRU repeats its prompt on the next line. The second line ends without a hyphen, signifying the end of the input specifier.

When you use BRU to restore files, the input specifier, along with the backup set defined in the previously entered Backup Set qualifier, names those files that you wish to copy back onto disk. The input specifier is entered in the same way as for a backup command. The device code identifies the particular tape drive in use. The individual file specifiers (if any) identify those files that you wish to extract from the particular backup set. If you do not specify any files, the entire contents of the backup set will be restored.

If you read the Utilities Manual for RSX-11M version 4.0, you will note some confusion on this issue. This manual specifically states that you must restore the entire contents of a backup set from tape to disk and then select (via PIP, for example) the individual files that you want. This is not true. You can select the individual files as part of the restore operation.

In general, the input file specifiers follow the normal rules for specifying files under RSX. There are, however, some important distinctions concerning wildcards. First, as already noted, you must specify the device; presumably **SY:** will suffice. If you specify a ufd and no file name or type, all files in that UFD are used—for example, **[300,1]** is the same as **[300,1]*.*;***. As already noted, if you omit the ufd as well as the file name and type, you will not simply get all the files in your current area—you will get all the files on the entire disk. (This again is based on using BRU for system level rather than individual user backup.) You should be careful not to do this, especially since you might not be able to abort BRU once you have started it. BRU also handles version numbers in a strange way. If you do not specify the version number, you do not get the latest version, as you would with other RSX utilities. Instead, you get all versions. Also, you cannot request the latest version by specifying a version number of zero—BRU will reject this. This is particularly bothersome for backup operations in which the input is on disk and

there is no reason why BRU should not be able to select the latest versions of your various files. You will find these peculiarities of BRU annoying, but you can at least accept them if you remember that they are intended to simplify the use of BRU for backing up an entire system disk.

The last and easiest part of the BRU command is the output specifier. You are allowed to identify only the output device itself. There are no defaults for the device code, so you must specify the tape drive for backup or **SY:** for restore operations. If you have files from more than one user area in a backup set, they will be restored into the appropriate user areas. The file names are automatically set to those of the input files. Again, think big. For restoring files, BRU is designed to restore an entire disk, one user area at a time. As noted above, however, you can pick individual files (or groups thereof by using wild cards) from a backup set.

Let's now put all of this together. Suppose, for example, that you want to back up all the command, FORTRAN, and MACRO-11 source files that are in your default directory. You want to put these into a backup set called VERS03MAR27, which is to be appended to a tape on which you have other backup sets. You have just loaded the tape onto drive MT1: so it is at the beginning of the tape. In MCR, an interactive command sequence to do the backup might look like this:

```
MCR>BRU
BRU>/MOU/APP/BAC:VERS03MAR27
From:SY:*.CMD,*.FTN,*.MAC
To:MT1:
BRU - Starting Tape 1 on MT1:
BRU - End of Tape 1 on MT1:
BRU - Completed
BRU>^Z
MCR>
```

Similarly, in DCL, the command sequence might look like this:

```
DCL>BACKUP /
Qualifier:MOU/APP/SAV:VERS03MAR27
From:SY:*.CMD,*.FTN,*.MAC
To:MT1:
BAC - Starting Tape 1 on MT1:
BAC - End of Tape 1 on MT1:
BAC - Completed
DCL>
```

Note that after accepting your entire command, BRU informs you that it is starting the first backup tape. Again, BRU is designed to back up

large amounts of data and is designed to use, if necessary, several reels of tape. The "end of tape" message tells you that BRU has finished writing to the first reel of tape, not that the tape is full. If it were full, you would be told to load a second volume. The "completed" message tells you that BRU has copied all your files to tape. After completing the backup operation, if you are in MCR, BRU gives you another prompt. At this point you could enter another backup or restore command. To leave BRU, you enter **CTRL/Z**. If you are in DCL, BRU is automatically terminated. Note that the messages you get from BRU are prefaced by the name "BAC"—this is because the DCL intermediate task BAC intercepts the messages from BRU and displays them in its own manner.

Having made this backup set, suppose you later want to restore the single file MAIN.FTN from it. At the time you want to do this, you have already loaded your tape on drive MT0: and used BRU to append another backup set to it. The tape is ready to read, but it must be rewound, since it is at the logical end of tape. Your command sequence would look like this:

```
MCR>BRU
BRU>/NOI/REW/BAC:VERS03MAR27
From:MT:MAIN.FTN
To:SY:


DCL>BAC /
Qualifier? NOI/REW/SAV:VERS03MAR27
From:MT:MAIN.FTN
To:SY:
```

You will find one last command sequence useful for BRU—obtaining a directory of your backup tape. To do this, you use the Directory qualifier. Note that in MCR, this is entered as the switch **/DIR**. In DCL, however, the switch DIR has a different meaning; to get a directory listing you must use the switch **/LIST**. To use the Directory switch, your tape must be at the beginning, so you might as well always use the Rewind switch also. (If your tape is at the beginning you do not need to rewind it, but it does not cost you anything either.) If you do not include the Backup Set qualifier, the directory will list the names of all the backup sets on the tape. If you do specify a particular backup set, then the directory will list all the files (the list will be ordered by user area) in that backup set. When you use the Directory qualifier, the input portion of the command is simply the device code for the tape drive. There is no output specifier, as the listing is always written to

your terminal. If you want a hard copy of the directory listing, you must use a printing terminal. To get a listing of the names of all the backup sets on your tape, you would use this command sequence:

```
MCR>BRU
BRU>/REW/DIR
From: ddnn:


DCL>BACKUP /
Qualifier? REW/LIST
From? ddnn:
```

where **ddnn:** is the particular tape drive. Similarly, to get a listing of all the files in one backup set, you would use this command sequence:

```
MCR>BRU
BRU>/REW/DIR/BAC:name
From: ddnn:


DCL>BACKUP /
Qualifier? REW/LIST/SAV:name
From? ddnn:
```

23.6 Conclusions and Comparisons

In the preceding sections we have examined four possible backup techniques. In terms of capabilities and usage, they differ significantly. You may well have all four of these backup utilities available to you. Which should you use? This decision must be a personal one. To help you, I offer some comparisons and some purely personal preferences.

First, you must decide what medium—magtape, DECtape, floppy, etc.—you will use for backup. It is quite possible that you will have no choice here. If you do have a choice, you should consider the storage capacity of the various volume types. In Table 3 (page 274) I listed the storage capacities of various disk devices. Cartridge disks are not commonly used for single-user backup—it is more likely that you will have DECtape or floppy disks. From the table, a typical volume can hold between 500 and 1000 blocks. By way of comparison, a very short tape (600 feet) at what is today a low recording density (800 bpi) can hold roughly 11,250 blocks. A full-size tape (2400 feet) at what is rapidly becoming the standard density (1600 bpi) can hold eight times as much as this. Thus, one full-size reel of tape can hold the equivalent of any-

where from 45 to 90 DECtapes or floppy disks. If you need to back up large amounts of data (not necessarily all at once, but in total) you will probably want to use magtape. Magtape does, however, have some disadvantages. First, it is much slower than a disk because it must be processed sequentially. Further, not all of the file handling capabilities that you are used to are available with tape. Specifically, files on magtape cannot be identified by ufd, and once on the tape, files cannot be deleted. A final consideration in the choice of medium is whether you might want to use the backup volume as a means of transporting your files to another installation and if so, what devices are available there. Magtape has been, and probably will be for some time to come, the universal medium for transferring data from one system to another.

Once you have chosen a backup medium, you must choose a backup utility. If you are using a disk volume for backup, you should use PIP. There are several reasons for this. First, you are by now already quite familiar with using PIP (or, if you use DCL, the equivalent commands such as **COPY**, **DELETE**, and **DIR**). Why learn yet another set of commands? Further, PIP is more versatile than any of the other techniques. The special wildcards discussed in Section 22.2 alone should convince you of this. For your purposes, PIP should be the best choice.

If you are using magtape, things are not as well defined. Each of the four techniques we have discussed has its own advantages. FLX continues to be widely used for transferring files from one system to another. It does not require the rigmarole of putting tapes into ANSI format. It has a very easy command form, especially for using DOS format (which is what you would use on a magtape). It is capable of distinguishing files by user area. Its main disadvantage is that it does not recognize version numbers. Further, all backup operations get put together into one large collection of files, which, coupled with the version number problem, leads to naming conflicts that are not easily overcome. PIP avoids the version number problem. In return, it requires you to use ANSI tapes, whereby it cannot recognize UFDs. The command syntax is one with which you are already familiar. Thus, the comparison between FLX and PIP should be based on whether you will have multiple UFDs or multiple versions. If you want to back up files from several areas and you do not intend to change any of these files afterward, use FLX. If you are working in only one area but want to make frequent backups as you update your files, use PIP.

If you need to be able to distinguish files by both UFD and version number, neither FLX nor PIP will be adequate. You should use either RMS or BRU. For your purposes, RMS and BRU offer essentially equal

capabilities. RMS requires ANSI formatted tapes; BRU does not. The RMS concept of a container file is equivalent to the BRU concept of a backup set. If you use MCR, you will find that the RMS command syntax closely resembles that of other utilities to which you are accustomed. From MCR, the BRU syntax is strange, but once you get used to it, the interactive procedure renders it tolerable. If you use DCL, the reverse will be true; you will find BRU easier to use. The RMS utilities are designed for backing up RMS-11 files but you can use them for normal Files-11 files without being aware of this. BRU is designed for backing up an entire disk. To use it for backing up just your own files, you have to remember to include some strange switches in your commands. Also, BRU does not allow you to select only the latest versions of your files for backup (you can, of course, purge your files prior to backing them up).

Perhaps the biggest difference between RMS and BRU is the way in which they handle multiple backup sets on one tape. RMS assumes that each backup you make should be put after any others that already exist on the tape. BRU is designed for backing up an entire disk at once, in which case one reel of tape may not be large enough to hold even one backup set. Thus, in BRU, the default is to overwrite any backup set that may already be on the tape. Unless you are working with huge amounts of data, the collection of files associated with one of your projects will occupy only a small amount of one reel of tape. One reel of tape will suffice for all your backups; one time you might put files for one project on it, and some time later you might add files for another project. RMS very nicely matches this style of use. With BRU, if you load your reel of magtape and enter an overly simple backup command, BRU will very quietly destroy all your old backups. You can, as noted, avoid this by using the Append switch, but the problem is that you must remember to add this switch to your command. (If you forget it—too bad.) For this reason alone, I hesitate to recommend BRU. The potential danger is simply too great, especially for casual use where you will not use BRU often enough to remember these details. All in all, RMS is probably more convenient for you to use than BRU, but you should try both and decide for yourself.

24

Object Libraries

When you compile a source file, you make an object file. This object file may then be used as an input to the Task Builder for inclusion in a task. This procedure is straightforward and often will be all that you will ever have to know about object files. The procedure can, however, be made more versatile by using object libraries.

An object library is a file that contains several closely related object modules. (Loosely speaking, a module is a single program unit, such as a subroutine or a function; I will define this more precisely.) For example, RSX contains a special library file of system functions; this file is LB:[1,1]SYSLIB.OLB. (The default file type for an object library file is OLB.) SYSLIB contains various system level functions that, even though you may not realize it, you will typically need when you build a task. Similarly, if you write a program in FORTRAN, you may use various subroutines or functions (for example, the SQRT function) that are supplied as part of the FORTRAN language. The collection of these (and other) functions is known as the Object Time System (OTS), which is contained in its own object library. Depending on whether your system has FORTRAN-IV or FORTRAN-77, the name of this library will be FOROTS or F77OTS. (If FORTRAN is used frequently on your system, your system manager may have added these functions into the main system library SYSLIB.)

SYSLIB and the FORTRAN OTS are examples of object libraries that are supplied as part of your RSX system. Another important use of object libraries is for object modules that you have written. Let's consider a few reasons why you might want to make your own object library. Suppose you write a lot of scientific analysis programs in FORTRAN. Some of these require special mathematical functions that you have to write

yourself. Over the years, you have developed a collection of such functions which are located in various files. Some of these files might be: NORMAL, containing the Normal density and probabilities; BESSEL, containing Bessel functions of order zero; GAMMA, containing the Gamma and log-Gamma functions; and BETA, containing the Beta function, which is evaluated using the log-Gamma function. When you write a program that requires one of these special functions, you must remember in which object file it is located, whether it uses any of your other special functions, and if so, where they are located. If you put all the functions into an object library (call it FUNCS), all you have to do is include this one library file in your Task Builder command. In this case, the main benefit to you of using an object library is that you do not have to choose which of many possible object files to use for making a particular task. As a somewhat different example, suppose you have written a collection of subroutines for using a plotter. To simplify editing and compiling, these are in several files—you might have one for initialization, one for basic pen moving commands, one for axis drawing functions, and one for drawing solid or dashed lines. Each time you build a task to use the plotter, you will probably want to use all of these. In this example, the use of an object library (call it PLOTSUBS) avoids the need to specify each file whenever you do a task build.

In general, it is appropriate to use an object library whenever you have a collection of related functions. The best way to handle these is to have many source files, as this simplifies making or changing an individual function. After compiling, you put all the object modules into one object library. This is done by compiling the source files (as you normally would) and then using a special system utility known as the Librarian to put the modules into a library file. Once you have done this, you no longer need the individual object files, and you can delete them. Typically, the single object library file will occupy less disk space than the collection of object files from which it was made; this savings in space is an additional benefit of using object libraries.

24.1 Using Object Libraries with the Task Builder

In Chapter 17 we discussed the use of the Library switch to denote a particular input file to the Task Builder as being an object library instead of an object file. For extremely simple applications, this may be all that you need to know about using object libraries for building a task. Re-

ferring to our earlier example of the object library FUNCS, if you have written a program to analyze the statistics of harmonic vibrations for suspension bridges, you might use a Task Builder command as simple as

```
MCR: FTB VIBRATE=VIBRATE,FUNCS/LB
DCL: LINK/FAST VIBRATE,FUNCS/LIB
```

In this case, your input consists only of the file containing the main program and your object library. If you follow the general principle of specifying the main program first, everything will be fine. For anything more complicated than this, things might not always work as you would like. To understand why, you must understand how the Task Builder works with object libraries.

Input files specified during a task build may be either object files or object libraries. These are handled differently by the Task Builder. An object file will often contain only one object module. It need not be limited in this manner, however. If it does contain more than one object module, the assumption is that all modules in it are needed. The Task Builder includes all modules in an object file in the task image being built. An object library has a more general purpose nature—it is assumed to contain a possibly large collection of object modules, not all of which may be needed. The Task Builder takes only those modules from an object library that are needed. This is the main reason for using object libraries: it is easier to specify one object library than it is to specify several object files, and this typically results in a smaller task.

What do I mean by a particular object module being needed or not during a task build? To make a task image, you first need a main program—that is, you need an object module that contains a definition of the point at which task execution will start. This main program will typically call several functions (subroutines), and these may in turn call other functions. The name of each function is known as an external reference. (Under RSX, external references may be at most six characters long.) For the task build to succeed, every external reference must be defined somewhere. These definitions come from object modules, which may be in object files or object library files. The modules are taken from the files you specify as input to the Task Builder, as well as from the System Library (SYSLIB), which is automatically included. If a module in a library file contains a definition of a currently unresolved external reference, then the Task Builder puts that module into the task being built; otherwise, the module is not used.

The important words in the above definition are "currently unre-

solved." The Task Builder processes your input files in exactly the order that you specify them. As it goes through each file, it will typically find definitions of some functions, which will resolve some external references. It will also find references to other functions, which will result in new unresolved references. Thus, as the files in your input list (or, more exactly, the object modules in these files) are read, the list of currently unresolved external references changes. When it comes to an object library, the Task Builder knows only about currently undefined references; it does not know about external references that may occur in files it has not yet read. The Task Builder takes only those modules from the library that it currently needs and then goes on; it does not go back to the library to look for other unresolved references. Thus, when you use object libraries, your task build will be sensitive to the order in which you specify your various input files. A Task Builder command with a set of input files specified in one order may succeed whereas another command with the same set of files in a different order may fail. For example, suppose that in the file TEST, you have a main program that calls a function CALC, which is in the file CALC. This function, in turn, uses the Normal probability functions, which are in your object library FUNCS. In this case, the command

```
MCR: FTB  TEST=TEST, CALC, FUNCS/LB
DCL: LINK/FAST  TEST, CALC, FUNCS/LIB
```

will work. What might appear to be an equivalent command,

```
MCR: FTB  TEST=TEST, FUNCS/LB, CALC
DCL: LINK/FAST  TEST, FUNCS/LIB, CALC
```

will, however, fail, for when FUNCS.OLB is processed, the references in CALC to the Normal probability functions will not have been noted.

This sequence of processing by the Task Builder was designed to provide a degree of flexibility that is normally required only for rather sophisticated applications. As far as you are likely to be concerned, it is an aggravation. To avoid problems, you should make your object library the last file that you specify as input when you build a task. Unless you are using more than one object library, you should not have any problems. I noted earlier that the Task Builder always reads the System Library; using similar logic, SYSLIB is read after all the files that you specify.

In its search for definitions of external references, the Task Builder does not go back to an object library once it has left it, but it does go

back and forth within a library. Thus, if the Task Builder extracts a module from a library and this requires other modules in the same library, the Task Builder will also take these, regardless of their relative order within the library. You must worry about the position of a library file specifier in your input list to the Task Builder, but you do not have to worry about the order of modules within the library.

If you have several object libraries, it is acceptable if functions in one library call functions in another. When this happens, you might say that the functions in one library are hierarchically higher than those in the other library. As long as you specify the higher level library first, everything will be all right. (Again, the functions in SYSLIB are considered to be at the lowest possible level, so it is always taken last.) If your libraries cannot be hierarchically divided, things will not be so nice. This would happen if you had a circular structure, where each library called functions in the other. In a case such as this, the best solution is to coalesce the two libraries into one. If you cannot do this, you will have to specify one library, then the other, and then the first again (and possibly the second again, etc.) to ensure that all necessary modules are found.

So far, I have spoken of including an entire object library as an input file. It is also possible to specify only particular modules from a library. When you do this, the Task Builder takes the named modules (whether it currently needs them or not) and ignores the other modules (whether it currently needs them or not). When you specify an object library in this manner, the format is

MCR: **object_library/LB:module:module...**

DCL: **object_library/INCLUDE:module:module...**

In MCR, you still use the switch **/LB**, but in DCL, you use the switch **/INC** instead of **/LIB**. In either case, you follow the appropriate switch by a list of module names, each preceded by a colon.

This ability to force inclusion of specific modules is used with the strictly sequential processing of input files to create special module combinations. Normally, this will be of no interest to you, but there is one notable exception. When you write a FORTRAN program, the OTS includes a number of functions for error checking and handling. For example, as part of the SQRT function, the OTS checks whether the input value is negative or not. If it is negative, the OTS prints an error message on your terminal and takes certain corrective action. The FORTRAN OTS similarly detects and handles a large variety of other errors.

In general, the error message that is printed to you consists of an error number followed by explanatory text. The collection of these text portions is contained in one module; for FORTRAN-77 this module is more than 1,000 words long. Since the maximum task size is 32k words, this represents a sizable fraction of your total task image, which in some cases, you may not be able to afford. You may build your task to include a dummy module that contains no textual descriptions.This forces the exclusion of the normal error message module, thereby saving over 1,000 words of memory. (This is one of the easiest and most profitable ways of reducing the size of a FORTRAN task; it will work for both FORTRAN-IV and FORTRAN-77.) If you elect this option and an error is detected when you run your task, the error number will be printed without any explanation. This is no big problem since all the possible errors are listed in numerical order in the FORTRAN User's Guide (not the FORTRAN Language Reference). To suppress the normal error message module, you force the Task Builder to include the short message module. For both FORTRAN-IV and FORTRAN-77, this is known as **$SHORT**. It is specified to the Task Builder as

```
MCR: LB: [1,1]SYSLIB/LB:$SHORT
DCL: LB: [1,1]SYSLIB/INC:$SHORT
```

or

```
MCR: LB: [1,1]F77OTS/LB:$SHORT
DCL: LB: [1,1]F77OTS/INC:$SHORT
```

or

```
MCR: LB: [1,1]F4POTS/LB:$SHORT
DCL: LB: [1,1]F4POTS/INC:$SHORT
```

depending on whether the FORTRAN OTS is part of your System Library or not. In MCR, a typical example for a system with a separate library for the FORTRAN OTS might look like this:

```
MCR>TKB
TKB>TEST=TEST
TKB>TESTSUBS,FUNCS/LB
TKB>LB:[1,1]F77OTS/LB:$SHORT
TKB>LB:[1,1]F77OTS/LB
TKB>//
MCR>
```

Note that in this example the library **F77OTS** is specified twice: the first time to extract the single module **$SHORT** and the second time to extract all other necessary modules. These two specifications must be in this order. Also note that in this example I use the regular Task Builder, not the Fast Task Builder. The Fast Task Builder supports the Library switch as shown in previous examples; it does not, however, support the module selection form of this switch. To save on task size with this technique, you must tolerate the extra time required to link via the regular Task Builder.

24.2 Modules and Entry Points

Up to now, I have been somewhat vague in my references to object modules and external references (or entry points). It is now time to examine the interrelations and distinctions among files, modules, and entry points. Loosely speaking, a module is a program unit, and an entry point is an address to which a program may jump. In simple examples, these are often synonymous. Suppose you have a file called BESSEL.FTN, which contains a single subroutine called BESSEL. In this case, one module will be associated with the file; this module will be called BESSEL. Further, a single entry point, again called BESSEL, will be associated with the module. The first and most obvious point is that the file name and the module name need not match. The file could be called BSL or GEORGE37 just as well as it is called BESSEL, although these other names might not be as meaningful to you. The module name is taken from the source code and is unaffected by your choice of file name.

The next point is that one object file may contain several object modules. A FORTRAN source file can contain as many subroutines or functions as you wish. When you compile it, you will get one object file, but each of these program units will result in a separate object module in that file. The name of each module will be taken from the corresponding Subroutine or Function statement. This multiplicity of modules per file is not possible with the MACRO-Assembler. By definition, one MACRO-11 source file defines one module. The module name is taken from the **.TITLE** directive. (You can, however, concatenate several object files using the basic PIP Copy command. The result is a single object file containing several modules.)

The next important point is that one module may contain several entry points. This is probably most common in MACRO-11 programs,

where entry points are synonymous with global symbols. Any symbol defined as global in an assembly language source file becomes an entry point in the corresponding object module. The entry point name is the symbol name. It is also possible to have multiple entry points in a FORTRAN module. By default there is always at least one entry point, corresponding to the Subroutine or Function statement that defines the module. By using the Entry statement (which effectively defines other subroutines or functions), it is possible to define other entry points in the module. (Note this feature is available only with FORTRAN-77.)

To summarize, each object file can contain one or more object modules; each module can have one or more entry points. The significance of this lies in the means whereby the Librarian and the Task Builder use object libraries. The Librarian deals with an object library primarily on the module level. It maintains the library as a collection of modules, performing insertions, deletions, or replacements by the module. The Task Builder views the library as being a collection of entry points. It looks through the library to resolve undefined external references. When it finds an entry point that it needs, it copies the corresponding module into the task image being built. To support this dualistic viewpoint, the Librarian maintains two tables, one of module names and one of entry points.

24.3 Making Your Own Object Libraries

The Librarian is the system utility that lets you create and maintain object libraries. In MCR, you invoke the Librarian by the command **LBR**. In DCL you use the command **LIBRARY**, which offers you a limited interface to the Librarian. If you are in MCR, you can use the Librarian in either the single- or multiple-line command form. In DCL, the **LIBRARY** command allows for only the single-line command form. In either MCR or DCL, you can use an indirect task command file with the Librarian, but if you are in DCL, you must use the MCR command forms.

For consistency in my presentation, I will show the single-line command forms for both MCR and DCL. If you are in MCR, you will probably want to use the Librarian in the multiple-line form, since you will typically need several successive commands to accomplish what you want. (Even if you need only one command, you may need several tries to get the syntax correct, since it is somewhat strange.) In this style, you enter various commands until you are done, and then exit via **CTRL/Z**.

An object library is stored in a special format (I have already mentioned the module and entry point tables). To set up the necessary structure for the object library file, you must use the Library Create command. The general form of this is

```
MCR: LBR library/CR:size:ept:mnt
DCL: LIB/CR: (BLOCKS:size,GLOBALS:ept,MODULES:mnt) library
```

Here and in the rest of this section, **library** denotes the name of the object library file. The file type OLB is assumed by default and need not be specified. In both MCR and DCL, **/CR** is the Create switch, which specifies the particular Librarian function to be effected by this command. The remaining parameters specify initial sizes for the library. In MCR they must be in the order shown and separated by colons. In DCL they may be in any order because they are identified by keywords, but the colons, commas, and parentheses are required. If you are willing to accept default values for these parameters, you need not specify them. Since you need to create a library file only once, however, you might as well specify values that will be somewhat more reasonable than the defaults.

The parameter **size** is the number of disk blocks to be reserved for the file, **ept** is the size of the entry point table, and **mnt** is the size of the module name table. It is not always easy to determine how big these should be. You can make reasonable estimates for whatever you plan to put into the library, but if you add more modules in the future, your initial allocations may become inadequate. This is not a disaster, as the Librarian offers you a way of redefining these. Try to make sensible choices that allow for a reasonable amount of growth but that are not excessively large. The size can be any value that you wish; the default is 100. The lengths of the entry point and module name tables are restricted to being multiples of 64, since one disk block can hold 64 table entries. (You can specify a value that is not a multiple of 64, but it will be rounded up to the next multiple.) The smallest values are each 64; it is likely, at least at first, that this will be more than ample for you. The default values are 512 for the entry point table and 256 for the module name table. If you are in MCR, the values that you enter for these three parameters will be assumed to be octal unless you include a decimal point.

For example, suppose you wish to make an object library for special mathematical functions. You have already written several functions (which are in several object files) that you wish to put into the library.

The combined size of the object files is 33 blocks. You plan on adding more functions later so the default size of 100 blocks seems reasonable. You figure that you will probably have at most 20 modules and 30 entry points, so the minimal values of 64 are indicated. FUNCS seems to be an appropriate name given the nature of the object library. To create this object library, you would use the command

```
MCR: LBR FUNCS/CR::64.:64.
DCL: LIB/CRE: (GLOB:64,MOD:64)
```

Note that by not specifying any **size** for the file, you are requesting the default size. If you wish to be more explicit, you could specify it:

```
MCR: LBR FUNCS/CR:100.:64.:64.
DCL: LIB/CRE: (BLOCKS:100,GLOB:64,MOD:64)
```

Note also the decimal points in the values **100.**, **64.**, and **64.** in the MCR forms.

Now that you have set up the object library, you are ready to put some modules into it. Modules are put into an existing object library via the Insert command. (You may, if you wish, put modules into the library as part of the Create command. Because the Create command is so complicated, we have not shown this.) In MCR, this is the default Librarian command. If you enter a command without any command switch, the Insert command is assumed. In DCL you must specify the Insert switch (**/INSERT**). The basic form of the Library Insert command is

```
MCR: LBR library/IN=file_specifier(s)
DCL: LIB/INS library file_specifier(s)
```

The **file_specifier(s)** determine the input files (assumed to be type OBJ) from which object modules are to be taken. Continuing with our example, suppose you have object files corresponding to the source files BESSEL.FTN, GAMMA.FTN, and NORMAL.FTN. BESSEL contains a single subroutine called BESSEL; GAMMA contains two subroutines called GAMMA and GAMLOG; NORMAL contains a function ZNORM and a subroutine PNORM with an entry statement for QNORM. Thus, you have a total of three object files that define five modules and six entry points. To insert all of this, you can use the command

```
MCR: LBR FUNCS/IN=BESSEL,GAMMA,NORMAL
DCL: LIB/INS FUNCS BESSEL,GAMMA,NORMAL
```

Note that the default file types are used for the library and the input files. In the MCR form, you can omit the switch /IN since Insert is assumed by default,

```
MCR: LBR FUNCS=BESSEL, GAMMA, NORMAL
```

You can use the Insert command as often as you like—that is, you can insert some modules now and others later. At no time can a module being inserted have the same name as one that is already in the library, nor can any of its entry points have the same name as an entry point defined by one of the existing modules. If you attempt an insert that would lead to one of these name duplications, the command will be rejected and an error message printed.

The Librarian has two listing commands that show you what is currently in an object library. The List Modules command lists only the module names; the List Entries command lists module names and entry points. In either case, the listing is organized by the module names (which appear in alphabetic order) and is preceded by a few lines that summarize the current status of the library file. The basic form of the List Modules command is

```
MCR: LBR library/LI
DCL: LIB/LIST library
```

The basic form of the List Entries command is

```
MCR: LBR library/LE
DCL: LIB/LIST/NAMES library
```

In these forms, the listing will appear on your terminal. If you want a permanent copy of the library listing, you can specify that it be put into a file instead of on your terminal. In this case, the command forms are

```
MCR: LBR library, listfile/LI
DCL: LIB/LIST: listfile library

MCR: LBR library, listfile/LE
DCL: LIB/LIST/NAMES: listfile library
```

Note that when a listing file is specified, it will be automatically printed unless the No Spool switch (/-SP) is also specified. This switch is available only from MCR,

```
MCR:LBR library,listfile/LI/-SP
MCR:LBR library,listfile/LE/-SP
```

There is a subtle point involved in the generation of a listing file. From the MCR command syntax, we see that the Librarian command involves two output files, where the first is the library file itself. (Although this command does not change the library file, it is nonetheless entered as an output file to conform to the syntax of the other Librarian commands.) In Section 11.3 we discussed the special MCR rules for defaults for multiple input files. Although I did not mention it there (it almost never is an issue), these rules apply to multiple output files as well. Thus, if you omit the device and UFD for the listing file, they will default to that of the library file. If the library file is in someone else's user area, you will be attempting to create the listing file in that area, not yours. Since you will presumably not have write access to that area, your command will fail with a privilege violation. This may leave you rather perplexed, especially if you enter the command from DCL, wherein the form of the multiple output files is totally obscured. This is not as far-fetched as it might at first seem. Once one person has written some useful functions and put them into an object library, it is common for co-workers to use them. (Why reinvent the wheel?) To find out what functions are available in the library, you would use the Library Listing command from your user area and fall into this trap.

Returning now to our example, if you use the List Entries command on your object library file FUNCS.OLB, you will see something like this:

```
Directory of file FUNCS.OLB;1
Object module library created by: LBR V06.00
Last insert occurred 13-FEB-84 at 13:16:59
MNT entries allocated: 64; Available: 59
EPT entries allocated: 64; Available: 58
File space available: 17206 words

** Module:BESSEL
     BESSEL
** Module:GAMLOG
     GAMLOG
** Module:GAMMA
     GAMMA
** Module:PNORM
     PNORM QNORM
** Module:ZNORM
     ZNORM
```

When you put modules into an object library, you should be reasonably certain that they are correct and final. Nonetheless, whether it be to correct errors or to add improvements, you may make changes to your source code after putting the object modules into the library. When this happens, you will want to remove old object modules and insert new ones. You can do this as a two-step (Delete/Insert) operation, but a simpler mechanism is offered by the Replace command.

The Replace command is similar in form to the Insert command in that you can specify several input files, each containing a variety of modules. Each module replaces the module of the same name in the library if it exists; if not, the module is simply inserted. Thus, the command action is actually "Replace if possible; otherwise insert." Note that when you use the Replace command, you cannot select one module from the object file and replace the corresponding module in the object library. All modules in the input file will either replace a corresponding module in the library or be included for the first time. The basic form of the Replace command is

```
MCR: LBR library/RP=file_specifier(s)
DCL: LIB/REP library file_specifier(s)
```

For example, if your original version of the file BESSEL contained only a subroutine that calculated the Bessel function of order 0 and you have now added a second subroutine to calculate the Bessel function of order 1, you could put the new object modules into the library FUNCS with the command

```
MCR: LBR FUNCS/RP=BESSEL
DCL: LIB/REP FUNCS BESSEL
```

Note that in MCR, the Replace command is signified by the switch **/RP** but that in DCL the switch is **/REPLACE** or **/REP**.

The Replace command should handle most of the situations in which you need to delete a module, but there may be times when you need to delete a module without inserting a replacement. You can do this with the Delete command. The form of this command is

```
MCR: LBR library/DE:module:module...
DCL: LIB/DEL library module:module...
```

Here, each **module** is the name of a module to be deleted. The module is removed from the module name table, and all associated entry points are removed from the entry point table.

It is normally necessary to use the Delete command only when you change the organization of entry points within modules. By definition, it is impossible to have a module name duplication when you do a Replace, but you can still obtain an entry name conflict (as when you reorganize things). Continuing with our example, when you first wrote the file GAMMA.FTN, it included two separate subroutines, which led to the two modules GAMMA and GAMLOG, each with a single entry point of matching name. Suppose now that you decide to rewrite these so that they share a certain amount of common code. In this case, you can rewrite the source file as a single subroutine called GAMMA with an Entry statement defining GAMLOG. When you compile this, you will get an object file containing a single module called GAMMA, which defines both entry points. If you attempt to put this into the object library with a Replace command, you will have two conflicting definitions of the entry point GAMLOG. The proper sequence of commands is to delete the module GAMLOG and then replace the module GAMMA:

```
MCR: LBR FUNCS/DE:GAMLOG
     LBR FUNCS/RP=GAMMA

DCL: LIB/DEL FUNCS GAMLOG
     LIB/REP FUNCS GAMMA
```

Alternatively, you may find it more straightforward to delete both old modules and then insert the single new module:

```
MCR: LBR FUNCS/DE:GAMLOG:GAMMA
     LBR FUNCS/RP=GAMMA

DCL: LIB/DEL FUNCS GAMLOG:GAMMA
     LIB/REP FUNCS GAMMA
```

When you delete modules, be it by the Library Delete or the Library Replace commands, it is important to note that the deleted modules are not physically removed from the object library file. The module names and entry points are removed from the library tables so that the object modules can no longer be used, but these modules continue to occupy space in the library file. If you make numerous changes to your library, this wasted space may become significant. You can recover it by using the Library Compress command. Conceptually, the Compress command rewrites the library file, moving all the currently active modules to the beginning of the file so that all the unused space is in one contiguous piece at the end of the file. This compression is not done in place; it is done by copying the active modules into a new object library. Following the compression, the old (uncompressed) object library file remains in your directory, and you should delete it.

Since the Library Compress command creates a new library file, you are given an opportunity to redefine the file and table sizes. Thus, even if you have not deleted any modules, you may need to use the Compress command if your original choice (in the Library Create command) of one or more of the size parameters was too small. In this case, the Compress command functions as an "Expand" command. The form of the Compress command is

```
MCR: LBR new_lib/CO:size:ept:mnt=old_lib
DCL: LIB/CO: (BLOCKS:size,GLOBALS:ept,MODULES:mnt) old_lib
     new_lib
```

Here, **new_lib** and **old_lib** are the file specifiers of the compressed and uncompressed libraries, respectively. In MCR, these must both be specified, even if the file names are to be the same. In DCL, if the new library name is omitted, it is set the same as the old library name. The parameters **size**, **ept** and **mnt** are defined the same as for the Create command except that with the Compress command, the default values are those for the old library file. For example, suppose you have made enough replacements of modules in library FUNCS so that the original allocation of 100 blocks has been exhausted. The total space actually being used is, however, still small so that after compression an allocation of 100 blocks should again be adequate. In this case, you can continue to use the original parameters. The Compress command is simplified by using defaults for these:

```
MCR: LBR FUNCS/CO=FUNCS
DCL: LIB/CO FUNCS
```

As a different example, suppose that you plan to add several functions into FUNCS so that a file size of 300 blocks and module name and entry point table sizes of 128 are more reasonable. In this case, you would use the command

```
MCR: LBR FUNCS/CO:300.:128.:128.=FUNCS
DCL: LIB/CO: (BLOCKS:300,GLOB:128,MODS:128) FUNCS
```

In both examples, when you are finished modifying your library, you should delete the old version of the object library file, which may be done via the command

```
MCR: PIP FUNCS.OLB/PU
DCL: PURGE FUNCS.OLB
```

25

Background Tasks and Batch Processing

So far, most of this book has been concerned with your use of the RSX operating system in an interactive and immediate manner. I stress here the words "interactive" and "immediate." You interact with the operating system: your CLI (or a utility) gives you prompts, asks you questions, and reports errors; you issue commands, supply responses, and take corrective actions. What you do is also immediate: you issue a command and something is done; you then issue another command. At times, it is useful to be able to do things in a noninteractive or nonimmediate manner.

I distinguish between noninteractive and nonimmediate because, although the two concepts are essentially inseparable, they represent different motivations for doing things in this manner. Some examples should clarify this. Suppose you have spent all morning editing programs in several files. You now want to compile all these programs, build a task, and run it. It is lunch time, however, and you do not want to sit at your terminal, entering one command after another. In this example, you would like to be able to do all these things in a noninteractive manner so that you can walk away and have lunch. You would still like things to be done rather quickly so that everything will be finished when you return from lunch. As a different example, suppose you have a task that will take many hours to run (for instance, a lengthy simulation). In this case, your primary interest is in running the task in a nonimmediate manner. It is no problem to interact with the system to enter the command to run the task. You would, however, like to be able to do other things, including logging out, without first waiting for the task to finish.

Loosely speaking, doing things in a noninteractive manner is known as "batch processing" and doing things in a nonimmediate manner is known as "running in background." (In contradistinction, the normal way of running a task is often described as running in foreground.) The RSX family of operating systems offers these capabilities to varying degrees. RSX-11M allows you to run your own tasks in background, but the procedure is somewhat clumsy. RSX-11M-PLUS and Micro/RSX offer a true batch processing capability, which includes running any task in background.

By "running in background" I will refer to two closely related but nonetheless distinct concepts. True background running consists of having your task active at the same time as other user tasks but at a lower priority. Off-hours running consists of running your task only at times when few if any other users will be on the system (such as late at night). In this case, the priority of your task is not significant. You run a task in background when you know that it will take a very long time to run or will, in some other way, use an inordinate amount of system resources. Many PDP-11 systems tend to be dedicated to a relatively small group of users. Thus, a common reason for putting a task in background is out of courtesy to your fellow users. PDP-11 systems are used less often for commercial timesharing, but if this applies to you, you will want to run large jobs during off-hours, as the charges for computer time are typically less. If your system is heavily loaded, a task that you start in background during busy hours may not receive any CPU time until much later in the day. Thus, the net effect will be much the same as if you had simply scheduled your task to start running during off-hours. There is, however, always the chance that the computer system will have unutilized periods during the day, and by having your task in background, it can take advantage of these.

In distinction to background running, batch processing often involves running tasks at the same priority as other users during the busy part of the day. Batch processing is advantageous because it lets you do things without really being there. The entire concept of having batch processing on a multiuser system such as RSX is rather interesting. Originally, computer systems were purely single-user systems. There was no operating system—instead, the user had complete control of the system. Batch processing represented a major advance over single-user systems. Many users submitted jobs (remember decks of punched cards?), which were fed in as a batch to the computer by operators who also returned the answers when they came out on the printer. The next big advance was the multiuser system capable of supporting many user terminals.

Systems such as these (e.g., RSX) eliminated the need to carefully prepare all input in advance, as they offered the ability to interact with the computer. Now, by including a batch processing capability, such multiuser systems are, in a sense, reviving a style of computer usage that was once abandoned as being obsolete. Batch processing is similar to running in background in that you will typically use it when the total running time is expected to be long. Batch processing, however, can be used when this total time is the sum of the running times of many different tasks, each of which may be short or long. Background running allows you to run a single task without being on the system; batch processing allows you to execute a complicated series of tasks.

25.1 Running a Task in Background

RSX includes a special form of the Run command that allows you to run a task in background. This is effected by specifying that the task, rather than being run immediately, is to be started at some time in the future. This is known as a deferred or delayed Run command. You can specify the time at which your task is to start either as an absolute time or as a relative (to the time at which you issue the Run command) value. The time delay may be as small as one second. The magnitude of the time delay is not functionally significant; all that matters is that you are not asking that your task be started immediately. An important aspect of using the delayed Run command is that, after issuing the command, you can log out without affecting the running of the task. This is not true of the immediate Run command discussed in Section 18.1; any task started in this manner will be aborted if you log out before it finishes.

Before discussing the many ramifications of running a task in background via the delayed Run command, I note that this is a privileged command. Further, to run a task this way, the task must be installed, which also is done via a privileged command. Assuming that you are not a privileged user, you must get a privileged user (such as your system manager) to install your task after you have built it and also to run it for you whenever you want to run it in background. (Some installations have added special capabilities to RSX that allow nonprivileged users to install tasks or to run in background. Find out whether this is possible on your system.) In the rest of this section I will assume that, one way or the other, you have the wherewithal to use these commands.

There are two forms of the delayed Run command. You can command

that a task be run at a certain time increment (relative to the time of issuance of the command) in the future. This increment can be measured in hours, minutes, or seconds and cannot exceed 24 hours. The form of this command is

MCR: **RUN task delay**

DCL: **RUN/DELAY: delay task**

The **delay** field is specified as an integer number of time units followed by the unit code, which is **H** (for hours), **M** (for minutes), or **S** (for seconds). Alternatively, you can direct that the task be activated at some absolute time in the future. This command is

MCR: **RUN task hh:mm:ss**

DCL: **RUN/SCHEDULE: hh:mm:ss task**

where the fields **hh**, **mm**, and **ss** specify the absolute (24 hour clock) time of day in hours, minutes, and seconds, respectively. Examples of these delayed Run commands are

MCR: **RUN TEST 6H**

DCL: **RUN/DELAY: 6H TEST**

which requests that task TEST be made active in six hours, and

MCR: **RUN TEST 22:30:00**

DCL: **RUN/SCH: 22:30:00 TEST**

which requests that task TEST be made active at 10:30 P.M. Note that in the MCR command forms, the time specification is separated from the task name by a space, not a slash; this is contrary to the syntax of most other MCR commands.

When RSX processes a delayed Run command, it does not start the task immediately. Rather, it puts the task, along with the desired start time, into a list known as the Clock Queue. This list is used to control all time-delayed task executions. You can use the Show Clock Queue command to see what (if any) requests are in this list. There are no options; the command is simply

MCR: **CLQ**

DCL: **SHOW CLOCK_QUEUE**

On early RSX systems, this is a privileged command although, since it is informational only, there is no need for this restriction. With version

4.0 of RSX-11M, this command was made available to the nonprivileged user as well.

In Section 18.5 we discussed the Abort command. This may be used only to abort an active task. If you have used a delayed form of the Run command to schedule a task and you then decide that the task should not be run, you cannot use the Abort command until the task is active. Rather than waiting for this to happen, you can simply cancel the request to run the task by using the Cancel command. The form of this command is the same for both MCR and DCL,

CAN task

This is a privileged command.

Earlier in this book I introduced the concept of the pseudo device TI:. Whenever a task is run in foreground (via a normal Run command), the terminal from which the Run command is issued is assigned to TI:. Suppose you are on terminal TT2: and you run a task using an immediate Run command. The task does interactive reads and writes to TI:. The operating system translates these into reads and writes to the physical device TT2:. This process is transparent to you; normally you never have to think about it. When a task is in background, however, the concept of TI: becomes meaningless. There is no user, and correspondingly there is no user terminal. The task still may attempt to read or write to TI:, and when this happens, the operating system has to choose some physical device to use. Under RSX, all I/O to TI: for a background task goes to the Console Output device CO:. This device is used to record a log of things that happen in the computer system—for example, all log-ins and log-outs are noted on CO:. (This recording, known as console logging, is not a part of Micro/RSX.) Typically, CO: will be a printing terminal located in the computer room and will be available only to the computer operator or system manager. Thus, when you run a task in background, all output written to TI: will appear in the Console Output log, interspersed among log-ins, log-outs and other messages. Any attempts to read data from TI: will fail unless you have access to CO: (assuming that it is a terminal, not a printer). In effect, a background task cannot use the pseudo device TI:.

The inability to use TI: with a background task is, by itself, not a problem. What is a major limitation is that, although the deferred Run command allows you to put a task in background, it does not provide a mechanism for passing a command line to the task. This severely limits the usefulness of the command. For example, you could use a

delayed Run command to start the Task Builder. Without a command line, this is equivalent to entering the MCR command

```
MCR>TKB
```

In this case, the Task Builder will try to get command input from you via TI:, which it will not be able to do. In normal interactive use, you can use a command such as

```
MCR>TKB @command_file
DCL>LINK @command_file
```

to direct the Task Builder to get all its user input from an indirect command file. There is, however, no analogous capability with the delayed Run command. In general, all system utilities are designed to accept interactive input, be it only a command file specifier. As a result, they cannot be run effectively in background via the delayed Run command.

What the delayed Run command is useful for is running a task that you have made. If it is your task, you can design it so that it does not require interactive input. At the simplest level, you can design the task so that it does not require any input at all. In this case, the input parameters are said to be hard-wired into the program. For example, suppose you wish to simulate the operation of a digital communications system. The input to the receiver is a signal to which random noise is added. When the noise is large, the receiver makes a mistake. You wish to simulate many such events and count how many errors the receiver makes. The randomness of the noise is modeled by using a random number generator. This program requires very few inputs—typically, the level of the signal relative to that of the noise and the total number of trials are sufficient. For a task that might run for several hours (you might want to simulate millions of events), it is not out of the question to change two statements in the source program that specify these values, then recompile and rebuild each time you want to change them. Your task will then be totally self-contained. You can schedule it to start running a few seconds or a few hours later with a deferred Run command. You can then log out from the system and forget about it. When your task starts, it will have all the information it needs.

Although conceptually this may be the easiest way to get input values into your program, it certainly is not the most convenient. It is very unusual for you to want to run a task in background exactly once—that is, for only one set of input parameters. As soon as you start changing

these inputs, the process of changing the source code, recompiling, and rebuilding will become a nuisance. In addition, whenever you do this, you will have to remove the old task and install the new one (see Section 18.3), which can only be done by a privileged user. It is much more common to write your program so that it reads its input values from a data file. In this manner, the data file, not the task, is changed whenever you want to run with a different set of inputs.

The easiest way to set up a program to take input from a data file is to write the program exactly as you would if it were to read its input from your terminal. The form of the Read statements would not change, but the specification of the source of the data would. Correspondingly, the data in the file would appear in exactly the form that you would use if you were to enter it interactively from a terminal. Thus, you could simply use a text editor to create or modify the data file.

The fact that a background task cannot interact with you means more than that it cannot get its input directly from you; it also means that it cannot give its output directly to you. This affects the design of your program in two ways. First, you must write your answers into a file rather than to your terminal. You may be familiar with this procedure. Even if you have a simple interactive task, it may generate large amounts of output. Rather than having it come out on your terminal, you might prefer to have it go into a data file so that you can have the file printed on a line printer (see Chapter 15). The same concept applies to a background task except that it is a requirement, not merely a convenience. What is less obvious is that the manner whereby your task accepts its input also must change. A typical interactive input sequence consists of a question or prompt (e.g., "Enter the value of...") from the task to you followed by a read of the answer that you type in. A background task has nowhere to write these prompts. Thus, you must either leave them out of your program or include them as you normally would but render them ineffective.

If you write a program specifically to be run in background, it is no big problem for you to follow these guidelines in your initial program design. You will not put in any Write statements for prompts. All Read statements for input parameters will refer to a logical unit number that you associate with a disk file. Similarly, all Write statements for your results will refer to another disk file. Without going into programming details, you do this by specifying logical unit numbers (LUNs) in your Read and Write statements. (Note that in FORTRAN, you cannot use the Accept, Type, or Print statements; you must use Read and Write statements). These LUNs are associated with specific disk files by using

a statement that opens the file. Opening a file directs the operating system to find the file (if you specify that it already exists, as you would for the input data file) or to create the file (if you specify that you want a new file, as you probably would for the output data file). Any subsequent I/O requests to the LUN that was specified in the Open statement are then directed to the file. In FORTRAN, you may open a file using the Open statement; alternatively, if you do not wish to do so specifically, the FORTRAN OTS will do so for you, choosing a default file name of FORnnn.DAT where nnn is the three-digit (including leading zeros) unit number. Thus, if you write a FORTRAN program that reads from unit 1 and writes to unit 2 and you do not bother to open data files, your task will read input parameters from the file FOR001.DAT and will write answers into the file FOR002.DAT.

You do not always write a program with the specific purpose of running it in background. Often you will first run it as a normal interactive task. For instance, in our example of simulating errors in a receiver, you might want to run several small cases to make sure that everything is correct before you do any long simulations. In this case, you should write your program so that you can run it as either an interactive or a background task.

One means of switching from foreground to background use is to rewrite the program itself. This implies having several copies of the source code, including one that interacts with TI: and one that uses disk files. This is very inconvenient, especially when you make other changes to your program, as you have to change all copies.

The better method is to make different tasks from one program. You do this by changing device assignments when the task is built. This avoids the problem of maintaining multiple copies of the source code. To do this, you must first be careful in your choice of logical units. The RSX operating system relies heavily on the concept of the user terminal TI:. This uses one logical unit for both terminal input and output. Since the input and the output are to the same physical device (your terminal, whichever one it might be), this is a somewhat logical choice.But input and output are fundamentally different, so it is a poor choice to tie the two together. In your program, you will typically want to have three separate LUNs—one for user output, one for user input, and one for writing answers. You specifically will not want to have both reads and writes to LUN 5 (which is the normal LUN for TI: under RSX), as this will not let you specify different devices for input and output.

As an example of how to do all of this, let's assume that you write your program so that all prompts to the user are written to LUN 1, all

inputs from the user are read from LUN 2, and all answers are written to LUN 3. You use the three lowest possible LUNs because this allows you to decrease the total number of units used by the task, which in turn decreases the size of the task. (Although it is good practice to use the lowest unit numbers, there is nothing special about our assignment of functions to these units; any other permutation would do just as well.) Given a program written in this manner, you can make a completely interactive task with a task build sequence such as this:

```
MCR>FTB
FTB>task=input_files
FTB>/
Enter Options:
FTB>ASG=TI:1:2:3
FTB>UNITS=3
FTB>ACTFIL=3
FTB>//
```

(For simplicity, I have shown only the MCR form. The key point in this example is the use of certain Task Builder options. Even if you are in DCL, you have to enter these in MCR format.) Here, the Assign option (**ASG**) specifies that all three logical units are to be assigned to the pseudo device TI:. Since you are not using any other units, you include the Units option (**UNITS**) which specifies that the task will use a total of only three (rather than the default of six) units. This allows the task to be smaller. Similarly, the Active Files option (**ACTFIL**) specifies that only three (rather than the default of four) files will simultaneously be open. This also saves on task size, as it reduces the number of I/O buffers included in the task. **UNITS** and **ACTFIL** are not critical, but since you will probably put all this into an indirect command file anyway, you might as well include them. (Note that if you have a more complicated program, such as one that reads auxiliary data from other files, you will have to use a larger number of units and possibly a larger number of active files as well. On the other hand, you can reduce the number of units and active files to two if you are really pressed for space in your task by structuring your program so that you first do all the user I/O on units 1 and 2, then close them, and then open unit 1 again, this time for the answers.)

In the example above, the answers are written to your terminal; that is why I called it fully interactive. If the answers are going to be lengthy (for example, a long table of results), you might prefer to write them into a disk file. In this case, you could make a partially interactive task by replacing the single **ASG** option line above with the two lines

```
FTB>ASG=TI:1:2
FTB>ASG=SY:3
```

Finally, if you want the task to run in background, you would use these device assignments:

```
FTB>ASG=SY:2:3
FTB>ASG=NL:1
```

Here, the user input (LUN 2) and the task output (LUN 3) are both assigned to your default disk SY:. The other assignment is a little bit special. It specifies that prompts (LUN 1) are to be written to the Null device. The Null device is a pseudo device that does not correspond to any physical device. Anything that is written to the Null device goes nowhere. The value of the Null device is that you can write to it with normal Write statements, exactly as you would to any other device. Thus, you can effectively disable the writing of prompts without changing the Write statements simply by directing them at task build time to the Null device.

Note in these examples that **ASG** specifies the device to use for a file, but it does not say anything about what the file name itself is. The file name is specified within your program by an Open statement. For example, if you specify via an Open statement that unit 2 is to be associated with file PARAMETER.DAT, the complete file specifiers for LUN 2 will be TI:[ufd]PARAMETER.DAT for the interactive task and SY:[ufd]PARAMETER.DAT for the background task, where ufd is your default directory. By convention, if the device portion of a file specifier is a terminal (TI: or TTnn:) or the Null device (NL:), the remainder of the file specifier is ignored. Thus, the effective file specifier for LUN 2 will be either TI: or SY:[ufd]PARAMETER.DAT. The importance of this is that you need not worry about creating any confusion by opening a file on either TI: or NL:, neither of which are physically capable of holding files.

If you are familiar with some of the more advanced programming techniques for RSX, you may have used some of the Executive References. These are a set of functions that allow you to interface directly with the innards of the operating system. As a final point, note that the Executive Reference RUN (**RUN$** for MACRO-11 programs; **CALL RUN** for FORTRAN) allows you to put an installed task into the Clock Queue. By writing a simple task that uses this function, you can run any installed task on a deferred basis. For some strange reason, you do not have to

be privileged to do this. This offers you a sneaky way around the privilege requirement for the delayed Run command.

25.2 Batch Processing via a Virtual Terminal

Under RSX-11M-PLUS and Micro/RSX, a concept known as the "virtual terminal" offers the user a true batch processing capability. This is true only of these operating systems; none of what we discuss in this section pertains to RSX-11M.

This batch processing capability allows you to do almost anything that you can normally do at a terminal, with the important distinction that you do not have to be logged in to do it. You can schedule the processing of your batch job to be immediate or to begin at any arbitrary time in the future. Your batch job may, for example, perform a lengthy sequence of compilations and task builds, or you may use it to schedule a single task (which need not be installed) for running during off-hours. You do not have to be privileged to use any of these capabilities.

The means by which batch processing is provided under RSX-11M-PLUS or Micro/RSX and the concept of a virtual terminal are inseparable—to understand one, you must understand the other. Consider first what you do at a normal terminal. You log in, enter commands, respond to questions, make decisions based on the results of previous commands, and eventually log out. Suppose that you were to make a record of everything that you typed in while working on the terminal. If you could get a stand-in to enter all this exactly as it appeared in your record, line by line, then you would get exactly the same results without actually being there. This is the principle underlying the virtual terminal. You create a disk file containing lines of input that, in essence if not in fact, look just like lines that you would enter at a terminal. You then direct the operating system to read this file, one line at a time, pretending that you entered each line from a real terminal. A special utility known as the Batch Processor processes each line in a manner similar to the way that MCR handles input from a real terminal. The disk file that is treated as though it were a real terminal is known as a virtual terminal and the sequence of commands that it contains is known as a batch job.

The most important thing to remember about using a virtual terminal is that your batch job must be complete. Whatever you might have to type during a normal terminal session must have a counterpart in your batch file. You should allow for no surprises; you should anticipate anything that might happen and correspondingly provide for it. For example,

if in your batch job you compile a file and then build a task, you must consider the possibility that the compiler will detect errors in your source code. This is not necessarily as difficult as it might sound, although it may seem strange at first.

Just as input from a virtual terminal actually comes from a disk file, output directed to a virtual terminal actually goes into another disk file. This is known as the log file. The log file contains a complete record of what happened when your batch job was run. It includes each command line in your batch job (with the time that it was executed), output directed to the virtual terminal, any data you may supply as part of your job, error messages, identifying information, etc.

Batch processing is controlled by the Queue Manager. As we saw in Chapter 15, the Queue Manager provides a print spooling capability by controlling various Line Printer Processors and various print queues. In an identical manner, it provides a batch processing capability by controlling various Batch Processors and various batch queues. Your batch job is processed by a Batch Processor; it is directed to do so by the Queue Manager. You will notice a distinct similarity between the commands used for print spooling and batch processing, especially since these commands are unlike all the other RSX commands we have studied.

To cause a batch job to be run, you submit it to the Queue Manager via the Submit command. The basic form of this command is the same for both MCR and DCL:

```
SUB batch_file
```

Here **batch_file** is the name of the file containing your batch job; the default file type is CMD. It is important to note that this is the same as the default file type for an indirect command file. This offers a great potential for confusion. It is unfortunate that a different file type, such as BAT, was not chosen as the default. You cannot change this, so you must do your best to avoid mistakes. One technique is to use the file type BAT for all batch job files. This reserves the file type CMD for indirect command files. The disadvantage is that you must remember to specify the file type whenever you use the Submit command. This is not a major imposition, but if you forget to do it, and if you have an indirect command file with the same name, it will be submitted to the Batch Processor. Due to differences in command syntax, the command file will be rejected when the Batch Processor attempts to interpret it as a batch job, but you will not know this until you go looking for your results. A second technique is to use the default file type but to use file names that (to you) distinguish batch job files from indirect command

files. (For example, you might set the first or the last three characters of each batch job file name to be BAT. You could then access all these files via special PIP wildcards, as in **BAT*.CMD** or ***BAT.CMD**.) The disadvantage is that you must choose file names that might not be the most natural or meaningful. I will use the file type BAT for batch job files in our examples, but you may prefer the other technique.

When you submit a batch job, it is put into a batch queue. Your system can have up to 16 batch queues. One, the default queue (BATCH) will always exist. Your system manager may create others. The individual queues are controlled by various Batch Processors; as with print spooling the interrelationships can be intricate. For simplicity, I will assume a one-to-one relationship between Batch Processor and batch queue. Each Batch Processor takes jobs out of its queue one at a time—that is, once a job has started, all other batch jobs after it in the queue must wait until it has finished. Thus, batch processing is not always well suited for running a single long task in background. It might be better to use the delayed Run command discussed in the previous section for this. If you want to use batch processing for a very lengthy job, check with your system manager to find out which batch queue you should submit it to. In the basic Submit command above, the default queue BATCH is assumed. I will assume this in our examples, but I will also show how to specify a different queue.

With the simple form of the Submit command shown above, certain default actions are assumed in addition to the choice of batch queue. For one, the batch job is declared to be eligible to start immediately. Also, a log file will be made. Following termination of the batch job, the log file will be automatically printed and then deleted. The log file will have the same name as the batch file (regardless of that file type), but the file type for the log file will be LOG. Suppose as a simple example that you have a batch job in a file called PROCESS.BAT. In either MCR or DCL, the command

```
SUB PROCESS.BAT
```

will submit this to the Queue Manager, which will give it to the appropriate Batch Processor for immediate processing. The job will be identified by the name PROCESS. A log file named PROCESS.LOG will be opened, and all commands, output and messages will be written into this file. When the job finishes, the file PROCESS.LOG will be spooled (sent to the Line Printer Processor), and after being printed, it will be deleted.

If you want to override any of these defaults, you must use a slightly

more elaborate form of the Submit command. The general command form is*

MCR: **SUB queue:job_name/options=batch_file**

DCL: **SUB/QUEUE:queue/NAME:job_name/options batch_file**

Here, **queue** is the name of the batch queue that you want to use. In MCR the terminating colon is required syntax; if you do not specify the queue, you also omit the colon. The **job_name** will be given to the batch job itself and to your log file. You can omit both the queue and job name specifiers but still include options. In that case, the command form is

MCR: **SUB /options=batch_file**

DCL: **SUB/options batch_file**

There are several possible options, but only three are likely to be of value to you. The After option allows you to control when your job will start. You specify that your job is not to start until after a given time. (There is, of course, no guarantee that it will start at that time—if your system is heavily loaded, your job may be further delayed.) Whether you are in MCR or DCL, the form of this option is cumbersome:

MCR: **/AF:hh:mm:dd-mmm-yy**

DCL: **/AFTER: (dd-mmm-yy hh:mm)**

Here, **hh** and **mm** specify the time of day in hours and minutes, **dd** is the number of the day, **mmm** is the first three letters of the month (in English), and **yy** is the year. All of this, including the plethora of punctuation, must be included, with the exception of **hh** and **mm**. If you omit these, the time of day will default to midnight (in MCR, all three colons must still be included). There is no way to make the date default to the current date; you must specify it. Note that this is exactly the same as for the Print command.

The other two options that you may find useful allow you to control the processing of the log file. In the Submit command, these are both appended to the **job_name** specifier. The No Log File option specifies that you do not want a log file. (Unless you are sure that everything will work properly, you should not suppress creation of the log file. It is safer to let it be made and subsequently delete it.) For both MCR and DCL, this option is simply **/NOLO**. The No Print option specifies that the log file should not be spooled and then deleted. Instead, it is made a permanent file in your user area. With this option, you may elect to

print the log file only if something did not work right and you need to see all the details. For both MCR and DCL, this option is simply **/NOPRIN.**

Let's continue with our example of a batch job in the file PROCESS.BAT. Suppose that you want to run this several times overnight. If the current day is June 25, 1985, you might use a series of Submit commands such as these:

```
MCR: SUB TRY1/NOPRIN/AF:22:00:25-JUN-85=PROCESS.BAT
     SUB TRY2/NOPRIN/AF::::26-JUN-85=PROCESS.BAT
     SUB TRY3/NOPRIN/AF:02::26-JUN-85=PROCESS.BAT

DCL: SUB /NAME:TRY1/NOPRIN/AF: (25-JUN-85 22) PROCESS.BAT
     SUB /NAME:TRY2/NOPRIN/AF: (26-JUN-85) PROCESS.BAT
     SUB /NAME:TRY3/NOPRIN/AF: (26-JUN-85 02) PROCESS.BAT
```

This will cause a total of three jobs to be scheduled by the Queue Manager. Job TRY1 will start after 10 P.M.; TRY2 will start after midnight, and TRY3 will start after 2 A.M. Log files TRY1.LOG, TRY2.LOG, and TRY3.LOG will be made and kept in your user area; they will not be printed.

So much for the mechanics of submitting a batch job. How do you prepare the job itself? In many ways, this is similar to writing an indirect command file. If you are familiar with using indirect command files (Chapter 20), you should have no problems (other than adapting to yet another syntax) mastering the use of batch jobs. Unfortunately, many of the fancy things that you might be used to doing with command files cannot be done with batch jobs.

The Batch Processor treats the contents of your batch file as a sequence of input lines. There are three basic categories of lines—special commands to the Batch Processor itself, commands to your CLI, and data for tasks that might need it. All commands, whether they are for the Batch Processor (Batch Specific commands) or your CLI, must begin with a dollar sign ($). All data must appear exactly as the task that will read it expects to see it.

The Batch Specific commands are interpreted directly by the Batch Processor and, as such, do not depend on whether you are in MCR or DCL. Commands for your CLI are entered in exactly the same way that you would enter commands for interactive use, except that each must be prefaced with a dollar sign.

Remember that your batch job must duplicate everything that you would do at a terminal. This includes logging in and out. For a virtual terminal, however, you use special commands for these. The first com-

mand in your batch file must be a Job command, and the last should be an End-of-Job command. These replace the Log-In and Log-Out commands that you would use at a normal terminal. These commands are simply

```
$JOB
```

and

```
$EOJ
```

Between the Job and End-of-Job, commands you can have any mixture of command and data lines that makes sense. This includes the @ command (remember to put a $ in front of it). Thus, you can invoke indirect command files from within a batch job.

Before going any further, let's consider a simple example. For brevity, I will present only the MCR command forms. Suppose you are developing a task called TASK22 that will process data in a file and write results into the file TASK22.OUT. The task is to be built (using the indirect command file TASK22.CMD) from the FORTRAN main program MAIN22 and from the Macro-11 files FILEIN and GETVALUES. After rewriting the source files, you want to compile, build, run the task, and then print the answers. A batch job to do this might look like this:

```
$JOB
$F4P MAIN22,MAIN22/-SP=MAIN22
$MAC FILEIN=FILEIN
$MAC GETVALUES=GETVALUES
$FTB @TASK22
$RUN TASK22
$PRI TASK22.OUT
$EOJ
```

If you were to put these commands in a file called TASK22.BAT, after editing you could do this:

```
MCR>SUB TASK22.BAT
MCR>BYE
```

After logging out, you could go have lunch, come back sometime later, and (presumably) find your results waiting for you.

To stress the point I made earlier, if you were to enter the commands

```
MCR>SUB TASK22
MCR>BYE
```

the file TASK22.CMD would be submitted to the Batch Processor. This would result in the immediate generation of the error message

```
BPR -- Syntax error -- $JOB does not appear first
```

which would be written into your log file. Following this, the job would be terminated. This is a common error. There is nothing inherently wrong with having files named TASK22.BAT and TASK22.CMD, just as there is nothing wrong with using the names TASK22.TSK and TASK22.OUT—the file names are all the same to indicate the interrelationship of the files. You just have to be careful.

In Section 25.1 we discussed some techniques for making data available to a task started by a delayed Run command. When you use a Batch Processor to run a task, a much simpler procedure is possible. You simply put the data into the batch job. Remember the principle behind the use of a virtual terminal. At a real terminal, you would type in the Run command and then type in data values as your task requested them. Correspondingly, in your file, you have one line containing the Run command, and the next lines have data values as required. Thus, the general format for including data for a task is:

```
$RUN task
data
.
.
.
data
$next command
```

Note that the **data** lines do not start with a dollar sign. All lines until the next line starting with a dollar sign are assumed to contain data. The number of lines of data that you include in your batch job should be the same as the number that the task will want to read.

The use of data in a batch job is not restricted to the running of one of your tasks. You may also supply data in this manner to system utilities. For example, you could start the Task Builder and then have data lines that would specify which files and options to use. If you did not have an indirect command file for building TASK22 in our example, you could instead do something like this:

```
$FTB
TASK22=MAIN22
FILEIN,GETVALUES
/
UNITS=8
```

```
ASG=SY:7:8
ACTFIL=6
//
$RUN TASK22
```

For many simple applications, what we have discussed so far—preparing a batch job, including the Job and End-of-Job commands, the CLI commands, and data, and submitting the job—is all that you need to know about using the Batch Processor. As such, it is a simple, convenient, and powerful tool. In the remainder of this section, we discuss error handling and control flow. You may need to know about these, but in general, they will not be as useful to you.

Let's return to our example of TASK22. What we have discussed is fine if everything works right. What happens if, for example, one of the compilations or assemblies fails due to an error in your source code? If you were working interactively at a normal terminal, you would probably determine the error in your program, correct it by editing, recompile, and then proceed. This is clearly too versatile a procedure to emulate on a virtual terminal. What can you do when errors occur during a batch job?

With the Batch Processor, the default action is to stop the job when an error occurs. Thus, in our example, if you had made some mistakes when you last edited MAIN22.FTN so that the compiler returned an Error status, the batch job would stop at that point. You can override this default by specifying that you want processing to continue regardless of errors. The rest of the job—the assemblies, task build, and execution—would then be done. Neither of these options is what you would really like. With the default, the assemblies are not done, so you cannot find out whether either FILEIN.MAC or GETVALUES.MAC also contain errors. If you override the default, the file MAIN22.OBJ used in the task build will not be what you expect it to be, making it meaningless or even dangerous to run the task.

In general, if your batch job compiles several files and then builds a task, what you would like is to do all the compilations, then note if an error occurred in any one of them, and proceed only if all of them were successful. There is no simple or sensible way of doing this with the Batch Processor. In a batch job, you can detect an error when it occurs, but you cannot remember it—you must act on it immediately. Further, the available actions are extremely limited. In contradistinction, the Indirect Command Processor is much more powerful. In an indirect command file, you can set and test logical symbols to achieve the desired processing flow. (Specifically, you would initialize a logical symbol in-

dicating success to the value True; you would then use the AND operator to accumulate the success of each compilation. The task build would then be conditional on the value of this symbol—i.e., on the success of all the preceding operations.) What you will find is that if you want to do anything fancy in a batch job, you will do it via an indirect command file that you run from within the batch job. The batch job itself then becomes just a shell that allows you to execute the command file in batch mode.

Although the capabilities are limited, it is still worthwhile to discuss how you can control the sequence of operations in a batch job. In general, any task run under RSX is able to return an exit status when it finishes running. (We discussed this concept, as it relates to indirect command files, in Section 20.3.) This status allows the task to inform the operating system whether it encountered any problems and if so, how severe they were. A task may return four possible status values: SUCCESS, WARNING, ERROR, and SEVEREERROR. You can assume that all system utilities return a status value. Typically, a task that you write will not return a status value, although you may make it do so via the Executive Reference "Exit with Status." A task that returns no status is assumed by the Batch Processor to have returned the value SUCCESS.

Your batch job is structured as a sequence of commands. Some of these are commands to the Batch Processor itself, and some are CLI commands. After each CLI command, the Batch Processor examines the status that was returned and uses this value to control further operations. Loosely speaking, this is what happens. Each time a status value is returned, a table of possible conditions and actions is examined to determine what to do next. This table contains entries for each of the three error codes (WARNING, ERROR, and SEVEREERROR). For each status code, the corresponding entry specifies whether the condition should be ignored or not and if the condition is not to be ignored, what should be done if it occurs. The Batch Processor maintains a list of tests and actions that it uses by default unless you direct otherwise. You can define the defaults by using the batch specific commands **$ON** and **$SET**, and you can override a default for one step in the job by using the command **$IF**. When your batch job starts, the default tests are initialized to enable the recognition of all errors and the default actions are initialized to stop the batch job. In our example of TASK22 I did not use any On, Set or If commands, so these initial defaults apply for the entire job. After the FORTRAN compilation, for example, the returned status will be examined. If it is any one of the three error conditions, the Batch Proc-

essor will stop the job. Only if the status is SUCCESS will the batch job be allowed to proceed to the next instruction (the first assembly). The status returned by this assembly will similarly be examined, etc.

In many cases, this stopping of the batch job upon detection of any error condition will be perfectly acceptable. You may, however, wish to have more control over what happens. For example, in a FORTRAN program, you may use a variable name longer than six characters; the compiler will generate a warning and use only the first six characters. If you like to use long variable names, you certainly would not want to have the job stopped upon generation of a warning by the FORTRAN compiler. Going one step further, you might want to ignore all warnings.

You can use the Set command to disable or reenable the default tests. You can use the On command to change the default tests and actions. You can use the If command to temporarily override any or all of the defaults. A Set or an On command applies until (if ever) it is changed. An If command applies to only the command that last returned a status and always overrides any On or Set commands.

The Set command can be either

```
$SET NO ON
```

which disables all the default tests, or

```
$SET ON
```

which enables them. The Set command does not change the details of the default tests. It just determines whether the tests are to be done or not.

The On and If commands have the same general form:

```
$ON status THEN action
$IF status THEN action
```

The possible status values that you can specify and actions that you can declare are the same for both commands. The **status** may be **WARNING**, **ERROR**, or **SEVEREERROR** (no space); these may be shortened to **WAR**, **ERR**, or **SEV**. Note that you cannot test for Success. You can declare three possible actions. You can stop the job (**STOP**); ignore the error (**CONTINUE**); or transfer control to another part of the job (**GOTO label**).

The On command sets the default action to be taken when the returned status is equal to or worse than the status you specify. It also

declares that a status better than the one you specify is to be ignored. For example, the command

```
$ON ERROR THEN STOP
```

sets the default tests to "Ignore a status of SUCCESS or WARNING; stop the job if the status is ERROR or SEVEREERROR." Normally, this example will be the only form of the On command that you will use. The If command overrides the default action for the particular status code specified. Probably your most common use of the If command will be

```
$IF WARNING THEN CONTINUE
```

You would use this command, for example, after a FORTRAN compilation to ignore warnings. It would not affect the default action (presumably **STOP**) for errors or severe errors.

The On and If commands allow you to specify **GOTO label** as an action to be taken. Somewhere in the batch job **label** must be defined in the command

```
$label:
```

The name of the label may be from one to six characters long. You will probably use the command **GOTO** only if you want to do something special (such as printing certain files) before terminating the job.

The **STOP**, **CONTINUE** and **GOTO** actions that you use in the On and If commands can also be used by themselves as batch-specific commands. Their forms remain the same except that, when used as separate commands, they are preceded by the mandatory dollar sign.

To illustrate these commands, we consider again our example of TASK22. Suppose that while the task is running it stores intermediate data in a temporary file. The purpose of this data is to help you debug your program if it fails. If the program runs properly, you do not need this information. The program can detect errors and returns an appropriate status value when it exits. You might modify our batch file shown earlier so that after the Compile and Build commands it looks like this:

```
$RUN TASK22
$IF SEV THEN GOTO DUMP
$IF ERR THEN GOTO DUMP
$PIP *.TMP;*/DE
$STOP
$DUMP
```

```
$PRI DEBUG.TMP/DEL
$STOP
$EOJ
```

This batch job will run your program and print the temporary file only if your program returns a status signifying an error.

Index

Related Titles Available from Digital Press

Working with RT-11, by David Beaumont, Anne Summerfield, and Julie Wright.

For new or potential users of this popular PDP-11 operating system.
$24.00 Order number EY-00021-DP.

Programming with RT-11. Volume I: Program Development Facilities, by Simon Clinch and Stephen Peters.

$28.00 Order number EY-00022-DP.

Programming with RT-11. Volume II: Callable System Facilities, by Stephen Peters, Kevin Small, Anne Summerfield, and Julie Wright.

Using RT-11 programming facilities and system services to build BASIC, FORTRAN and MACRO programs.
$32.00. Order number EY-00023-DP.

Tailoring RT-11: System Management and Programming Facilities, by Simon Clinch, Stephen Peters, Kevin Small, and Anne Summerfield.

A resource book for systems managers and advanced programmers.
$36.00. Order number EY-00024-DP.

Structured Programming in MACRO-11, by Bob Southern.

Hands-on instruction in assembly language programming for beginners. Packed with exercises and examples.
$21.00. Order number EY-00032-DP.

Designing Applications for the Professional 300 Series: A Developer's Guide, by John Lucas.

A guide for designers and programmers of applications intended to run under the RSX-like P/OS operating system.
$35.00. Order number EY-00030-DP.

Digital Press books are available at your local technical bookstore. To order by Mastercard or VISA, call toll free 1-800-343-8321, or write Digital Press Order Processing, Digital Equipment Corporation, 12A Esquire Road, Billerica, Massachusetts 01862.
Prices listed are U.S. list prices only and are subject to change without notice. For current prices and further information, call 617/663-4152.