UNISYS

A Series
Task Management
Programming Guide

Release 3.9.0

Priced Item

September 1991

Printed in U S America
8600 0494–000

UNISYS

Product Information Announcement

o New Release o Revision ● Update o New Mail Code

Title

A Series Task Management Programming Guide

This Product Information Announcement announces Update 1 to the September 1991 publication of the *A Series Task Management Programming Guide*. The update is relative to the Mark 4.0.0 System Software Release, dated July 1992.

The major changes described in this update to the reference manual are the following:

- Tasking security status

  This new category of security status enables a program to execute with most of the privileges of a message control system (MCS), without actually being an MCS.

- CALLCHECKPOINT

  User programs can now initiate a checkpoint by invoking the exported MCP procedure CALLCHECKPOINT. The CALLCHECKPOINT procedure is available to user programs written in all the languages that support libraries.

- OSI remote tasking

  A Series remote tasking is now supported across Open Systems Interconnection (OSI) networks.

- MP (Mark Program) system command

  This system command has been extended to replace the functions of the CP (Control Program), MC (Make Compiler), and PP (Privileged Program) system commands.

- C libraries

  It is now possible to specify whether a C library has a temporary or permanent freeze. Further, it is now possible to pass parameters of ALGOL type Boolean to a C library parameter of type int.

- Library usage

  The Y (Status Interrogate) system command now displays the user programs linked to a library. A program can obtain the same information through a new form of the type 0 (Mix Entries) GETSTATUS call.

- Test and Debug System (TADS)

  TADS is now provided for the C and COBOL85 programming languages.

Various technical changes have been made to improve the quality and usability of the document.

Remove	Insert
iii through iv	iii through iv
xiii through xxiv	xiii through xxiv
3–5 through 3–6	3–5 through 3–6

continued

Announcement only:

Announcement and attachments:
AS199

System: A Series
Release: Mark 4.0.0 July 1992

Part number: 8600 0494–010

Remove	Insert
3–15 through 3–16	3–15 through 3–16
4–1 through 4–2	4–1 through 4–2
	4–2A through 4–2B
4–3 through 4–4	4–3 through 4–4
5–3 through 5–4	5–3 through 5–4
	5–4A through 5–4B
5–5 through 5–6	5–5 through 5–6
5–9 through 5–10	5–9 through 5–10
5–13 through 5–18	5–13 through 5–18
6–5 through 6–6	6–5 through 6–6
6–11 through 6–16	6–11 through 6–16
7–1 through 7–6	7–1 through 7–6
8–3 through 8–4	8–3 through 8–4
9–9 through 9–10	9–9 through 9–10
	9–10A through 9–10B
10–5 through 10–6	10–5 through 10–6
10–13 through 10–16	10–13 through 10–16
	10–16A through 10–16B
11–1 through 11–10	11–1 through 11–10
	11–10A through 11–10D
11–15 through 11–20	11–15 through 11–20
12–1 through 12–2	12–1 through 12–2
12–7 through 12–8	12–7 through 12–8
	12–8A through 12–8B
12–9 through 12–10	12–9 through 12–10
16–21 through 16–22	16–21 through 16–22
18–3 through 18–10	18–3 through 18–10
	18–10A through 18–10B
18–11 through 18–12	18–11 through 18–12
18–21 through 18–22	18–21 through 18–22
18–41 through 18–42	18–41 through 18–42
	18–42A through 18–42B
18–45 through 18–52	18–45 through 18–52
Glossary–19 through 24	Glossary–19 through 24
Bibliography–1 through 4	Bibliography–1 through 2
Index–1 through 24	Index–1 through 24

Changes are indicated by vertical bars in the margins of the replacement pages.

Retain this Product Information Announcement as a record of changes made to the base publication.

To order additional copies of this document

- United States customers call Unisys Direct at 1-800-448-1424
- All other customers contact your Unisys Subsidiary Librarian
- Unisys personnel use the Electronic Literature Ordering (ELO) system

UNISYS

A Series
Task Management
Programming Guide

Release 3.9.0

September 1991

Printed in U S America
8600 0494–000

Priced Item

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication may be forwarded using the Product Information card at the back of the manual, or may be addressed directly to Unisys, Product Information, 25725 Jeronimo Road, Mission Viejo, CA 92691.

Page Status

Page	Issue
iii through iv	–010
v through xii	–000
xiii through xxi	–010
xxii	Blank
xxiii	–010
xxiv	Blank
1–1 through 1–11	–000
1–12	Blank
2–1 through 2–25	–000
2–26	Blank
3–1 through 3–4	–000
3–5 through 3–6	–010
3–7 through 3–14	–000
3–15 through 3–16	–010
3–17 through 3–28	–000
4–1 through 4–4	–010
4–5 through 4–28	–000
5–1 through 5–2	–000
5–3 through 5–6	–010
5–7 through 5–8	–000
5–9 through 5–10	–010
5–11 through 5–12	–000
5–13 through 5–17	–010
5–18	Blank
6–1 through 6–4	–000
6–5 through 6–6	–010
6–7 through 6–10	–000
6–11 through 6–15	–010
6–16	Blank
7–1 through 7–5	–010
7–6	Blank
8–1 through 8–2	–000
8–3 through 8–4	–010
8–5 through 8–8	–000
9–1 through 9–8	–000
9–9 through 9–10B	–010
9–11 through 9–16	–000
10–1 through 10–4	–000
10–5 through 10–6	–010
10–7 through 10–12	–000
10–13 through 10–16B	–010
10–17	–000

continued

Page Status

continued

Page	Issue
10–18	Blank
11–1 through 11–10D	–010
11–11 through 11–14	–000
11–15 through 11–20	–010
11–21 through 11-22	–000
12–1 through 12–2	–010
12–3 through 12–6	–000
12–7 through 12–10	–010
12–11 through 12–15	–000
12–16	Blank
13–1 through 13–4	–000
14–1 through 14–2	–000
15–1 through 15–5	–000
15–6	Blank
16–1 through 16–20	–000
16–21 through 16–22	–010
16–23 through 16–24	–000
17–1 through 17–39	–000
17–40	Blank
18–1 through 18–2	–000
18–3 through 18–12	–010
18–13 through 18–20	–000
18–21 through 18–22	–010
18–23 through 18–40	–000
18–41 through 18–42B	–010
18–43 through 18–44	–000
18–45 through 18–52	–010
18–53 through 18–79	–000
18–80	Blank
19–1 through 19–20	–000
20–1 through 20–2	–000
Glossary–1 through 18	–000
Glossary–19 through 23	–010
Glossary–24	Blank
Bibliography–1 through 2	–010
Index–1 through 24	–010

Unisys uses an 11-digit document numbering system. The suffix of the document number (1234 5678–*xyz*) indicates the document level. The first digit of the suffix (*x*) designates a revision level; the second digit (*y*) designates an update level. For example, the first release of a document has a suffix of –000. A suffix of –130 designates the third update to revision 1. The third digit (*z*) is used to indicate an errata for a particular level and is not reflected in the page status summary.

About This Guide

Purpose

This guide describes the following types of operating system features that can be accessed using programming languages:

- Tasking features

  These are features that enable processes to initiate, monitor, and control other processes. Examples of such features are the CALL, PROCESS, and RUN statements, task variables, and task attributes. Features related to job restarting and process history also fall into this category.

- Interprocess communication features

  These are features that enable user-defined information to be passed between processes, or that help regulate the timing of parallel processes. Examples of such features are events, libraries, and parameter passing.

Scope

This guide introduces basic concepts about tasking and interprocess communication, and gives examples of many of the concepts discussed. However, the detailed reference information for many of these topics resides in other manuals. Of particular note are the following topics:

- Task attributes

  This guide introduces the functions of many task attributes. However, the full descriptions of these attributes reside in the *A Series Task Attributes Programming Reference Manual*.

- Port files

  This important interprocess communication technique is only briefly introduced in this guide. For a full description of how to use port files, refer to the *A Series I/O Subsystem Programming Guide*.

- Job queues

  These mechanisms for controlling the behavior of Work Flow Language (WFL) jobs are described in detail in the *A Series System Administration Guide*.

- Programming language syntax

  This guide discusses features that are available in the A Series implementations of a number of programming languages, including ALGOL, COBOL74, and WFL. The detailed syntax information for these features is in the manuals for these programming languages.

Audience

The audience for this guide consists of applications programmers who are familiar with at least one high-level programming language, such as ALGOL, C, COBOL74, FORTRAN77, Pascal, or WFL.

Prerequisites

Before reading this guide, you should have a general familiarity with the concepts discussed in the *A Series Systems Functional Overview*.

How to Use This Document

Because this guide addresses a variety of tasking and interprocess communication techniques, it is unlikely that you will need to read the whole guide. For an overview of tasking techniques, refer to Section 1,"Understanding Basic Tasking Concepts." For an overview of interprocess communication techniques, refer to Section 13, "Understanding Interprocess Communication." The information in these sections should help you determine which of the other sections are relevant to your immediate needs.

Error messages related to tasking and interprocess communication are discussed throughout this guide. The index at the end of this guide includes all these error messages, and refers to the pages where they are discussed.

Statements about ALGOL in this guide apply also to DCALGOL, DMALGOL, and BDMSALGOL unless otherwise specified.

The ANSI-68 version of COBOL is referred to as COBOL(68) in this guide. This convention helps to distinguish between ANSI-68 COBOL and the newer COBOL implementations (COBOL74 and COBOL85).

Organization

This guide is divided into the following parts and sections.

Part I. Tasking

The sections in this part describe basic concepts and features related to initiating, monitoring, and controlling processes.

Section 1. Understanding Basic Tasking Concepts

This section defines procedures, procedure entrance and initiation, task variables, and task attributes. This section also gives an overview of the benefits and limitations of tasking.

Section 2. Understanding Interprocess Relationships

This section discusses the concepts of inclusion, dependency, flow of control, jobs and tasks, coroutines, process families, and predefined task variables.

Section 3. Tasking from Interactive Sources

This section explains how to initiate, monitor, and control processes by using the Command and Edit (CANDE), Menu-Assisted Resource Control (MARC), and operator display terminal (ODT) interfaces. This section also discusses the inheritance of task attributes from each of these sources, and introduces task attributes that enable a process to communicate with an operator.

Section 4. Tasking from Programming Languages

This section discusses the tasking capabilities of major languages. Capabilities discussed include initiating various types of processes and using task attributes. Some examples are also given.

Section 5. Establishing Process Identity and Privileges

This section introduces the task attributes, such as USERCODE, MIXNUMBER, and NAME, that establish process identity. This section also discusses the various categories of security status that a process can have, and explains how to assign such a security status to a process.

Section 6. Monitoring and Controlling Process Status

This section discusses the various states a process can have, such as active, scheduled, suspended, and terminated. This section also explains the methods of monitoring and controlling process state using programming languages.

Section 7. Controlling Processor Usage

This section discusses task attributes that affect process priority and record or limit processor usage. This section also relates processor usage to the various accounts that can be displayed through system commands.

Section 8. Controlling Process Memory Usage

This section discusses task attributes that affect memory estimates for a process and limit the memory usage of a process.

Section 9. Controlling Process I/O Usage

This section discusses task attributes that affect various aspects of process I/O activity. Included are discussions of attributes that affect disk files, printed output, data comm I/Os, and the total I/O activity of a process.

Section 10. Determining Process History

This section explains the various types of normal and abnormal terminations that a process can have, and the means for determining how a process terminated. This section also explains how to initiate and specify the contents of a program dump.

Section 11. Restarting Jobs and Tasks

This section discusses restarting WFL jobs automatically, using checkpoints to store intermediate states of a process, and using the RESTART task attribute to reinitiate a process after a fault termination.

Section 12. Tasking across Multihost Networks

This section explains how to initiate WFL jobs or other processes that run on a remote host, how to monitor the status of a remote process, and how to ensure that a remote process runs successfully in the environment of a remote host.

Part II. Interprocess Communication

The sections in this part discuss the various ways that processes can communicate user-defined information to each other.

Section 13. Understanding Interprocess Communication

This section introduces the types of objects that are most useful for sharing information between processes, methods used to make these objects available to more than one process, and the means of synchronizing access to shared objects.

Section 14. Using Task Attributes

This section discusses the task attributes for passing user-defined information between processes.

Section 15. Using Global Objects

This section discusses the scope of declarations in WFL and ALGOL, and the ability of internal tasks to access and share these global objects.

Section 16. Using Events

This section discusses the use of events to regulate the timing of asynchronous processes. Topics include the available state, the happened state, interrupts, and buzz loops.

Section 17. Using Parameters

This section discusses the effect of parameter passing on the scope of a declaration, actual and formal parameters, parameter passing modes, and the use of tasking parameters in interprocess communication.

Section 18. Using Libraries

This section explains basic concepts relating to libraries, and provides examples of libraries and calling programs in each language. Topics include user programs, library programs, duration and sharing, properties of a library object, library attributes, library-related task attributes, and parameter-type matching.

Section 19. Using Shared Files

This section summarizes the capabilities of port files, host control (HC) files, and HYPERchannel® (HY) files. This section also introduces the use of interprocess communication techniques for regulating access to disk files. See the *A Series I/O Subsystem Programming Guide* for details about all these topics.

Section 20. Communication across Multihost Networks

This section explains which interprocess communication techniques are available for use between processes that run on separate host systems.

In addition, this guide includes a glossary, a bibliography, and an index.

Related Product Information

***A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation* (form 8600 0098)**

This manual describes the basic features of the Extended ALGOL programming language. This manual is written for programmers who are familiar with programming concepts.

***A Series COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation* (form 8600 0296)**

This manual describes the basic features of the Standard COBOL ANSI-74 programming language, which is fully compatible with the American National Standard, X3.23-1974. This manual is written for programmers who are familiar with programming concepts.

A Series Task Attributes Programming Reference Manual* (form 8600 0502). Formerly the *A Series Work Flow Administration and Programming Guide

This manual describes all the task attributes available on A Series systems. It also gives examples of statements for reading and assigning task attributes in various programming languages.

HYPERchannel is a registered trademark of Network Systems Corporation.

***A Series Work Flow Language (WFL) Programming Reference Manual* (form 8600 1047)**

This manual presents the complete syntax and semantics of WFL. WFL is used to construct jobs that compile or run programs written in other languages and that perform library maintenance such as copying files. This manual is written for individuals who have some experience with programming in a block-structured language such as ALGOL and who know how to create and edit files using CANDE or the Editor.

Contents

Section 3. Tasking from Interactive Sources

Section 4. Tasking from Programming Languages

Section 10. Determining Process History

Section 11. Restarting Jobs and Tasks

Section 17. Using Parameters

Section 18. Using Libraries

Section 19. Using Shared Files

Section 20. Communication across Multihost Networks

Tables

Part I
Tasking

Section 1
Understanding Basic Tasking Concepts

A Series tasking features are inherent in the overall system architecture. Various A Series programming languages and operations interfaces provide you with access to different subsets of the tasking capabilities of A Series systems. This section presents an overview of A Series tasking features and discusses the advantages and limitations of these features.

Tasking Concepts

The following subsections discuss the relationships between programs and processes, and the methods you can use to monitor and control process behavior.

Programs and Processes

A *program* is a sequence of statements written in any of a number of languages, including ALGOL, BASIC, C, COBOL74, COBOL85, FORTRAN77, Pascal, and Work Flow Language (WFL). The file in which you write and store these statements is referred to as a *source file*. By compiling the source file, you cause the creation of an *object code file*.

By using any of a number of commands or statements, you can cause a particular object code file to be *initiated*. That is to say, you cause the system to start performing the instructions in the object code file. At this point, the object code file is being *executed*. However, in a sense, nothing is happening to the object code file itself. The system merely reads instructions from the object code file; the contents of the file remain unchanged.

There is, nonetheless, a dynamic entity called a *process*, which is separate from the object code file, but which reflects the current state of the execution of the object code file. A process stores the current values of variables used by the program, as well as information about which procedures have been entered and which statement is currently being executed. (Procedures are discussed under "Internal and External Processes" later in this section.)

Each process exists in the system memory, and consists of several distinct structures that are discussed in Section 8, "Controlling Process Memory Usage."

The distinction between object code files and processes is a very important one on A Series systems. This is because, at any given time, there can be multiple processes that are executing the same object code file; these are referred to as *instances* of that object code file. A new instance is created each time a user or an existing process submits a statement that initiates the object code file.

Because many instances of the same object code file can be running at the same time, the object code file title is not sufficient to uniquely identify a process. Therefore, in system command displays, the various processes are identified both by an object code file title and by a unique four-digit number called the *mix number*. For further information on mix numbers, refer to Section 5, "Establishing Process Identity and Privileges."

Even if processes are executions of the same object code file, the processes are completely separate entities and do not interact with each other. For example, suppose the object code file called OBJECT/PROG includes a declaration of an integer variable named N, as well as various statements that assign values to N. In this case, each instance of OBJECT/PROG has its own copy of variable N in memory. When one process changes the value of N, there is no change to the value N has for the other processes.

The fact that processes are separate and maintain their own copies of variables generally prevents confusion and simplifies program design. However, there can also be cases where you want processes to have shared access to a particular variable. For these cases, the A Series systems provide a variety of interprocess communication techniques, which are described in Part II of this guide.

Tasking consists of using various A Series features to initiate, monitor, and control processes. You can perform tasking functions by entering commands through various system operation interfaces, or by writing programs that initiate, monitor, and control the execution of other programs.

Task Attributes

Task attributes are entities that record various properties of a process, such as its usercode, mix number, priority, printing defaults, and so on.

There are a limited number of task attributes, which are defined by the operating system and have fixed meanings. Each process possesses all of these task attributes, but the values of the task attributes can vary. For example, each process has a USERCODE task attribute, but where one process might have a USERCODE value of JASMITH, another process might have a USERCODE value of JANEDOE.

Task attributes record or modify many aspects of process execution, including security, processor usage, memory usage, and I/O activity. You can assign task attributes to a process either through commands entered at an interactive source, or through statements in a program.

This guide introduces many of the important uses of task attributes. The remaining sections in Part I of this guide introduce task attributes within discussions of general functional areas, such as processor usage, memory usage, and so on. For detailed information about any of these task attributes, you can refer to the *A Series Task Attributes Programming Reference Manual*, which presents the task attributes in alphabetical order.

Interactive Tasking

You can perform tasking functions through any of the following interactive interfaces:

- Command and Edit (CANDE)

  This is a command-driven environment that provides file handling and tasking capabilities.

- Menu-Assisted Resource Control (MARC)

  This is a menu-driven interface to system operations functions.

- Operator display terminals (ODTs)

  These are terminals that support an interface called *system command mode*.

Each of these products provides the following general types of tasking capabilities:

- A command or menu selection that allows you to initiate any object code file by name. Examples are the RUN command in CANDE and MARC.
- Syntax for specifying *task equations*, which are task attribute assignments applied to a process when it is first initiated.
- Task attribute *inheritance*, which causes a process to receive task attributes associated with the initiating source.
- Various commands or selections for monitoring the status and resource usage of processes, or for intervening in process execution in various ways.

The tasking capabilities of CANDE, MARC, and the ODT are described in Section 3, "Tasking from Interactive Sources."

Note that many commands entered by users can indirectly cause a process to be initiated. For example, the Communications Management System (COMS) initiates instances of direct window programs in response to variations in the message traffic from users. Similarly, the system initiates processes to execute some specialized system commands, such as LOG.

This guide does not attempt to describe all such cases of indirect tasking. CANDE, MARC, and the ODT are all introduced in this guide because they provide direct, generalized tasking interfaces. With these products, you can initiate any object code file, as well as monitor and control any process (to the extent allowed by system security).

Programmatic Tasking

You can perform tasking functions using any of the following programming languages: ALGOL, APLB, COBOL(68), COBOL74, PL/I, and WFL. This guide provides details about the tasking capabilities of the newer and more popular of these languages, namely ALGOL, COBOL74, and WFL.

Each of these languages provides you with the following types of tasking capabilities:

- Statements that allow you to initiate any object code file by name. Examples are the CALL, PROCESS, and RUN statements in ALGOL and COBOL74.
- Constructs for reading and assigning the task attributes of a process before the process is initiated, while it is running, and after it completes execution.

The tasking capabilities of each of these languages are described in Section 4, "Tasking from Programming Languages."

At this point you might be aware of the potential for some ambiguity in the use of task attributes within programs. For example, every process has a USERCODE task attribute. If you write a program that makes an assignment to the USERCODE task attribute, how does the system know which process the USERCODE should be applied to?

The answer is that ALGOL, COBOL74, and WFL all provide a special type of variable called a *task variable*. A task variable is also known as a *control point* in COBOL74. You can declare one or more task variables in a program, each with a distinct name. When you use a process initiation statement, you include a reference to a task variable in that statement. The task variable thereafter becomes associated with the new process.

Statements that use task attributes always specify a task variable name as well as a task attribute name. In this way, it is always clear which process is being referred to.

When one process initiates another process, many of the task attributes of the initiating process are transferred to the new process. This transference is called *inheritance*. Details about the task attributes that are inherited, and under what circumstances they are inherited, are given in the *A Series Task Attributes Programming Reference Manual*.

Process Termination

A process typically ends when the last instruction in the object code file is executed. This is referred to as a *normal termination*.

However, a process can also terminate prematurely for any of a number of reasons. For example, you can use the DS (Discontinue) system command to terminate a process. A process can also terminate because a flaw in program design causes it to attempt to do something impossible, such as dividing by zero. Additionally, all processes are terminated in the event of a system halt/load. All of these types of terminations are referred to as *abnormal terminations* because the inference is that something went wrong.

When you initiate a process, you usually want to be able to find out later whether it ran successfully or not. The system provides a number of facilities to help you determine whether the process ran successfully, and why it failed if it was not successful. These facilities include the HISTORYTYPE, HISTORYCAUSE, and HISTORYREASON task attributes, and the program dump facility. These facilities are described in Section 10, "Determining Process History."

Sometimes you might want to rerun a process that terminated abnormally. For example, if the process was terminated by a system halt/load, then the underlying program might be perfectly sound. Restarting the process could enable it to complete its work successfully. However, a number of design issues must be considered for processes that are intended to be restartable. These design issues, and the means of restarting processes, are explained in Section 11, "Restarting Jobs and Tasks."

Internal and External Processes

Up to this point, this section has discussed only cases where an object code file is executed from beginning to end as a single process. However, A Series systems give you the option of causing individual procedures to be initiated as separate processes. These processes fall into two general categories: internal and external processes.

The following subsections describe the various types of internal and external processes. For a discussion of the varying capabilities of these types of processes, refer to the discussion of inclusion in Section 2, "Understanding Interprocess Relationships."

Internal Processes

Many programming languages give you the ability to create groups of declarations and statements within a program, and to assign a name to each group. In ALGOL, these groupings are referred to as *procedures*. In WFL, these groupings are referred to as *subroutines*. However, the basic concept is similar in both cases, and the term "procedure" in this guide refers equally to ALGOL procedures and WFL subroutines.

Other programming languages offer similar types of structures, but ALGOL and WFL are the only languages that give you a choice between the following two methods of invoking a procedure:

- Procedure entrance

  The syntax for entering a procedure consists of using the procedure name as if it were a statement. Entering a procedure causes the procedure to be executed as part of the same process that invoked the procedure. When the process finishes executing the procedure, the process *exits* that procedure.

- Procedure initiation

  The syntax for initiating a procedure consists of using a CALL, PROCESS, or RUN statement in ALGOL, or a PROCESS <subroutine> statement in WFL. Initiating a procedure causes it to be executed as a new process, separate from the process that invoked the procedure. This new process is referred to as an *internal process* because it is executing part of the same object code file as the initiating process.

Of these methods, procedure entrance has the advantages of simplicity and low impact on system resources, as discussed under "Limitations of Tasking" later in this section. On the other hand, procedure initiation allows you to use parallel processing or to assign the new process different task attribute values than those of the initiating process. These features are introduced under "Advantages of Tasking" later in this section.

Note that, if you use the Binder utility to bind a procedure from a subprogram into a host program, that procedure is thereafter considered an internal procedure of the host program. If the host program is an ALGOL program, the host program can either enter or initiate the bound procedure. If the procedure is initiated, the resulting process is considered to be an internal process. For information about the Binder utility, refer to the *A Series Binder Programming Reference Manual.*

External Processes

An external process is one that results when a statement in a program initiates an external procedure. An external procedure is one that resides in a program other than the program containing the statement that invokes the procedure. External procedures are of three types:

- Separate programs

  Any program, taken as a whole, can be thought of as an external procedure when it is invoked by a statement in a different program. A separate program is always executed as a separate process; that is, a process can initiate, but cannot enter, a separate program. WFL, ALGOL, and COBOL74 all allow you to initiate separate programs. In ALGOL and COBOL74, you must specify dummy procedures, called *declared external procedures*, in statements that initiate separate programs.

- Passed external procedures

  These are procedures passed into the program as parameters. You can write programs in ALGOL that accept procedures as parameters from the initiating program. Statements in the receiving ALGOL program can either enter or initiate a passed procedure.

- Library procedures

  These are procedures that are provided by a special type of program called a *library*. Libraries make procedures available for use by other programs. Statements in an ALGOL program can either enter or initiate a library procedure. Programs written in other languages can enter, but cannot initiate, a library procedure. The methods for writing libraries and programs that use libraries are discussed in Section 18, "Using Libraries."

Program Structure

Each program is viewed by the operating system as having a certain *block structure*. The block structure of the program can have implications for the critical block definition and for the ability of processes to communicate through global objects. For further information on these topics, refer to "Critical Blocks" in Section 2, "Understanding Interprocess Relationships" and to Section 15, "Using Global Objects."

The term *flow of control* refers to the order in which the statements of a program are executed. Most statements perform an action and then pass control to the immediately following statement. However, some statements can pass control to structures residing elsewhere in the program.

A *block* is a program, or program subunit, that can contain a group of declarations and a group of statements. The declarations create objects that are for local use by the statements in the block. There are two kinds of blocks: procedures and simple blocks.

A *procedure* is a block that can be executed using a procedure invocation statement, which passes control to the start of the procedure. When the procedure finishes executing, control automatically returns to the procedure invocation statement, and passes to the next statement in the program.

This abstract definition of a procedure corresponds to the way procedures are viewed by the A Series operating system. Procedures are called by different names in the syntax of the various programming languages. This definition of a procedure corresponds, for example, to a PROCEDURE in ALGOL, a PROCEDURE or FUNCTION in Pascal, a SUBROUTINE or FUNCTION in FORTRAN or FORTRAN77, or a nested program in COBOL85. It also corresponds to a complete program written in any of these languages.

Note that a complete program written in COBOL(68) or COBOL74 is also considered a procedure. However, a paragraph or a section in these languages is not considered a procedure. It is true that a PERFORM statement resembles a procedure invocation statement in that it causes control to pass through the paragraph or section and then return to the PERFORM statement. However, paragraphs and sections cannot include declarations and thus are not treated as procedures by the operating system. Therefore, the various properties of procedures discussed in this guide do not apply to COBOL(68) or COBOL74 paragraphs or sections.

A *simple block* is a block that cannot be specified in a procedure invocation statement. Simple blocks exist only in ALGOL, where they appear among the statements in the program, rather than among the declarations. The beginning and end of a simple block are marked by the keywords BEGIN and END. A simple block is executed in sequence between the statements that immediately precede and follow the simple block.

Note that a BEGIN...END group is considered to be a simple block only if it contains at least one declaration. Otherwise, it is considered a compound statement. Compound statements do not affect tasking or interprocess communication issues, and will not be further discussed in this guide.

Some languages, including WFL and ALGOL, allow blocks to be declared within other blocks. This practice is referred to as *nesting*. A block that contains a nested block is said to be *global* to that nested block. The most global block is referred to as the *outer block* of the program.

The *lexical level* of a block is a measure of how deeply the block is nested. By default, the outer block of a program has a lexical level of 2; however, compiler control options can be used to cause the outer block to be compiled with a higher lexical level. Each procedure has a lexical level one higher than the outer block or procedure in which it is declared.

Advantages of Tasking

The benefits of tasking fall into the general areas of simplifying system operations, increasing programmer productivity, and improving performance of an application.

Simplifying System Operations

Many applications involve running a sequence of programs, one after another in a certain set order. Often it is necessary to specify parameters and task attribute assignments for each of the programs. An operator can initiate the programs individually, providing the needed parameters and task attribute assignments in each case. However, this proves to be too time consuming in an environment where many applications are run during a given work shift.

An alternative, which reduces the labor required of the operator, is to write a small program whose only purpose is to initiate a series of other programs. Such a program can provide a standard set of parameters and task attribute assignments. You can write such a program in ALGOL, COBOL74, or WFL. This enables the operator to initiate a single program and leave it to initiate all the others.

WFL is particularly suitable for implementing such programs because WFL programs typically pass through *job queues*. An operator can use the MQ (Make or Modify Queue) system command to create job queues and assign various job queue attributes to them. The use of job queues enables the operator to submit jobs when it is convenient, while relying on the system to initiate jobs at specified times or according to specified criteria. Job queues are further discussed under "Selecting the Queue for a Job" in Section 4, "Tasking from Programming Languages."

Increasing Programmer Productivity

Tasking techniques can improve programmer productivity by modifying the behavior of existing programs, by allowing you to use programs as modules in a larger application, and by allowing multiple programming languages to be used in an application.

Modifying Program Behavior

Sometimes a program is designed to run in a particular environment, and later that environment changes. For example, a program might be designed to read a file on a family named DATAPK. Later, you might want to run a copy of that program on a different system that does not have a family with that name. Rewriting the source program and recompiling it can be a time-consuming process. Fortunately, many such behaviors can be modified through task attribute assignments.

For example, there is a task attribute called FAMILY that causes a process to use files on a different family than it otherwise would. Suppose a process expects to find all its input files on the family named DATAPK. You can assign the FAMILY task attribute a value of "DATAPK = CONTROL OTHERWISE DISK". This causes the process to look for all its input files on the family named CONTROL instead of the family named DATAPK.

You can assign a task attribute to a process in any of the following ways, none of which requires recompiling or rewriting the program that is being initiated:

- If you are running a program from CANDE or MARC, you can append task attribute assignments to the RUN command that initiates the program.
- You can use a WFL *MODIFY* statement to assign default task attribute values to an object code file. The system assigns these task attribute values each time the object code file is run.
- ALGOL, COBOL74, and WFL all allow you to assign task attributes to a task variable. If you then specify this task variable in a statement that initiates a separate program, the task attribute assignments are applied to the new process.

The *A Series Task Attributes Programming Reference Manual* gives examples of these methods of assigning task attributes.

Using Programs as Modules

A *module* is a body of code that can be reused in a variety of different contexts. The use of modules simplifies the programmer's job by making it unnecessary to repeat large amounts of code. One advantage of tasking is that it allows you to use an entire program as a module in one or more larger applications.

For example, you could have a report-formatting and printing program. You might also have a program that retrieves customer data from a database, and another program that does an inventory analysis. The customer data program and the inventory analysis program could both use process initiation statements to invoke the report-formatting and printing program and cause it to create reports using the data collected.

Tasking is only one of the methods that A Series systems provide for allowing code to be reused by different programs. Some of the other methods are

- Compile-time options

  You can use a $INCLUDE option in a program source file. At compilation, the compiler inserts text from a separate source file specified by the $INCLUDE option. This option is discussed in the manuals for each programming language.

- Binding

  This technique enables you to to insert a compiled procedure from one object code file into a separate object code file. This technique is documented in the *A Series Binder Programming Reference Manual*.

- Libraries

  This technique enables a process to dynamically invoke a procedure in another running process. This technique is described in the Section 18, "Using Libraries."

All of these methods have their virtues. Compared to the $INCLUDE option or binding, tasking has the advantage of enabling you to maintain the shared module separately from the programs that call on it. You can make changes to the module without having to recompile another program or rerun the Binder.

On the other hand, both the $INCLUDE option and binding have the advantage of enabling you to insert an external procedure directly into the source or object program. Because the inserted procedure is treated by the system as an internal procedure, the main program can enter the procedure rather than initiating it. This results in savings of processor time and memory.

Compared to libraries, tasking has a slight performance advantage in some situations. Initiating a program carries a certain cost in terms of processor time, memory, and so on. The cost of entering a library procedure varies, and can be higher or lower than the cost of initiating a process. For the first call on a particular library, the system must initiate the library process and establish a linkage between the calling program and the library. Once the library is running, it is more economical to enter a library procedure than to initiate a process.

Another advantage of the tasking method arises in situations where there already exists a program that performs a function needed by your application. You can initiate that program as a process without having to rewrite or recompile the program that performs the function. Changing the program into a library would require rewriting, and binding the program into another program requires using the Binder utility.

Using Multiple Languages in an Application

Different programming languages have different unique capabilities. These might make it easier to implement some types of routines in one language, and other types of routines in another language. If the same application requires routines in two or more different languages, then those routines have to be stored in separate source files and compiled separately.

One way to enable an application to use modules written in different languages is through tasking. You can accomplish this by using statements that initiate separate object code files. For example, you can write a COBOL74 program that initiates another program written in ALGOL.

A nice thing about this technique is that A Series systems also enable you to pass parameters between programs written in different languages. The operating system allows parameters to match as long as they are of compatible types. Section 17, "Using Parameters," explains which parameter types are considered compatible by the operating system.

Alternatively, you could use binding or libraries to create an application that uses modules written in different languages. The advantages of using tasking instead of binding or libraries are introduced under "Using Programs as Modules" earlier in this section.

Improving Application Performance

The definition of *performance* for an application has two general aspects: measurements of the resource usage of an application and measurements of the elapsed time of the application. Resource usage includes total processor time, average memory usage, and

so on. Elapsed time means the total clock time a batch program takes to run, or the average time an online program takes to respond to a transaction.

If you find that the elapsed time of an application is of crucial importance to your business, you can use tasking features to help decrease the elapsed time by allowing the application to use system resources more intensively. The two features that allow you to do this are process priorities and parallel processing.

A Series systems are designed to be able to execute large numbers of processes simultaneously. However, each central processor can execute only one process at a time. The operating system frequently reevaluates the processes waiting for processor service, and assigns the processor to the process with the highest priority. You can use task attributes and system commands to control some aspects of process priority, as discussed in Section 7, "Controlling Processor Usage."

Parallel processing consists of dividing your application into two or more processes that run concurrently. Parallel processing enables the application to use system resources more intensively than a single process can. This increased intensity of system resource usage results because each process typically alternates among using the central processor, I/O processor, and other resources. With parallel processes, one process can use the central processor while the other is waiting for an I/O to complete, and so on.

You can create parallel processes by designing one process to initiate another process of type PROCESS or type RUN. These process types are discussed in Section 2, "Understanding Interprocess Relationships."

Limitations of Tasking

If you do not need any of the benefits of tasking described in the preceding subsection, you can simply implement your entire application as a single program, and use only procedure entrance statements rather than procedure initiation statements. Procedure entrance uses fewer system resources than procedure initiation, and allows your application to complete faster and interfere less with other running applications.

Some of the expenses involved in initiating a procedure are

- It takes slightly more processor time than entering a procedure.
- It causes several hundred words of save memory to be allocated for the new process stack.
- It causes the system to create additional system log entries, and thus adds to general system overhead.
- It adds to the number of entries visible to the operator in a mix display. It thus tends to complicate the system operator's efforts to monitor the system.

The performance differences between entering and initiating a procedure are not great if the procedure is to be executed only once. However, for a procedure that is invoked many times, the performance loss can slow an application noticeably.

Section 2
Understanding Interprocess Relationships

The relationship between a process and its initiator is defined in terms of three major properties, which are defined in the following subsections. These properties are inclusion, flow of control, and dependency. These properties affect the speed and efficiency with which a process is executed, and the ability of the initiator to interact with the process. You can control these properties in two ways:

- By choosing among the various process-initiation statements that are available
- By choosing a program structure appropriate to the type of process desired

This section examines these choices and their implications for a family of processes.

Several of the discussions that follow refer to the term *parent*. This term is defined fully under "Dependency" in this section. For now, it is enough to know that the initiator of a process is usually also the parent of that process.

Inclusion

Section 1, "Understanding Basic Tasking Concepts," introduced the distinction between internal and external procedures, and the concept that initiating procedures results in internal or external processes. The differing properties of internal and external processes are referred to in this guide as *inclusion* properties. The following are the inclusion properties of internal and external processes:

- An internal process must be dependent. Similarly, external processes that result from initiating library procedures or passed external procedures must be dependent. Only external processes that result from initiating separate programs can be either dependent or independent. Any attempt to initiate a procedure that is not a separate program as an independent process causes the error "NON - EXTERNAL RUN". For an explanation of the difference between dependent and independent processes, refer to "Dependency" in this section.
- In ALGOL and WFL, internal procedures have access to variables declared globally in the program. These global variables can serve as a medium for interprocess communication if the internal procedure is initiated. For information about this interprocess communication technique, refer to Section 15, "Using Global Objects."
- Several task attributes that are inherited by internal processes are not inherited by external processes. These task attributes include LIBRARY, NAME, OPTION, STACKSIZE, and TADS. For a discussion of task attribute inheritance, refer to the *A Series Task Attributes Programming Reference Manual*.

Flow of Control

In Section 1, "Understanding Basic Tasking Concepts," control was defined as the path execution takes among the various statements of a program. In a broader sense, control is the path execution takes among the statements of a procedure and any procedures initiated by that procedure. The programmer specifies the type of control path to be used by choosing the corresponding process initiation statement.

The control path determines whether the initiating process and new process execute in parallel or by taking turns. If they are executing by turns, the control path specifies when and how often they take turns before the new process terminates. The following subsections discuss the types of control paths that are available on Unisys A Series systems.

Synchronous Processes

When a synchronous process is initiated, control is transferred from the initiating process to the new process. In other words, the initiating process stops executing and the new process begins executing. The initiating process is still considered active during this period and its process stack still exists. When the new process terminates, the initiating process begins executing again, starting with the first statement after the process initiation statement.

Examples of statements that initiate synchronous processes are the CALL statement in ALGOL or COBOL74 and the RUN statement in Work Flow Language (WFL). Synchronous processes are sometimes referred to as coroutines, but more properly the term *coroutine* has a different use. (Refer to "Coroutines" in this section for details.)

The initiating process can set the attributes of a synchronous process only at initiation time and can interrogate the attributes only after the synchronous process has terminated.

Synchronous processes can be simpler to design than coroutines or asynchronous processes because you do not have to deal with certain complexities of timing that arise for these other types of processes.

Asynchronous Processes

When an asynchronous process is initiated, the necessary memory structures are created for the new process. Thereafter, the new process and the initiator execute in parallel. Although they execute at the same time, they do not necessarily execute at the same speed. It is for this reason that the new process is called *asynchronous.*

Examples of statements that initiate asynchronous processes are the PROCESS statement in ALGOL or COBOL74, and the PROCESS RUN or PROCESS <subroutine> statement in WFL.

Asynchronous processes are useful because, in many situations, two or more processes running in parallel can do needed work in less elapsed time than a single process. What

is saved in elapsed time does not necessarily translate into savings in processor or I/O time, however.

The task attributes of an asynchronous process can be read or assigned by its initiator while the asynchronous process is executing. This makes it possible for the initiator to intervene in the execution of the asynchronous process.

A disadvantage to initiating processes asynchronously is that, except in WFL, the programmer must take special measures to prevent a critical block exit error from occurring. (See the discussion of "Critical Blocks" in this section.)

Also, initiating processes asynchronously can create ambiguous timing situations because it is impossible to predict exactly how long a process will take to execute. If an asynchronous process and its initiator share a data item, such as a global variable, and both change the value of that data item, it will be difficult to predict the order in which the changes will occur.

Various methods are used to regulate the timing of asynchronous processes. These methods are discussed in Section 16, "Using Events."

Coroutines

The term *coroutines* refers to a group of processes that exist simultaneously but take turns executing, so that only one of the processes is executing at any given time. Coroutines offer some of the advantages of asynchronous processes, but generally are easier to design because coroutines execute in a sequential order that prevents any ambiguities of timing. The use of coroutines offers the following benefits:

- The ability to execute a procedure repeatedly without incurring the processor time required to enter or initiate the procedure each time
- The ability to execute a procedure repeatedly without losing the values of objects declared in the procedure between each execution

Note, however, that coroutines use the processor less efficiently than do asynchronous processes. Only one coroutine runs at a time, and there might be periods when the processor is unused because the coroutine is waiting for an I/O operation to complete. Furthermore, the statements coroutines use to transfer control to other coroutines use more processor time than the event-related functions that asynchronous processes can use to suspend or resume each other.

Creating Coroutines

An ALGOL or COBOL74 process can create a coroutine by executing a CALL statement. The new process and its initiator are referred to as coroutines. When the initiator executes a CALL statement, the initiator temporarily ceases execution and its stack state becomes "TO BE CONTINUED". The stack state can be displayed by using the Y (Status Interrogate) system command. A coroutine with this stack state is referred to as a *continuable coroutine.*

The new process has one of the stack states that indicate the process is being processed, or soon will be, such as ALIVE or READY. The new process is referred to as an *active coroutine.*

The total number of coroutines increases each time an active coroutine executes a CALL statement. The new process created is an active coroutine and all others are continuable coroutines.

The concept of a coroutine is closely related to that of a synchronous process, as defined in "Synchronous Processes" in this section. Every synchronous process is also a coroutine; however, not every coroutine is a synchronous process. An asynchronous process can execute a CALL statement and thus become a continuable coroutine.

Using Continue Statements

An active coroutine can transfer control to a continuable coroutine by executing an ALGOL *CONTINUE* statement or a COBOL74 *CONTINUE* or *EXIT PROGRAM* statement. For convenience, these are all referred to as *continue statements* in the following discussion.

The other programming languages (BASIC, FORTRAN, FORTRAN77, Pascal, PL/I, RPG, and WFL) do not provide continue statements. Therefore, processes other than ALGOL or COBOL74 processes can be considered coroutines only in a restricted sense. For example, a WFL job can create a synchronous offspring by executing a RUN statement. The stack state of the WFL job then becomes "TO BE CONTINUED". However, the system does not allow the offspring to use a continue statement to transfer control to the WFL job. Instead, the system automatically continues the WFL job when the offspring terminates. This act is referred to as an *implicit continue* and is discussed further in "Continuing the Partner Process" in this section.

To understand the effects of a continue statement, suppose an active coroutine called A executes a continue statement that specifies a continuable coroutine called B. When the continue statement is executed, the coroutine A ceases execution and coroutine B resumes execution. In other words, coroutine A becomes a continuable coroutine and coroutine B becomes an active coroutine. Control passes from coroutine A to coroutine B.

Coroutine B can later reverse this situation by executing a continue statement that passes control back to coroutine A. However, control does not always have to pass back and forth between the same pair of processes. For example, coroutine B might continue another coroutine called C and that coroutine might then continue coroutine A.

Control can pass between coroutines any number of times. In the course of its lifetime, a coroutine can execute many continue statements applying to any number of other processes. However, for a continue statement to be successful, it must be executed by an active coroutine and it must specify a continuable coroutine. The continue statement results in an "ILLEGAL VISIT" error if it transfers control to a process that is not a continuable coroutine.

Coroutines usually belong to the same process family because continue statements must explicitly or implicitly specify the task variable of the process to be continued. A

process usually has access only to the task variables of processes in its own process family. Process families are defined under "Process Families" in this section. The means of accessing the task variables of related processes are discussed under "Accessing Task Variables" in this section.

Determining Where Execution Resumes

When any coroutine continues an ALGOL coroutine, the ALGOL coroutine resumes at the point where it left off. Thus, if an ALGOL coroutine executes a CALL statement, it later resumes with the first statement after the CALL statement. If an ALGOL coroutine executes a CONTINUE statement, it later resumes with the first statement after the CONTINUE statement.

By contrast, a COBOL74 coroutine can resume execution at either of two points. If a COBOL74 coroutine executes a CONTINUE statement or an EXIT PROGRAM RETURN HERE statement, then the coroutine later resumes at the point where it left off. However, if a COBOL74 coroutine executes a simple EXIT PROGRAM statement, then the coroutine later resumes with the first statement in the program. (Certain limitations on the EXIT PROGRAM statement are discussed under "Continuing the Partner Process" later in this section.)

Block Structure and Coroutines

Continue statements can occur in any of the procedures executed by a process. For example, a process can execute a continue statement and, after being continued later on, can enter another procedure and execute another continue statement. Both of those continue statements can transfer control to the same coroutine, or they can transfer control to different coroutines.

If a coroutine uses a continue statement to resume its parent, and the parent exits the critical block for that coroutine, then the parent is terminated with a "CRITICAL BLOCK EXIT" error. The methods of preventing a critical block exit are discussed under "Critical Blocks" in this section.

Continuing the Partner Process

There are two types of continue statements: specific continue statements and general continue statements.

A specific continue statement is one that specifies a task variable. An ALGOL example of a specific continue statement is CONTINUE (T1). A COBOL74 example of a specific continue statement is CONTINUE T1. Either of these statements continues the coroutine specified by the task variable T1.

A general continue statement does not specify a task variable. In ALGOL, the general continue statement is CONTINUE. In COBOL74, the general continue statement is EXIT PROGRAM or EXIT PROGRAM RETURN HERE.

The effect of the general continue statement is to continue the *partner process*. The partner process is the process specified by the PARTNER task attribute. This task

attribute is said to be *task-valued* because it accesses the task variable of a particular process. For a synchronous process, the system assigns the initiating process as the partner process by default. You can design a program to assign a different task variable to the PARTNER task attribute. Thereafter, any general continue statements affect the process with that task variable.

When a synchronous process terminates, the system implicitly continues the partner process. This is the reason the initiating process usually resumes after a synchronous process terminates. However, if a synchronous process has another task variable assigned to the PARTNER task attribute, then the system continues that partner process rather than the initiating process.

Setting the PARTNER task attribute to a process other than the initiator is not recommended. Such a practice causes general continue statements or implicit continues to consume more processor time than they otherwise would. This practice also leads to source code that is difficult to understand and maintain.

A process can interrogate the PARTNEREXISTS task attribute to determine whether the current partner process is in a continuable state. This can be a useful method for avoiding "ILLEGAL VISIT" errors.

For further information regarding the PARTNER and PARTNEREXISTS task attributes, see the discussions of these attributes in the *A Series Task Attributes Programming Reference Manual*.

Communication between Coroutines

When an active coroutine becomes a continuable coroutine, or vice versa, objects declared by the coroutine retain their values and are not reinitialized.

Nevertheless, the values of objects declared by a continuable coroutine can be changed by any active coroutine having access to those objects. For example, if a process executes a CALL statement, passing call-by-reference parameters, the process becomes a continuable coroutine. The offspring process is an active coroutine and can change the values of the call-by-reference parameters. The offspring process can use this method to communicate information to the parent process. When the parent process is continued, it can check to see if the parameter values were changed.

Similar considerations apply to the task attributes of a coroutine. An active coroutine can read or assign the task attributes of other coroutines, including continuable coroutines. When a continuable coroutine is continued, it can check its task attribute values to see if any were changed.

Complex Coroutine Structures

The continue statements implemented on A Series systems enable you to develop complex coroutine structures that do not exactly correspond to the classical model of coroutines. A complex coroutine structure is one in which two or more active coroutines exist at the same time. In a simple coroutine structure, only one of the coroutines is active at a time.

A complex coroutine structure can result, for example, if a process called INITP initiates an asynchronous offspring called PROCP, and then initiates a synchronous offspring called CALLP. While INITP is waiting for CALLP to complete, INITP is in a "TO BE CONTINUED" state. PROCP can, therefore, execute a continue statement that causes INITP to resume. In this case, PROCP becomes a continuable coroutine and INITP and CALLP are active coroutines at the same time.

In general, the use of complex coroutine structures is not recommended because they lack the simplicity that is the primary benefit of using coroutines.

Dependency

The last of the three main properties the programmer can specify for a process is *dependency*. To understand the concept of dependency, the programmer must first be familiar with the following related concepts.

- Critical objects

  Every process makes use of certain objects originally declared by another process. These include the task variable, the procedure that the process is executing, and any objects passed as actual parameters to the process. In this guide, these objects are referred to as the critical objects of the process.

- Parents

  When a process is initiated, it receives these critical objects from a process called the parent. In most cases, the initiator of a process is also the parent of that process. The exact method for determining which process is the parent of a particular process is given under "Critical Blocks" later in this section.

Dependency is the relationship between a process and its parent that determines how these critical objects are stored. For an independent process, the system creates copies of these critical objects when the process is initiated. For a dependent process, the system creates references to the objects stored by the parent.

The programmer can specify the dependency of a process by choosing an appropriate process initiation statement. The dependency of a process remains the same throughout execution; if it is initiated as dependent, it cannot later become independent, or vice versa.

To initiate an independent process, you can use an ALGOL or COBOL74 *RUN* statement or a ??RUN (Run Code File) system command. Also, a WFL job submitted through a START statement is executed as an independent process.

To initiate a dependent process, you can use a CALL or PROCESS statement in ALGOL or COBOL74, or a RUN statement in Command and Edit (CANDE), Menu-Assisted Resource Control (MARC), or WFL.

Many implications result from the choice to initiate a process as dependent or independent. However, the most crucial difference is that an independent process can continue to exist after its parent has terminated. A dependent process must terminate before its parent does.

The second most crucial difference between dependent and independent processes is that a dependent process and its parent can communicate through shared objects, whereas an independent process and its parent cannot.

Communications Effects

Some objects declared by the parent process can be shared with a dependent process, but not with an independent process.

For example, a parent can declare a task variable and include it in a process initiation statement executed by the parent. For a dependent process, the task variable remains associated with the process for as long as the process exists. After the dependent process terminates, the task variable continues to store the final task attribute values of the dependent process (though later assignments can change these values). The parent can use the task variable to access the task attributes of the process before initiation, while the process is in use, or after the process terminates. However, for an independent process, the task variable ceases to be associated with the process once initiation is complete. Only task attributes assigned to the task variable before initiation have any effect on the independent process.

Similarly, a procedure declared in the parent can be initiated only as a dependent process. A separate program, on the other hand, can be initiated as a dependent or independent process. Thus, an independent process is always an external process.

Like any external process, an independent process is unable to access objects declared globally in the parent. On the other hand, a dependent process, if it is also internal, can access objects declared globally in the parent.

Finally, any parameters passed to an independent process must be passed by value. A dependent process can be passed parameters by name, by reference, or by value.

Flow of Control Effects

The dependency of a process affects the ability of the process to be synchronous or asynchronous, and the ability of the parent to exit certain blocks without incurring an error.

Synchronization

An independent process is always asynchronous. The initiator of an independent process continues execution without waiting for the independent process to terminate. By contrast, a dependent process can be synchronous or asynchronous, depending on the type of initiation statement that is used. Another difference is that an independent process can continue executing after its parent has terminated, whereas a dependent process must terminate before its parent does.

Critical Blocks

Another flow of control issue related to dependency is the prevention of critical block exits. To understand exactly what a critical block exit is and why it is important, you must first understand the following basic concepts:

- Critical objects

  This concept is introduced under "Dependency" earlier in this section. You should be aware that the critical objects of a process can be stored in more than one process stack, and they can be stored in more than one activation record in a process stack. If any block that declares one of these critical objects is exited, the corresponding activation record is removed and that critical object ceases to exist. This block exit causes the process that is using that critical object to terminate abnormally.

- Critical block

  This is a block that includes a definition of at least one critical object and is so positioned that it is normally exited before any other blocks that declare critical objects are exited. If you ensure that the parent does not exit the critical block prematurely, then the other blocks declaring critical objects also are not exited prematurely.

At this point, the definition of a parent can be further refined as follows: the parent is the process that owns the critical block of a specified process. In other words, the parent has entered the critical block and not yet exited that block. A dependent process is said to be an *offspring* of its parent.

You need to be concerned with the critical block for a process only if that process is an asynchronous dependent process or a coroutine. If the process is either of these, you must take steps to ensure that the critical block is not exited before the process terminates.

By contrast, if a process is independent, it is not affected by critical block exits. If the process is synchronous, then the parent ceases execution until the process terminates and therefore has no opportunity to exit the critical block prematurely.

Effects of a Critical Block Exit

When a parent exits an offspring's critical block, the parent is discontinued and the error message "CRITICAL BLOCK EXIT" is displayed. When the parent terminates, all its offspring processes currently in use are discontinued and a "PARENT PROCESS TERMINATED" error message is displayed.

Defining the Critical Block

The critical block of a process usually occurs somewhere in the program containing the statement that initiated the process. Within that program, the critical block is the procedure of the highest lexical level that contains any of the following items:

- The declaration of the task variable specified in the process initiation statement.
- The declaration of the procedure that was initiated, if it is an internal procedure, a passed external procedure, or an imported library procedure. The position of a declared external procedure has no effect on the critical block definition.
- The declarations of any actual parameters passed to the process. (It makes no difference whether the parameters are passed as call-by-name, call-by-value, or call-by-reference.)
- Any *thunk* generated for the process by the compiler. A thunk, which is also referred to as an accidental entry, is generated if the procedure initiation statement passes a constant or an expression to a call-by-name parameter. The thunk is located in the procedure containing the procedure-initiation statement. For an illustration of the effect of a thunk on the critical block definition, refer to Example 3 in "Critical Block Examples" later in this section.

Note that the definition of the critical block can be affected if any of the critical objects are passed as parameters from one procedure to another. If a critical object is passed as a parameter to a procedure, then for purposes of defining the critical block, the formal parameter that receives the critical object must be considered to be the declaration of that critical object. For an illustration, refer to Example 4 in "Critical Block Examples" later in this section.

There is one exception to the rule about the effects of passing critical objects as parameters. If a task variable is passed as a parameter to an external procedure, the critical block is affected by the declaration of the actual parameter rather than the formal parameter. This exception holds true for all types of external procedures: separate programs, passed external procedures, and imported library procedures. This exception also makes it possible for the procedure-initiation statement to reside in a different program than the critical block does. For an illustration, refer to Example 5 in "Critical Block Examples" later in this section.

The initiator of a process might or might not also be the parent of that process. This issue is illustrated by Examples 1 and 2 in "Critical Block Examples" later in this section.

Preventing ALGOL Critical Block Exits

In ALGOL, the programmer can prevent a critical block exit by including a statement such as the following at the end of the critical block:

```
WHILE T.STATUS GTR VALUE(TERMINATED) DO
   WAITANDRESET(MYSELF.EXCEPTIONEVENT);
```

In this example, T is the task variable of the dependent process. This statement causes the parent to wait on its own EXCEPTIONEVENT task attribute, which is automatically

caused by the system whenever the offspring changes status. The program then checks the status of the offspring and returns to a waiting state if the offspring has not yet terminated.

Preventing COBOL74 Critical Block Exits

A COBOL74 process cannot receive a critical block exit error for exiting a paragraph or a section because paragraphs and sections are not blocks. However, a COBOL74 process can incur a critical block exit error if the process

- Terminates while one of its offspring is in-use
- Exits a bound-in procedure that is the critical block for an offspring
- Exits an imported library procedure that is the critical block for an offspring

Statements such as the following can be included at the end of a COBOL74 program to prevent it from terminating before an offspring terminates:

```
PROCWAIT SECTION.
P2.
    WAIT AND RESET UNTIL ATTRIBUTE EXCEPTIONEVENT OF MYSELF.
    IF ATTRIBUTE STATUS OF TASK-VAR-1 IS GREATER THAN
       VALUE TERMINATED THEN GO PROCWAIT.
STOP RUN.
```

The preceding example assumes that an asynchronous offspring was initiated using task variable TASK-VAR-1. The COBOL74 program waits on its own EXCEPTIONEVENT task attribute, which is automatically caused whenever the offspring changes status. The program then checks the status of the offspring and returns to a waiting state if the offspring has not yet terminated.

Automatic Protection from WFL Critical Block Exits

The programmer does not need to include any special statements in WFL jobs to prevent critical block exits. WFL implicitly waits for the termination of asynchronous processes initiated by the job. The implicit wait occurs at the end of the subroutine that executed the process initiation statement.

Critical Block Examples

The following examples illustrate various factors that affect the definition of the critical block for a process. The more typical cases are presented first.

Example 1

In most cases, the initiator of a process is also the parent of that process. However, this is not always the case. The following ALGOL program is an illustration of the difference between the parent and the initiator:

```
100 PROCEDURE TRUEPARENT;
110 BEGIN
120    TASK T1, T2;
130    REAL I;
140
150    PROCEDURE WAITFOR(T);
160    TASK T;
170    BEGIN
180       WHILE T.STATUS GTR VALUE(TERMINATED) DO
190          WAITANDRESET(MYSELF.EXCEPTIONEVENT);
200    END;
210
220    PROCEDURE OFFSPRING(X);
230    REAL X;
240    BEGIN
250       X := 1;
260    END;
270
280    PROCEDURE INITIATOR;
290    BEGIN
300       PROCESS OFFSPRING(I) [T2];
310    END;
320
330    PROCESS INITIATOR [T1];
340    WAITFOR(T1);
350    WAITFOR(T2);
360 END.
```

In this example, the procedure TRUEPARENT initiates the procedure INITIATOR as an asynchronous process. INITIATOR then initiates the procedure named OFFSPRING. In this situation, the initiator of OFFSPRING is INITIATOR, but the parent is TRUEPARENT.

TRUEPARENT is considered the parent because the declarations of the procedure OFFSPRING, the task variable T2, and the actual parameter I all occur in the outer block of TRUEPARENT.

Example 2

In the following ALGOL example, the process called INITIATOR is both the initiator and the parent of the process named OFFSPRING. INITIATOR is considered the initiator because INITIATOR includes the task initiation statement that initiates the OFFSPRING procedure. INITIATOR is considered the critical block for OFFSPRING because the task initiation statement passes OFFSPRING a parameter declared within INITIATOR. An invocation of the WAITFOR procedure is added to INITIATOR to prevent a critical block exit.

```
100 PROCEDURE OUTERBLOCK;
110 BEGIN
120    TASK T1, T2;
130
140    PROCEDURE WAITFOR(T);
150    TASK T;
160    BEGIN
170       WHILE T.STATUS GTR VALUE(TERMINATED) DO
180          WAITANDRESET(MYSELF.EXCEPTIONEVENT);
190    END;
200
210    PROCEDURE OFFSPRING(X);
220    REAL X;
230    BEGIN
240       X := 1;
250    END;
260
270    PROCEDURE INITIATOR;
280    BEGIN
290       REAL R;
300       PROCESS OFFSPRING(R) [T2];
310       WAITFOR(T2);
320    END;
330
340    PROCESS INITIATOR [T1];
350    WAITFOR(T1);
360 END.
```

Example 3

The following is an ALGOL example of a case where the presence of a thunk affects the critical block definition for a process:

```
100 PROCEDURE OUTERBLOCK;
110 BEGIN
120    TASK T1, T2;
130    REAL A, B, C, D;
140
150    PROCEDURE WAITFOR(T);
160    TASK T;
170    BEGIN
180       WHILE T.STATUS GTR VALUE(TERMINATED) DO
190          WAITANDRESET(MYSELF.EXCEPTIONEVENT);
200    END;
210
220    PROCEDURE OFFSPRING(X);
230    REAL X;
240    BEGIN
250       C := X;
260    END;
270
280    PROCEDURE INITIATOR;
290    BEGIN
300       PROCESS OFFSPRING(A + B) [T2];
310       WAITFOR(T2);
320    END;
330
340    A := 2;
350    B := 5;
360    PROCESS INITIATOR [T1];
370    WAITFOR(T1);
380 END.
```

In the preceding example, X is a call-by-name formal parameter of the procedure OFFSPRING. The statement that invokes OFFSPRING passes the expression (A + B) to the parameter. This creates a thunk at the point of the procedure initiation. The thunk causes the INITIATOR procedure, rather than the OUTERBLOCK procedure, to be considered the critical block of OFFSPRING. Because the statement at line 360 initiates INITIATOR rather than entering it, INITIATOR becomes a separate process that is the parent of OFFSPRING.

You can avoid some thunks by making the formal parameter call-by-value rather than call-by-name. For example, you can avoid the thunk in the preceding example by adding a line to the procedure heading of the procedure OFFSPRING at line 220. The revised procedure heading appears as follows:

```
PROCEDURE OFFSPRING(X);
VALUE X;
REAL X;
```

This change has the side effect of making OUTERBLOCK the critical block, instead of INITIATOR.

Example 4

In the following ALGOL example, the location of the critical block is affected by a formal parameter specification:

```
100 PROCEDURE OUTERBLOCK;
110 BEGIN
120    TASK T1, TVAR;
130    REAL I;
140
150    PROCEDURE WAITFOR(T);
160    TASK T;
170    BEGIN
180       WHILE T.STATUS GTR VALUE(TERMINATED) DO
190          WAITANDRESET(MYSELF.EXCEPTIONEVENT);
200    END;
210
220    PROCEDURE OFFSPRING(X);
230    REAL X;
240    BEGIN
250       X := X + 1;
260       WAIT ((10));
270    END;
280
290    PROCEDURE INITIATOR(T2);
300    TASK T2;
310    BEGIN
320       PROCESS OFFSPRING(I) [T2];
330       WAITFOR(T2);
340    END;
350
360    PROCESS INITIATOR(T1) [TVAR];
370    WAITFOR(TVAR);
380 END.
```

In this example, INITIATOR is the critical block for the procedure OFFSPRING, because the task variable T2 is declared in the procedure heading of INITIATOR. It makes no difference that the actual parameter T1 is declared in the outer block. It is

the formal parameter T2 that is mentioned in the procedure invocation statement, and therefore the declaration of T2 takes precedence.

Example 5

In the following ALGOL examples, the critical block is located in a different program than the one that contains the process-initiation statement. The following is program OBJECT/CALL:

```
100 BEGIN
110   TASK T, T1;
120   PROCEDURE OB (T);
130     TASK T;
140     EXTERNAL;
150   REPLACE T.NAME BY "OBJECT/CALL/2.";
160   PROCESS OB (T1) [T];
170   WHILE T1.STATUS GTR VALUE(TERMINATED)
180     DO WAITANDRESET (MYSELF.EXCEPTIONEVENT);
190 END.
```

The previous program initiates a separate program called OBJECT/CALL/2, passing a task variable as a parameter. The following is the program OBJECT/CALL/2:

```
100 PROCEDURE OB (T);
110   TASK T;
120 BEGIN
130 PROCEDURE X;
140   EXTERNAL;
150 REPLACE T.NAME BY "OBJECT/TASK.";
160 PROCESS X [T];
170 END.
```

The preceding program uses its task variable parameter to initiate a third program. The procedure declaration at lines 130 and 140 does not affect the critical block definition, because it is an external procedure declaration. Note that since the process initiation statement is PROCESS, and no WAIT statement follows it, the preceding program finishes executing while the third program, OBJECT/TASK, is still running. However, no CRITICAL BLOCK EXIT error occurs.

The following is the third program, OBJECT/TASK:

```
100 BEGIN
110   EBCDIC ARRAY FORMALARRAY[0:119];
120   REPLACE FORMALARRAY BY MYSELF.EXCEPTIONTASK.NAME;
130   DISPLAY (FORMALARRAY);
140   WAIT(MYSELF.ACCEPTEVENT);
150 END.
```

This program displays the name of its EXCEPTIONTASK, which, by default, is the same as the parent. The name it displays is OBJECT/CALL, which is therefore the parent. Because OBJECT/CALL is the parent, no CRITICAL BLOCK EXIT occurs when OBJECT/CALL/2 terminates.

Recall the rule about passing task variables to external procedures that is discussed under "Defining the Critical Block" earlier in this section. It is the declaration of the actual task variable parameter, at line 110 in OBJECT/CALL, that affects the critical block definition. The critical block is therefore the outer block of OBJECT/CALL. However, the process initiation statement occurs in OBJECT/CALL/2. This is the only type of situation where it is possible for the process-initiation statement and the critical block to reside in separate programs.

Process Families

A *process family* is a group of processes that have relationships based on dependency. These relationships have many effects, including effects on interprocess communication, handling of printer output, and enforcement of resource usage limits.

Familial Relationships

Each process belonging to a process family is called a *member* of that process family. Every process family includes a single independent process as its founding member. The process family also includes any dependent offspring of that independent process, any dependent offspring of those offspring, and so on.

Familial terms are used to describe the relationships between the members of a process family. Of these, *parent* and *offspring* are defined under "Critical Blocks" earlier in this section. A related term is *sibling*. Offspring processes that have the same parent are referred to as siblings.

Each offspring of a process is considered a *descendant* of the process. Any offspring of the descendants of a process are also considered descendants of the original process.

Conversely, the parent of a process is considered to be an *ancestor* of the process, and any ancestors of the parent are also considered to be ancestors of the same process. Processes having a common ancestor, but not a common parent, are referred to as *cousins*. The independent process in a process family is the common ancestor of all the processes in that family.

Finally, processes are said to be *related* if they belong to the same process family, and *unrelated* if they do not.

A dependent process is dependent on the continued existence of all its ancestors, not only its parent. This is true because a type of domino effect occurs if any of the ancestors terminates. The immediate offspring of the terminated process are discontinued with a "PARENT PROCESS TERMINATED" error. The offspring of the discontinued processes are, in turn, discontinued with the same error, and so on.

In contrast, a member of a process family does not depend on the continued existence of any of its descendants. For example, the descendants of a process can terminate abnormally without affecting the process.

Jobs and Tasks

The independent process in a process family is called the *job* for that family. The dependent processes in a process family are referred to as *tasks*.

Note that, in some older publications, you might find the term *task* used with a different meaning than the one defined here. In addition to the meaning given here, *task* has sometimes been used to refer to any process, to the offspring of some particular process, or to any discrete unit of work. These usages are generally avoided in this guide, except in the terms *task attribute* and *task variable*, which have been retained because they are well known. (More properly, these terms would be *process attribute* and *process variable* because they can apply to either jobs or tasks.)

Certain services that the system provides for a process family are linked to the job for the family. The job provides the following services:

- Job logging

  The job has a job file associated with it that stores the job log. The job log includes information about the activities of all the processes in the process family. When the job terminates, the system can issue a printout of the job log, called the job summary. (The job file for a WFL job includes additional information, which is described under "Special Types of Jobs" later in this section.)

- Printer output

  By default, any printer or punch backup files created by process family members are saved until the job terminates; they are then grouped together as a single entry in the print queue.

Operators or programmers can use the following means to determine whether a process is a job or a task:

- Process messages

  The system displays a "BOJ" message when a job is initiated and an "EOJ" message when the job is terminated. For a task, the corresponding messages are "BOT" and "EOT".

- Job displays

  The J (Job and Task Display) system command displays all the process families that currently exist. The members of each process family appear in hierarchical order, beginning with the job.

- Job number

  A task has a job number that differs from the mix number and indicates the job or session associated with the task. For a job, the job number and mix number are equal. The operator can see the job number and mix number in the output of many system commands. A process can also read these values from the JOBNUMBER and MIXNUMBER task attributes.

- Process type

  A process can determine whether a particular process is a job by reading the TYPE task attribute. For WFL jobs, the value is JOBSTACK; for other jobs, the value is RUN. For tasks, the value is CALL or PROCESS.

Special Types of Jobs

The following subsections describe WFL jobs, BDBASE tasks, and MCS sessions, all of which are special types of jobs and entities that resemble jobs.

WFL Jobs

A program written in WFL is usually executed as an independent process. Because of this, the execution of a WFL program is referred to as a *WFL job*. The TYPE task attribute of a WFL job usually has a value of JOBSTACK.

When a WFL job is submitted from one of the available sources, the system initiates the WFL compiler. (The sources for submitting WFL jobs include START commands in CANDE and MARC sessions, and various statements in programming languages.) The WFL compiler creates the job file for the WFL job.

The job file for a WFL job contains several kinds of information that are not included in the job file for any other kind of job. In addition to the logging information, a WFL job file includes the following:

- A copy of the WFL source program.
- Object code for the job. The job file also serves as the code file for a WFL job.
- Data specifications used by the job. A data specification is a portion of the WFL source program that can be used as an input file by one or more of the offspring of the job.
- Job restart information.

WFL jobs have several other properties not shared by any other type of process. For details, refer to Section 4, "Tasking from Programming Languages."

BDBASE Tasks

Setting the BDBASE option of the OPTION task attribute causes a task to assume some characteristics of a job. The exact effects of the BDBASE option depend on whether it is assigned before or after initiation of the task. If BDBASE is assigned before task initiation, then the task receives the following joblike characteristics:

- Its own job file.
- Ability to produce a job summary.
- A mix number equal to its job number.
- "BOJ" and "EOJ" messages.
- Automatic printing, when the BDBASE task terminates, of any backup files created by the BDBASE task or its descendants. Note that this behavior applies only to backup files whose PRINTDISPOSITION file attribute has the default value of EOJ.

If BDBASE is assigned after task initiation, then its only effect is to cause default printing of backup files when the task terminates. Even if BDBASE is assigned before initiation, it does not make the task into a true job. A BDBASE task differs from a job in the following ways:

- The BDBASE task usually is not an independent process. (There is no point in setting BDBASE for an independent process, because such a process already has all job capabilities.)
- The JOBNUMBER value for a descendant of a BDBASE task does not equal the MIXNUMBER of the BDBASE task. Rather, the JOBNUMBER equals the MIXNUMBER of the job at the head of the process family.
- The MYJOB task variable never refers to a BDBASE task. For details, refer to "MYJOB Task Variable" in this section.

In the past, the main use of the BDBASE option was to cause printer backup files produced by a task to print when the task terminated, rather than being saved until the job terminated. However, other Print System features now enable you to provide the same control over printing, without assigning any other joblike characteristics to the task. For further information, refer to the discussion of printing in Section 9, "Controlling Process I/O Usage."

MCS Sessions

CANDE and MARC sessions have the following job characteristics:

- Job summaries that are produced at the end of the session and that summarize the activities of all tasks initiated from the session
- Default printing, when the session ends, of backup files produced by tasks initiated from that session
- A mix number, also called the *session number*, that is inherited by the JOBNUMBER task attribute of tasks initiated from the session

However, CANDE and MARC sessions are not really jobs because, in fact, they are not even processes. Each session is merely a dialogue between the user and the CANDE or MARC software. The MYJOB task attribute has a special meaning for tasks initiated from CANDE and MARC sessions. For further information, refer to "Access to Ancestral Processes in CANDE" and "Access to Ancestral Processes in MARC" in Section 3, "Tasking from Interactive Sources."

Accessing Task Variables

The system automatically provides several task variables, called *predeclared task variables,* for use by a process. The process can use these task variables to access task attributes of certain related members of the process family.

MYSELF Task Variable

A process can access its own task attributes by way of the predeclared task variable MYSELF.

MYSELF has a special meaning for processes that are descendants of CANDE or MARC sessions. For more information, refer to Section 3, "Tasking from Interactive Sources."

MYJOB Task Variable

A process can use the predeclared task variable MYJOB to access the task attributes of its job. When a job uses MYJOB, it has the same meaning as the MYSELF task variable.

If a BDBASE task, or a descendant of a BDBASE task, uses the MYJOB task variable, MYJOB does not refer to the BDBASE task. Instead, MYJOB refers to the independent process that is the eldest ancestor of the BDBASE task and, therefore, the real head of the process family. In other words, MYJOB refers to the job.

MYJOB has a special meaning for processes that originate from CANDE or MARC sessions or from an ODT. For more information, refer to Section 3, "Tasking from Interactive Sources."

Exception Task

Every process has an associated *exception task* with which it has a special relationship. There are two aspects to this relationship:

- Whenever the value of the STATUS task attribute of the process changes, the system notifies the exception task by causing the EXCEPTIONEVENT task attribute of the exception task.
- A process can access the task attributes of its exception task by way of its own EXCEPTIONTASK task attribute. For example, the following ALGOL statement assigns a value to the TASKVALUE task attribute of the exception task:

```
MYSELF.EXCEPTIONTASK.TASKVALUE := 5;
```

The parent of a dependent process is the default exception task of the process. An independent process, by default, is its own exception task; however, in this case, the exception task relationship embodies only the second of the aspects in the previous list. The EXCEPTIONEVENT of the independent process is not caused when the status of the independent process changes.

A dependent process can use the EXCEPTIONTASK task attribute to access the task variable of any of its ancestors. The process can specify EXCEPTIONTASK repeatedly to access ancestors two or more generations back (for example, the grandparent, great-grandparent, and so on). The following statement assigns an attribute to the grandparent of the process:

```
MYSELF.EXCEPTIONTASK.EXCEPTIONTASK.SW1 := TRUE;
```

A process can override the default exception task and assign a different process as the exception task. The following ALGOL statement specifies that the process identified by the task variable TVAR be treated as the exception task:

```
MYSELF.EXCEPTIONTASK := TVAR;
```

The process assigned as the exception task must be either the process itself or an ancestor, sibling, or cousin of the process. The exception task cannot be a descendant of the process. An attempt to assign a descendant as the exception task results in the error "UP LEVEL TASK ASSIGNMENT".

Unisys recommends that only the process itself or one of its ancestors be assigned as the exception task. If a sibling or cousin is assigned as the exception task, then any attempt to access the exception event of the exception task causes a "NON ANCESTRAL TASK REFERENCE" error. For example, in such a situation, the following statement would cause an error:

```
CAUSE (MYSELF.EXCEPTIONTASK.EXCEPTIONEVENT);
```

Assigning a process that is not the parent as the exception task can also have more subtle side effects. Suppose the task is called T and the parent contains a statement such as the following:

```
WHILE T.STATUS GTR VALUE(TERMINATED) DO
  WAITANDRESET(MYSELF.EXCEPTIONEVENT);
```

This statement causes the parent to wait until its exception event is caused, at which point it checks the status of T. If T has terminated, the next statement in the parent is executed. If T has not terminated, the parent goes back into a waiting state.

The problem is that, if a parent is not also the exception task for its offspring, then any changes in the offspring's status do not cause the parent's exception event. Instead, changes in the task's status cause the exception event of the process assigned as the exception task. Therefore, the parent continues waiting indefinitely, regardless of any changes in the task's status.

The same problem can occur in a WFL job that is waiting for an asynchronous task to complete. Consider the following WFL statement:

```
DO WAIT UNTIL TVAR IS COMPLETED;
```

This statement checks the status of the task TVAR whenever the WFL job's exception event is caused. If the WFL job's exception event is never caused, then the job waits indefinitely, regardless of changes in the status of the task.

Sometimes, however, it is not desirable for the exception event of a process to be caused whenever the status of any of its offspring changes. For example, the process might be waiting for a HI (Cause EXCEPTIONEVENT) system command. In this case, each of the offspring could be assigned itself as its exception task. This assignment prevents any of the offspring from accidentally causing its parent's exception event.

The MCS that controls a session is the parent of any tasks initiated from that session. By default, therefore, the MCS is also the exception task for any tasks initiated from that session.

Partner Processes

The *partner process* is the process specified by the task-valued task attribute PARTNER. For a synchronous process, the default value of this attribute is the initiator. However, a process can assign any task variable to this attribute. A process can use the PARTNER task attribute as a convenient means of accessing the task attributes of the partner process. For example, the following ALGOL statement assigns a value to the TASKVALUE task attribute of the partner process:

```
MYSELF.PARTNER.TASKVALUE := 3;
```

The partner process has a special significance for coroutines. For details, refer to "Continuing the Partner Process" in this section.

Other Task Variables

A programmer can make it possible for two sibling or cousin processes to access each other's task variables by declaring the task variables in a common ancestor of the two processes. Internal processes can access task variables that are declared globally in the same object code file as the internal procedure declaration. Task variables can also be passed as parameters to offspring processes.

Private Processes

A *private process* is a process whose task attributes cannot be altered by any of its descendant processes. Assigning the private process option to the OPTION task attribute causes the process to become a private process. Any descendant process that attempts to access the task attributes of a private process is terminated with the error "NON OWNER WRITE ACCESS OF A PRIVATE TASK".

Both CANDE and MARC are private processes.

Setting Resource Limits

Any resource limits attached to a job are propagated downward through all the job's descendants. Resource limits are stored in the values of the task attributes DISKLIMIT, ELAPSEDLIMIT, MAXCARDS, MAXIOTIME, MAXLINES, MAXPROCTIME, PRIORITY, RESOURCE, SAVEMEMORYLIMIT, TEMPFILELIMIT, and WAITLIMIT. Information about the amount of resources a particular process has used is stored in the task attributes ACCUMIOTIME, ACCUMPROCTIME, ELAPSEDTIME, and TEMPFILEMBYTES. If the accumulated usage of a resource rises above the maximum allowed, the process terminates abnormally. Most of these resource limits are propagated in two ways:

- When a task is initiated, by default each resource limit for the task is assigned the difference between the parent's own limit for the resource and the parent's accumulated usage of the resource. For example, if the parent's MAXPROCTIME is 100 and its ACCUMPROCTIME is 75, then the task is assigned a MAXPROCTIME of 25. The parent's own MAXPROCTIME and ACCUMPROCTIME values are not affected. The parent can assign resource limits to the task through task equation, but the values are ignored unless they specify lower limits than the task would receive by default.
- When a task terminates, the values of its accumulated usage attributes are added to the accumulated usage attributes of the task's job. If this addition causes any accumulated usage attribute of the job to be assigned a value greater than the corresponding maximum usage attribute, the job is abnormally terminated. The termination of the job in turn causes the termination of all the other members of the process family.

The resource-limiting attributes of a task cannot be set above the values of the corresponding attributes of the job. The MAXPROCTIME and MAXIOTIME task attributes can be set above the job values for an inactive task, but when the task is initiated, the values of these task attributes are automatically reduced to a value within the allowed limits.

If a job is a WFL job, then its resource-limiting attributes can inherit values specified by the queue attributes of the job queue from which the WFL job was initiated. For further information on queue limits, refer to Section 4, "Tasking from Programming Languages."

Section 3
Tasking from Interactive Sources

An interactive tasking source is one that enables you to enter at a terminal commands that initiate, monitor, and control processes. This section reviews the tasking capabilities of the most important sources for interactive tasking: Command and Edit (CANDE), Menu-Assisted Resource Control (MARC), and the operator display terminal (ODT).

The information in this section can help you decide which of these interfaces best serves your needs. This section also explains considerations to keep in mind when writing programs that are intended to be initiated from these sources.

Note that many users access applications primarily through Communications Management System (COMS) direct windows. This interface is not reviewed here because the direct window interface does not provide any direct means to control processes. Rather, COMS initiates and controls direct window programs automatically, within various parameters set by the system administrator. For information about direct window programs, refer to the *A Series Communications Management System (COMS) Programming Guide*.

CANDE

CANDE is a message control system (MCS) that enables you to interactively perform functions such as file editing, program compilation, and program execution. You initiate communications with CANDE by logging on at a terminal controlled by CANDE, or by opening a CANDE window dialogue on a terminal controlled by COMS. Your interactions with CANDE between the times you log on and log off are referred to as a *session*. CANDE assigns each session an identifying number called the *session number*.

CANDE Tasking Capabilities

CANDE offers a number of process-initiation commands, as well as other commands for monitoring or controlling processes. For details about any of the commands discussed in the following subsections, refer to the*A Series CANDE Operations Reference Manual*.

Initiating Dependent Processes from CANDE

You can initiate a task from a CANDE session by using the RUN command. (EXECUTE is a synonym for the RUN command.) The RUN command can pass only a single string parameter to a program.

The CANDE *RUN* command is unique in that it usually specifies a program by its source file title rather than by its object code file title. CANDE takes the file title specified in the RUN command and looks for an object code file with the same title, except that

the object code file title is prefixed by "OBJECT/". For example, the object code file OBJECT/TEST can be initiated by the command *RUN TEST*.

However, if you prefix the file title with a dollar sign ($), then CANDE interprets the file title as an object code file title. You can use this form of the RUN statement to initiate programs whose object code file title does not begin with "OBJECT/". An example of such a command is *RUN $ACCOUNTS/INPUT*, which initiates the object code file named ACCOUNTS/INPUT.

CANDE also assumes that the file title is an object code file title if the file title is nonusercoded. You can indicate that a file title is nonusercoded by including an asterisk (*) at the start of the title. For example, you can initiate an object code file titled *SYSTEM/FILEDATA with the command *RUN *SYSTEM/FILEDATA*.

If you omit the file title from the RUN statement, CANDE assumes the current work file is the source program. If no object code file with the related file title exists, or if the object code file does not reflect recent changes to the work file, then CANDE automatically compiles the work file and executes the resulting object code file.

The task is asynchronous (that is, it runs in parallel with the CANDE software that initiated it). However, the process appears to the user to be a synchronous task because most CANDE commands are not available while the task is running. Only control commands (commands, such as ?Y, that start with a question mark) can be used. It is not possible to issue file maintenance or editing commands or to initiate another task until the first task terminates.

An alternative to the RUN command is the UTILITY command. The UTILITY command behaves like the RUN command in most respects. However, the UTILITY command enables you to append to it unquoted text that is passed as a string parameter to the program. If you do not append any text, the UTILITY command passes an empty string parameter. The following are examples of UTILITY commands and the equivalent RUN commands:

```
U DAILY UPDATE OUTPUT = PRINTER
RUN DAILY/UPDATE("OUTPUT=PRINTER")

U DAILY UPDATE
RUN DAILY/UPDATE("")
```

The UTILITY command also automatically passes certain task equations and file equations to the program initiated. These equations make it possible for the program to use the unsaved work file, work source, or work object associated with the session. Certain utilities, such as the Editor, are designed to accept these task and file equations. Such programs must be initiated with the UTILITY command instead of the RUN command. For details about the task and file equations that are passed, refer to the UTILITY command discussion in the *A Series CANDE Operations Reference manual*.

Initiating Compilations from CANDE

You can use the COMPILE command to compile a program. This command allows you to specify the compiler to use, the input file titles, the object code file title, and task equations for the compiler and the resulting object code file. For example:

```
COMPILE DAILY/UPDATE/PATCH AS DAILY/UPDATE/NEW WITH COBOL74;
  COMPILER FILE SOURCE = DAILY/UPDATE/SOURCE;
  PRIORITY = 40;
```

This example initiates the COBOL74 compiler, specifying a primary input file called DAILY/UPDATE/PATCH and a secondary input file called DAILY/UPDATE/SOURCE. The object code file that results is called OBJECT/DAILY/UPDATE/NEW. The compiler stores the PRIORITY assignment in the resulting object code file, so that OBJECT/DAILY/UPDATE/NEW receives a default PRIORITY value of 40 whenever it is run.

The COMPILE command can be used more simply than it is in the preceding example. Suppose that DAILY/UPDATE is your work file. Simply entering *COMPILE* in your CANDE session is sufficient to compile your work file. CANDE chooses the compiler that matches the file type of the source file. The resulting object code file consists of the source file title with "OBJECT/" prefixed (for example, OBJECT/DAILY/UPDATE.)

The COMPILE command cannot cause the execution of the resulting object code file. However, a simple RUN command compiles and runs the work file if no object code file exists.

Initiating Utilities from CANDE

The RUN and UTILITY commands can be used to initiate a variety of system utility programs such as FILECOPY, LOGGER, and so on. However, CANDE also includes a number of specialized commands that you can use to initiate particular utilities. The following are the commands and the names of the corresponding utilities:

Command	Utility
BACKUPPROCESS	Backup Processor
DCSTATUS	DCSTATUS
LFILES	FILEDATA
LOG	LOGANALYZER

Submitting WFL Jobs from CANDE

You can submit WFL programs from CANDE sessions using the START or WFL command. The START command submits a WFL program that is stored in a disk file. The WFL command enables you to enter WFL statements directly at the terminal.

The START command can pass any number or type of parameters that are expected by the WFL program. In addition, you can use the *FOR SYNTAX* clause for syntax

checking. This clause causes the program to be compiled, but not executed, and displays information about any syntax errors in the WFL program. You can also assign the STARTTIME task attribute to delay initiation of the program. However, you cannot assign any other task attributes to the program.

While the WFL program is compiling, only CANDE control commands are available. If you enter any other CANDE commands during this period, CANDE queues the commands and executes them when the compilation is finished. However, after the WFL program is compiled and entered in a job queue, all CANDE commands are available again. The WFL program executes as a job and can have a job summary or printer backup files associated with it. By default, these files are queued for printing when the WFL program terminates.

You can use the WFL command to submit one or more WFL statements. Simply enter *WFL*, followed by the WFL statements. You can omit the *?BEGIN JOB* and *?END JOB* statements. The WFL statements can include all the constructs defined in WFL with the exception of data specifications and STARTTIME specifications.

When you submit WFL input by way of the WFL command, only CANDE control commands are available while the WFL input compiles and executes. Any other CANDE commands that you enter during this period are queued for later execution. By default, any backup files created by the WFL process are saved with the CANDE session. The files are queued for printing when you end the CANDE session.

The CANDE *ADD*, *COPY*, and *PRINT* commands correspond to the WFL commands of the same names. When you enter any of these commands, CANDE passes it to WFL for execution.

Access to Task Attributes in CANDE

For each session, CANDE stores information about a few selected task attributes. CANDE requests some of this information from the user at log-on time and obtains most of the rest from usercode attributes defined in the USERDATAFILE. CANDE assigns these task attribute values to any process initiated by that session (for example, by a CANDE *RUN* command). The task attributes stored by CANDE include the following:

ACCESSCODE	JOBNUMBER	PRINTDEFAULTS
CHARGE	JOBSUMMARY	PRIORITY
CONVENTION	JOBSUMMARYTITLE	SOURCESTATION
DESTNAME	LANGUAGE	STATION
FAMILY	NOJOBSUMMARYIO	USERCODE

For a task initiated from a CANDE session, CANDE assigns a JOBNUMBER equal to the session number. A job initiated from a CANDE session receives a JOBNUMBER equal to its MIXNUMBER.

You can use CANDE commands to change the values of some of the session attributes. By using these commands, you create new defaults that are applied to all tasks initiated later in that session. The ACCESS, CHARGE, DESTNAME, FAMILY, and LANGUAGE

commands each display or assign the session attribute of the same name. Additionally, the PDEF command displays or assigns the PRINTDEFAULTS session attribute.

You can also assign task attributes to specific processes through the use of task equations. Task equations can be appended to most CANDE process initiation statements, including RUN, UTILITY, COMPILE, and the various special-purpose commands for initiating utilities. Task equations can assign values to all but task-valued or event-valued task attributes, such as EXCEPTIONTASK or EXCEPTIONEVENT. If a task equation conflicts with task attribute inheritance, the task equation takes precedence. For example, the following CANDE command assigns to a process a LANGUAGE value different from the LANGUAGE value of the session:

```
RUN DRIVER;LANGUAGE = FRANCAIS
```

For information about the task attributes available in CANDE, refer to the *A Series CANDE Operations Reference Manual.*

Monitoring and Controlling Processes in CANDE

Any messages generated by a task initiated from a CANDE session are automatically displayed at that session, including any "BOT", "EOT", DISPLAY, and RSVP messages and error or warning messages. However, for processes indirectly associated with a session, the display of messages is optional. Processes indirectly associated with a session include WFL processes initiated by a *START* or *WFL* command, the descendants of such processes, and the descendants of any task initiated from a session.

The CANDE session option *MSG* controls the display of messages by processes indirectly associated with a session. While MSG is set, all messages generated by such indirect processes are displayed at the session. While MSG is reset, all such messages are suppressed. CANDE sets the MSG option to TRUE if the usercode attribute CANDEGETMSG is set for the usercode of the session. You can also set the MSG option to TRUE for a session by entering a CANDE *SO MSG* command. You can use the equivalent CANDE control command, *?SO MSG*, even when the station is busy.

A number of CANDE control commands are available for monitoring and controlling particular processes. You can use these commands to monitor or control any process that has the same usercode as the session usercode. This includes processes initiated from the current session as well as processes initiated from other sources, such as MARC or the ODT.

Most CANDE commands related to process control correspond to system commands with similar names. Some restrictions and differences in spelling apply to the CANDE versions of these commands. For further information, refer to "Tasking Command Equivalents" later in this section.

The system assigns a unique mix number, also known as the *session number*, to each CANDE session. The CANDE session does not appear as a process in mix display commands. However, the session mix number does appear in the output from two system commands: Y (Status Interrogate) and C (Completed Mix Entries). The output from these commands shows both the job number and the mix number of a process. If

the process is a task, and it was initiated from a CANDE session, then the job number shown is actually the CANDE session number.

Saving CANDE Commands for Later Use

You can achieve some of the convenience of programmatic task initiation and control by saving CANDE commands in a file for later use. You can use the DO or SCHEDULE command to execute the commands in the file. You can reuse the file as many times as desired.

The DO command takes effect immediately and prevents you from using most other commands in the session until the DO file is completed. However, you can use the SCHEDULE command to cause the file to be executed separately from your current session or at a later time.

Files that store CANDE commands are different from programs in that they are not compiled and are not executed as separate processes. Their process control abilities are more limited than those of WFL, ALGOL, or COBOL74 programs, because no conditional statements or variables are available.

CANDE Programming Considerations

When you design a program to be run from CANDE, you need to be aware of CANDE features affecting parameter passing, task attribute access, and terminal communications.

Receiving Parameters from CANDE

If you are designing a program to be initiated from CANDE, be aware that the program can receive only one parameter from the RUN or UTILITY command that initiates it. This parameter appears as a string to the user, but in the program it must be declared as type Real Array (or compatible parameter type) with an unspecified lower bound. For information about Real Array parameters and compatible parameter types, refer to Section 17, "Using Parameters."

Access to Ancestral Processes in CANDE

If you initiate a task from a CANDE session, and that task accesses its own EXCEPTIONTASK task attribute, the system interprets EXCEPTIONTASK as a reference to the CANDE MCS. The task can use the EXCEPTIONTASK task attribute to query the values of the task attributes of the CANDE MCS. However, if the task attempts to modify the task attributes of the CANDE MCS, the task is terminated with a task attribute error. This error occurs because CANDE runs with the private process option of the OPTION task attribute set to TRUE.

For a task initiated through a CANDE *RUN* command, the MYSELF task variable, the MYJOB task variable, and the PARTNER task attribute all have the same meaning. In most cases, each of these constructs refers to the task itself. However, when

these constructs are used to access the JOBSUMMARY, JOBSUMMARYTITLE, and NOJOBSUMMARYIO task attributes, these task attributes affect the job summary of the CANDE session. For example, if the task sets the JOBSUMMARY attribute of the MYSELF attribute to SUPPRESSED, the job summary for the session is suppressed.

The following ALGOL example assigns a value of SUPPRESSED to the JOBSUMMARY task attribute of the session. If this program is initiated by a CANDE *RUN* command, the program prevents a job summary from being printed when the session ends.

```
BEGIN
  MYJOB.JOBSUMMARY := VALUE(SUPPRESSED);
END.
```

For WFL statements submitted through a CANDE *WFL* command, the MYJOB task variable refers to the WFL compiler process. The NAME of the WFL compiler process in this case is *CANDE WFL*, prefixed by the usercode of the session. The MYSELF task variable refers to the task that is executing the compiled WFL statements. The NAME of this task is *WFLCODE*, prefixed by the usercode of the session. MYSELF(JOBNUMBER) returns the CANDE session number, but MYJOB(MIXNUMBER) returns the mix number of the WFL compiler process.

When statements submitted through the WFL command use the MYJOB construct to alter job summary-related task attributes, these changes affect the job summary of the CANDE session. However, if the MYSELF variable is used to access these task attributes, there is no effect on the job summary of the CANDE session.

The MCSNAME task attribute of tasks initiated from CANDE sessions typically returns a value of *SYSTEM/CANDE*, which might or might not be preceded by an asterisk (*). Note that the MCSNAME value can be different if CANDE was installed at your site under a different name.

Communicating with CANDE Terminals

CANDE automatically assigns the logical station number (LSN) to the STATION and SOURCESTATION task attributes of processes initiated from CANDE sessions. Most programs initiated from CANDE can therefore declare and open a remote file at the originating CANDE session without having to explicitly assign the STATION task attribute or otherwise indicate where the remote file is to be opened.

However, for WFL jobs submitted by a START command, CANDE does not assign the LSN to the STATION task attribute. CANDE does not assign a STATION value to such WFL jobs because these WFL jobs are typically intended to run independently of the originating CANDE session. In fact, a WFL job or offspring task that opens a remote file at a CANDE session becomes to some extent dependent on that session. If the user logs off and the process attempts to write to the remote file, the process receives an I/O error.

It is nevertheless possible for a WFL job submitted through a CANDE *START* command to initiate a task that opens a remote file. However, you must take certain precautions to enable the task to open the remote file successfully. The simplest precaution is for the job to assign its own SOURCESTATION value to its STATION value before initiating any tasks. For example:

```
?BEGIN JOB;
MYSELF(STATION = MYSELF(SOURCESTATION));
RUN OBJECT/X;
RUN OBJECT/Y;
?END JOB
```

An alternate precaution is for the job to file-equate the TITLE attribute of a program's remote file to the job's SOURCENAME task attribute value. Like the STATION assignment shown previously, this file equation causes the remote file to be opened at the originating station. In the following example, REM is the internal name of a remote file used by the program OBJECT/PROG:

```
?BEGIN JOB;
  RUN OBJECT/PROG;
    FILE REM(TITLE = #MYSELF(SOURCENAME));
?END JOB
```

MARC

MARC is a COMS transaction processor that enables you to perform system operations and tasking functions. You initiate communications with MARC by opening the MARC window. Depending on the way your terminal is defined to COMS, the MARC window might appear automatically after you log on to COMS. If it does not, you might still be able to open the MARC window by entering the command *?ON MARC*. Your interactions with MARC between the time you open the MARC window and the time you log off or close the window are referred to as a *session*. MARC assigns each session an identifying number called the *session number*.

MARC Tasking Capabilities

MARC provides the only menu-assisted interface to tasking. You can use MARC menu selections or commands to submit WFL jobs or to initiate programs written in any language.

MARC offers commands and menu selections for initiating dependent processes, submitting WFL jobs, and initiating utilities. Once the process is initiated, MARC displays the TAsk command in the Action field of the current screen. By transmitting this command, you can display a special screen called TASKSTATUS. You can use the TASKSTATUS screen to monitor and control the process.

Because the system administrator can modify MARC to add or delete functions, some features mentioned here might not be available at your site. The descriptions apply to the version of MARC supplied by Unisys.

The following paragraphs provide an overview of MARC tasking capabilities. For further details about these features, refer to the *A Series Menu-Assisted Resource Control (MARC) Operations Guide*.

Initiating Dependent Processes from MARC

You can enter RUN in the choice field of the MARC home menu to initiate a program as a dependent process. This selection can initiate a program written in any language except WFL. Entering this selection displays the RUN screen. You can use the RUN screen to specify the object code file title, any parameter that is to be passed, and any assignment to the TASKVALUE task attribute. You enter TASKVALUE assignments in the Value field of the screen.

An alternate method of initiating dependent processes is by using the RUN command. You can enter this command in the Action field of a screen or on the COMND screen. The syntax of this command is similar to the WFL *RUN* statement, except that the command can pass only a single parameter. Depending on the requirements of the program being initiated, the parameter can be a string of characters enclosed in quotation marks (") or a number with no quotation marks. The following are both valid examples:

```
RUN OBJECT/RECOMM("REPORT=DAILY")

RUN OBJECT/TELEMAX(346)
```

Initiating Compilations from MARC

You can initiate compilations from MARC in either of the following ways:

- By using the MARC *WFL* command to submit a WFL *COMPILE* statement. For details, refer to "Submitting WFL Jobs from MARC" later in this section.
- By using the EDIT screen to initiate an Editor session. While in the Editor, you can use the Editor COMPILE command to initiate a compilation.

Initiating Utilities from MARC

You can initiate utilities by using either the RUN screen or the RUN command. However, you can also use either of two special screens, UTIL or TOOLS, which list many utilities as selections. By choosing one of the selections on the UTIL or TOOLS screen, you cause the corresponding utility to be initiated. If parameters are needed, MARC prompts you to supply them.

Submitting WFL Jobs from MARC

You can use the START selection on the MARC screen to submit a WFL program that is stored in a disk file. Entering this selection displays the START screen. Use this screen to enter the file title of the WFL program and any parameter values to be passed

to the program. You can also use this screen to enter a value for the STARTTIME task attribute of the WFL program.

WFL programs stored in disk files can also be initiated by way of the START command. The START command can pass parameters to the WFL program, but cannot include a STARTTIME specification.

You can use the MARC *WFL* command to submit WFL statements directly at the terminal. Simply type the word *WFL*, followed by the statements that constitute the WFL program. You can omit the *?BEGIN JOB* and *?END JOB* statements. The program cannot include any WFL constructs except data specifications or a STARTTIME specification. For example, the following WFL input initiates another program and assigns it a task attribute:

```
WFL RUN OBJECT/INVENTORY;FAMILY DISK = DPMAST OTHERWISE DISK
```

Monitoring Processes Initiated from MARC

When you initiate any dependent process, WFL job, or utility from a MARC session, the TASK command appears as a prompt on the current screen. Entering *TASK* in the Action field displays the TASKSTATUS screen. This screen displays information about the process and includes a field in which you can enter process control commands. You can leave the TASKSTATUS screen at any time by entering one of the screen traversal commands, such as HOME or GO, that are displayed. As long as the process is running, you can return to the TASKSTATUS screen by using the TASK command.

The TASKSTATUS screen includes fields that display various types of information for the process. The following are the fields and their meanings:

- The Task field displays the mix number and the name of the process.
- The Parameter field, if it appears, displays the value of the parameter passed to the process.
- The Task Status field displays the current stack state of the process. For a discussion of what the stack states mean, refer to Section 6, "Monitoring and Controlling Process Status."
- The Elapsed field displays the time elapsed since the process was initiated.
- The Processor field displays the processor time used by the process.
- The I/O field displays the accumulated I/O initiation time for the process.
- The area below the Elapsed, Processor, and I/O fields displays messages generated by the process, including "BOT", "EOT", DISPLAY, and RSVP messages.

You can enter process control commands in the Action field. The list of available actions below the Action field includes the most common system commands used for process monitoring and control. You can enter any of the listed commands without having to prefix them with the mix number of the process; MARC automatically prefixes the command with the mix number listed in the Task field. You can also enter system process control commands that are not listed as actions, but you must prefix them

with the mix number of the process. For a list of system commands related to process monitoring and control, refer to "Tasking Command Equivalents" in this section.

If you submit a WFL job by way of the START screen or the START command, then the process control commands are displayed only during the compilation of the job. However, you can enter these commands even after they no longer appear as prompts, provided that you prefix them with a mix number. You can prefix them with the mix number of the job or of any task initiated by the job. The TASKSTATUS screen continues to display any messages generated by the job as it executes. You can initiate another process as soon as the job has finished compiling and has been inserted in a job queue.

However, if you submit a WFL job by way of the MARC *WFL* command, the process control commands continue to be displayed as the job executes. Also, it is not possible to initiate new processes until the job terminates.

If you initiate a process that initiates offspring, then any messages created for the offspring are included with the other process messages on the TASKSTATUS screen. You can enter process control commands for the offspring in the Action field, but you must always prefix the command with the mix number of the offspring process.

You can usually learn the mix number of the offspring by looking at its "BOT" message in the process messages display. However, if MARC has scrolled this message off the screen, you can learn the mix number by entering the VIEW command in the Action field. This command causes MARC to display the TASKVIEW screen, which lists the mix numbers and the names of the original process and all its descendants in a hierarchical order.

You cannot enter process control commands on the TASKVIEW screen. You can display the TASKSTATUS screen for a particular offspring by entering the mix number of the offspring in the Action field of the TASKVIEW screen. You can then enter process control commands on that TASKSTATUS screen. Alternatively, you can return from the TASKVIEW screen to the original TASKSTATUS screen by entering the RETURN command in the Action field.

Monitoring Other Processes in MARC

All system commands related to process monitoring and control can be entered through MARC, except for the primitive commands (commands preceded by two question marks). You can use these commands to monitor or control processes initiated from the current MARC session or processes initiated from other sources, such as CANDE or an ODT.

You can enter system commands on the COMND screen or in the Action field of any screen that displays "COmnd" as a prompt. However, system commands that you enter through MARC are screened for security. Many system commands are available only if the usercode of the session has privileged, SYSTEMUSER, or security administrator status. For details, refer to "Tasking Command Equivalents" in this section.

Each MARC session receives a unique mix number, also called the *session number*, which appears in the output from some system commands, including mix display commands. The MARC session does not appear as a process in mix display commands. However, the

session mix number does appear in the output from two system commands: Y (Status Interrogate) and C (Completed Mix Entries). The output from these commands shows both the job number and the mix number of a process. If the process is a task, and it was initiated from a MARC session, then the job number shown is the MARC session number.

Communicating with Interactive Processes in MARC

A special window called a *task window* is created if a remote file is opened by a process run from a MARC session. In most cases, when the process opens the remote file, MARC automatically displays the task window. The current screen disappears and MARC displays the following message:

```
Enter ?MARC for task status
```

If the process writes to the remote file, the messages appear in the task window. If you type and transmit any text in the task window, MARC interprets this as input to the remote file. The only exceptions are the ?MARC command and other Communications Management System (COMS) commands that are prefixed with question marks.

You can return to the TASKSTATUS screen by entering the ?MARC command. You can return to the task window by entering the TASK command in the Action field of any screen.

If you are on the task window when the process terminates, then MARC returns you to the originating screen. In some cases, MARC prompts you to press the SPCFY key before making this transfer. For information about why this happens, refer to "Communicating with MARC Terminals" later in this section.

Note that if you submit a WFL job through the START command and the job initiates a task that opens a remote file, you are not automatically transferred to the task window when the remote file is opened. When the task opens the remote file, a message of the following form appears on the TASKSTATUS screen:

```
<time> <mix number> Remote window <remote window name> OPEN.
INTNAME = <internal name>.  PROGRAM = <object code file title>.
```

Note the <remote window name> value in this message. You can transfer to the remote window by entering a command of the form:

```
?ON <remote window name>
```

You can return to the TASKSTATUS screen by entering the following command:

```
?ON MARC
```

The shorter form, ?MARC, is not accepted in this situation.

Access to Task Attributes in MARC

For each session, MARC stores information about a few selected task attributes. MARC requests some of this information from the user at log-on time and obtains the rest from usercode attributes defined in the USERDATAFILE. MARC assigns these task attribute values to any process initiated by that session (for example, by a MARC *RUN* command). The task attributes stored by MARC include the following:

BACKUPFAMILY	JOBNUMBER	PRINTDEFAULTS
CHARGE	JOBSUMMARY	SOURCESTATION
CONVENTION	JOBSUMMARYTITLE	STATION
DESTNAME	LANGUAGE	USERCODE
EXCEPTIONTASK	NOJOBSUMMARYIO	
FAMILY	PRIORITY	

For a task initiated from a MARC session, MARC assigns a JOBNUMBER equal to the session number. A job initiated from a MARC session receives a JOBNUMBER equal to its MIXNUMBER.

Certain of the session attributes established for MARC dialogue 1 are inherited by any sessions started in other MARC dialogues; these session attributes are USERCODE, ACCESSCODE, CHARGE, FAMILY, and LANGUAGE.

MARC provides commands and menu selections that you can use to set the values of the following attributes: DESTNAME, FAMILY, JOBSUMMARY, JOBSUMMARYTITLE, LANGUAGE, NOJOBSUMMARYIO, and PRINTDEFAULTS. The other attributes in the previous list cannot be accessed by the user.

You can also assign task attributes to specific processes by using task equations. You can enter task equations in MARC in either of the following ways:

- FILEEQUATE screen

  The RUN screen includes boxes you can fill to indicate that file equations or task attribute assignments are needed. If file equations are needed, the FILEEQUATE screen is displayed. You can enter any number of file equations. Implicitly, these are assignments to the FILECARDS task attribute. If task attribute assignments are needed, the TASKATTR screen is displayed. This screen includes fields for assigning selected task attributes. Only the following task attributes can be assigned: BDNAME, DESTNAME, LANGUAGE, MAXLINES, OPTIONS, STATION, TADS, and SW1 through SW8.

- RUN command

  When you initiate a task by using a RUN command, you can include task equations that assign task attribute values for the task. The following RUN command includes several task equations:

```
RUN OBJECT/PROGA;TASKVALUE=1;DISPLAYONLYTOMCS=TRUE;FILE OUT=OUT/FILE;
```

MARC Programming Considerations

When you design a program to be run from MARC, you need to be aware of MARC features affecting parameter passing, task attribute access, and terminal communications.

Receiving Parameters from MARC

If you are designing a program to be initiated from MARC, be aware that the program can receive only one parameter from the RUN screen or RUN command that initiates it. If the user encloses the parameter in quotation marks ("), MARC passes the parameter as type Real Array with an unspecified lower bound. If the user does not enclose the parameter in quotation marks, MARC passes the parameter as type Real. For information about the parameter types in each language that are compatible with the Real and Real Array types, refer to Section 17, "Using Parameters."

Access to Ancestral Processes in MARC

If you initiate a task through the MARC *RUN* command and that task accesses its own EXCEPTIONTASK task attribute, the system interprets EXCEPTIONTASK as a reference to the MARC library, *SYSTEM/MARC/COMMANDER. The task can use the EXCEPTIONTASK task attribute to query the values of the task attributes of the MARC MCS. However, if the task attempts to modify the task attributes of the MARC MCS, the task is terminated with a task attribute error. This error occurs because MARC runs with the private process option of the OPTION task attribute set to TRUE.

For tasks initiated through a MARC *RUN* command, the MYJOB task variable and the PARTNER task attribute act as synonyms for the MYSELF task variable. When such a task uses MYJOB or PARTNER to access any task attributes, the task attributes accessed are those of the task itself. However, if the task changes the values of the job summary-related task attributes, the changes affect the job summary of the MARC session. The job summary-related task attributes are JOBSUMMARY, JOBSUMMARYTITLE, and NOJOBSUMMARYIO.

For WFL statements submitted through a MARC *WFL* command, the MYJOB task variable refers to the WFL compiler process. The NAME of the WFL compiler process in this case is *MARC WFL*, prefixed by the usercode of the session. The MYSELF task variable refers to the task that is executing the compiled WFL statements. The NAME of this task is *WFLCODE*, prefixed by the usercode of the session. MYSELF(JOBNUMBER) returns the MARC session number, but MYJOB(MIXNUMBER) returns the mix number of the WFL compiler process.

When statements submitted through the WFL command use the MYJOB construct to alter job summary-related task attributes, these changes affect the job summary of the MARC session. However, if the MYSELF variable is used to access these task attributes, there is no effect on the job summary of the MARC session.

Note: When you use the JOBSUMMARY command to display the current JOBSUMMARY value for the session, the output does not reflect any JOBSUMMARY assignments made by tasks of the session. Nevertheless, such assignments made by tasks do affect the job summary of the session unless overridden by a later JOBSUMMARY command.

The MCSNAME task attribute of tasks initiated from MARC sessions typically returns a value of *SYSTEM/COMS*, which might or might not be preceded by an asterisk (*).

Communicating with MARC Terminals

MARC automatically assigns the logical station number (LSN) to the STATION and SOURCESTATION task attributes of processes initiated from MARC sessions. The one exception is that, for WFL programs submitted by a START command, MARC does not assign the LSN to the STATION task attribute.

Most programs initiated from MARC can therefore declare and open a remote file without having to explicitly assign the STATION task attribute or otherwise indicate where the remote file is to be opened. However, when you use a START command to submit a WFL job from MARC, the job and its descendants must take precautions before attempting to open any remote files. The simplest precaution is for the job to assign its own SOURCESTATION value to its STATION value before initiating any tasks. For example:

```
?BEGIN JOB;
MYSELF(STATION = MYSELF(SOURCESTATION));
RUN OBJECT/X;
RUN OBJECT/Y;
?END JOB
```

An alternate precaution is for the job to file-equate the TITLE attribute of a program's remote file to the job's SOURCENAME task attribute value. Like the STATION assignment shown previously, this file equation causes the remote file to be opened at the originating station. In the following example, REM is the internal name of a remote file used by the program OBJECT/PROG.

```
?BEGIN JOB;
  RUN OBJECT/PROG;
    FILE REM(TITLE = #MYSELF(SOURCENAME));
?END JOB
```

The "Communicating with Interactive Processes" subsection pointed out that MARC opens a task window to enable a process to communicate with a user through a remote file. You can use the AUTOSWITCHTOMARC attribute to affect the handling of the task window for users. If you set the AUTOSWITCHTOMARC task attribute to TRUE, then users of the program are automatically transferred from the task window to the originating screen when the process terminates. If AUTOSWITCHTOMARC is FALSE, then the user must press the SPCFY key to return to the originating screen.

ODT

An operator display terminal (ODT) is any data comm terminal or workstation that is connected to the system through one of the following types of data link processors: the ODT-DLP or the UIP-DLP. The system provides ODTs with access to two operational modes: *system command mode* and *data comm mode*. When an ODT is in data comm mode, you can log on to COMS and use various programs that run under COMS, such as MARC. When an ODT is in system command mode, you can enter system commands or view automatic displays of system information.

The following subsections discuss tasking capabilities and programming considerations for an ODT running in system command mode. For details about any of the system commands, refer to the *A Series System Commands Operations Reference Manual*.

ODT Tasking Capabilities

The ODT provides you with the capability to submit WFL jobs and initiate dependent or independent processes. The ODT also enables you to conveniently monitor all the processes in the system mix.

Submitting WFL Jobs from an ODT

You can submit WFL programs at an ODT by using any of the following methods:

- Typing in an entire WFL program, including a BEGIN JOB statement at the start, and then transmitting it. (There is no need to include an END JOB statement.)
- Entering one or more WFL statements preceded either by a question mark (?) or by the letters "CC". (The BEGIN JOB is not necessary in this case.)
- Entering one of a certain group of WFL statements that do not require a BEGIN JOB or any other prefix when used at the ODT. These include COMPILE, COPY, PROCESS, RERUN, RUN, and START.
- Using the LD (Load Control Deck) system command, which submits a WFL program that is stored on tape.

When you submit a WFL program through the LD command, the system executes the program as a WFL job, which goes through the job queue mechanism. When you submit WFL statements through the other methods listed previously, the system usually executes the input as a WFL job. However, the system can execute some statements directly, without creating a WFL job. Such statements do not pass through the job queue mechanism, and therefore are not affected by job queue attributes. For a list of these statements, refer to Section 4, "Tasking from Programming Languages."

For further details about submitting WFL programs from an ODT, refer to the *A Series Work Flow Language (WFL) Programming Reference Manual*

Initiating Processes from an ODT

You can use the ??RUN (Run Code File) primitive system command to initiate a program as an independent process. The program can be written in any language except WFL. The resulting process receives its own job file and job summary.

Note that if you enter *RUN* without the two question marks, the system treats this as the WFL *RUN* statement. The system creates a WFL job to execute the RUN statement and enters the job in a job queue. The job can be delayed by the queue mix limit or affected by other job queue attributes. Further, the job affects the job queue active count. Therefore, you might prefer to use ??RUN to initiate processes, such as MCSs, that you do not wish to go through the job queue mechanism.

Initiating Compilations from an ODT

You can initiate compilations at an ODT by using the WFL *COMPILE* statement. The system responds to this command by creating a WFL job that includes the COMPILE statement and sending it through the job queue mechanism for initiation.

Initiating Utilities from an ODT

Utilities can be initiated at the ODT by way of the ??RUN command or the WFL *RUN* statement. There are other system commands that initiate specific utilities, such as the TDIR (Tape Directory) command, which initiates the FILEDATA utility to list the directory of a tape, and the DA (Dump Analyzer) system command, which initiates the DUMPANALYZER utility.

Two WFL statements that initiate specific utilities can be entered at the ODT. The LOG statement initiates the LOGANALYZER utility, and the PB statement initiates the BACKUP utility. To use the WFL *PB* statement at the ODT, you must prefix it with a question mark (?); otherwise, the system interprets it as the PB (Print Backup) system command, which does not initiate the BACKUP utility.

Monitoring and Controlling Processes at an ODT

Of all the interactive sources for process initiation, the ODT provides the most complete selection of commands for monitoring and controlling processes. The operator can use these system commands to monitor or control all the processes on the system, including processes initiated from any of the sources discussed in this section. These system commands are listed under "Tasking Command Equivalents" in this section.

A unique feature of the ODT is Automatic Display mode. You initiate and control this mode by using the ADM (Automatic Display Mode) system command. You can use this feature to cause various types of information to be displayed at intervals, such as active entries, waiting entries, completed entries, and process messages. This feature allows you to monitor processes from beginning to end without having to enter commands repeatedly.

By default, Automatic Display mode displays seven lines of A (Active Mix Entries) system command output, three lines of W (Waiting Mix Entries) system command

output, two lines of S (Scheduled Mix Entries) system command output, five lines of C (Completed Mix Entries) system command output, and devotes the remainder of the display to MSG (Display Messages) system command output. By default, the system updates the contents of the display every nine seconds. You can use the ADM command to cause different system commands to be displayed or to change the time interval for updates to the display.

Access to Task Attributes from an ODT

You can include task equations after a WFL task initiation statement submitted from the ODT. Also, if you type in a complete WFL job at the ODT, you can include task attribute assignments in the job attribute list. However, you cannot include task equations after the ??RUN command.

When you initiate a process from the ODT, the process typically does not inherit any of the task attributes that it would if you initiated the process from a MARC or CANDE session. For example, the USERCODE, ACCESSCODE, CHARGE, and FAMILY values of the process are usually null, unless explicitly assigned.

However, usercode attributes are inherited in the following two cases:

- If you submit a WFL job that includes a USERCODE assignment in the job attribute list, then the following task attributes of the WFL job inherit values from the corresponding usercode attributes: ACCESSCODE, CHARGE, CLASS, FAMILY, PRINTDEFAULTS, and PRIORITY. This inheritance can be overridden by assignments to these attributes in the job attribute list.
- You can use the TERM (Terminal) system command to assign a terminal usercode to an ODT. This usercode is inherited by WFL jobs submitted from the ODT, unless overridden by a USERCODE assignment in the job attribute list. The job also inherits values for the same set of task attributes listed in the previous item in this list.

Note that programs initiated by a ??RUN command do not inherit the terminal usercode or any other usercode attributes.

Special types of security status apply to nonusercoded processes and certain WFL statements when they are entered at the ODT. These privileges are discussed in Section 5, "Establishing Process Identity and Privileges."

ODT Programming Considerations

When you design a program to be run from the ODT, you need to be aware of ODT features affecting parameter passing, task attribute access, and terminal communications.

Receiving Parameters from an ODT

If you are designing a program to be initiated by the ??RUN primitive system command, be aware that the program cannot receive any parameters.

If the program is to be initiated by a WFL *RUN* statement entered at an ODT, the program can receive the four parameter types passed by WFL: Boolean, integer, real, and string. The string parameter should be declared in the program as a real array (or compatible parameter type) with an unspecified lower bound. For information about real array parameters and compatible parameter types, refer to Section 17, "Using Parameters."

Access to Ancestral Processes in the ODT Environment

For a process initiated by the ??RUN primitive system command, the MYJOB task variable and the EXCEPTIONTASK and PARTNER task attributes are all references to the process itself.

For a process initiated by a WFL *RUN* statement at an ODT, MYJOB, EXCEPTIONTASK, and PARTNER are all references to the WFL job that was created by the system to execute the RUN statement. The name of this WFL job consists of the first 17 characters of the WFL input you submitted.

Communicating with an ODT

Interactive programs that are designed for use at remote terminals might not run successfully if initiated from the ODT. You must design the program somewhat differently if it is to be initiated at an ODT. If the process opens a file with KIND = REMOTE, it is discontinued with an "UNKNOWN FILE/STATION" error. The process should open a file with KIND = ODT instead. A process can determine whether it was initiated from an ODT or a remote terminal by interrogating the SOURCEKIND task attribute.

A process can open a file either at a labeled ODT or at a scratch ODT. A labeled ODT is one that has been assigned a label by the LABEL (Label ODT) system command. A scratch ODT is one that has not been assigned such a label.

To open a file at a labeled ODT, a process should first set the TITLE file attribute to match the label assigned to the ODT. In addition, the NEWFILE file attribute value should be FALSE or else unspecified. If NEWFILE is unspecified, the MYUSE file attribute value should be IN or IO. When the process runs, the system opens the remote file at any ODT with a matching label. If none of the ODTs has a matching label, the process is suspended with a "NO FILE <file title> (SC)" RSVP message. The process resumes execution when an operator uses the LABEL command to label an ODT with the requested file title.

To open a file at a scratch ODT, a process should set the NEWFILE file attribute to TRUE, or leave NEWFILE unspecified and set MYUSE to OUT. The value of the TITLE file attribute makes no difference in this case. If the process was initiated from an ODT, and that ODT is a scratch ODT, the system opens the file at that ODT. Otherwise, the system selects another scratch ODT and opens the file there.

To open a file at a particular ODT, regardless of whether that ODT is labeled or scratch, the process can assign the UNITNO file attribute a value equal to the physical unit number of the ODT. The system opens the file at the requested ODT even if the ODT

is labeled and the label does not match the TITLE file attribute. However, note that use of the UNITNO file attribute is restricted on systems running InfoGuard security enhancement software at the S2 level; refer to the *Security Administration Guide* for details.

To open a file at the ODT where the process was initiated, regardless of whether that ODT is labeled or scratch, the process should first read the physical unit number from its own SOURCESTATION or ORGUNIT task attribute value. The process can then assign the physical unit number to the UNITNO file attribute, as described previously.

When a process opens an ODT file, automatic display mode at the ODT is temporarily suspended. However, system commands continue to be available. You can enter text into the ODT file by preceding the text with a GS character. The GS character is also known as the *delta* character and looks like an upward-pointing triangle. (Do not confuse the GS character with the circumflex character, which resembles an inverted letter V.) Refer to the documentation for your terminal to find out whether your terminal supports the GS character, and which key it is mapped to.

You can indicate that there is no more input, and cause an end-of-file condition, by entering the GS character, followed by ?END.

When the process closes the ODT file, the system removes the label from the ODT and resumes Automatic Display mode. You can also resume Automatic Display mode while the ODT file is still open by entering an ADM OK command at the ODT.

An example of a program that uses an ODT file is given in the ORGUNIT description in the *A Series Task Attributes Programming Reference Manual.*

Tasking Command Equivalents

MARC and the ODT allow you to enter almost all of the same system commands for process initiation, monitoring, and control. In addition, CANDE allows you to enter process control commands that correspond fairly closely to system commands.

The system commands available in MARC for process control are spelled the same as those available at an ODT, and have the same functionality, with the following exceptions:

- Security

  If the Communications Management System (COMS) security category COMMANDCAPABLE is defined, then system commands can be submitted in MARC only by users defined as COMMANDCAPABLE. Further, some commands are available only to users with SYSTEMUSER or privileged status. Some other commands are filtered: in other words, they are limited to monitoring and controlling processes with the same usercode as the MARC session. For further information about COMMANDCAPABLE, SYSTEMUSER, and privileged status, refer to the *A Series Security Administration Guide.*

- Spelling

  The MSG (Display Messages) system command is spelled SMSG in MARC.

CANDE process control commands differ from the corresponding system commands in the following ways:

- Spelling

  The CANDE process control commands each begin with a single question mark (?). In addition, the following spelling differences exist:

  - ?JA corresponds to the J (Job and Task Structure) system command.
  - ?CS corresponds to the mix number system command, which is formally known as the COMPILE STATUS (Information for Compiler Task) command. Note that the ?CS command in CANDE is not related to the CS (Change Supervisor) system command.
  - ?MXA corresponds to the MX (Mix Entries) system command. ?MXA can be abbreviated as ?MX or ?M.

- Implicit mix numbers

  For commands that apply to a dependent process initiated directly from the CANDE session, you can omit the mix number from the command. For example, instead of entering *?1234 Y*, you can enter simply *?Y*.

- Security

  In general, the CANDE process control commands can monitor or control only processes running with the same usercode as the CANDE session. If you attempt to apply a CANDE process control command to a process running with a different usercode, CANDE displays the message "INVALID NUMBER". However, CANDE makes one exception to this restriction. If you initiate a process in a CANDE session, and that process later changes its own usercode, CANDE still enables you to apply process control commands to that process.

- Mix display options

  The CANDE mix display commands (?C, ?JA, ?LIBS, and ?MXA) do not provide the following options of the equivalent system commands: ALL, IN, MCSNAME, QUEUE, and USER. However, the ALL option is implicitly set for all CANDE mix display commands. Furthermore, CANDE mix display commands do offer one feature that the corresponding system commands do not: the ability to specify a logical station number (LSN), which limits the display to processes originating from the specified station.

Table 3–1 shows the equivalent commands in these three interfaces and briefly states the function of each command. In Table 3–1, the abbreviations (f), (pu), and (su) are used in the MARC column to indicate commands that are filtered or that require SYSTEMUSER status or privileged status. For complete descriptions of these commands, refer to the *A Series System Commands Operations Reference Manual*, the *A Series Menu-Assisted Resource Control (MARC) Operations Guide* , and the *A Series CANDE Operations Reference Manual*. For a general introduction to process monitoring and control from an ODT, refer to the *A Series System Operations Guide*.

Table 3–1. Interactive Tasking Functions

Functional Area	ODT	MARC	CANDE	Specific Function
Initiating Processes	LD	LD (su & pu)	None	Initiate a WFL job from tape.
	??RUN	None	None	Initiate an object code file as an independent process.
	RUN	RUN	RUN, UTILITY	Initiate an object code file as a dependent process.
	<WFL statements>	WFL	WFL	Submit WFL statements.
	START	START	START	Submit a WFL program stored in a file.
Managing Queued WFL Jobs	DS	DS (f)	?<mixno> DS	Discontinue a queued WFL job.
	FS	FS (su)	None	Force initiation of a queued WFL job.
	MOVE	MOVE (su)	None	Change order of queued WFL jobs.
	PF	PF (su)	None	Display FETCH message associated with a WFL job.
	PQ	PQ (su)	None	Discontinue all the WFL jobs in a queue.
	PR	PR (su)	None	Change the priority of a queued WFL job.
	SQ	SQ (f)	?SQ	Display the WFL jobs in a queue.
	STARTTIME	STARTTIME (f)	?<mixno> STARTTIME	Assign a start time to a queued WFL job.
	Y	Y (f)	?<mixno> Y	Display information about a queued WFL job.

continued

Legend

f	Filtered if not SYSTEMUSER
pu	Privileged status required
su	SYSTEMUSER status required
mixno	Mix number

Table 3–1. Interactive Tasking Functions (cont.)

Functional Area	ODT	MARC	CANDE	Specific Function
Monitoring the Mix	ADM	None	None	Periodically display system mix and other items.
	C	COMND C (f)	?C	Display completed entries.
	DBS	DBS (su)	None	Display database stacks.
	J	J (f)	?JA	Display active mix entries, grouped into process families.
	LIBS	LIBS (f)	?LIBS	Display library processes.
	MSG	SMSG (f)	?MSG	Display process messages.
	MX	MX (f)	?MXA	Display active, scheduled, and waiting mix entries.
	S	S (su)	?S	Display scheduled mix entries.
	W	W (f)	?W	Display waiting mix entries.
Displaying Process Status	Y	Y (f)	?Y	Display current status of a process.
	<mixno>†	<mixno>	?<mixno> or ?CS	Display the status of a compilation.
	OT	OT (f)	?OT	Display contents of a selected word in the process stack.
Displaying Process Resource Usage	CU	CU (f)	?CU	Display current memory usage of a process.
	TI	TI (f)	?TI	Display accumulated processor, I/O, presence bit, ready queue, and elapsed times for a process.

† The <mixno> syntax is formally known as the COMPILE STATUS (Information for Compiler Task) system command.

continued

Legend

f	Filtered if not SYSTEMUSER
pu	Privileged status required
su	SYSTEMUSER status required
mixno	Mix number

Table 3–1. Interactive Tasking Functions (cont.)

Functional Area	ODT	MARC	CANDE	Specific Function
Communicating with a Process	HI	HI (f)	?HI	Cause process EXCEPTIONEVENT and optionally assign a TASKVALUE.
	AX	AX (f)	?AX	Pass a string of text to a process.
	IB	IB (su)	None	Display instruction block associated with a WFL job.
	PF	PF (su)	None	Print a FETCH message associated with a WFL job.
Modifying an Active Process	PR	PR (su)	None	Change the priority of a process.
	ST	ST (f)	?ST	Suspend execution of a process.
	DS	DS (f)	?DS	Abnormally terminate execution of a process.
Responding to Suspended Processes	AX	AX (f)	?AX	Pass a string of text to a process.
	DS	DS (f)	?DS	Discontinue a process.
	FA	FA (f)	?FA	Modify file attributes used by a process.
	FM	FM (su)	None	Change the printer form used by a process.
	FR	FR (f)	?FR	Specify that a tape reel is the last of a multireel set.
	IL	IL (su)	None	Change the physical unit used for an input file.
	NF	NF (f)	?NF	Return an open error to a process opening a file that is not an optional file.

continued

Legend

f	Filtered if not SYSTEMUSER
pu	Privileged status required
su	SYSTEMUSER status required
mixno	Mix number

Table 3–1. Interactive Tasking Functions (cont.)

Functional Area	ODT	MARC	CANDE	Specific Function
Responding to Suspended Processes (cont.)	NOTOK	NOTOK (f)	?NOTOK	Prevent the process from attempting a given action, but do not discontinue the process.
	OF	OF (f)	?OF	Indicate an optional file is not present.
	OK	OK (f)	?OK	Cause a suspended process to attempt to resume processing.
	OU	OU (su)	None	Change the physical unit used for an output file.
	RM	RM (f)	?RM	Remove a file specified in a DUP LIBRARY message.
	UL	UL (su)	None	Assign an unlabeled tape file to a particular process.
Saving and Restarting Processes	BR	BR (su)	None	Display checkpoint eligibility or initiate a checkpoint.
	OK	OK (f)	?OK	Allow automatic restart of a process.
	DS	DS (f)	?DS	Deny automatic restart of a process.
	RERUN	WFL RERUN	WFL RERUN	Initiate manual restart of a process.

Legend

f	Filtered if not SYSTEMUSER
pu	Privileged status required
su	SYSTEMUSER status required
mixno	Mix number

Communicating with an Operator

You can design a process to display information to an operator or accept information from an operator. You can accomplish this communication through any of the following methods:

- By accepting parameters from the operator in the statement that initiates the process. This topic is discussed earlier in this section under "Receiving Parameters from CANDE," "Receiving Parameters from MARC," and "Receiving Parameters from an ODT."
- By performing read and write operations on a remote file or ODT file. This topic is discussed in the following subsections of this section: "Communicating with CANDE Terminals," "Communicating with MARC Terminals," and "Communicating with an ODT."
- By using certain statements and task attributes that the system provides for operator communications. These methods are discussed in the following subsections.

Displaying Information to Operators

A process can display information to operators using any of the following features: DISPLAY statements, instruction blocks, and fetch specifications.

DISPLAY statements are the most commonly used of these methods. The DISPLAY statement is implemented in ALGOL, COBOL74, and WFL. This feature is also available as the Display procedure in Pascal. The following is a WFL example of this statement:

```
DISPLAY "INCORPORATING NEW DATA - MAY TAKE AWHILE";
```

The output from a DISPLAY statement is referred to as a *DISPLAY message*. The DISPLAY message appears as one of the entries in the response to the MSG (Display Messages) system command. If the process is initiated from a CANDE or MARC session, the DISPLAY message is automatically displayed at the session. The programmer can use the DISPLAYONLYTOMCS task attribute to limit the display of the message to the originating session. If this task attribute is TRUE, then the DISPLAY message does not appear at the ODT.

You can use instruction blocks to store information that an operator can display at any time. By contrast, DISPLAY messages are only temporarily visible to the operator, because the MSG command displays only the most recent system messages. Instruction blocks are created using the INSTRUCTION statement, which is available only in WFL. The following is an example of this statement.

```
INSTRUCTION 3 TESTTAPE IS IN TAPE RACK 3.;
```

An operator can use the IB (Instruction Block) system command to display instruction blocks for a WFL job. For example, a command of the form *7645 IB* displays the most

recent instruction block for the WFL job with mix number 7645. A command of the form *7645 IB 3* displays instruction block 3 for that WFL job.

The disadvantage of instruction blocks is that nothing prompts the operator to use the IB command. The operator has to know in advance that instruction blocks exist for a particular WFL job. If you want to be sure that an operator sees a message, you can use the FETCH task attribute. This task attribute can be used only in WFL jobs, and only in the job attribute list at the start of the job. You can assign any arbitrary string of text to this attribute. The following is an example of a FETCH assignment:

```
FETCH = "THIS JOB NEEDS THREE TAPE DRIVES";
```

If the operating system option NOFETCH is not set, then when a WFL job containing a FETCH assignment reaches the head of a job queue, the system suspends the job rather than initiating it. The job appears in the W (Waiting Mix Entries) system command display with an RSVP message of REQUIRES FETCH. The operator can use the PF (Print Fetch) system command to display the FETCH specification, and the OK (Reactivate) system command to cause the job to be initiated.

If NOFETCH is set, then the system does not suspend jobs with FETCH specifications. However, the PF system command can still be used to display FETCH specifications.

If you enter a PF command for a process that has no FETCH specification, the system displays the message "NO FETCH STATEMENT".

Accepting Information from Operators

A process can be passed information by an operator using either the HI (Cause EXCEPTIONEVENT) or the AX (Accept) system command.

EXCEPTIONEVENT is an event-valued task attribute, meaning that it has either of two states: HAPPENED or NOT HAPPENED. The HI command *causes* the EXCEPTIONEVENT, meaning that the value is changed to HAPPENED. This action has no effect on process execution unless the program is specifically designed to monitor the status of the EXCEPTIONEVENT. Only programs written in WFL, ALGOL, or COBOL74 have access to this attribute.

A program can monitor the EXCEPTIONEVENT in any of the following ways:

- To suspend execution until the EXCEPTIONEVENT is caused, the process can use a simple wait statement such as *WAIT(MYSELF.EXCEPTIONEVENT)* in ALGOL or *WAIT;* in WFL.
- To suspend execution until either the EXCEPTIONEVENT or some other event occurs, the process can use a complex wait statement that lists the EXCEPTIONEVENT as one of several events.
- To continue doing other work until the EXCEPTIONEVENT is caused, the process can attach an interrupt to the EXCEPTIONEVENT.

In addition to causing the EXCEPTIONEVENT, the HI command can also pass an assignment to the TASKVALUE task attribute of the process. For example, the

command *3874 HI 14* causes the EXCEPTIONEVENT of the process with mix number 3874 and assigns a TASKVALUE of 14. To design a process to use this type of input, you must first use a wait statement or interrupt to monitor the EXCEPTIONEVENT. Whenever the EXCEPTIONEVENT occurs, the process can read its own TASKVALUE and take appropriate action.

Because the programmer controls the way an application responds to a HI command, the operator has no direct way of discovering whether a HI command is needed or what effect it has. Another feature is available that allows the process itself to prompt the operator for certain types of input. This feature is the ACCEPT statement.

The ACCEPT statement displays a string of text to the operator and suspends execution of the process. The process appears in the W (Waiting Mix Entries) system command display, where it can attract the attention of an operator. Execution resumes when the operator uses an AX (Accept) system command to pass another string of text to the process.

In some situations, you might find it more convenient for a process to continue executing until AX input is available from the operator. This goal can be achieved in any of the following ways:

- If the operator is familiar with the program, and knows that an AX command is required later, he or she can enter the AX command without waiting for the process to become suspended. The system saves the text that was input by the operator. When the process executes an ACCEPT statement, the process retrieves this saved text and immediately continues executing.
- By using a conditional ACCEPT statement. This form of ACCEPT checks for AX text previously submitted by the operator. The conditional ACCEPT returns a Boolean value indicating whether such text was found. The process continues executing normally, regardless of whether an AX text was available.
- By using the ACCEPTEVENT task attribute. The system causes the ACCEPTEVENT of a process whenever the operator enters an AX command for that process. A process can monitor the ACCEPTEVENT using wait statements or interrupts, similar to those used for monitoring the EXCEPTIONEVENT. Whenever the ACCEPTEVENT is caused, the process can execute an ACCEPT statement to capture the AX input.

Section 4
Tasking from Programming Languages

The A Series implementations of several programming languages include Unisys extensions for process initiation and control. You can use these features to

- Initiate related suites of programs, so there is no need for an operator to initiate them individually
- Divide an application into two or more cooperating, parallel processes for faster execution

The languages with the most advanced process initiation and control capabilities are WFL, ALGOL, and COBOL74. Of these, WFL is the simplest to use, and also has the advantage of passing through the job queue mechanism and offering automatic job restart after a halt/load. On the other hand, ALGOL and COBOL74 offer sophisticated features such as user-declared events, interrupts, port files, and a large variety of parameter types. Each of these languages provides access to task attributes.

This section describes the tasking capabilities of WFL, ALGOL, and COBOL74 in some detail and provides brief examples of tasking programs written in each of these languages. Additionally, this section includes a brief overview of the tasking capabilities of other languages supported by A Series systems.

Work Flow Language (WFL)

Work Flow Language (WFL) is a programming language that is designed specifically for use in task initiation and control. WFL is a block-structured language with syntax similar to ALGOL, although WFL is simpler and easier to learn.

The following subsections explain how WFL jobs are submitted and how they can be used to initiate other processes.

For further information about WFL, refer to the *A Series Work Flow Language (WFL) Programming Reference Manual*.

Submitting WFL Input

WFL statements can be stored in disk or tape files or in arrays in programs written in other languages. You can also enter and transmit WFL statements at a terminal. Regardless of how WFL statements are stored or submitted, a group of one or more WFL statements is referred to as *WFL input*.

WFL input must be submitted with special-purpose statements such as START and ZIP. You cannot use general-purpose initiation statements such as CALL, PROCESS, and RUN to initiate a WFL job.

The system can compile WFL input and execute it as a job or a task, or it can skip the compilation and simply interpret the WFL input. The statement you use to submit the WFL input and the statements contained in the WFL input together determine how the system executes that input.

Table 4–1 summarizes the factors that determine how the system executes WFL input. The various sources that can submit WFL input are listed at the left. The headings of the two right hand columns give information about the contents of the WFL input. The following are the meanings of these headings:

- The Single Interpretive Statement column indicates WFL input consisting of a single statement that is one of the following statements: ALTER, CHANGE, PRINT, REMOVE, RERUN, SECURITY, or START. The WFL input can also include a FAMILY job attribute assignment, but cannot include any other job attributes. For example, the following input is treated as a single interpretive statement:

```
FAMILY DISK = SYSPK ONLY;CHANGE (JASMITH)ORDS TO (JASMITH)OLDORDS;
```

- The Other Statements column indicates WFL input that consists of either more than one statement or a single statement that is not one of the interpretive statements. A WFL input also falls into this category if it includes assignments to job attributes other than the FAMILY attribute. For example, the following input would fall into this category:

```
JOBSUMMARY = SUPPRESSED;CHANGE (JASMITH)ORDS TO (JASMITH)OLDORDS;
```

Table 4–1. WFL Execution Modes

Sources for Submitting WFL Input	Single Interpretive Statement	Other Statements
CALL SYSTEM WFL (COBOL74)	Interpreted	Job
CALL SYSTEM WITH ZIP (COBOL(68))		
With Array:	Interpreted	Job
With File:	Job	Job
CONTROLCARD function (DCALGOL)		
With [38:01] = 1 and [07:08] = 4 (Array Input):	Interpreted	Task
Otherwise:	Interpreted	Job
LC (Load Control) System Command	Job	Job
START Statement (CANDE, MARC, or WFL)	Job	Job
WFL Command (CANDE or MARC)	Interpreted	Task
WFL Statements Entered at the ODT	Interpreted (except PRINT, which is executed as a job)	Job
ZIP Statement (ALGOL, FORTRAN, or RPG)		
With Array:	Interpreted	Job
With File:	Job	Job

If the system executes the WFL input as a job, it first calls an independent runner called CONTROLCARD to compile the job and create a job file. CONTROLCARD invokes the WFL compiler, which is a procedure exported by the system library WFLSUPPORT. CONTROLCARD runs in a special high-priority category that prevents it from being scheduled or suspended by the system if there is a shortage of available memory. The job file that CONTROLCARD creates contains more information than a typical job file, as discussed in Section 2, "Understanding Interprocess Relationships."

The system then inserts the job file in a job queue. (For a description of the job queue mechanism, refer to "Selecting the Queue for a Job" later in this section.) Later, the system selects the job file from the job queue and initiates it as a job (an independent process). When the job terminates, the system usually prints the job summary and any backup files associated with the job and its tasks. The system then deletes the job file.

If the system executes the WFL input as a task, the system initiates CONTROLCARD to compile the input and create an object code file. The system then initiates the WFL input as a task (a dependent process); the task does not pass through the job queue mechanism. When the task terminates, the system removes the object code file.

By default, no job summary or backup files are associated with the WFL task. For example, if the WFL task was initiated from a CANDE session, then backup files produced by the WFL task or its descendants are associated with the CANDE session and queued for printing only when the session is ended.

If the system handles the WFL input interpretively, then CONTROLCARD executes the WFL statement without creating a WFL job or a WFL task. In this case, CONTROLCARD neither creates a job file or an object code file, nor does it use the job queue mechanism.

Selecting the Queue for a Job

A *job queue* is a list of WFL jobs that are awaiting initiation. Job queues are defined by the system administrator and managed by the operating system.

The purpose of job queue definitions is to allow the system administrator to set up some general parameters affecting the flow of WFL jobs on the system. Because a WFL job is typically an agent for initiating batch programs, the job queue system by implication can be used to regulate the initiation of batch programs in general.

Before defining the job queues, the system administrator usually analyzes the batch programs run on the system in terms of their patterns of resource usage and their relative urgency. The administrator then defines a separate job queue for each set of batch programs that show similar characteristics. For example, if there is a payroll application that has to finish processing before a precise deadline, the administrator might assign the application to a high-priority job queue. The administrator uses an MQ (Make or Modify Queue) system command to define the job queue.

For a complete explanation of job queues and using job queues in system administration, refer to the *A Series System Administration Guide*. For information about using system commands to monitor and interact with jobs in queues, refer to the *A Series System Operations Guide*. The following subsections describe the features of job queues that are of most direct interest to a programmer.

Deciding on the Queue for a Job

Depending on the policies that are in effect at your site, you might be required to ask your system administrator which job queue to submit a particular WFL job to. However, if the system administrator allows you to decide on the job queue, then you need to examine the job queue definitions to determine which queue is most suitable to your job.

The system command for displaying job queue definitions is QF (Queue Factors). The following is an example of a QF command and the response:

```
QF 4

     QUEUE 4:
       MIXLIMIT = 2
       DEFAULTS:
         PRIORITY = 50
         PROCESSTIME = 100
       LIMITS:
         PRIORITY = 60
         PROCESSTIME = 200
```

In this example, 4 is the job queue number. This number uniquely identifies a job queue. If the QF command does not specify a number, the output displays the definitions of all job queues on the system.

The MIXLIMIT value specifies, roughly, the maximum number of jobs and descendant tasks initiated through this job queue that can be running concurrently. If the actual number of jobs and tasks originating from this job queue equals or exceeds the MIXLIMIT value, the system temporarily ceases initiating jobs from this job queue. After one or more of the jobs and tasks in this job queue terminates, the system resumes initiating jobs from this job queue.

The DEFAULTS and LIMITS portions of the job queue definition specify default values and maximum values for various task attributes that restrict the resource usage of a process.

The job queue defaults are inherited by the corresponding task attributes of a WFL job. However, the job can override this inheritance with assignments in the job header; that is, assignments that follow the BEGIN JOB construct but precede any of the declarations and statements in the job. Consider the following example:

```
?BEGIN JOB;
  CLASS = 4;
  PRIORITY = 55;
  TASK T;
  MYSELF(MAXPROCTIME = 150);
  RUN OBJECT/PROG;
?END JOB
```

Assume that this job is submitted through the job queue that was previously shown in the QF command example. Queue 4 has default values for both PRIORITY and PROCESSTIME (which corresponds to the MAXPROCTIME task attribute). The PRIORITY assignment in the job is part of the job header, and therefore overrides the PRIORITY queue default. However, the MAXPROCTIME assignment in the job is not part of the job header. Therefore, the job does inherit the default MAXPROCTIME of 100 at initiation. The statement that assigns MAXPROCTIME a value of 150 has no affect, because the system does not allow a process to increase its MAXPROCTIME value after initiation.

Now consider the following job:

```
?BEGIN JOB;
  CLASS = 4;
  PRIORITY = 75;
  MAXPROCTIME = 3ØØ;
  TASK T;
  RUN OBJECT/PROG;
?END JOB
```

The system would never accept this job into queue 4, because the job header assigns values to PRIORITY and MAXPROCTIME that are both higher than the queue limits for these attributes. Since the CLASS attribute explicitly requests queue 4, the system rejects the job and displays a "Q-DS" message. (The CLASS attribute is explained under "Requesting the Queue for a Job," later in this section.)

The following are the job queue attributes that establish resource usage limits, and the task attributes that correspond to the job queue attributes:

Job Queue Attribute	Task Attribute	Effect
CARDS	MAXCARDS	Limits the number of cards the job and its tasks can punch
DISKLIMIT	DISKLIMIT	Limits the space the job and its tasks can allocate for disk files
ELAPSEDLIMIT	ELAPSEDLIMIT	Limits the amount of time a job can be in use
IOTIME	MAXIOTIME	Limits the amount of processor time that can be devoted to initiating I/O operations for the job and its tasks
LINES	MAXLINES	Limits the number of lines the job and its tasks can print
PROCESSTIME	MAXPROCTIME	Limits the amount of processor time that a process can use for computations
PRIORITY	PRIORITY	Specifies the relative urgency of jobs and tasks as compared to other processes in the mix
SAVEMEMORYLIMIT	SAVEMEMORYLIMIT	Limits the amount of save memory the job and its tasks can use

continued

continued

Job Queue Attribute	Task Attribute	Effect
TEMPFILELIMIT	TEMPFILELIMIT	Limits the space the job and its tasks can allocate for temporary disk files
WAITLIMIT	WAITLIMIT	Limits the amount of time the job and its tasks can remain waiting after executing a WAIT statement

If the actual resource usage of the job or its tasks exceeds one or more of the resource usage limits, the system discontinues the process that exceeded the limit. The point of this behavior is to encourage you to reexamine the job queue definitions and submit the job through the appropriate job queue.

In summary, you can determine an appropriate job queue for a job by estimating the resource usage requirements of the job and choosing a job queue whose resource usage limits are adequately high. There are, however, some additional restrictions that you need to be aware of:

- The system administrator can assign two attributes to your usercode that specify which job queues you are allowed to use. These attributes are CLASSLIST and ANYOTHERCLASSOK. If ANYOTHERCLASSOK is set, then CLASSLIST is interpreted as a list of the job queues you are forbidden to use. If ANYOTHERCLASSOK is not set, then CLASSLIST is interpreted as a list of all the job queues you are allowed to use. You should ask the system administrator whether these attributes are defined for your usercode.
- The system administrator can use the UQ (Unit Queue) system command to specify that all WFL jobs submitted from a particular device be routed into a particular job queue. ODTs and card readers are examples of input devices that can be specified in a UQ command. The inquiry form of the UQ command can be used to display the unit queue assignments in effect on the system.
- The job queue definition can include a FAMILY attribute that corresponds to the FAMILY task attribute. However, the FAMILY queue attribute is not exactly a default or a limit. Rather, it excludes any job from the job queue if the job header includes a FAMILY assignment different from the FAMILY queue attribute. You can use the QF command to determine whether a job queue has a FAMILY queue attribute.

Requesting the Queue for a Job

If you have decided that a specific job queue is most appropriate for your job, then you can request the job queue through a CLASS assignment in the job header. For example, the following job requests queue 10:

```
?BEGIN JOB;
  CLASS = 10;
RUN OBJECT/PROG;
?END JOB
```

If the job does not include a CLASS assignment, it can inherit a value from the CLASS usercode attribute. An inherited CLASS value has the same effect as an assigned CLASS value.

The system evaluates the eligibility of a job for a requested job queue on the basis of the factors discussed previously: queue resource usage limits, usercode class limits, unit queue assignments, and the FAMILY value. If the job qualifies for the requested queue, the system places the job in the queue. If the job does not qualify for the requested queue, the system rejects the job and displays the message "Q-DS."

If the job has no assigned or inherited CLASS value, the system attempts to find an appropriate job queue to place the job in. The method the system uses for making this selection depends on whether the operating system compile-time option QFACTMATCHING is set.

If the job has no CLASS assignment and QFACTMATCHING is set, then the system examines the various job queues to determine their eligibility for receiving the job. The system selects the first job queue that meets the following criteria:

- Any resource limits specified for the queue are greater than or equal to the corresponding resource limits in the WFL job header. For example, if the queue has a PRIORITY limit of 50, the job must either have no PRIORITY assignment in the job header or a PRIORITY assignment less than 51.
- The job queue must be one that is legal for a job with this usercode.

If the job has no CLASS assignment and QFACTMATCHING is reset, then the system selects the default job queue. The system administrator defines the default job queue using the DQ (Default Queue) system command. If no default queue has been defined, the system checks all the job queues, just as it would if QFACTMATCHING were set.

Whether QFACTMATCHING is set or not, the system performs an additional check. If the job queue selected by the system has a FAMILY attribute and the job also has a FAMILY assignment in the job header, the system checks to see whether they match. If they do not specify identical family values, the system rejects the job and displays a "Q-DS" message.

Specifying a Start Time

You can use the STARTTIME task attribute to specify the earliest time and date that a particular job can be selected from a job queue. This task attribute can be assigned only to WFL jobs. It can be assigned in the task attribute list of the WFL job or in the statement that initiates the WFL job. You can also use the STARTTIME (Start Time) system command or the CANDE *?STARTTIME* command to assign this attribute to a job in a job queue. However, any changes made using these commands are not maintained across a halt/load.

When you initiate a job with a STARTTIME specification, the job is compiled immediately and placed in an appropriate job queue. The job remains in the job queue at least until the date and time specified by the STARTTIME. You can use the SQ (Show Queue) system command to display the STARTTIME of jobs in a queue. The following is an example of the output for the command SQ 2:

```
QUEUE 2
 6643 Ø1 TEST/WFL (#ØØØ1)
         QUEUED: 12/19/89 AT 15:41:31   STARTTIME = 18:ØØ:ØØ ON 12/2Ø/89
```

The STARTTIME specification provides a convenient means of scheduling a job for a time when the system load is lighter, such as in the evening or during a weekend. STARTTIME is also a convenient means of scheduling jobs that must run at regular intervals, such as every morning. The following example job, which is stored in the file (JASMITH)WFL/RUN, restarts itself on a daily basis:

```
?BEGIN JOB WFL/RUN;
  RUN OBJECT/PROG;
  START (JASMITH)WFL/RUN;STARTTIME = 1Ø:ØØ ON +1
?END JOB
```

Structuring the WFL Job

A complete WFL job is considered a block, and each subroutine declared in the job is also a block. The WFL job can enter or initiate subroutines. WFL automatically protects against critical block exits by performing an implicit wait at the end of the block that contains a task initiation statement. Control does not exit this block until all tasks initiated in that block have terminated.

WFL includes CASE, DO UNTIL, GO, IF, and WHILE DO statements that you can use to direct the flow of control in a job. By using these statements together with task attribute interrogations, a WFL job can provide conditional control over tasks. For example, the job can initiate the SYSTEM/PATCH utility as a task. When SYSTEM/PATCH terminates, the job can interrogate the task attributes of the SYSTEM/PATCH task. If the attribute values indicate that SYSTEM/PATCH ran without errors, the job can compile the merged source program. If the compilation is free of errors, the job can run SYSTEM/XREFANALYZER to produce an analysis of cross-references in the program.

Initiating Dependent Processes from WFL

In WFL, the RUN statement can be used to initiate an object code file as a synchronous dependent process. The TYPE task attribute of the resulting process shows a value of CALL. The initiated program can be written in any language except WFL.

The *PROCESS* keyword is used as a modifier in front of other initiation statements to cause the process to run asynchronously. Thus, a PROCESS RUN statement initiates an asynchronous task. The TYPE task attribute of the task has a value of PROCESS.

WFL cannot initiate a program as an independent process. Also, a WFL job is never considered to be a coroutine; that is, a WFL job and its offspring cannot use CONTINUE statements to pass control back and forth.

There are some noteworthy differences between task initiation in WFL and task initiation in ALGOL or COBOL74. In the latter two languages, RUN initiates an independent process and PROCESS initiates an asynchronous dependent process. Another difference is that WFL does not use external procedure declarations. Also, there is no need to include a NAME task attribute assignment in WFL; the name of the object code file to be executed is specified in the RUN statement.

WFL jobs can also initiate internal procedures. An internal procedure in WFL is referred to as a *subroutine.* If the *PROCESS* keyword precedes a subroutine invocation statement, the system initiates the subroutine as an internal, asynchronous, fully dependent process. (If you do not use the *PROCESS* keyword, the subroutine invocation statement enters, rather than initiates, the subroutine.)

Initiating Compilations from WFL

A WFL job can initiate compilations by using the COMPILE statement. The COMPILE statement initiates a compiler and specifies the object code file to be compiled. The COMPILE statement can also include an object code file disposition, which specifies whether the object code file is to be executed once it is compiled, and whether the object code file is to be saved. The COMPILE statement can also be used to invoke the Binder. BIND is a synonym for the COMPILE statement.

Initiating Utilities from WFL

In addition to the RUN statement, WFL provides various special-purpose initiation statements. These statements include ADD, COPY, LOG, and PB. The COPY and ADD statements each initiate the visible independent runner LIBRARY/MAINTENANCE to copy a file. The LOG statement initiates the LOGANALYZER utility, and the PB statement initiates the BACKUP utility.

Initiating Interactive Processes from WFL

A WFL job can initiate an interactive process, but you might need to include a STATION task attribute assignment for the interactive process to run properly. The STATION task attribute specifies the logical station number (LSN) of the station where any remote files used by the process are to be opened. WFL jobs initiated through a CANDE or MARC *START* command do not inherit the LSN associated with the remote terminal where they are initiated. You can remedy this problem by including the following statement at the start of the job:

```
MYJOB (STATION = MYSELF(SOURCESTATION));
```

This statement assigns the LSN of the station that initiated the job to the STATION attribute of the job. This STATION value is inherited by all tasks initiated by the job.

(Note that this assignment is lost across a halt/load. For details, refer to Section 11, "Restarting Jobs and Tasks.")

Submitting Other WFL Jobs

A WFL job can include a START statement to initiate another WFL job. The START statement can initiate only WFL programs that are stored on disk files. This statement can include any of the parameter types that WFL recognizes. The START statement can also include an assignment to the STARTTIME task attribute, which specifies when the WFL job should be initiated.

Access to Task Attributes in WFL

A WFL job can include a job attribute list, which specifies task attributes to be applied to the job before initiation. Certain task attributes, if included in this list, can help determine the job queue in which the job is placed. The CLASS task attribute has the most direct effect on job queue selection; for more information about the CLASS attribute, refer to "Selecting the Queue for a Job" later in this section.

A WFL job can specify initial values for the attributes of a task if you include a task equation list in a task initiation statement. All task initiation statements in WFL (including RUN, COPY, and COMPILE) allow the use of task equations.

A WFL job can also use task variables to interrogate or modify the task attributes of a process. The task variable becomes associated with a task by being included in the task initiation statement. Assignments to the task variable before task initiation have the same effect as task equations. A job can monitor and control an asynchronous task while it is executing by accessing its task variable. After a task terminates, the job can interrogate the task variable to return task history information.

A WFL job can use the predeclared task variable MYSELF to access the job's own task attributes. A job can also use the predeclared task variable MYJOB, which has the same meaning as MYSELF unless it is referred to in an asynchronous subroutine. For an asynchronous subroutine, MYJOB refers to the parent WFL job and MYSELF refers to the subroutine's task attributes.

The COMPILE statement can specify task attributes that are stored in the object code file created by the compilation. These task attribute values are used each time the object code file is executed, unless the values are overridden by task equations at run time. Also, a WFL job can use the MODIFY statement to assign task attributes to an object code file that already exists.

WFL jobs can directly access all task attributes except for task-valued or event-valued task attributes and the HISTORYREASON task attribute.

In general, the syntax for accessing task attributes in WFL is simpler than that used in ALGOL. Mnemonic-valued task attributes return string values rather than integers. Pointer-valued task attributes also return string values. Attributes that record resource usage, such as ACCUMPROCTIME, return values in units of seconds instead of 2.4 microseconds.

Using File Equations in WFL

Assignments to the FILECARDS task attribute are referred to as file equations. In WFL jobs, FILECARDS can be abbreviated to FILE. Using this task attribute, the job can modify the attributes of the logical files used by the task. The TITLE attribute can be used to cause the task to use a different physical file than it otherwise would.

You can include a construct called a *global file assignment* in a WFL job to cause an offspring to use a file declared in the WFL job. A global file assignment assigns a particular file declared by the WFL job to a particular internal name used by the offspring. Whenever the offspring attempts to use the file with that internal name, the system causes it to use the global file instead. This mechanism amounts to a hidden call-by-reference parameter because the job and its offspring use the same logical file.

A unique feature of WFL is the ability to include data specifications in the WFL source program. Whenever an offspring attempts to read from a card reader file, it reads instead from a data specification, if one is available. You can also use data specifications to replace other kinds of input files used by an offspring. To do this, you must include a file equation in the statement that initiates the offspring. The file equation must assign the input file a KIND file attribute value of READER. The offspring then reads lines from the data specification as if they were lines of the input file; for this reason, data specifications are also known as *pseudo-reader files*.

Responding to Error Conditions in WFL

Use the ON TASKFAULT statement to specify actions to be taken if a task terminates abnormally or if a compilation is terminated for syntax errors. WFL can also interrogate the values of the STATUS and HISTORYTYPE task attributes after a task terminates to determine the type of termination and take appropriate action.

Communicating with Other Processes in WFL

WFL jobs can communicate with their tasks by using any of several methods. The following list reviews each method of interprocess communication:

- Globally declared objects

  A subroutine initiated with a *PROCESS* < *subroutine* > statement can access objects declared globally to the subroutine in the WFL job.

- Parameters

  The RUN statement can include Boolean, integer, real, or string parameters. By default, these parameters are call-by-value parameters. However, you can specify that a parameter is call-by-reference by including the word *REFERENCE* after the parameter. A WFL job and an asynchronous task can communicate by interrogating and modifying the value of a call-by-reference parameter.

- Events

  A WFL job cannot declare events or interrupts and cannot access event-valued task attributes directly. However, a WFL job can use the WAIT statement, which can wait on many different types of implicitly declared events. For example, the simple form of the WAIT statement waits on the job's exception event. A job can also use WAIT statements to wait for a task to terminate or for one of the task attributes to attain a specified value. A WFL job can also access the LOCKED task attribute. LOCKED is a Boolean task attribute that acts like an event.

- Libraries

  Libraries cannot be written in WFL, nor can WFL use libraries written in other languages.

- Port files and disk files

  WFL jobs cannot read from or write to files. A WFL job can create a single disk file and specify the contents of that file by using the DECK statement. However, the DECK statement, if used, must be the only statement in the job. Another useful feature is the ability of WFL to create a dummy file by simply declaring a file, opening it, and closing it. Such files can be used as flags to other processes. For example, a WFL job can perform a file-residence inquiry to determine whether a file with a certain title exists.

For details about any of these interprocess communication methods, refer to Part II of this guide, "Interprocess Communication."

Restarting WFL Jobs

A WFL job automatically restarts if interrupted by a halt/load. WFL is the only language with this automatic restart capability. WFL also plays an important role in the restarting of checkpointed ALGOL or COBOL(68) processes. These processes must be offspring of a WFL job in order to be checkpointed. Also, the WFL *RERUN* statement is the means used to restart a checkpointed process. For further information, refer to Section 11, "Restarting Jobs and Tasks."

WFL Example

The following example illustrates some WFL capabilities for task initiation and control:

```
?BEGIN JOB AUTOPB/HELP(STRING SOURCE, STRING PATCH);
  JOBSUMMARY = SUPPRESSED;
  DISPLAYONLYTOMCS = TRUE;
  CLASS = 15;
TASK T;
STRING RUN1, HELPTITLE;
HELPTITLE := (PATCH & "/LEVEL1/HELPBOOK");
RUN1 := ("SOURCE=" & SOURCE
       & ",PATCH=" & PATCH
       & ",OUT=" & PATCH & "/LEVEL1/ED"
       & ",HELP=" & HELPTITLE
       & ",MESSAGEFILE=" & PATCH & "/LEVEL1/MESSAGES");
DISPLAY "RUNNING AUTOPB WITH " & RUN1;
RUN OBJECT/AUTOPB ON DOCMAST(RUN1) [T];
  FILE TEACHUTILNAME=*SYSTEM/HELP/UTILITY ON DOCMAST;
IF T(TASKVALUE) NEQ 1
   THEN BEGIN
          DISPLAY "HELPBOOK NOT CREATED; PRINTING ERRORS FILE";
          RUN *OBJECT/AUTOLP ON DOCMAST;
             TASKVALUE = 1;
             FILE SOURCE = #PATCH/LEVEL1/MESSAGES;
        END;
?END JOB
```

The main point of this job is to run a program called AUTOPB. The AUTOPB program accepts two input files, SOURCE and PATCH, and produces three output files, OUT, HELP, and MESSAGEFILE.

The job accepts two string parameters that provide the titles of the SOURCE and PATCH files. Using these, the job constructs an elaborate string parameter to pass to AUTOPB. This string parameter defines the titles for all the input and output files.

AUTOPB sets its own TASKVALUE to 1 unless it finds errors in the input files. The job inspects the TASKVALUE after AUTOPB terminates and prints out the MESSAGEFILE if there are errors.

ALGOL

ALGOL is a structured, high-level programming language with advanced computational and I/O capabilities. ALGOL also provides the most complete process initiation and control capabilities of any language available on A Series systems.

Closely related to ALGOL are several extended versions of the ALGOL language. DCALGOL is an extended ALGOL that includes some system control and data comm interfaces. DMALGOL includes special constructs for data management software. BDMSALGOL contains extensions for accessing Data Management System II (DMSII)

databases. In the following discussion, the features described are available in each of these languages, except where otherwise noted.

For further information about the ALGOL tasking features discussed in the following subsections, refer to the *A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

Structuring an ALGOL Program

The following ALGOL structures are considered blocks:

- Any complete ALGOL program. A complete ALGOL program can be initiated but cannot be entered.
- Any typed procedure; that is, any procedure designed to be invoked as a function that returns a value. (For example, Boolean procedures or real procedures.) Typed procedures can be entered but cannot be initiated.
- Any untyped procedure; that is, any procedure that does not return a value. Untyped procedures can be entered or initiated.
- A simple block, which is any group of declarations and statements that appears between the words BEGIN and END and is not preceded by a procedure heading. (An exception is the outer block of the program, which is not considered a simple block.) Such a block cannot be entered or initiated. The block is executed when control passes either from the previous statement in the program or from a GO TO statement elsewhere in the program. (Note that a BEGIN...END statement is not treated as a block if it does not include any declarations. In this case, it is simply a compound statement.)

When you initiate one of these ALGOL structures, the system creates a process stack. When you enter one of these ALGOL structures, the system creates an activation record. When a BEGIN...END block that includes declarations is executed, the system also creates an activation record.

An ALGOL program that initiates an asynchronous process should usually include a wait statement to prevent the critical block from being exited while the offspring is in use. An example of this wait statement is given in Section 2, "Understanding Interprocess Relationships."

ALGOL includes an abundance of flow-of-control statements, such as CASE, DO, FOR, IF and WHILE. By using these statements together with task attribute interrogations, an ALGOL program can provide conditional control over tasks.

Initiating Processes from ALGOL

An ALGOL program can initiate any untyped procedure, including imported library procedures, passed external procedures, and separate programs.

To initiate another object code file, an ALGOL program must declare an external procedure and a task variable. The program must also assign the title of the object code file to the NAME attribute of a task variable. The program can then initiate the

object code file with a process initiation statement that specifies the declared external procedure and task variable that were previously prepared.

Three process initiation statements are available. The CALL statement initiates a dependent, synchronous process. The PROCESS statement initiates a dependent, asynchronous process. The RUN statement initiates an independent process.

You can implement coroutines in ALGOL through the use of CALL and CONTINUE statements. The CALL statement creates an active coroutine and changes the initiating process into a continuable coroutine. Coroutines can pass control back and forth by using CONTINUE statements.

Initiating Compilations from ALGOL

ALGOL does not provide any statement specifically for initiating compilations. However, an ALGOL program can submit a WFL job that includes a COMPILE statement. Alternatively, an ALGOL program can initiate a compiler like any other program, with a CALL, PROCESS, or RUN statement. An example of this method is given under "ALGOL Examples" later in this section.

Initiating Utilities from ALGOL

ALGOL does not provide any statements specifically for initiating utilities. However, the CALL, PROCESS, and RUN statements can initiate any utility and pass any parameters that are required by that utility. An example of an ALGOL program that initiates the LOGANALYZER utility is given under "ALGOL Examples" later in this section.

Initiating Interactive Processes from ALGOL

An ALGOL program initiated from a MARC or CANDE session inherits the STATION task attribute of the session. The STATION attribute is in turn inherited by any processes initiated by the ALGOL program. As a result, the processes initiated by the ALGOL program can open a remote file at the originating terminal without having to make any special remote file assignments. However, an ALGOL program initiated from a WFL job or from an ODT might not inherit a STATION value. For further information, refer to "Work Flow Language (WFL)" in this section and to the ODT discussion in Section 3, "Tasking from Interactive Sources."

Submitting WFL Jobs from ALGOL

You can use the ZIP statement to submit a WFL job for execution. You can store the WFL job source in a disk file or in an array in the ALGOL program itself. Note that messages produced by the WFL job or its descendants will not be forwarded to the CANDE or MARC session that originated the ALGOL program. However, you can use the CANDE *?MSG* command or the MARC *SMSG* command to display these messages.

Access to Task Attributes in ALGOL

An ALGOL program can declare task variables for use in accessing the task attributes of offspring processes. Every process-initiation statement must specify a task variable, which thereafter is associated with the new process. An ALGOL program can interrogate or assign task attribute values of the task variable before or after the task variable is used in a process initiation statement. Assignments made to a task variable before initiation are saved and applied to the process at initiation time.

An ALGOL program can use the predeclared task variables MYSELF and MYJOB to access its own task attributes and those of its job.

An ALGOL program can interrogate and modify task attributes that store any of the possible data types, such as Boolean, integer, and so on. The task attribute types available in ALGOL include two types that are not available in WFL: event and task.

Communicating with Other Processes from ALGOL

ALGOL programs have full access to all of the interprocess communication methods discussed in this guide, including globally declared objects, call-by-reference or call-by-name parameters, events and interrupts, port files, and libraries. For details about any of these interprocess communication methods, refer to Part II of this guide, "Interprocess Communication."

Restarting ALGOL Processes

An ALGOL program can include a CHECKPOINT statement that creates a checkpoint file. The checkpoint file stores information about the current state of a process. You can use the checkpoint file after a halt/load to restart the process. For further information, refer to Section 11, "Restarting Jobs and Tasks."

DCALGOL Features

In addition to the ALGOL features previously discussed, DCALGOL includes the CONTROLCARD function, which you can use instead of the ZIP statement to submit WFL jobs for execution. The CONTROLCARD function has several capabilities that are unavailable through ZIP. For example, the CONTROLCARD function can

- Specify whether the WFL job should be a dependent or independent process
- Compile the job for syntax checking only, without executing it
- Specify that any messages generated by the job be routed to an MCS for display in the originating session
- Define the invalid character to be something other than a question mark (?)
- Submit a job that is stored as a message in a DCALGOL queue

Additionally, the process that submits the CONTROLCARD function can determine whether the WFL job compiled without syntax errors. If a WFL job submitted through

CONTROLCARD has syntax errors, the system assigns the value 1 to the TASKVALUE of the process that submitted the job.

A privileged DCALGOL process can also duplicate the process initiation and control capabilities that are available at an ODT. You can use the DCKEYIN statement to submit system commands to the operating system. The GETSTATUS and SETSTATUS functions directly invoke the operating system interfaces that are accessed by system commands. For information about ODT process initiation and control capabilities, refer to Section 3, "Tasking from Interactive Sources."

ALGOL Examples

The following sample program initiates a separate program called REPORTER. The REPORTER program is initiated twice, both times as an asynchronous task, and is passed a different parameter each time. The sample program then uses a WAITANDRESET statement to prevent a critical block exit.

```
BEGIN
EBCDIC ARRAY DAILYTYPE[Ø:5],
             WEEKLYTYPE[Ø:6];
TASK T, T2;
PROCEDURE REPORTS (ACTUALARRAY);
   EBCDIC ARRAY ACTUALARRAY[*];
EXTERNAL;

REPLACE T.NAME BY "(JASMITH)OBJECT/REPORTER ON DATAPK.";
REPLACE DAILYTYPE[Ø]  BY "DAILY";
PROCESS REPORTS (DAILYTYPE) [T];

REPLACE T2.NAME BY "(JASMITH)OBJECT/REPORTER ON DATAPK.";
REPLACE WEEKLYTYPE[Ø] BY "WEEKLY";
PROCESS REPORTS (WEEKLYTYPE) [T2];

WHILE (T.STATUS GTR Ø OR T2.STATUS GTR Ø) DO
   WAITANDRESET (MYSELF.EXCEPTIONEVENT);
END.
```

The following is an example of initiating a compilation from an ALGOL program. The sample program passes an array parameter and makes FILECARDS assignments to tell the compiler what files to use:

```
BEGIN

TASK CTASK;
ARRAY SHEET[Ø:32];

PROCEDURE ALGOLCOMPILER(SHEET);
  ARRAY SHEET[*];
  EXTERNAL;

REPLACE CTASK.NAME BY "*SYSTEM/ALGOL ON DISK.";
REPLACE CTASK.FILECARDS BY
  "FILE CARD (KIND=DISK, TITLE=ALGOL/TASK);"
  "FILE CODE (KIND=DISK, TITLE=OBJECT/ALGOL/TASK);"
   48"ØØ";

REPLACE SHEET BY Ø FOR 33 WORDS;
SHEET[8] := VALUE(LIBRARY); % This statement specifies the
                            % object code file disposition.
SHEET[Ø] := Ø & 1[47:1];

CALL ALGOLCOMPILER(SHEET) [CTASK];

END.
```

The following is an example of initiating a utility from ALGOL. This sample program includes a statement that initiates LOGANALYZER:

```
BEGIN
TASK T;
PROCEDURE LOGRUN (FORMAL_OPTIONS);
  ARRAY FORMAL_OPTIONS[*];
  EXTERNAL;
ARRAY ACTUAL_OPTIONS[Ø:19];
REPLACE T.NAME BY "*SYSTEM/LOGANALYZER ON DISK.";
REPLACE ACTUAL_OPTIONS BY "PRINTER JOB 126Ø",48"ØØ";
CALL LOGRUN (ACTUAL_OPTIONS) [T];
END.
```

The following ALGOL example submits WFL programs for execution in two different ways. The first ZIP statement submits the WFL program stored in array WFLARRAY. The second ZIP statement submits the WFL program stored in the file WFL/TEST. Note the use of triple quotes (""") in WFLARRAY wherever a single quote (') is to occur in the WFL program.

```
BEGIN
  EBCDIC ARRAY WFLARRAY[1:120];
  FILE WFLFILE(KIND=DISK,NEWFILE=FALSE,DEPENDENTSPECS=TRUE,
               TITLE="WFL/TEST.");
  REPLACE WFLARRAY BY
    "CLASS=2;JOBSUMMARY=SUPPRESSED;ELAPSEDLIMIT=120;"
    "MYSELF(STATION=MYSELF(SOURCESTATION));"
    "DISPLAY (""HI""");";
  ZIP WITH WFLARRAY;
  ZIP WITH WFLFILE;
END.
```

COBOL74

COBOL is available in three different A Series implementations: COBOL(68), COBOL74, and COBOL85. These correspond to the ANSI-68, ANSI-74, and ANSI-85 levels of COBOL, respectively. Note that COBOL(68) is frequently referred to simply as COBOL in other A Series documentation. The suffix (68) is used in this guide to differentiate the ANSI-68 version of COBOL from the other versions.

Of the A Series COBOL implementations, COBOL74 and COBOL(68) incorporate a full range of tasking capabilities. COBOL74 is the newer of these two languages and the preferred language for writing new COBOL tasking applications. COBOL85 does not have tasking capabilities at this time. In the following subsections, statements about COBOL74 apply equally to COBOL(68) except where otherwise noted.

For further information about COBOL74, refer to the *A Series COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation*. For further information about COBOL(68), refer to the *A Series COBOL ANSI-68 Programming Reference Manual*.

Structuring a COBOL74 Program

In a typical COBOL74 program, the outer block of the program is the only block of the program. Paragraphs and sections within a COBOL74 program are not considered blocks, because executing a paragraph or a section does not result in the creation of an activation record.

A COBOL74 program can contain more than one block only if the Binder is used to bind a procedure from a separate object code file into the program. The bound-in procedure could be another COBOL74 program or a procedure from a program written in some other language. A COBOL74 program can enter, but cannot initiate, a bound-in procedure.

The following rules govern COBOL74 access to external procedures:

- Separate programs

  A COBOL74 program can declare external procedures and use them to initiate separate programs.

- Passed external procedures

  COBOL74 does not provide any method for passing procedures as parameters. Therefore, a COBOL74 program generally has no access to passed external procedures. Thunks are the only exception to this rule. When a program passes a constant or an expression by name to a COBOL74 program, the system creates a thunk. Whenever the COBOL74 program interrogates the parameter, the system executes the thunk on the COBOL74 program's process stack.

- Imported library procedures

  A COBOL74 program can enter, but cannot initiate, a procedure imported from a library.

A critical block exit error can occur if the COBOL74 program terminates before an asynchronous offspring or a coroutine. For information about how to prevent such critical block exits, refer to Section 2, "Understanding Interprocess Relationships."

Initiating Processes from COBOL74

A COBOL74 program can initiate separate programs as processes, but cannot initiate internal sections and paragraphs.

Separate object code files are initiated by statements that have the following general form:

```
<verb> <task variable> WITH <section name> [USING <parameter list>]
```

The verb in this statement can be CALL, which initiates a synchronous, dependent process; PROCESS, which initiates an asynchronous, dependent process; or RUN, which initiates an independent process. EXECUTE is a synonym for RUN.

The task variable in this statement is a data item declared with a usage of TASK, CONTROL-POINT, or CP.

The section name in this statement must have been previously defined in the DECLARATIVES portion of the PROCEDURE DIVISION. The section name definition in the DECLARATIVES must be followed by a USE EXTERNAL statement.

The COBOL74 program must also associate an object code file title with a <section name> by one of the following methods:

- By using a mnemonic name in the SPECIAL-NAMES paragraph. This is the preferred method.
- By using a MOVE statement to assign the object code file title to the identifier that was specified in the USE EXTERNAL statement in the DECLARATIVES.
- By assigning the NAME task attribute to the task variable before task initiation. The title assigned must be a string that is enclosed in quotes and terminated with a period.

The *USING <parameter list>* clause passes parameters to the initiated program. If no parameters are to be passed, you can omit this clause.

Using Coroutines in COBOL74

You can implement coroutines in COBOL74 through the use of CALL, CONTINUE, and EXIT PROGRAM statements. The CALL statement creates a synchronous task that is an active coroutine and changes the parent process into a continuable coroutine. The task can return control to its parent by executing an EXIT PROGRAM statement. The parent can return control to its task by executing a *CONTINUE <task variable>* statement.

The EXIT PROGRAM statement, in addition to transferring control to the parent, also specifies where execution resumes when the parent later continues the task. The simple form EXIT PROGRAM specifies that the task resumes from the beginning. The EXIT PROGRAM RETURN HERE form specifies that the task resumes with the statement that follows the EXIT PROGRAM statement.

Entering Individual COBOL74 Procedures

COBOL74 allows the use of certain special formats for the CALL statement that enter, rather than initiate, a procedure.

A COBOL74 program can use a CALL statement with one of the following forms to enter a bound-in procedure:

```
CALL <section name>.
CALL <section name> USING <parameter list>.
```

A COBOL74 program can use any of several forms of the CALL statement to enter an imported library procedure. The following is an example:

```
CALL "PROCEDUREDIVISION OF OBJECT/COBOL74/PROG" USING PARAM1.
```

By contrast, the GO and PERFORM statements do not enter procedures. They simply transfer control to a selected paragraph or section without creating an activation record.

Initiating Compilations from COBOL74

COBOL74 does not include any statement specifically for initiating compilations. However, a COBOL74 program can submit a WFL job that includes a COMPILE statement. Alternatively, a COBOL74 program can initiate a compiler like any other program, with a CALL, PROCESS, or RUN statement.

Initiating Utilities from COBOL74

COBOL74 does not include any statements specifically for initiating utilities. However, the CALL, PROCESS, and RUN statements can initiate any utility and pass any parameters that are required by the utility.

Initiating Interactive Processes from COBOL74

A COBOL74 program initiated from a MARC or CANDE session inherits the STATION task attribute of the session. The STATION attribute is, in turn, inherited by any processes initiated by the COBOL74 program. As a result, these processes can open a remote file at the originating terminal without having to make any special remote file assignments.

However, a COBOL74 program initiated from a WFL job or from an ODT might not inherit a STATION value. For further information, refer to "Work Flow Language (WFL)" earlier in this section and to "Communicating with an ODT" in Section 3, "Tasking from Interactive Sources."

Submitting WFL Jobs from COBOL74

A COBOL74 program can submit WFL jobs with a statement of the following form:

```
CALL SYSTEM WFL USING <identifier>
```

The identifier in this statement must be associated with a data item that contains the complete WFL source program.

Note that when a COBOL74 program submits a WFL job, any messages produced by the WFL job or its descendants are not forwarded to the CANDE or MARC session that originated the COBOL74 program. However, you can use the CANDE *?MSG* command or the MARC *SMSG* command to display these messages.

The syntax for submitting WFL jobs is slightly different in COBOL(68). In that language, you must use a statement of the following form, where <identifier> is the name of an array or file containing the source WFL program:

```
CALL SYSTEM ZIP <identifier>.
```

Access to Task Attributes in COBOL74

A COBOL74 program can access task attributes by using a task variable. A COBOL74 program can create a task variable by declaring a data item with a USAGE of TASK, CP, or CONTROLPOINT in the DATA DESCRIPTION entry.

The MYSELF and MYJOB task variables are available in COBOL74 and enable a COBOL74 program to access its own task attributes or those of its job.

A COBOL74 program can assign task attribute values using the CHANGE statement (the preferred method), the MOVE statement, or the SET statement. A COBOL74 program can interrogate task attributes using the MOVE statement. COBOL(68) supports MOVE and SET statements, but does not support the CHANGE statement.

COBOL74 programs can use all types of task attributes, including event-valued and task-valued task attributes.

Invoking COBOL74 Programs

Most COBOL74 programs can be invoked in either of two ways: through process initiation statements or through the library linkage mechanism. If the COBOL74 program is invoked through the library linkage mechanism, the program automatically freezes and exports the PROCEDURE DIVISION. This automatic freeze occurs even though the program does not include a FREEZE statement or export declaration. For further information about COBOL74 library capabilities, refer to Section 18, "Using Libraries."

Communicating with Other Processes from COBOL74

COBOL74 programs have access to almost all the interprocess communication methods discussed in this guide, including call-by-reference parameters, events and interrupts, port files, and libraries. The only interprocess communication method that does not apply to COBOL74 is the use of globally declared objects, because COBOL74 cannot initiate an internal procedure. For details about any of these interprocess communication methods, refer to Part II, "Interprocess Communication."

Restarting COBOL(68) Processes

A COBOL(68) process can use a CHECKPOINT statement to create a checkpoint file that describes the current process state. You can use the checkpoint file after a halt/load to restart the process. The CHECKPOINT statement is not available in COBOL74. For further information, refer to Section 11, "Restarting Jobs and Tasks."

COBOL74 Examples

The following COBOL74 program initiates a separate COBOL74 program called OBJECT/COBOL/TEST using the task variable TASK-VAR-1:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
Ø1 TASK-VAR-1  USAGE IS TASK.
Ø1 EXT-NAME        PIC X(8Ø).
PROCEDURE DIVISION.
DECLARATIVES.
PROC-EXTERNAL SECTION.
   USE EXTERNAL EXT-NAME AS PROCEDURE.
END DECLARATIVES.

START-HERE SECTION.
P1.
    MOVE "OBJECT/COBOL/TEST." TO EXT-NAME.
    PROCESS TASK-VAR-1 WITH PROC-EXTERNAL.

PROCWAIT SECTION.
P2.
    WAIT AND RESET UNTIL ATTRIBUTE EXCEPTIONEVENT OF MYSELF.
    IF ATTRIBUTE STATUS OF TASK-VAR-1 IS GREATER THAN
       VALUE TERMINATED THEN GO PROCWAIT.
STOP RUN.
```

The following COBOL74 program also invokes OBJECT/COBOL/TEST; but this program invokes OBJECT/ALGOL/TEST as an imported library procedure rather than as a task. OBJECT/COBOL/TEST is executed as part of the calling process.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
START-HERE SECTION.
P1.
    CALL "PROCEDUREDIVISION IN OBJECT/COBOL/TEST".
STOP RUN.
```

The following is the program OBJECT/COBOL/TEST, which can be invoked by either of the preceding two programs:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 MIXNO  BINARY PIC 9(11).
PROCEDURE DIVISION.
START-HERE SECTION.
P1.
    MOVE ATTRIBUTE MIXNUMBER OF MYSELF TO MIXNO.
    DISPLAY MIXNO.
STOP RUN.
```

The following COBOL74 program submits WFL input in array form for execution. The WFL statements are stored in an array of picture items. Note that if any of the WFL statements includes a quotation mark ("), the quotation mark must be represented by two quotation marks ("") in the MOVE statement that stores the statement in the array. The use of double quotation marks is necessary because the compiler interprets a single quotation mark as the end of the WFL input rather than as part of the WFL input.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
Ø1 PARAM.
   Ø5 PARAM-1      PIC X(8Ø).
   Ø5 PARAM-2      PIC X(8Ø).
   Ø5 PARAM-3      PIC X(8Ø).

PROCEDURE DIVISION.
START-HERE SECTION.
P1.
MOVE "CLASS=2;JOBSUMMARY=SUPPRESSED;ELAPSEDLIMIT=12Ø;" TO PARAM-1.
MOVE "MYSELF(STATION=MYSELF(SOURCESTATION));" TO PARAM-2.
MOVE "DISPLAY (""HI AGAIN"");" TO PARAM-3.
CALL SYSTEM WFL USING PARAM.

STOP RUN.
```

The following COBOL74 program initiates a utility. This example also shows how to pass parameters to a task from a COBOL74 program.

```
    IDENTIFICATION DIVISION.
    ENVIRONMENT DIVISION.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
     Ø1 TASK-VAR-1  USAGE IS TASK.
     Ø1 EXT-NAME        PIC X(8Ø).
     Ø1 ACTUALPARAM     PIC X(19).
    LOCAL-STORAGE SECTION.
    LD PARAMS.
     Ø1 FORMALPARAM     PIC X(19).
    PROCEDURE DIVISION.
    DECLARATIVES.
    PROC-EXTERNAL SECTION.
        USE EXTERNAL EXT-NAME AS PROCEDURE
        WITH PARAMS USING FORMALPARAM.
    END DECLARATIVES.
    START-HERE SECTION.
    P1.
        MOVE "*SYSTEM/LOGANALYZER ON DISK." TO EXT-NAME.
        MOVE "PRINTER JOB 126Ø" TO ACTUALPARAM.
        CALL TASK-VAR-1 WITH PROC-EXTERNAL USING ACTUALPARAM.
        STOP RUN.
```

The following COBOL74 example initiates a compilation:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
 Ø1 TASK-VAR-1  USAGE IS TASK.
 Ø1 EXT-NAME        PIC X(8Ø).
 Ø1 VALUE-ONE       PIC 9(11) BINARY VALUE 1.
 Ø1 ACTUALPARAM.
     Ø3 PARAMWORD BINARY PIC 9(11) OCCURS 33.
LOCAL-STORAGE SECTION.
LD PARAMS.
 Ø1 FORMALPARAM.
    Ø3 FORMALWORD BINARY PIC 9(11) OCCURS 33.
PROCEDURE DIVISION.
DECLARATIVES.
PROC-EXTERNAL SECTION.
    USE EXTERNAL EXT-NAME AS PROCEDURE
    WITH PARAMS USING FORMALPARAM.
END DECLARATIVES.
START-HERE SECTION.
P1.
    MOVE "*SYSTEM/ALGOL ON DISK." TO EXT-NAME.
    MOVE VALUE LIBRARY TO PARAMWORD (9).
```

```
MOVE VALUE-ONE TO PARAMWORD (1) [ØØ:47:Ø1].
CHANGE ATTRIBUTE FILECARDS OF TASK-VAR-1 TO
   "FILE CARD (KIND=DISK,TITLE=ALGOL/TASK);".
CHANGE ATTRIBUTE FILECARDS OF TASK-VAR-1 TO
   "FILE CODE (KIND=DISK,TITLE=OBJECT/ALGOL/TASK);".
CALL TASK-VAR-1 WITH PROC-EXTERNAL USING ACTUALPARAM.
STOP RUN.
```

In this example, the COBOL74 program initiates the compiler directly as a task. An alternative would be for the program to submit in array form a WFL program that contains a COMPILE statement.

Other Languages

The other user languages available on A Series systems are APLB, BASIC, C, COBOL85, FORTRAN, FORTRAN77, Pascal, PL/I, and RPG. These languages are not primarily intended for process initiation and control. However, most of these languages have one or more of the following tasking capabilities:

- Submitting WFL jobs

  If a program can submit a WFL job, the job can, in turn, initiate and control programs written in any language.

- Invoking library procedures

  You can implement ALGOL or COBOL74 libraries that export procedures that initiate or control processes. Any language that can use libraries can invoke these procedures.

- Using bound-in procedures

  You can bind ALGOL procedures or complete COBOL74 programs into programs written in other languages. These bound-in procedures can be designed to initiate and control processes.

The following are the tasking capabilities of each language:

- APLB

  Provides the task utility, which can be used to initiate tasks and to read or write task attributes. Also provides the zip utility for submitting WFL jobs. For further information, refer to the *A Series APLB Programming Reference Manual*.

- BASIC

  Has no tasking capabilities. This language is described in the *A Series BASIC Programming Reference Manual*.

- C

  Has no process-initiation capabilities. C programs can invoke library procedures in libraries that are written in other languages. For further information, refer to the *A Series C Programming Reference Manual*.

- COBOL85

  Has no process-initiation capabilities. COBOL85 programs can invoke procedures in libraries that are written in other languages. Additionally, you can add tasking features to a COBOL85 program by binding in ALGOL procedures or COBOL74 programs. For further information, refer to the *A Series COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation.*

- FORTRAN

  Can include ZIP statements that are used to submit WFL jobs for execution. FORTRAN programs can invoke library procedures in libraries that are written in other languages. Additionally, you can add tasking features to a FORTRAN program by binding in ALGOL procedures or COBOL74 programs. For further information, refer to the *A Series FORTRAN Programming Reference Manual.*

- FORTRAN77

  Can invoke library procedures in libraries that are written in other languages. Additionally, you can add tasking features to a FORTRAN77 program by binding in ALGOL procedures or COBOL74 programs. For further information, refer to the *A Series FORTRAN77 Programming Reference Manual*

- Pascal

  Can invoke library procedures in libraries that are written in other languages. Additionally, you can add tasking features to a Pascal program by binding in ALGOL procedures or COBOL74 programs. For further information, refer to the *A Series Pascal Programming Reference Manual, Volume 1: Basic Implementation.*

- PL/I

  Provides the process-initiation statements that are standard to this language. PL/I programs can read or assign task attributes. PL/I programs can also invoke library procedures in libraries that are written in other languages. For further details, refer to the *A Series PL/I Reference Manual.*

- RPG

  Can include ZIP statements that submit WFL jobs for execution. An RPG program can use the external indicators U1 through U8 to interrogate the SW1 through SW8 task attributes. For further information, refer to the *A Series Report Program Generator (RPG) Programming Reference Manual, Volume 1: Basic Implementation.*

Section 5
Establishing Process Identity and Privileges

Process identity is the term used in this guide for a number of task attributes and other features that uniquely identify a process and its capabilities. The "Process Identity" subsection of this section describes the various aspects of process identity and their implications for security, billing, and operations. The "Process Security Classes" subsection explains the security classes a process can belong to, and the privileges associated with each of these classes.

Process Identity

Some of the aspects of process identity, such as mix numbers, are assigned by the operating system. You can control other aspects of process identity, such as the usercode, although the system provides default values for these aspects if you do not.

Mix Number and Stack Number

In Section 1, "Understanding Basic Tasking Concepts," it was pointed out that there can be multiple processes running that are instances of the same object code file. Thus, the object code file title cannot serve as a unique identification for a process. Instead, the system assigns two identifying numbers to a process: the *mix number* and the *stack number*.

The mix number is a 4-digit number that the system assigns to each process when the process is initiated. The name arises because all the processes running on the system are collectively referred to as the system *mix*, and the mix number distinguishes a process from the other processes in the mix. Mix numbers identify processes in the system log as well as in many system commands and CANDE commands that affect running processes or provide information about them.

Additionally, mix numbers serve to identify entities that are not processes, but which resemble processes from the point of view of the operator or user. Thus, Menu-Assisted Resource Control (MARC) and Command and Edit (CANDE) assign a mix number to each session at log-on time. (The mix number of a session is also known as the *session number*.) The system assigns mix numbers to WFL jobs when they are first entered into a job queue. This mix number remains the same when the WFL job leaves the queue and begins executing.

The mix numbers of processes, sessions, and queued jobs are chosen from the same pool of numbers, so no two of these different entities have the same mix number. Each process, session, or queued WFL job is generally assigned a mix number one higher than the last assigned mix number. However, a jump in the numbering can occur after a

halt/load. This jump occurs because the system periodically reserves a range of numbers for use by new processes. After a halt/load, the system avoids reusing any number in the range reserved at the time of the halt/load.

When the mix numbers reach 9999, the numbering starts over. Certain low numbers are reserved and are not used. Also, a number cannot be reused if the last process to which it was assigned is still in the mix.

Processes can determine their own mix numbers, or the mix number of a related process, by interrogating the MIXNUMBER task attribute. However, the mix number has little use in programmatic tasking. A process accesses the task attributes of another process by specifying a particular task variable, rather than by specifying a particular mix number.

Like the mix number, the stack number of a process is a number assigned by the operating system. The stack number is unique to the process and remains constant for the lifetime of the process. However, while the mix number is intended primarily for use by system operators, the stack number of a process is intended primarily for internal use by the operating system. Yet the stack number is visible to operators and programmers in the following contexts:

- The stack number appears in the system log records for Major Type 0, Minor Type 1 (Establish Identity) and Major Type 1, Minor Types 1 (BOJ), 2 (EOJ), 3 (BOT), and 4 (EOT). The stack number is expressed in hexadecimal format.
- The output from the OT (Inspect Stack Cell) system command includes the stack number for the process. The stack number is expressed in hexadecimal format. Thus, in the following output, the stack number is 011B:

  ```
  Ø11B STACK CELL 2Ø= 7 Ø962465ØEØØ3 (HEX)
  ```

- The PROCESSID function in ALGOL returns the stack number of the process. The stack number is expressed in decimal format.
- The stack number can appear in memory dump analyses created by DUMPANALYZER. The stack number is expressed in hexadecimal format.

Usercode, Access Code, and Charge Code

USERCODE, ACCESSCODE, and CHARGE are three closely-related task attributes that help to specify the identity and privileges of a process.

The USERCODE task attribute stores a value that is intended to identify the user who initiated the process. In actual practice, more than one user of the system can use the same usercode, but only if all the users agree to do so. This is because you must know the password associated with a usercode to use the usercode, and only the owner of the usercode can tell you the password.

Usercodes are created by the security administrator for the system, usually through the use of the MAKEUSER utility. The security administrator can associate a variety of *usercode attributes* with each usercode. Some of these usercode attributes confer various types of special security privileges, as described under "Process Security Classes" later in this section.

Other usercode attributes interact with the values of various task attributes. Some of these usercode attributes provide default values for the corresponding task attributes. Other usercode attributes define a range of permitted values for a task attribute, or specify whether the task attribute must have a value. The following are these usercode attributes and the task attributes that are related to them:

Usercode Attribute	Task Attribute
ACCESSCODELIST, ACCESSCODENEEDED	ACCESSCODE
CHARGECODE, CHARGEREQ, USEDEFAULTCHARGE	CHARGE
CANDEDESTNAME	DESTNAME
CLASS, CLASSLIST, ANYOTHERCLASSOK	CLASS
CONVENTION	CONVENTION
DEPTASKACCOUNTING	DEPTASKACCOUNTING
FAMILY	FAMILY
FILEACCOUNTING	FILEACCOUNTING
LANGUAGE	LANGUAGE
PRINTDEFAULTS	PRINTDEFAULTS
PRIORITY	PRIORITY
SAVEMEMORYLIMIT	SAVEMEMORYLIMIT
TEMPFILELIMIT	TEMPFILELIMIT

The values supplied by usercode attributes are propagated to their corresponding task attributes in the following ways:

- MARC and CANDE read some usercode attributes when you log on, and store the corresponding task attribute values for your session. Thereafter, if you initiate a process from that session, the process inherits the task attributes of the session.
- If a WFL job includes a USERCODE assignment in the job header, the WFL job inherits the attribute values associated with the usercode.

For details about the effects of usercode attributes on task attributes, refer to the task attribute descriptions in the *A Series Task Attributes Programming Reference Manual*.

The ACCESSCODE task attribute serves as a form of secondary identification, in addition to the usercode. This identification is relevant only when a process attempts to use a file that is guarded by a guard file; refer to "Nonprivileged Status" later in this section for further details.

The CHARGE task attribute serves as a form of group identification for billing purposes. Thus, all the people working in a particular department might have usercodes with the same CHARGECODE usercode attribute. The system records the CHARGE attribute of each process in the system log. This makes it possible for site personnel to write billing programs that analyze the system usage on a charge code by charge code basis. For further information about billing programs, refer to the *A Series System Administration Guide*.

You can override the propagation of most usercode attributes to task attributes by explicitly assigning task attributes to the process in question. However, the system enforces some consistency checks to ensure that the USERCODE, ACCESSCODE, and CHARGE attribute values are consistent with each other. For details about these consistency checks, refer to the descriptions of these attributes in the *A Series Task Attributes Programming Reference Manual*.

A process can change its own usercode while it is running by making an assignment to the USERCODE attribute. Such an assignment must specify the password as well as the usercode. The system verifies the correctness of the usercode and password before making the usercode assignment.

Name

The name of a process is stored in the NAME task attribute of the process. The value of this attribute is, by default, the same as the title of the object code file that the process is executing. The process name appears in system log entries generated for the process. The process name also appears in the output from system mix display commands such as A (Active Mix Entries), W (Waiting Mix Entries), and C (Completed Mix Entries).

In addition to aiding the operator, the process name can affect the ability of the process to use some files. If a file has a guard file associated with it, the guard file can include a PROGRAM clause that specifies access rights for processes with a given name.

In some cases, the NAME value for a process can be different from its object code file title. This can occur if a WFL process or an ALGOL process initiates an internal procedure. The initiating process can make an arbitrary assignment to the NAME attribute of the new process before initiating it.

The initiating process can even assign the internal process the NAME of an entirely different program. This method enables the process to circumvent the PROGRAM clause in a guard file. To prevent such abuses, a CODEFILE clause is also available for guard files. This clause ignores the process name and instead specifies access rights for processes having a particular object code file title. For details, refer to the *A Series Security Features Operations and Programming Guide*.

Object Code File

An operator can use the MP (Mark Program) system command to assign any of several options to an object code file. Some of these options confer special types of security status on a process, and these options are the following:

- COMPILER. This option marks an object code file with compiler status. Object code files can also be marked with compiler status by the MC (Make Compiler) system command, which is scheduled for deimplementation. The effects of compiler status are described under "Compiler Status" later in this section.
- CONTROL. This option marks an object code file with control program status. Object code files can also be marked with control program status by the CP (Control Program) system command, which is scheduled for deimplementation. The effects of control program status are discussed under "Controlling Process Priority" in Section 7, "Controlling Processor Usage."
- PU. This option marks an object code file with privileged status. Object code files can also be marked with privileged status by the PP (Privileged Program) system command, which is scheduled for deimplementation. The effects of privileged status are discussed under "Privileged Status" later in this section.
- SECADMIN. This option marks an object code file with security administrator status. Object code files can also be marked with security administrator status by the PP (Privileged Program) system command, which is scheduled for deimplementation. The effects of security administrator status are described under "Security Administrator Status" later in this section.
- TASKING. This option marks an object code file with tasking status. The effects of tasking status are described under "Tasking Status" later in this section.

When an object code file is initiated, the resulting process receives the privileges that were assigned to the object code file. The process can make some of the procedures in the object code file available to other processes by initiating an internal procedure, by initiating a process and passing a procedure parameter, or by becoming a library and exporting procedures. Any of these processes temporarily assumes the privileges assigned to the object code file while it is executing procedures from the object code file.

The following subsections explain how these privileges are propagated to processes from object code files.

Transparent Object Code File Privileges

Most of the options available through the MP (Mark Program) system command have only two states: set or reset. However, the MP command enables you to specify a third state for the PU, SECADMIN, and TASKING options. This third state is called *transparent*. The following are MP commands and the security categories they assign:

MP Command	Security Category
MP <file title> + PU	Privileged
MP <file title> + PU TRANSPARENT	Privileged transparent
MP <file title> – PU	Nonprivileged
MP <file title> + SECADMIN	Security administrator
MP <file title> + SECADMIN TRANSPARENT	Security administrator transparent
MP <file title> – SECADMIN	Non-security administrator
MP <file title> + TASKING	Tasking
MP <file title> + TASKING TRANSPARENT	Tasking transparent
MP <file title> – TASKING	Nontasking

Each option can be in only one state at a time: enabled, disabled, or transparent. However, the three options (PU, SECADMIN, and TASKING) do not have to be in the same state. The following command assigns privileged status and security administrator transparent status, and removes tasking status:

```
MP <file title> + PU, + SECADMIN TRANSPARENT, - TASKING
```

The concept of transparent status is intended primarily for libraries, to enable the actions of a library to be applied with the status of the user program that invokes the library. If a procedure resides in an object code file that has one of these options in the transparent state, then

- If the procedure is initiated, the resulting process is treated as if the option were disabled.
- If the procedure is entered, it inherits the enabled or disabled state of the option of the invoking procedure. Privileged, security administrator, or tasking status can be inherited through a series of privileged transparent procedures.

For example, if a privileged program initiates a procedure in a privileged transparent library, the procedure is executed as nonprivileged. However, if the privileged program enters the same procedure instead of initiating it, the procedure is executed as privileged.

For information about how privileged transparent status applies to file access rights, refer to Section 19, "Using Shared Files."

Delayed Effects of Object Code File Privileges

A process can receive privileged, security administrator, or compiler status from an object code file only if the process uses a code segment dictionary that was created after the object code file was marked with the specified status. Marking an object code file with privileged or compiler status does not affect processes that are already in progress. Even a process initiated after the object code file is marked might not receive the specified privileges in some situations. For further information, refer to the discussion of code segment dictionary sharing in Section 8, "Controlling Process Memory Usage."

Copying Privileged Object Code Files

If you copy an object code file marked with privileged, security administrator, or compiler status, the copy retains the same privileges as the original. However, the system administrator can limit the ability to copy or execute such object code files by using the RESTRICT (Set Restrictions) system command. For details, refer to the discussion of the RESTRICT command in the *A Series Security Administration Guide*.

Originating Source

When you initiate a process through a peripheral device, the system records the type of peripheral device in the SOURCEKIND attribute. There is one situation in which the SOURCEKIND value can make an important difference in the capabilities of the process. If the SOURCEKIND value is ODT, the system accords the process ODT status, which is described under "Process Security Classes" in this section.

Additionally, the system records the physical unit number or logical station number (LSN) of the originating peripheral device in the SOURCESTATION task attribute. The value of this attribute allows messages generated by a process to be routed back to the station that originated the process, so that you can easily monitor the progress of your processes.

MARC and CANDE similarly assign the LSN of a session to the STATION task attribute of any tasks (but not jobs) initiated from that session. Refer to Section 9, "Controlling Process I/O Usage" for a discussion of the effects of this attribute.

The system also records the name of the originating station in the SOURCENAME task attribute. The station name can be more stable than the LSN, which often changes after a halt/load or COMS quit.

Process Security Classes

A Series software provides a number of security features that you can use to regulate the ability of processes to access other users' files or perform other restricted actions. Processes are classified according to security classes, and each security class allows the process to perform a somewhat different set of restricted actions.

The following subsections describe the capabilities of each of the process security classes and explains how a process can be assigned to a particular class. For further information

about any of the security features discussed, refer to the *A Series Security Features Operations and Programming Guide* and the *A Series Security Administration Guide*.

The following are the security classes a user process can belong to: nonprivileged, privileged, nonusercoded, operator display terminal (ODT), security administrator, and compiler. A process can belong to more than one of these classes, although certain classes are mutually exclusive. In addition, a process can belong to different security classes at different points in its execution.

Additional security classes exist for operating system processes. For information about system library security and library linkage classes, refer to Section 18, "Using Libraries."

For a discussion of certain special security issues that arise from the sharing of logical files between processes, refer to Section 19, "Using Shared Files."

Nonprivileged Status

The default security class for a process is nonprivileged. On a typical system, the vast majority of processes fall into this class. A nonprivileged process can perform any of the following actions:

- Inspect or modify any object within the extended addressing environment of the process. For information about the addressing environment, refer to Section 15, "Using Global Objects," and Section 17, "Using Parameters."
- Create, remove, open, close, read, write, copy, or access the file attributes of data files.
- Initiate, copy, remove, open, close, read, or access the file attributes of object code files.
- Use the nonprivileged form of the GETSTATUS directory call. The nonprivileged form of this call provides information only about directories having the same usercode as the process.
- Use the *VOLUME CHANGE* form of the WFL *VOLUME* statement to affect tape volumes whose FAMILYOWNER value is the same as the usercode of the process.
- Use the WFL ARCHIVE command to back up, roll out, or restore files that have the same usercode as the process.

The ability of a nonprivileged process to access a particular disk file is determined by the values of certain task attributes and file attributes. The following task attributes affect file access rights:

- USERCODE

  The USERCODE value generally grants the process access to files that are stored under the usercode. Certain USERCODE values can also grant special privileges, as discussed under "Privileged Status" and "Nonusercoded Status" later in this section.

- ACCESSCODE

  The ACCESSCODE value can grant the process access to some files that are protected by guard files, as discussed later in this subsection.

- NAME

  The value of this task attribute can grant the process access to some files that are protected by guard files, as discussed later in this subsection.

- FILEACCESSRULE

  The effects of this task attribute are discussed in Section 19, "Using Shared Files."

The process that creates a disk file can assign security-related file attributes to determine which nonprivileged processes can access the file. Thereafter, only privileged processes or processes running with the same usercode as the file can change the values of these security-related file attributes. Following are brief descriptions of the security-related file attributes:

- TITLE

  This file attribute includes the usercode under which the file is stored. For nonusercoded files, an asterisk (*) is included instead of a usercode. Only privileged or nonusercoded processes can create a nonusercoded file.

- SECURITYTYPE

  This file attribute specifies whether a process must have the same usercode as the file in order to access the file. A value of PUBLIC allows any process to access the file. A value of PRIVATE enables nonprivileged processes to access the file only if the processes are running under the same usercode as the file. For nonusercoded files, a value of PRIVATE enables only privileged processes and nonusercoded processes to access the file. A value of GUARDED or CONTROLLED specifies that a guard file is used to determine which nonprivileged processes can access the file.

- SECURITYUSE

  This file attribute specifies whether nonprivileged processes having a usercode different from the file can read from or write to the file. SECURITYUSE does not restrict the ability to initiate an object code file. SECURITYUSE has effect only if the SECURITYTYPE file attribute value is PUBLIC.

- SECURITYGUARD

  For files with a SECURITYTYPE value of GUARDED or CONTROLLED, the SECURITYGUARD file attribute specifies the title of the guard file to be used.

These file attributes are described in detail in the *A Series File Attributes Programming Reference Manual*.

Guard files can be created using the GUARDFILE utility, which is described in the *A Series Security Features Operations and Programming Guide*. A guard file can include detailed information about the types of access allowed to various nonprivileged processes. The guard file can include USERCODE or ACCESSCODE clauses that discriminate between processes on the basis of the corresponding task attributes. The guard file can also include a PROGRAM clause that discriminates between processes on the basis of the NAME task attribute value.

If a guard file is used, it overrides the value of the SECURITYUSE attribute.

If the InfoGuard tape volume security feature is enabled on the system, then the rights of a nonprivileged process to access a particular tape file are regulated by the task attributes and file attributes listed in the previous discussion as well as by the tape volume attributes FAMILYOWNER, PERMANENTLYOWNED, and MATCHONLYSERIALNO. The tape volume attributes can be assigned only by a privileged user or a privileged process with the WFL *VOLUME* statement. The security administrator can enable tape volume security by using the SECOPT (Security Options) system command to set the security option TAPECHECK to AUTOMATIC. If tape volume security is not enabled, then a nonprivileged process can open a tape file on any tape unit that is not currently in use by another process.

An additional security restriction for disk files is *system file* status. The operating system marks disk files that are part of the acting system software as system files. Examples of system files are the object code file of the current MCP, the job description file, and the current system log. An application process cannot remove or change the title of any system file. Some files have a modified form of system file status. Thus, the USERDATAFILE has system file status and additionally is protected from being read by any application process (only system software can read this file).

Privileged Status

A privileged process has the capabilities of a nonprivileged process, as well as the following capabilities:

- The ability to access physical files stored under other usercodes, regardless of the SECURITYTYPE, SECURITYUSE, and SECURITYGUARD file attribute values. Any guard files are ignored.
- The ability to use the WFL ARCHIVE command to backup, roll out, or restore files regardless of their usercode.
- The ability to initiate nonusercoded processes and create nonusercoded files.
- The ability to survive most task attribute access errors.

Privileged status also grants several other capabilities on systems where the InfoGuard security administrator feature is not enabled. On systems where the security administrator feature is enabled, these capabilities are wholly or partially reserved for processes with security administrator status. (Refer to "Security Administrator Status" later in this section.) The following are the capabilities:

- The ability to create or alter usercode definitions.
- The ability to access certain system interfaces, including the DCKEYIN, GETSTATUS, and SETSTATUS functions in DCALGOL.
- The ability to read from the USERDATAFILE.
- The ability to remove the current INFOGUARDSUPPORT library object code file. This file has a modified form of system file status that enables it to be removed by privileged processes, but does not allow title changes.

Note that the following types of file access are *not* granted by privileged status: the ability to remove or change the titles of most system files, and the ability to write to object code files. Further security restrictions can apply if the privileged process accesses the file through a shared logical file, as discussed in Section 19, "Using Shared Files."

A process is automatically considered privileged if it is running under a privileged usercode. The usercode of a process is stored in the USERCODE task attribute. An operator can assign privileged status to a usercode by running the MAKEUSER utility or using the MU (Make User) system command. A usercode can also be assigned privileged status by a program that uses the USERDATA function in DCALGOL. For further information about these features, refer to the *A Series Security Administration Guide*.

A process usually inherits the usercode of the session or process that initiated it. A different usercode can be assigned by task attribute assignment, use of the DCALGOL *USERDATA* function, or use of the WFL *USER* statement. However, in each of these cases, the statement that assigns the usercode must also specify a password, which is checked for validity. Only processes with special privileges can assign a usercode without specifying a password. Message control systems (MCSs) and processes with tasking status use this feature when assigning a usercode to a process initiated by a session.

If a process is not running under a privileged usercode, then the ability of a process to perform a privileged action is determined by the privilege status of the object code file that contains the request.

A process can execute code from several different object code files. This is the case if the process has entered either a library procedure or a passed external procedure. (For an introduction to external procedures, refer to Section 1, "Understanding Basic Tasking Concepts.") The various object code files might not have the same privilege status. The current privilege status for the process is determined by the privilege status of the object code file containing the procedure that was most recently entered. This procedure contains the code that is currently being executed. For further details about this concept, refer to "Object Code File" earlier in this section.

Note that a privileged program has no special privileges when accessing files on a remote host. For example, suppose a process sets the HOSTNAME attribute of a file to specify a remote host, and then attempts to open that file. This action is executed with privilege on the remote host only if the process usercode is privileged on that host.

Nonusercoded Status

A nonusercoded process is one whose USERCODE task attribute value is a null string. By default, a process runs without a usercode if you initiate it from one of the following sources: a nonusercoded MARC session, a card reader, or a load control tape.

In addition, a process initiated from an ODT is nonusercoded by default unless one of the following conditions is true:

- The ODT has been assigned a terminal usercode by the TERM (Terminal) system command. The terminal usercode is the default usercode for most processes initiated at that ODT.

  However, processes initiated at an ODT by a primitive system command default to a null usercode, even if there is a terminal usercode associated with the ODT. ??COPY (Copy Files) and ??RUN (Run Code File) are two primitive system commands that initiate processes.

- The process is a remote WFL job and the system has a host usercode. Host usercodes are assigned by the HU (Host Usercode) system command.

Processes initiated by a nonusercoded process are, by default, also nonusercoded.

Processes initiated by usercoded processes are, by definition, always usercoded. It is possible for a process to assign a null usercode to a task variable that is not in use, and then initiate a process with that task variable. However, the null usercode value in the task variable is overridden by task attribute inheritance, and the new process runs with the usercode of its initiator.

It is possible for a usercoded process to be assigned a null usercode after initiation. However, only a privileged process can assign a null usercode to an in-use process. Thus, for example, a privileged process can change its own usercode to a null usercode. When the usercode of a privileged process is changed to a null usercode, the process retains its privileged status.

A privileged process can also initiate a task with a nonprivileged usercode, and then change the usercode of the task to a null while the task is running. The task then assumes nonusercoded security status. Processes that are nonusercoded from the time they are first initiated also have nonusercoded security status.

A process with nonusercoded status has the same capabilities as a nonprivileged process, with the following additions:

- The ability to create nonusercoded files; that is, files whose TITLE file attributes begin with an asterisk (*) instead of a usercode.
- The ability to initiate a nonusercoded process; that is, a process whose USERCODE task attribute value is a null string.
- The ability to use the UNITNO file attribute, even on a system running with the security option S2RESTRICTIONS set.

Further, certain WFL statements are treated as privileged when submitted by a nonusercoded process. These statements, and other conditions affecting their privilege status, are shown in Table 5–1. This table refers to two concepts not discussed previously:

- Single-statement WFL inputs. These are single WFL statements entered directly at an ODT, entered in CANDE or MARC with the *WFL* prefix, or submitted in array form by a ZIP statement in a program.
- ODT status. This concept is defined under "ODT Status" later in this section.

Table 5–1. WFL Statements Executed with Privilege

WFL Statements	Conditions Granting Privilege
ADD, COPY	Privileged if the process is nonusercoded.
CHANGE, REMOVE, RERUN, SECURITY, START	Privileged if a nonusercoded, single-statement WFL input.
PRINT	Privileged if a nonusercoded, single-statement WFL input that does *not* have ODT status.
VOLUME	Privileged if either of the following is true: • The process is nonusercoded and has ODT status. • The process has ODT status, only the VOLUME ADD or VOLUME DELETE form of the command is used, and the statement affects only volumes with the same usercode as the process.

ODT Status

A process is said to have *ODT status* if it was initiated from an ODT, or if it is descended from a process initiated from an ODT. The exception to this rule is that processes initiated with the ??RUN (Run Code File) primitive system command do not receive ODT status, nor do the descendants of such processes.

Processes initiated from an ODT frequently run without a usercode and receive nonusercoded status, as discussed under "Nonusercoded Status" earlier in this section.

Regardless of whether it has a usercode, a process with ODT status is granted access to all GETSTATUS calls in DCALGOL. This access includes the privileged form of the GETSTATUS directory call. (The privileged form of this call can return information about directories stored under any usercode.)

Certain WFL statements are treated as privileged when submitted by a process with ODT status. For a list of these statements, and other conditions affecting their privilege status, refer to Table 5–1, "WFL Statements Executed with Privilege".

SYSTEMUSER Status

A process receives SYSTEMUSER status if it is running under a usercode whose SYSTEMUSER usercode attribute is set. SYSTEMUSER status enables a process to use the DCKEYIN, GETSTATUS, and SETSTATUS functions in DCALGOL, even if the process does not have privileged status. A process can use these functions to submit system commands and perform other system operations functions.

By default, SYSTEMUSER status gives access to all the possible DCKEYIN, GETSTATUS, and SETSTATUS calls. However, certain restrictions can apply on a system running InfoGuard security enhancement software. Refer to the following subsection, "Security Administrator Status."

Security Administrator Status

On a system where InfoGuard security enhancement software is installed, the system administrator can enable a special security administrator status. If security administrator status is enabled for the system, then certain system commands that would otherwise be available to any privileged or SYSTEMUSER process are instead reserved for use only by processes with security administrator status. The DCKEYIN and SETSTATUS functions corresponding to these system commands are similarly restricted. In addition, the ability to create or alter usercode definitions, which would otherwise be available to any privileged user, is restricted to processes with security administrator status.

The security administrator can also use the RESTRICT command to prevent or limit the use of certain system commands. For information about the RESTRICT command, refer to the *A Series System Commands Operations Reference Manual.*

The system administrator can enable security administrator status on the system by setting the system SECADMIN option. This option is set using the *??SECAD* system command. Once the SECADMIN option is set, a process assumes security administrator status if either of the following conditions are true:

- The process is running with a usercode for which the SECADMIN attribute is set in the USERDATAFILE.
- The process is executing code from an object code file that has been marked with system administrator status. This concept is discussed further under "Object Code File" earlier in this section.

For further information about security administrator capabilities, refer to the *A Series Security Administration Guide.*

Compiler Status

A process with compiler status is allowed to create an object code file or write to an existing object code file. You can mark an object code file with compiler status by using the *MP <file title> + COMPILER* form of the MP (Mark Program) system command. An operator can use this command to mark any program with compiler status, whether or not the program is really a compiler.

If a process without compiler status attempts to write to an object code file that is a permanent file, the write operation is not performed and the process is abnormally terminated. A process without compiler status can write to an object code file that is a temporary file. However, if the process attempts to lock the file, the system changes the file from an object code file into a data file. (For information about the concepts of permanent and temporary files, refer to the *A Series I/O Subsystem Programming Guide*.)

Note that a compiler program has no special privileges when accessing object code files on a remote host. For example, suppose you initiate a compiler and file equate the HOSTNAME attribute of the CODE output file to a remote host. The compiler receives a file attribute error when it attempts to create the object code file. A compiler must create object code files on the host where the compiler is running.

Message Control System Status

Message control systems (MCSs) differ from other interactive programs in that they interface directly to the data comm subsystem (rather than opening a remote file) in order to send or receive messages from terminals. This interface is possible because MCSs are written in DCALGOL, an extended version of ALGOL with special data comm capabilities. The system extends a number of special privileges to MCSs.

How an MCS Acquires Its Privileges

The MCS security privileges and MCS priority are not granted to a program simply because it is written in DCALGOL; the system must also recognize the program as an MCS. Two things are necessary for the system to recognize a program as an MCS:

- Each MCS on a system must be named in the data comm network definition for that system. Only one MCS of a given name can be active.
- The MCS must invoke the DCALGOL *DCWRITE* function to initialize its primary queue. Every MCS must have such a queue and must initialize it in order to be recognized as an MCS.

Priority of an MCS

An MCS automatically runs in the same priority category that control programs run in. This priority category gives the MCS higher priority than WFL jobs and application programs. However, the priority of an MCS is lower than that of any invisible independent runner. The priority of MCSs relative to each other is determined by the PRIORITY task attribute. For an explanation of process priority, refer to Section 7, "Controlling Processor Usage." For a discussion of how and when this special priority can be inherited by offspring of an MCS, refer to "Inheritance of MCS Status" later in this section.

Privileges of an MCS

MCS status includes all the privileges associated with privileged status, as discussed under "Privileged Status" earlier in this section. This is true even if the MCS has not been marked as a privileged program. The following paragraphs describe other special privileges and features of MCS status.

An MCS is allowed the following privileges with regard to the DCALGOL *USERDATA* function:

- Ability to temporarily assume the usercode of a user by calling USERDATA function 3 (Validate Usercode/Password). If the MCS sets bit [1:1] of the locator parameter to USERDATA, then the MCS temporarily loses its MCS privileges. When bit [1:1] is set and bit [0:1] is also set, then the MCS assumes the privileges that the requested usercode would normally have; if [1:1] is set but bit [0:1] is reset, then the MCS runs as nonprivileged.

  When an MCS uses USERDATA function 3 to temporarily assume a usercode, the MCS does not appear in GETSTATUS mix request calls that request mix entries with that usercode. The MCS also does not appear in the output from system commands that display the mix and that request mix entries with that usercode.
- Ability to change a user's password with USERDATA function 6, subfunction 1, which is normally disallowed on a system using password generation.
- Ability to call the USERDATA function to validate a usercode/password combination or, optionally, a usercode without the password. This USERDATA function allows the MCS to run with the specified usercode so the MCS can perform a function on behalf of that usercode.
- Ability to call the USERDATA function to validate a usercode/chargecode combination.
- Ability to call the USERDATA function to validate an accesscode/accesscode password combination.
- Ability to call USERDATA function 9 (Privileged Fetch and Examine).
- Ability to specify that the last log-on information for a usercode should be updated as a result of the current USERDATA call.
- Ability to survive USERDATA errors that would normally be fatal. The errors are returned in the USERDATA error result field.

An MCS receives the following special privileges with regard to other restricted DCALGOL functions:

- Ability to call the DCWRITE function, which handles station message traffic.
- Ability to call the MCSLOGGER function, which creates sessions or logs session activity.
- Ability to survive SETSTATUS errors that would otherwise be fatal. The SETSTATUS error reporting mechanism returns the error to the MCS process.
- Trusted status that causes the operating system not to perform validation on any mix numbers specified by the MCS in SETSTATUS calls.

An MCS receives the following special privileges with regard to initiating processes:

- Ability to survive errors in initiating an external object code file, such as security errors, that would otherwise be fatal. Also, the ability to attempt to initiate a missing external code file without becoming suspended with a NO FILE condition. The MCS can determine if initiation was successful by inspecting the task variable used in the process initiation statement. If initiation failed, the STATUS task attribute has a value of BADINITIATE. The reason for the failure is reported in the HISTORYTYPE, HISTORYCAUSE, and HISTORYREASON task attributes.
- Ability to pass a single array parameter to an offspring process by value instead of by reference.
- Trusted status that causes the operating system not to perform validation of any sheet array parameter the MCS passes when initiating a compiler.

An MCS receives the following special privileges with regard to task attribute access:

- Ability to survive task attribute errors that would normally be fatal. The MCS can determine whether an error occurred, and the type of error, by interrogating the TASKERROR task attribute of the task variable that was accessed.
- Ability to commit task attribute errors in reading a task attribute without any error messages resulting.
- Ability to assign values to the BACKUPFAMILY, JOBNUMBER, and SOURCESTATION task attributes. Also, ability to set the ACCESSCODE task attribute to a null string, and to set the FILEACCESSRULE task attribute to a value of ACTOR.

An MCS has the following special privileges with regard to file access:

- Ability to invoke the exported MCP procedure CHANGESECURITY, which changes the security attributes of files.
- Ability to survive security errors that occur while changing file security.
- Ability to call the exported MCP procedure DRCDETERMINEUSERLIMITS, which reports on the file usage limits imposed on a user by the disk resource control (DRC) system.
- Ability to exceed DRC limits at file open time without any error messages being displayed.

An MCS has sufficient privilege to access library objects that have a linkage class of 3. For information about library linkage classes, refer to "Security Considerations" in Section 18, "Using Libraries."

Inheritance of MCS Status

In some cases it can be useful for an MCS to initiate one or more tasks to handle some of its work. By default, these tasks do not receive any special privileges as a result of being initiated by an MCS. These tasks also do not receive the special MCS priority category, and by default they receive a PRIORITY task attribute value of 50 rather than inheriting the PRIORITY value of the MCS.

However, the MCS can grant MCS privileges and priority to any of its offspring by assigning a value of TRUE to the INHERITMCSSTATUS task attribute of the offspring.

Tasking Status

Tasking status grants a process most of the privileges associated with MCS status, without the process actually having to be an MCS. Tasking status is well suited to interactive programs that service multiple users and need to be able to assume the identity of those users temporarily.

A process receives tasking status when both of the following conditions are true:

- The process is executing code from an object code file that has been marked with tasking status. You can mark object code files with tasking status by using the MP (Mark Program) system command. For information about how processes receive privileges from their object code files, refer to "Object Code File" earlier in this section.
- The process is nonusercoded. If an object code file with tasking status is initiated under a usercode, the process runs with the privileges of that usercode. The process receives tasking status when and if it changes its own USERCODE task attribute value to a null string.

Tasking status provides the same privileges and restrictions as MCS status, with the following exceptions:

- Tasking status does not grant access to the DCALGOL functions DCWRITE and SETUPINTERCOM.
- Tasking status does not allow a process to make assignments to the SOURCESTATION task attribute.
- Tasking status does not cause the process to run in the MCS priority category.
- Multiple instances of the same tasking program can be running at the same time.

Further, the INHERITMCSSTATUS task attribute does not cause tasking status to be inherited.

Section 6
Monitoring and Controlling Process Status

During its lifetime, a process can pass through several distinct states. These states characterize whether the process is currently executing, and if not, why not. You can design programs to monitor and modify process states through the use of task attributes and related expressions. You can also monitor process status through the use of system commands.

Understanding Process Status

From the time a process is initiated until it terminates, it is considered an *in-use* process. An in-use process can pass through several process states. These process states indicate whether the process is current executing, and if not, why not.

When a process is not in use, the task variable for that process stores any of several process states. These process states specify if the process has not been initiated, if initiation failed, or if the process has terminated. It is possible for the task variable to store this information because the task variable of a process exists before the process is initiated and continues to exist after the process terminates. In the following WFL example, task variable T is created when the system executes the declaration TASK T, and continues to exist until the system executes the END JOB statement:

```
?BEGIN JOB;
TASK T;
RUN OBJECT/PROG [T];
.
.
.
?END JOB
```

Information about process status is available through several mechanisms, including the STATUS task attribute, the task state expression in WFL, mix display commands, and the STACK STATE line of the Y (Status Interrogate) system command output. Each of these programming constructs and system commands uses a slightly different terminology to portray process status. Table 6–1 shows the possible values of the STATUS task attribute and the corresponding status values returned by the other

process monitoring methods. The meanings of the various process states are discussed in the subsections following the table.

Table 6–1. Process States

STATUS Task Attribute	WFL Task State	Mix Display Commands	STACK STATE in Y Display
NEVERUSED	None	None	None
SCHEDULED	Both of these: SCHEDULED INUSE	Both of these: S (Scheduled Mix Entries) MX (Mix Entries)	One of these: SCHEDULED SELECTED
ACTIVE	Both of these: ACTIVE INUSE	All of these: A (Active Mix Entries) J (Job and Task Display) MX (Mix Entries)	One of these: ALIVE HOLDING READY TO BE CONTINUED WAITING ON AN EVENT
SUSPENDED	Both of these: STOPPED INUSE	Both of these: W (Waiting Mix Entries) MX (Mix Entries)	WAITING ON AN EVENT
FROZEN	Both of these: ACTIVE INUSE	LIBS (Library Task Entries)	FROZEN
GOINGAWAY†	None	None	None
BADINITIATE	ABORTED	C (Completed Mix Entries)	None
TERMINATED	One or more of these: COMPLETED COMPLETEDOK COMPILEDOK ABORTED	C (Completed Mix Entries)	None

† GOINGAWAY is a write-only value. That is, assigning GOINGAWAY actually changes the STATUS value to ACTIVE, with additional effects that are described under "Thawing a Library" later in this section.

STATUS Task Attribute

The following are explanations of the STATUS task attribute values shown in Table 6–1.

- NEVERUSED

  The task variable being interrogated has never been used in a process initiation statement, or has been reinitialized since it was last used. Refer to "Preparing a Task Variable for Reuse" later in this section.

- SCHEDULED

  A process initiation statement has been executed, but the system is delaying initiation of the process. For further information, refer to "Preventing Process Scheduling" later in this section.

- ACTIVE

  Process execution is proceeding normally. In the Y command output, this status is expressed through any of several more specific values, which are described under "Y (Status Interrogate) Stack States" later in this section.

- SUSPENDED

  The process is waiting on an event that might require operator intervention. For example, the process might be trying to open a tape file, and the operator might need to mount the appropriate tape on a drive. Or the process might have executed an ACCEPT statement, which requires a response from an operator. For further information, refer to the discussion of responding to waiting entries in the *A Series System Operations Guide*.

- FROZEN

  The process is a frozen library process. It might be a permanent or temporary library. For information about library processes, refer to Section 18, "Using Libraries."

- GOINGAWAY

  You can assign this value to a frozen library process to cause it to resume execution as soon as possible. For details, refer to "Thawing a Library" later in this section.

- BADINITIATE

  An unsuccessful attempt was made to initiate the process. Process initiation can fail, for example, if the specified object code file is missing or if the object code file expects different parameters than are passed by the initiator. Note that BADINITIATE is only one of the terminations that can cause the WFL task state of ABORTED to return a value of TRUE. See the following description of TERMINATED.

- TERMINATED

  The process initiated successfully and later terminated. You cannot tell from the STATUS value whether the process completed normally or whether some circumstance caused the process to fail. For further information about process terminations, refer to "WFL Task State Expression" later in this section and to Section 10, "Determining Process History."

WFL Task State Expression

The task state expression in WFL returns a Boolean value indicating whether a process is in a specified state. For example, the following statement fragment takes a specified action if the process with task variable T1 has completed execution.

```
IF T1 IS COMPLETED THEN...
```

Some of the task states that can be queried correspond to single STATUS task attribute values. Other task states correspond to two or more STATUS task attribute values. Thus, at any given time, it is possible that more than one of the possible task state expressions will return a value of TRUE. The following are values that can be used in a WFL task state expression, and the conditions that cause them to evaluate to TRUE:

- SCHEDULED

  The system is delaying initiation of the process. The STATUS task attribute value is SCHEDULED.

- ACTIVE

  The process is executing normally. The STATUS task attribute value is ACTIVE.

- STOPPED

  The process is waiting on an event that might require operator action. The STATUS task attribute value is SUSPENDED.

- INUSE

  The process is in use; that is, it has been initiated but has not yet terminated. The STATUS task attribute value is SCHEDULED, ACTIVE, or SUSPENDED.

- COMPLETED

  The process terminated. The STATUS task attribute value is TERMINATED or BADINITIATE.

- COMPLETEDOK

  The process completed execution normally, but if it was a compilation, it might not have compiled the program successfully. The STATUS task attribute value is TERMINATED and the HISTORYTYPE task attribute value is NORMALEOTV or SYNTAXERRORV.

- COMPILEDOK

  The process completed execution normally. If the process was a compilation, it compiled the program successfully. The STATUS task attribute value is TERMINATED and the HISTORYTYPE task attribute value is NORMALEOTV.

- ABORTED

  The process terminated abnormally, for example, because of a fault or because of operator entry of a DS (Discontinue) system command. The STATUS task attribute value is TERMINATED or BADINITIATE, and the HISTORYTYPE task attribute value is DSEDV.

The COMPLETEDOK, COMPILEDOK, and ABORTED values give you the ability to determine whether a process completed successfully. For further information on determining how and why a process terminated, refer to Section 10, "Determining Process History."

Mix Display Commands

Several different system commands are available for displaying all the processes that are in a particular state. The following are the system mix commands and the process states they display:

- A (Active Mix Entries)

  Displays processes that are running normally. The STATUS task attribute value is ACTIVE.

- C (Completed Mix Entries)

  Displays processes that have recently terminated. The STATUS task attribute value is TERMINATED or BADINITIATE.

- J (Job and Task Display)

  Displays all processes that are running normally. The processes are grouped into process families in the display. The STATUS task attribute value is ACTIVE.

- LIBS (Library Task Entries)

  Displays frozen library processes. The STATUS task attribute value is FROZEN.

- MX (Mix Entries)

  Displays all in-use processes that are not frozen libraries. The STATUS task attribute value is SCHEDULED, ACTIVE, or SUSPENDED.

- S (Scheduled Mix Entries)

  Displays processes that the system is delaying initiating. The STATUS task attribute value is SCHEDULED.

- W (Waiting Mix Entries)

  Displays processes that are waiting on an event that might require operator intervention. The STATUS task attribute value is SUSPENDED.

In addition to being available through individual system commands, these displays are available as part of the ADM (Automatic Display Mode) output. For information about using ADM to track processes, refer to the *A Series System Operations Guide*.

Y (Status Interrogate) Stack States

The Y (Status Interrogate) system command returns several types of information about a process, including the mix number, usercode, program name, and stack state. In the following example, the stack state is WAITING ON AN EVENT:

```
Status of Task 3251\4441 AT 16:15:28
Program name: *OBJECT/ED ON DOCPK
Priority = 50
Origination: SA154/CANDE/3 (LSN 288)
MCS: SYSTEM/CANDE
Usercode: JASMITH
Chargecode: 6825
Stack State: Waiting on an event
```

The following are explanations of the STACK STATE values shown in Table 6–1:

- ALIVE

  The process is currently bound to a processor. That is, it is actually being processed rather than being in any type of waiting state. You can see this state displayed only on a multiprocessor system, because on a single processor system, the CONTROLLER independent runner has to take over the only processor to execute the Y command.

- FROZEN

  The process is a frozen library.

- HOLDING

  The process is waiting on interrupts. This type of waiting is described in Section 16, "Using Events."

- READY

  The process is in the ready queue and will proceed as soon as a processor is available. It is not unusual for a process to be in this state, because each central processor on the system can be executing only one process at a time, and the mix can contain many processes. The priority of a process can affect the amount of time it spends in the ready queue; refer to Section 7, "Controlling Processor Usage."

- SCHEDULED

  The system is delaying initiation of the process for any of various reasons. For information about process scheduling, refer to "Preventing Process Scheduling" later in this section.

- SELECTED

  The process is being initiated.

- TO BE CONTINUED

  This value indicates a process that has initiated a synchronous task that has not yet completed, or a process that has executed a CONTINUE statement and is waiting on its coroutine.

- WAITING ON AN EVENT

  If no RSVP line appears in the Y command output, then this value means the process is waiting for an I/O operation to be completed or for a particular event to be caused. For a general discussion of events, refer to Section 16, "Using Events."

  If an RSVP line appears in the Y command output, then the WAITING ON AN EVENT value means that the process is waiting on an event that might require operator intervention. The RSVP line lists system commands that might be helpful responses to the situation. For more information about responding to waiting entries, refer to the *A Series System Operations Guide*.

Monitoring Changes in Process Status

A process can monitor the status of its offspring by waiting on its own EXCEPTIONEVENT task attribute. This method works because, whenever the status of a process changes, the system causes the EXCEPTIONEVENT of the parent of the process.

If you design a process to wait on its own EXCEPTIONEVENT, it can resume execution and check the status of its offspring each time the EXCEPTIONEVENT is caused. For example, to wait for a task to terminate, the parent process can execute the following ALGOL statement:

```
WHILE T.STATUS GTR VALUE(TERMINATED) DO
   WAITANDRESET(MYSELF.EXCEPTIONEVENT);
```

Note that a WAITANDRESET statement is used rather than a simple WAIT statement. If a simple WAIT statement were used, then the WHILE loop would execute an infinite number of times after the first time the EXCEPTIONEVENT was caused.

The most typical reason for using such WAIT statements is to prevent critical block exits for ALGOL or COBOL74 programs that initiate asynchronous tasks. Critical block exits are discussed in Section 2, "Understanding Interprocess Relationships."

It is not necessary to take steps to prevent critical block exits in WFL. WFL automatically waits at the end of each block if any processes initiated by statements in the block are still in use. However, there can be other reasons for a WFL job to wait on the termination of an asynchronous task. For example, suppose you have an application consisting of three programs. The first two programs create files that are used as input by the third program. The following WFL job runs the first two programs in parallel, and waits for them to complete before initiating the third program:

```
100 ?BEGIN JOB;
110   TASK T1, T2, T3;
120   PROCESS RUN OBJECT/RUNEX [T1];
130   PROCESS RUN OBJECT/TADCOM [T2];
140   WHILE T1 ISNT COMPLETED OR T2 ISNT COMPLETED DO
150     WAIT;
160   PROCESS RUN OBJECT/DIALUP [T3];
170 ?END JOB
```

Note that the statement at line 150 is simply WAIT, rather than WAITANDRESET (MYSELF.EXCEPTIONEVENT) as it would be in ALGOL. This difference arises because WFL has no syntax for directly accessing the EXCEPTIONEVENT task attribute, or events in general for that matter. However, the simple WAIT in WFL has the effect of implicitly waiting on and resetting the EXCEPTIONEVENT.

WFL provides some other useful expressions for monitoring process status. You can design a WFL job to wait for a task to terminate, to wait for the task to assume a particular status, or to wait for any attribute of the task to assume a desired value. For details, refer to the discussion of the WAIT statement in the *A Series Work Flow Language (WFL) Programming Reference Manual*.

Controlling Process Status

You can accomplish some changes to process status through programmatic assignments to the STATUS task attribute. Additionally, you can prevent many instances of process scheduling or suspension through careful program design.

Terminating a Process

You can interactively terminate a process by entering the DS (Discontinue) system command. You can can design a program to terminate a process by assigning the STATUS task attribute a value of TERMINATED. The following is a WFL program that terminates a task if it becomes suspended:

```
100 ?BEGIN JOB ACCOUNTS/JOB;
110   JOBSUMMARY = SUPPRESSED;
120   CLASS = 2;
130 TASK T;
140 PROCESS RUN OBJECT/DAILY/ACCOUNTS [T];
150 WHILE NOT DONE DO
160 BEGIN
170   WAIT;
180   IF T IS STOPPED THEN
190      BEGIN
200        T(STATUS = TERMINATED);
210        MYSELF(JOBSUMMARY = UNCONDITIONAL);
220      END;
230   IF T IS COMPLETED THEN
240      DONE := TRUE;
250 END;
260 ?END JOB
```

The presumption behind this WFL program is that OBJECT/DAILY/ACCOUNTS is a program that does not normally become suspended at any point in its run. If this particular program becomes suspended, it means that something has gone wrong and it is something that an operator cannot easily fix. Further, it is assumed that job queue 2, which this job is initiated from, has a mix limit of 1. Thus, if OBJECT/DAILY/ACCOUNTS becomes suspended, it is impossible for any more WFL jobs to be initiated from that job queue until an operator notices the situation and discontinues the process.

The WHILE statement at lines 150 to 250 is included to prevent this job from ever uselessly blocking up the job queue. Within the WHILE statement, the WAIT statement at line 170 causes the WFL job to wait until its own EXCEPTIONEVENT is caused. The system automatically causes the job's EXCEPTIONEVENT when the STATUS value of any of the job's offspring changes. When the status value of the offspring changes, the statement at line 180 uses the task state expression to determine if OBJECT/DAILY/ACCOUNTS is suspended; if so, then the statement at line 200 assigns a STATUS of TERMINATED to discontinue OBJECT/DAILY/ACCOUNTS. The statement at line 210 causes printing of the job summary. For information about job summaries, refer to Section 10, "Determining Process History."

The statement at lines 230 to 240 causes the loop to be exited when OBJECT/DAILY/ACCOUNTS terminates (whether it terminated normally or was discontinued).

Thawing a Library

Thawing a library is the act of changing a permanent or control library into a temporary library. A temporary library automatically resumes execution as soon as it has no more users. By contrast, a permanent library remains frozen indefinitely, and a control library remains frozen until it exits the control procedure. Thawing a library is thus a first step toward removing a permanent or control library process from usage (for example, because you want a newer version of the library program to be used). Thawing the library is less drastic than discontinuing the library process with a DS (Discontinue) system command or a STATUS assignment of TERMINATED.

You can design a program to thaw a library process by assigning either of two values to the STATUS task attribute: ACTIVE or GOINGAWAY. Table 6–2 summarizes the differences in the effects of these two assignments.

Table 6–2. Effects of GOINGAWAY and ACTIVE Assignments

Effects	GOINGAWAY Assignment	ACTIVE Assignment
Time Execution Resumes	When there are no more users	When there are no more users
New Users of Shared Libraries	Are linked to a new invocation of the latest version of the library object code file	Are linked to the existing library process
STATUS Task Attribute	ACTIVE	FROZEN
WFL Task State	ACTIVE, INUSE	ACTIVE, INUSE
Mix Commands	A, J, MX	LIBS
Y Stack State	WAITING ON AN EVENT	FROZEN

The key difference between GOINGAWAY and ACTIVE assignments is that the GOINGAWAY assignment prevents any additional user processes from linking to the library process. For libraries that are shared by many users, this can make a big difference in how long the library process takes to resume execution. If you use an assignment of ACTIVE instead, new processes can continue linking to the (newly temporary) library, with the result that the library remains frozen indefinitely.

Note that neither the GOINGAWAY assignment nor the ACTIVE assignment actually changes the STATUS task attribute to the requested value. After a GOINGAWAY assignment, the STATUS value is ACTIVE. GOINGAWAY is therefore never returned as a value when a process reads the STATUS task attribute. By contrast, after an ACTIVE assignment, the STATUS remains FROZEN until there are no more users of the library. Then the library STATUS changes to ACTIVE and the process resumes execution.

You can thaw a library interactively with the THAW (Thaw Frozen Library) system command. This command has the same effect as assigning a value of ACTIVE to the STATUS task attribute.

Suspending and Resuming Processes

You can interactively suspend execution of a process with the ST (Stop) system command, and resume execution of the process with an OK (Reactivate) system command. You can design a program to achieve the same result by assigning the STATUS task attribute a value of SUSPENDED or ACTIVE. However, note that if the process is suspended by the system, the OK command or ACTIVE assignment frequently is not enough to resolve the cause of the suspension. In this case, the process is suspended again by the system without progressing any further in its execution.

One reason to suspend a process programmatically is for testing. For example, you can add a statement in a program that causes it to be suspended at a certain point where a problem has been occurring. Then you can force a memory dump or program dump through the DUMP (Dump Memory) system command, with the knowledge that the dump will reflect the state of the process at a selected point in its execution.

Parallel processes can also use assignments of SUSPENDED or ACTIVE as a means of coordinating their activities. However, A Series systems provide special variables called *events* that are better suited to coordinating parallel processes. For details, refer to Section 16, "Using Events."

Preparing a Task Variable for Reuse

As was stated earlier in this section, a task variable can be in use by only one process at a time. However, it is possible to reuse a task variable so long as the first process terminates before the task variable is used in another process initiation statement. The side effects that can result from such reuse of task variables are discussed in the *A Series Task Attributes Programming Reference Manual*.

Suffice it to say here that the side effects involve task attribute values that are retained from one use of the task variable to the next. To restore all the task attributes of the task variable to their default values, you can assign the STATUS task attribute a value of NEVERUSED. In WFL, you also have the option of using an INITIALIZE statement, which has the same effect as the STATUS assignment.

Preventing Process Scheduling

A process is said to be SCHEDULED when it has been submitted for initiation, but the system is delaying initiation of the process. Scheduling can have any of several causes, the most common of which is a lack of available memory on the system. If the system estimates that a particular process will require more memory for efficient execution than is currently available, the system places the process in a scheduled state until more memory becomes available.

There are two methods you can use to help prevent a process from being scheduled because of a shortage of available memory:

- You can override the system's memory estimate for the process through assignments to the CORE and STACKSIZE task attributes. For details, refer to Section 8, "Controlling Process Memory Usage."
- You can assign the process with control program status by marking its object code file with the *MP <file title> + CONTROL* form of the MP (Mark Program) system command. For information about control program status, refer to Section 7, "Controlling Processor Usage."

For further information about the causes of scheduling, refer to the process scheduling discussion in the *A Series System Administration Guide*.

Preventing Process Suspension

Many cases where a process becomes suspended by the system can be prevented with a little planning. To be sure, there are situations that cannot be anticipated that might make it necessary for the system to suspend a process. One such example is an extreme shortage of available memory. However, a good many cases of process suspension result from such causes as failed attempts to open files or ACCEPT statements that require immediate input from the operator.

In many of these cases, it might be preferable to allow the process to become suspended. The advantage to this is that the process appears in the W (Waiting Mix Entries) display with a message explaining why it is suspended. This is desirable if the situation is one that an operator can easily remedy. A common example of such a situation is one where a process is attempting to open a tape file. When the process appears in the W display, the operator is prompted to mount the appropriate tape.

However, if you are interested in automating operations at your site as much as possible, then you might find the techniques discussed in the following subsections to be useful. For further information about the file attributes mentioned in the following discussions, refer to the *A Series File Attributes Programming Reference Manual*.

Checking File Residence

You can design a program to read the AVAILABLE or RESIDENT attributes of a file before attempting to open the file. RESIDENT returns a value of TRUE or FALSE to indicate whether the file is available. AVAILABLE returns a numeric value indicating whether the file can be opened, and if not, why not.

If the file is available, the program can execute an OPEN statement. If the file is not available, the program can skip the OPEN statement and take whatever recovery actions are deemed appropriate by the programmer.

Using AUTORESTORE for Disk Files

You can use the AUTORESTORE task attribute to request that the system automatically attempt to restore any missing disk file requested by a process. Automatic restoration can prevent the process from becoming suspended with a NO FILE condition. Refer to the discussion of disk file usage in Section 9, "Controlling Process I/O Usage."

Using a Serial Number for Tape Files

When a process opens a tape file, the process can become suspended even if the requested tape is already mounted on an available tape drive. The suspension occurs if the process does not give the system sufficient information to identify the particular tape to search for the file. If the process becomes suspended, the operator can use system commands such as IL (Ignore Label) or OU (Output Unit) to specify the correct tape drive so process execution can resume.

Nothing you can do removes the need for someone to mount a tape containing the tape file on a tape drive. However, you can set things up so that an operator does not have to take any further action beyond mounting the tape and later removing it. You can write the program to assign a value to the SERIALNO attribute of the tape file. When the process attempts to open the file, the system checks to see whether a tape with that SERIALNO value is mounted on any of the available drives. If the tape is mounted, then the system looks for the requested file on that tape, without ever suspending the process.

SERIALNO is also available as an option in the WFL *COPY* statement. The following is an example:

```
COPY (JASMITH)= FROM SYSPK(PACK) TO LABCON(TAPE,SERIALNO="LABIN");
```

This example creates a tape named LABCON with a SERIALNO value of "LABIN". The SERIALNO value can include letters as well as digits. Any letters in the string must be capitalized.

Sometimes you want the program to write output to a tape, but you do not really care which tape, as long as it goes to a tape that is not otherwise in use. In this case, you can leave the SERIALNO value empty and set the FILEUSE file attribute to OUT. If the SERIALNUMBER operating system option is not set, then the system writes the file to any scratch tape that is mounted and not in use. An operator can set or reset the SERIALNUMBER option with the OP (Options) system command.

Using UNITNO and OMITTEDEOF for Unlabeled Tape Files

By default, any tapes created by an A Series system have ANSI-standard tape labels. These tape labels store identification information for the tape. However, you might have occasion at some time to use a tape on an A Series system that was created by a different type of computer system. If the different computer system did not create an ANSI-standard tape label, you must design your program to read the tape as an

unlabeled tape. You can also use this technique to enable a program to read a tape whose label has become corrupted.

When a process attempts to open an unlabeled tape, the process typically becomes suspended until an operator enters a UL (Unlabeled) system command. This command specifies the tape drive to use for the file. You can prevent the need for the operator to enter this command. However, you must use a different technique than was previously described for labeled tapes. The SERIALNO attribute has no meaning for unlabeled tapes.

Instead, if you know the physical unit number of the drive where the correct tape will be mounted, you can design the program to assign the physical unit number of that tape drive to the UNITNO file attribute. A file open operation then opens any tape that happens to be on the specified tape drive. This method should not be used unless you can ensure that the correct tape will be mounted on the tape drive when the program runs. Note also that access to unlabeled tapes and to the UNITNO file attribute might be restricted on systems running InfoGuard security enhancement software at the S1 or S2 level; refer to the *A Series Security Administration Guide* for details.

A process can also be suspended when it reaches the end of an unlabeled tape file. This happens because, depending on the circumstances, a tape mark can indicate the end of the file or simply the end of a reel. If the LABEL file attribute value is OMITTED, the tape mark is interpreted to mean that the file continues on another tape reel. The process becomes suspended until the operator enters a UL command (to specify where the next reel is located) or an FR (Final Reel) system command.

If you know in advance that the unlabeled tape file will be confined to a single reel, you can prevent the process from suspending at the end of the file. To do this, you must declare the file with a LABEL value of OMITTEDEOF. In this case, when the process reads to the end of the file, the system returns an end-of-file condition on the read operation. The process can check the result of the read operation and take appropriate action. This method saves the operator the trouble of entering the FR command.

Using the AUTORM Option

A process can become suspended if it attempts to enter a file into the disk directory and a file of the same title already exists. The system displays a "DUP LIBRARY" RSVP message for the process. The process does not proceed any further until an operator enters an RM (Remove) system command. The RM command causes the system to remove the existing file. You can save the operator from having to enter an RM command by setting the AUTORM option. AUTORM can be set for a process through assignments to the OPTION task attribute, or for the whole system through the OP (Options) system command. The AUTORM option causes the system to automatically remove any old duplicate files that a process encounters. For further information about disk directories and the AUTORM option, refer to "Entering a File in the Directory" in Section 19, "Using Shared Files."

Using the ORGUNIT Value for ODT Files

A process can become suspended when it executes a statement that opens an ODT file. For information about how to prevent this process suspension from occurring, refer to the discussion of ODT terminal communications in the Section 3, "Tasking from Interactive Sources."

Using Conditional ACCEPT Statements

A process can become suspended when it executes an ACCEPT statement to prompt the operator for input. For information about how to prevent this suspension from occurring, refer to the discussions of the conditional ACCEPT statement and the ACCEPTEVENT task attribute in Section 3, "Tasking from Interactive Sources."

Section 7
Controlling Processor Usage

You can control two aspects of the processor usage for a process: process priority and total processor usage. In addition, you can monitor the processor usage of a particular process to gain an understanding of the resource usage patterns of the process.

Controlling Process Priority

A Series systems are designed to efficiently execute large numbers of processes simultaneously. However, each system incorporates a limited number of processors, including central processors, I/O processors, and data link processors (DLPs). Each system also has a finite amount of main memory. On a heavily used system, all the processes in the mix are competing for the use of these system resources.

However, it may be that not all these processes are equally urgent from the user's point of view. A Series systems provide the concept of *priority* to allow you to specify which processes should receive preference in the competition for system resources.

The primary effect of process priority occurs in cases where more than one process is ready to use a central processor. Each central processor executes only one process at a time, but divides its time among all the processes on the system. The system maintains a list, called the *ready queue*, of all processes that are waiting for a processor, arranged in priority order.

A processor continues executing a particular process until one of three things happens: the process reaches a natural stopping point (for example, because it is waiting for an I/O to complete), a higher-priority process appears in the ready queue, or the process exceeds its time slice and a process of equal priority is present in the ready queue. The processor then retrieves the higher-priority process from the ready queue and begins executing it.

The priority of a process is determined by several factors, only some of which can be controlled by the user. For example, some system software processes have a higher priority than can be assigned to an ordinary application process. For a complete overview of factors affecting process priority, refer to the *A Series System Administration Guide*.

One aspect of priority that you can control, within certain limits, is the PRIORITY task attribute value. The PRIORITY task attribute has a range of values from 0 to 99, with the higher values indicating higher priority. The default value is 50. You can assign a PRIORITY value to a process anytime before initiation, either through task equations or assignments to a task variable. Once a process is initiated, any programmatic assignments to the PRIORITY task attribute change the task attribute value without affecting the actual priority at which the process executes. The new PRIORITY task attribute value is returned when the task attribute is read, and displayed in the output of various system commands.

The only way to effectively change the priority of an in-use process is with the PR (Priority) system command. This command changes the PRIORITY task attribute value and also causes the system to enforce the new priority value.

One point to bear in mind about this attribute is that its effects are absolute rather than proportional. That is to say, the system always gives the processor to the highest-priority process that is ready to use it. A PRIORITY value of 51 gives as much advantage over a PRIORITY of 50 as a PRIORITY value of 99 does. If the process with the PRIORITY of 51 is very processor-intensive, it could prevent the process with PRIORITY 50 from receiving any processor time at all. For this reason, you should be cautious about raising the PRIORITY value of a processor-intensive process.

On the other hand, it is sometimes helpful and appropriate to raise the priority of interactive processes. An interactive process is one that is largely driven by input from a user at a terminal. Such a process typically spends most of its time waiting for the user to enter commands. Once the user does enter a command, the user typically has to wait for a response before being able to accomplish any further useful work. If the processor usage of the process is small and occasional, you can improve response time by raising the priority with relatively little impact on overall system performance.

For information about how to determine whether a process is processor-intensive, refer to "Understanding Processor Usage Accounting" later in this section.

The system administrator can place some constraints on the values you are able to assign to the PRIORITY task attribute. For example, the administrator can assign a PRIORITY limit to a job queue. If you write a WFL job that is initiated from that job queue, the job cannot request a PRIORITY value higher than the job queue PRIORITY limit. Similarly, the system administrator can assign a value to the PRIORITY attribute of your usercode. CANDE and MARC read the PRIORITY attribute of your usercode when you log on. When you initiate a task from a CANDE or MARC session, CANDE and MARC do not allow you to assign the PRIORITY task attribute a value higher than your PRIORITY usercode attribute.

Aside from the PRIORITY task attribute, the major feature you can use to manipulate process priority is the *MP <file title> + CONTROL* form of the MP (Mark Program) system command. This option marks an object code file as a control program. Thereafter, whenever that program is initiated, it runs in the same priority category that message control systems (MCSs) do. This category gives higher priority than WFL jobs or application programs have, but lower priority than invisible independent runners.

The system also places WFL jobs in a special priority category. WFL jobs receive higher priority than all application programs, but lower priority than control programs, MCSs, and invisible independent runners.

The system uses the PRIORITY task attribute only when comparing processes that are in the same priority class. Thus, a WFL job running with a PRIORITY value of 1 still has a higher priority than an ordinary process with a PRIORITY of 99.

An additional effect of control program status is that it prevents the system from *scheduling* a process (that is, delaying initiation of the process) when there is a shortage of available memory. If you mark too many programs with control program status, the result can be that system memory becomes overloaded, with a resulting adverse effect on system performance. Therefore, you should use caution in marking programs with control program status.

The system places WFL jobs in the high priority class because their only purpose in most cases is to initiate tasks; the sooner the job initiates each task, the sooner the system can evaluate the priority of each task on its own merits. However, it would not be possible to rewrite a typical application in WFL to take advantage of its priority. WFL is specialized for tasking functions and has no ability to read from or write to files.

Because the PRIORITY task attribute and control program assignments have a potential to affect overall system performance, you should generally consult with the administrator of your system before raising the priority of any particular process.

Limiting Processor Usage

You can use the MAXPROCTIME task attribute to set a limit on the amount of processor time that a process can use. The accumulated processor time for a process is stored in the ACCUMPROCTIME task attribute. When ACCUMPROCTIME reaches a value equal to that of MAXPROCTIME, the system discontinues the process and displays the error message EXC PROC TIME.

The main use of the MAXPROCTIME task attribute is to ensure that WFL jobs are placed in the proper job queues. For example, suppose there is a high-priority job queue that is intended for short jobs. The system administrator can use the PROCESSTIME job queue attribute to provide default and limiting values for the MAXPROCTIME task attribute of all WFL jobs that use the job queue. If you submit an extremely processor-intensive job through that job queue, the system discontinues the job when it exceeds the MAXPROCTIME value. This gives you an incentive to resubmit the job through a different job queue. For an introduction to the subject of job queues, refer to the discussion of WFL in Section 4, "Tasking from Programming Languages."

Understanding Processor Usage Accounting

Programs vary a lot in terms of their patterns of processor usage. Understanding the processor usage of a program can help you to decide the priority at which it should run. It can also help you to diagnose inefficiencies in program design or problems in overall system performance.

The system divides the processor usage of a process into several categories, which can be displayed through system commands, examined through task attributes, or read in the system log.

The system command that displays processor usage information is the TI (Times) command. The following is an example of the output:

```
5825 TI

     TIMES FOR 5825
       PROCESS   = ØØ:ØØ:37 LIMIT Ø:Ø1:2Ø
       IO        = ØØ:ØØ:Ø1 LIMIT Ø:Ø2:4Ø
       READYQ    = ØØ:ØØ:56
       INITPBIT  = ØØ:ØØ:Ø6 3217 OPERATIONS
       OTHERPBIT = ØØ:ØØ:Ø2 1521 OPERATIONS
       ELAPSED   = ØØ:11:4Ø
```

In the TI command output, all the times are expressed in a format of <hours>:<minutes>:<seconds>. The following are the meanings of these fields in the TI command output:

- PROCESS

  The accumulated processor usage of the process, with the exception of the process time spent on presence-bit operations. (See the following descriptions of INITPBIT and OTHERPBIT.) The LIMIT time, if displayed, corresponds to the MAXPROCTIME task attribute value.

- IO

  The accumulated I/O usage for the process. The LIMIT time, if displayed, corresponds to the MAXIOTIME task attribute value.

- READYQ

  The accumulated ready queue time for the process. Ready queue time is the time spent waiting for the processor to become available. If this value is excessive, it indicates either that the processor is overloaded or that other higher priority processes are dominating the processor.

- INITPBIT

  The amount of processor time spent on initial presence-bit operations. These are operations that create arrays, files, and code segments for this process. This value is followed by a count of the number of presence-bit operations.

  If the value of INITPBIT is high compared to the value of PROCESS, this can be a symptom of poor program structure. For example, if a large local array is declared in a procedure that is entered repeatedly, then much processor time is spent recreating the array each time the procedure is entered, thus resulting in a high INITPBIT value. You can prevent this problem by declaring the array globally to the procedure, or by declaring the array with an OWN clause (in ALGOL programs only).

- OTHERPBIT

  The amount of processor time spent on noninitial presence-bit operations for this process. Noninitial presence bit operations read arrays and code segments back into main memory after they have been overlaid. The value of OTHERPBIT can vary widely for different runs of the same program, depending on the memory demands that are made by other active processes. If this value is very high, it might indicate that memory is overloaded and the system is thrashing.

- ELAPSED

  The amount of real time that has passed since the process was initiated. This value is stored in a different form in the ELAPSEDTIME task attribute.

  The value of ELAPSED can be greater than or less than the sum of the other values listed in the TI display. The ELAPSED value can be greater because it includes time spent waiting on events, and this waiting time is not displayed separately. The ELAPSED value can be less because, in some cases, a process might be using the processor and performing one or more I/O operations at the same time.

Most of the resource usage information that can be displayed for a process can also be interrogated through task attributes.

The ACCUMPROCTIME task attribute returns the accumulated processor time. The value does not include processor time spent on presence-bit operations. The INITPBITTIME, INITPBITCOUNT, OTHERPBITTIME, and OTHERPBITCOUNT task attributes return the times and counts for presence-bit operations. The ACCUMIOTIME task attribute returns the accumulated I/O time for the process. The ELAPSEDTIME task attribute returns the total elapsed time.

The values these task attributes return are expressed in units of 2.4 microseconds, except if the attributes are read from WFL, which expresses the values in units of seconds.

The system log (SUMLOG) records several categories of processor usage for each process. This information includes the processor time, I/O time, ready queue time, and p-bit times and counts. This information is stored in the Major Type 1, Minor Type 2 (EOJ) and Minor Type 4 (EOT) log entries. For a description of these log entry types, refer to the *A Series System Software Support Reference Manual.*

Section 8
Controlling Process Memory Usage

Process execution takes place in a memory environment that is shared with all the other processes in the mix. Understanding that environment can help you to improve process performance and prevent a process from impairing overall system performance.

This section is aimed at programmers, and concentrates on the aspects of process memory usage that can be affected by task attributes and object code file location.

Understanding Process Memory Usage

A process consists of several distinct components, some of which reside in main memory and some of which can reside in virtual memory. The system uses presence-bit operations to create or re-create some of the process components in main memory.

Main Memory and Virtual Memory

The effective memory capacity of an A Series system consists of the following two components:

- Main memory

  This is the total amount of memory that is physically present.

- Virtual memory

  This is an additional amount of memory whose existence is simulated by temporarily copying segments of main memory out to disk. The use of virtual memory enables the system to handle more processes than can fit into main memory at the same time.

To facilitate memory management, the system classifies each of the segments of main memory into one of the following three categories:

- Available memory

  This is memory that is not assigned to an in-use process. The system is free to allocate this memory as the need arises.

- Overlayable memory

  This is memory that is assigned to in-use processes, but which can nevertheless be overwritten if necessary. For data segments, the system must copy the data to a different location in main memory or to an overlay disk file before reusing the memory segment. For code segments, the system can simply overwrite the code segment with other code or data. The system can read the code segment back in from the object code file the next time it is needed.

- Save memory

  Save memory consists of structures that, for performance reasons, must be kept in main memory at all times. The system never copies these segments out to disk, nor does it move them around in main memory except for stack stretches. (Refer to "Preventing Stack Stretches" later in this section.)

When the processes in the mix require far more memory than exists as main memory, the processor is forced to spend a lot of time performing overlays. When the time spent performing overlays begins to significantly impair system performance, the situation is called *thrashing*.

Process Components

Every running process includes the following basic structures in memory:

- Process information block (PIB)

  This structure contains process control information visible only to the operating system. The PIB also contains a reference to the TAB.

- Task attribute block (TAB)

  This structure stores the task variable for the process and includes the values of all task attributes. In addition to the TAB of the process, the system creates a separate TAB for each task variable the process declares. Thus, reusing a task variable can slightly reduce the memory usage of a process. For cautions related to task variable reuse, refer to the *A Series Task Attributes Programming Reference Manual*.

- Process stack

  This structure includes storage areas, or descriptors pointing to outside storage areas, for all variables declared by the process. The top of the process stack also serves as a working storage area that the processor can use when evaluating expressions. For information about estimating and limiting process stack size, refer to "Controlling Process Scheduling," "Preventing Stack Stretches," "Protecting against Looping Processes," and "Restricting Save Memory Usage" later in this section.

- Code segment dictionary

  This structure includes descriptors pointing to the locations of

  - The code segments used by the process.
  - Constant data used by the process, such as value arrays and translate tables.
  - Sequence numbers for all the code segments, if the program was compiled with the LINEINFO compiler options set. (For information about LINEINFO, refer to Section 10, "Determining Process History.")

  For further information about code segment dictionaries, refer to "Controlling Code Segment Dictionary Sharing" later in this section.

Presence-Bit Operations

When the processor constructs an array, a logical file, or a code segment for a process in memory, this action is referred to as a *presence-bit operation.* An initial presence-bit operation is one that creates an array or a code segment because the related procedure has just been invoked. A noninitial presence-bit operation is one that copies an array or a code segment back into main memory from disk.

The number of initial presence-bit operations performed by a process, and the processor time they take, are relatively stable from one run of a program to the next (provided that each run results in the same sequence of procedure entrances). However, the number of noninitial presence-bit operations performed by a process depends to a large extent on how much memory is being used by all the other processes in the mix. When memory is crowded, more noninitial presence-bit operations are performed.

You can monitor the number of presence-bit operations for a process, and the processor time spent on them, by using the TI (Times) system command, by interrogating task attributes, or by reading system log entries. Refer to the discussion of processor usage accounting in Section 7, "Controlling Processor Usage."

Controlling Code Segment Dictionary Sharing

The system generally causes processes to share the same code segment dictionary if the processes are executions of the same program. This technique reduces total memory usage and thus reduces the system overhead for memory management. The result is that all the processes in the mix are able to run more quickly.

There are a few situations in which the system does not use the same code segment dictionary for processes that are executing the same program. Understanding these situations can help you to conserve memory and control process privileges.

To decide whether two processes are executions of the same program, the system compares the object code file title for each process. Suppose you have one copy of OBJECT/PROG on the family SYSPK, and another copy on a family called DOCPK. In this case, the family part of the object code file title is different. The system therefore regards these as two different programs. If people are using both programs simultaneously, the system has to create two separate code segment dictionaries. You can eliminate this duplication, and thus reduce system overhead, by placing a single object code file in a central location where all the users have access to it.

Even if two processes have the same object code file title, the system still assigns them different code segment dictionaries in the following cases:

- If either of the processes is running in Test and Debug System (TADS) mode. A process runs in this mode if you compile the program with the TADS compiler option set and run the program with the TADS task attribute set. TADS mode gives ALGOL, C, COBOL74, COBOL85, or FORTRAN77 processes access to the TADS facility for debugging programs. You can prevent unnecessary duplication of code segment dictionaries by using TADS mode only for the rare cases when you are actually doing debugging.
- If the object code file is overwritten. An object code file can be overwritten if, for example, you recompile the program or use a COPY statement to replace it with a different program having the same title. If the object code file of a running process is overwritten, the system retains the old object code file as a temporary file. The running process continues to use its code segment dictionary and the old object code file. However, any new processes that are initiated with the same object code file title receive a code segment dictionary reflecting the new object code file. The main point to bear in mind is that updating or removing an object code file has no effect on processes that are already running.

The MP (Mark Program) system command can be used to assign various options to object code files. These options are described in Section 5, "Establishing Process Identity and Privileges." Be aware that these options do not affect new instances of the program if an old version of the code segment dictionary is lingering in memory. The code segment dictionary remains in memory as long as any process is using it. Further, for programs marked with the RP (Resident Program) system command, the code segment dictionary remains in memory until the next system halt/load.

If you assign a new security status to a program, and the program is frequently used or has resident program status, you might consider the following method of updating the code segment dictionary. You can copy the object code file over itself with a COPY statement such as the following:

```
COPY (JASMITH)OBJECT/PROG AS (JASMITH)OBJECT/PROG
```

Subsequent instances of the program will reflect the new privilege status.

Controlling Process Scheduling

A process is said to be scheduled when it has been submitted for initiation, but the system is delaying initiation of the process. Scheduling can occur for any of a number of reasons, most of which are not preventable by the programmer. For an explanation of these reasons, refer to the discussion of process scheduling in the *A Series System Administration Guide.*

One type of process scheduling that you can prevent, to some extent, is scheduling due to a lack of available memory. The system performs this type of scheduling if it estimates that a particular process requires more memory for efficient execution than is currently

available. The system places the process in a scheduled state until more memory becomes available.

The initial memory estimate for a process is created by the compiler and stored in the object code file. The memory estimate is an estimate of the average amount of memory that must be available for the process to run efficiently (that is, without excessive overlays). This ideal amount of memory is referred to as the *working set* of the process.

Each time the object code file is executed, the system writes an updated memory estimate into the object code file. The updated estimate is based on the average of the existing estimate and the memory usage during the current run. The effect is to gradually refine and improve the accuracy of the memory estimate each time the object code file is run.

The memory estimate for a process consists of two separate statistics: the estimated process stack size, and the estimated memory usage for data and code segments. You can override the process stack size estimate through an assignment to the STACKSIZE task attribute. You can override the data and code estimate through an assignment to the CORE task attribute. By assigning large or small values to the STACKSIZE and CORE attributes, you can make it more or less likely that the system will schedule a process when it is submitted for initiation.

It is rarely necessary or desirable for you to make assignments to the STACKSIZE and CORE task attributes. It is true, for example, that you can help ensure that a process will not be scheduled by setting STACKSIZE and CORE to artificially low values. However, doing so could cause a system to begin thrashing, with the result that system performance could dramatically worsen.

The following are situations in which it might make sense to assign STACKSIZE and CORE values:

- When initiating a program that is newly compiled. The memory estimate in such an object code file has not been refined through repeated use.
- When initiating a program that is stored on a read-only disk. Many types of disk drives have a switch that enables an operator to put the disk drive in read-only mode. If an object code file is stored on a read-only disk, the system is not able to update the memory estimate in the object code file after each run.
- When initiating an program whose memory usage varies widely from one run to the next. This can be the case if the memory usage depends on the type and quantity of the data passed to the program for processing.

Even in these situations, there is no point in your assigning a CORE or STACKSIZE value unless you have some information about what the working set of the program really is. You can get some general idea of the memory usage of a program by running it and examining statistics with the LOGANALYZER utility. You can use the LOGANALYZER *MIX* option to return log entries for a particular process. In the Major Type 2, Minor Types 4 and 5 (EOJ and EOT) log entries, you can find figures for the average memory usage of a process.

Processes" earlier in this section. This might also happen if the process uses large numbers of arrays and files.

You can prevent a process from exceeding a planned level of save memory usage by assigning a value to the SAVEMEMORYLIMIT task attribute. If the save memory usage of the process exceeds the limit set by this attribute, the system discontinues the process and displays the message "USER SAVE MEMORY LIMIT EXCEEDED".

If a process is discontinued with the "USER SAVE MEMORY LIMIT EXCEEDED" error, you should check to see if it was running normally or looping. If it was running normally, you can consider program design measures to reduce the save memory usage. Alternatively, you can raise the SAVEMEMORYLIMIT value and plan to run the process at a time when the system is not very busy.

The system administrator can place some limits on the SAVEMEMORYLIMIT value your processes can have. For example, the system administrator can assign a SAVEMEMORYLIMIT value to your usercode. This value becomes the maximum SAVEMEMORYLIMIT value for all processes initiated with your usercode. If you assign a different SAVEMEMORYLIMIT value to a process, the system uses the lower of your SAVEMEMORYLIMIT assignment and the usercode SAVEMEMORYLIMIT assignment.

You might also find that the system administrator has assigned a SAVEMEMORYLIMIT value to a job queue you use for your WFL jobs. If the SAVEMEMORYLIMIT is assigned as a job queue default, you can override it with a different SAVEMEMORYLIMIT assignment in the job header of your WFL job. If the SAVEMEMORYLIMIT is assigned as a job-queue limit, your WFL job is rejected from the job queue if the job header includes a higher SAVEMEMORYLIMIT assignment. For more information about job queues, refer to the discussion of WFL in Section 4, "Tasking from Programming Languages."

Section 9
Controlling Process I/O Usage

The I/O activity of a process is primarily determined by various I/O statements that the process executes. These include statements for reading from, writing to, opening, and closing files. For an overview of I/O features available in A Series programming languages, refer to the *A Series I/O Subsystem Programming Guide.*

There are also a number of task attributes that affect various global aspects of process I/O activity. For example, you can use task attributes to establish default locations for files used by a process, or to specify defaults for handling printer output produced by a process. This section introduces the functions of task attributes that affect process I/O activity and some related system commands.

Establishing the Default Usercode for Files

One of the effects of the USERCODE task attribute is to supply a default usercode for all files used by a process. For example, suppose a process runs with a USERCODE value of FERMAT. Suppose also that this process attempts to open a file with a TITLE file attribute of"INPUT/DATA ON DBFAM". In this case

- If the NEWFILE file attribute is TRUE, the system creates the file under usercode FERMAT and changes the TITLE file attribute to "(FERMAT)INPUT/DATA ON DBFAM".
- If the NEWFILE file attribute is FALSE, the system searches for the file first under the title "(FERMAT)INPUT/DATA ON DBFAM". If no file of that title exists, the system searches for the file under the title "*INPUT/DATA ON DBFAM".

A process can override the default behavior by assigning a usercode as part of the TITLE file attribute before attempting to open the file. For example, a process could assign TITLE the value (LUANN)INPUT/DATA ON DBFAM. In this case, the system searches for the file only under usercode LUANN.

Modifying File Attributes

File attributes are entities that describe the properties of files on A Series systems. For example, file attributes specify the title of the file and the physical device type on which it resides (such as disk or tape). Programs can specify attributes for a file in the file declaration. Programs can also add to or change file attributes with file attribute assignment statements later in the program.

After you have written and compiled a program, you might later find that you would like the program to start using a different set of file attributes than were originally specified in the program. One method for doing this is to rewrite and recompile the program.

This method can be time consuming for the programmer, and can make heavy use of system resources such as processor time and memory.

Alternatively, you can modify the file attributes used by a program through constructs called *file equations*. For example, suppose a program uses a file called IN and another file called OUT. In a CANDE *RUN* command, you could use file equations to specify different titles for these files in the RUN statement that initiates the program. The following is an example:

```
RUN REPORT1;FILE IN = (HKANE)INDATA, OUT = (HKANE)OUTDATA
```

File equations thus enable you to modify the file attributes used by a program without having to rewrite or recompile the program. However, in order to use a file equation you first have to know the internal name of the file. The internal name of the file is determined by the value of the INTNAME file attribute. If the program does not specify a value for INTNAME, then INTNAME defaults to the value of the file identifier used for the file in the program. You can determine the internal name of a file by looking at the file declaration in the program source file. Thus, either of the following ALGOL declarations creates a file with an internal name of SOURCE:

```
FILE CUSTDATA(INTNAME = "SOURCE.");
FILE SOURCE;
```

The syntax for file equations in CANDE, MARC, and WFL is almost identical. For example, to change the device kind of the file with the internal name of SOURCE, you can append the following to a RUN statement submitted through any of these sources:

```
FILE SOURCE(KIND = REMOTE);
```

The flexibility provided by file equations can be so convenient that programmers sometimes design a program with the intention that the user will use file equations. For example, in the documentation for various A Series compilers and utilities, you can find descriptions of the internal names of files used by these compilers and utilities. These internal names are documented so that you can use them in file equations.

Note that the same file attribute can be assigned different values by file declarations, file attribute assignment statements, and file equations. In these cases, the values assigned through file equations override those specified in the file declaration. File equations are in turn overridden by any conflicting file attribute assignment statements executed by the program. A programmer can prevent file equations from having effect simply by specifying file attributes through file attribute assignment statements rather than through attribute assignments in the file declaration.

When you specify file equations for a process, the system stores the equations in the FILECARDS task attribute of the process. For further information about FILECARDS, refer to the *A Series Task Attributes Programming Reference Manual*.

One of the file attributes that it is frequently useful to change at run time is the FAMILYNAME file attribute. You can save yourself the trouble of including

FAMILYNAME equations for each file in the program by using the FAMILY task attribute instead. Refer to "Specifying Family Substitution" later in this section. Also, you can establish default values for the file attributes related to printing by using the PRINTDEFAULTS task attribute, as described under "Programmatic Control Over Printing" later in this section.

You might find occasionally that you initiated a process and forgot to specify the correct file equations. The system suspends the process if both the following conditions are true:

- The process attempts a open operation with the WAIT option specified or with no specific open option.
- The process is unable to open the specified file because of a missing or incorrect file attribute value.

You cannot use file equations to remedy this problem, because file equations must be specified at process initiation. Instead, you can use the FA system command to supply the needed file attribute values. For example, suppose a process is suspended because it tried to open a file SOURCE with KIND = TAPE, and the file is a disk file. The Y system command output looks like this:

```
STATUS OF JOB 5692\5692 AT 16:34:45
CLASS = 2
PRIORITY = 50
ORIGINATION: SB154/CANDE/3 (LSN 320)
MCS: SYSTEM/CANDE
USERCODE: JASMITH
CHARGECODE: MANUFACTURING
STACK STATE: WAITING ON AN EVENT
PROGRAM NAME: WFL/TEST
RSVP: NO FILE SOURCE (MT) #1
REPLY: FA,UL,IL,OK,DS
```

Note that the name SOURCE, which appears on the RSVP line, is the file title rather than the internal name. However, it does not matter if you do not know the internal name in this case. When you specify file attribute assignments in an FA command, the system automatically applies the assignments to the file the process is trying to open. The following FA command enables the process to open the file and resume running normally:

```
5692 FA KIND = DISK
```

For detailed descriptions of all the file attributes available on A Series systems, refer to the *A Series File Attributes Programming Reference Manual*.

Controlling Disk File Usage

You can use task attributes to take advantage of some of the unique features of A Series disk storage, including the concept of disk families, disk directories, and the disk resource control system.

Specifying Family Substitution

Disk *families* are groups of disk units that are labeled with a common name and treated as a logical unit. Disk families are defined through system configuration and system commands. Once a family has been defined, a program can use the FAMILYNAME file attribute to specify that a file is located on that family.

It is quite often the case that all the input and output files used by a process are located on one, or possibly two, disk families. Now, suppose that you include FAMILYNAME file attribute assignments in the program for each file used by the program. The system administrator might later decide to change the name of a disk family, or might ask you to place your files on a different family. Further, you might need to run your program on a different host system, where no family of the original name exists. For any of these reasons, it might become desirable for the program to look for its files on a different family than is specified in the program code.

The simplest way to make a process use a different family is by assigning the FAMILY task attribute. This task attribute specifies a target family and one or two substitute families to be searched for files. For example, suppose a process expects to find its files on the family SYSPK. This is considered the target family. To make the process look for its files on the family PARTS instead, you could use the assignment "FAMILY SYSPK = PARTS ONLY".

Note that this FAMILY value affects only files with a FAMILYNAME value of SYSPK. For example, if the file has a FAMILYNAME of DBFAM, then the process still looks for the file on DBFAM.

Note also that only one FAMILY value can be in effect at a time. For example, suppose the existing FAMILY value of a process is "FAMILY SYSPK = PARTS ONLY". In this case, an assignment such as "FAMILY DBFAM = PACK ONLY" disables family substitution for the SYSPK family and enables substitution for the DBFAM family.

If a program does not specify a FAMILYNAME for a disk file, the system searches for the file on the family named DISK. If you want the users of a program to specify a FAMILY value, you can leave the FAMILYNAME unspecified for all the files. The user can override the default FAMILYNAME of DISK with a FAMILY task attribute assignment such as "FAMILY DISK = DBFAM ONLY".

Sometimes it is useful to specify two substitute families in the FAMILY value. For example, you might have a WFL job that runs utilities stored on the family named DBFAM, which in turn use data files stored on the family named SYSPK. In this case, you can use a FAMILY statement like the one in the following WFL job:

```
?BEGIN JOB;
  FAMILY DISK = DBFAM OTHERWISE SYSPK;
RUN OBJECT/DAILY/RUN;
RUN OBJECT/REPORT/GENERATOR;
?END JOB
```

Because the FAMILY assignment is in the job header, the system searches for OBJECT/DAILY/RUN and OBJECT/REPORT/GENERATOR on DBFAM family and

then on SYSPK family. The FAMILY task attribute value is inherited by both tasks, which search for their data files on DBFAM and SYSPK families.

When a family statement specifies two substitute families, the first is referred to as the *primary family* and the second as the *alternate family*. In the previous example, DBFAM is the primary family and SYSPK is the alternate family.

When a process attempts to create a new file on the target family, the system creates the file on the primary family instead. When a process attempts to open or execute an existing file on the target family, the process searches for the file first on the primary family and then on the alternate family. If the TITLE file attribute of the existing file does not specify a usercode, the system searches for the file in the following locations, in the order shown:

1. On the primary family, under the usercode of the process
2. On the primary family, as a nonusercoded file
3. On the alternate family, under the usercode of the process
4. On the alternate family, as a nonusercoded file

If the TITLE attribute of a file does not specify a usercode, and the NEWFILE file attribute is TRUE, the system creates the file on the primary family under the usercode of the process.

Another method for overriding the FAMILYNAME file attribute is through file equations, as described under "Modifying File Attributes" earlier in this section. The following are two advantages to using FAMILY instead of file equations for this purpose:

- A single FAMILY assignment affects all the files in the program that have the specified target FAMILYNAME. Using file equations, you must specify each file individually.
- The FAMILY assignment overrides the target FAMILYNAME wherever it is mentioned in the program. By contrast, file equations are applied when a file is first declared. The program can later use file attribute assignment statements to override the values supplied through file equations.

Preventing File Duplications

The system does not allow two permanent disk files with the same title to exist on the same disk family. In order to handle attempts to duplicate disk file titles, most system administrators set the system option AUTORM. If a process attempts to enter a file in the disk directory for a family, but a file with the same name already exists in that family's disk directory, then the AUTORM option causes the existing file to be removed. For further information, refer to the discussion of preventing process suspension in Section 6, "Monitoring and Controlling Process Status."

Automatically Restoring Missing Disk Files

If your site uses the archiving subsystem to perform system backups, you can use the AUTORESTORE task attribute to reduce the likelihood that a process will be suspended for attempting to open a nonresident disk file.

If a process with an AUTORESTORE value of TRUE attempts to open a disk file, and the disk file is not present on the requested family, the system checks to see if there is an archive record specifying the location of a backup copy of the file. The system issues a request for an operator to mount the necessary tape. When the tape is mounted, the system copies the file back onto disk. At this point, the process that was attempting to use the tape resumes execution.

If the system is unable to restore the file for any reason, the process becomes suspended and appears in the W (Waiting Entries) system command display with a "NO FILE" RSVP message.

For an overview of the system archiving and AUTORESTORE features, refer to the *A Series System Administration Guide.*

Limiting Disk Usage

The system administrator can use the disk resource control (DRC) system to limit the disk usage of each user. For each usercode, the administrator can establish the maximum amount of space the user can use on each family. The limits are applied in a somewhat different manner for permanent and temporary disk files.

For permanent disk files, the limits imposed by the system administrator apply to the total of all the user's files on a given family. Any process that attempts to increase the total file usage beyond the limit receives an I/O error. For example, suppose the system administrator has established a limit of 2 megabytes on the disk usage for usercode CHAN on DBFAM family. Suppose there are already 1999999 bytes of permanent files under CHAN usercode on DBFAM, and a process attempts a write operation that requires another area to be allocated for one of these files. In this case, the write operation fails.

For temporary disk files, the limits imposed by the system administrator apply to individual processes running under the specified usercode. The administrator specifies the limit by assigning a TEMPFILELIMIT attribute to the usercode. This in turn sets a limit on the value that can be stored by the TEMPFILELIMIT task attribute of processes running under the usercode. If a process attempts to increase its temporary file usage beyond the number of megabytes specified by TEMPFILELIMIT, the process receives an I/O error.

For example, if the TEMPFILELIMIT for usercode CHAN is 3 megabytes, there can be two different processes running with CHAN usercode that each use 2 megabytes for temporary files. The total temporary file usage is thus 4 megabytes. This is not a violation of the TEMPFILELIMIT because the limit is enforced on a process-by-process basis.

Note also that, unlike the permanent disk file limits, the TEMPFILELIMIT cannot be linked to a particular disk family. The process might allocate its temporary files on any family. For example, if the TEMPFILELIMIT is 3 megabytes, and the process has allocated 2 megabytes of temporary files on DBFAM, the process can allocate no more than 1 megabyte on SYSPK.

At any given time, the TEMPFILEMBYTES task attribute records the total number of disk megabytes in use by the process for temporary files. The process can interrogate this task attribute to determine the process is nearing the TEMPFILELIMIT value. Alternatively, you can design the process to include I/O error handling that enables the process to recover from temporary file limit errors.

For information about permanent and temporary disk files, and about I/O error handling, refer to the *A Series I/O Subsystem Programming Guide*. For more information about the DRC system, refer to the *A Series Disk Subsystem Administration and Operations Guide*.

Controlling Printing

One aspect of process control is the ability to direct the printer output generated by a task. A process can control the printer output of an offspring through the use of relevant task attributes, such as PRINTDEFAULTS and FILECARDS.

The following subsections briefly introduce the printing control features of A Series systems and the role that task attributes play in printing control. The statements made about printer output in these subsections also apply to punch output, unless otherwise stated. For complete details about controlling printer and punch output, refer to the *A Series Print System (PrintS/ReprintS) Administration, Operations, and Programming Guide*.

For information about printing job summaries, refer to Section 10, "Determining Process History."

Default Handling of Printer Output

The system handles printer output in certain typical ways if operators, programmers, and users do not use printing-related statements to request different treatment. The following subsections describe the typical handling of printer and punch output.

Storing Printer Backup Files Temporarily

The system temporarily stores the printer backup files created by a process on a backup medium before printing them.

A process can use the BACKUPKIND file attribute to specify the kind of medium on which backup files are to be created. If the BACKUPKIND is DISK or PACK, the backup file is created on the family with that name and is automatically printed later.

If the BACKUPKIND value is TAPE, TAPEPE, TAPE7, or TAPE9, then the process is suspended and displays an RSVP message asking the operator to mount a tape. When the tape is mounted, the backup file is created on the tape. The system does not print the backup file automatically. However, you can later use SYSTEM/BACKUP or a WFL *PRINT* command to cause the file to be printed.

If the BACKUPKIND value is DONTCARE, then the LPBDONLY operating system option and the BACKUP option of the OPTION task attribute determine how the backup file is handled. If either or both of these options are set, the backup file is created on the family DISK. If both of these options are reset, the backup file is routed directly to a printer. If no printer is available, the process is suspended until a printer becomes available. The operator can use the OP (Options) system command to set or reset the LPBDONLY option.

The operator can use the SB (Substitute Backup) system command to specify a substitute backup medium for each possible BACKUPKIND value. The SB setting can convert any BACKUPKIND value to any other BACKUPKIND value. For example, SB can specify that all files with a BACKUPKIND of DISK be created on PACK instead. Note that if the LPBDONLY operating system option is set, SB substitutions for DISK also affect any backup files that have a BACKUPKIND of DONTCARE.

The SB command can also convert any BACKUPKIND value to DLBACKUP. This value cannot be specified directly by the BACKUPKIND file attribute; only the SB setting can cause this value to be applied to a backup file. The DLBACKUP value causes the backup file to be created on the family specified by the *DL BACKUP ON <family name>* form of the DL (Disk Location) system command.

Titling of Printer Backup Files

If no title is specified by the process that created a backup file, the system automatically assigns the backup file a title of the following form:

```
*BD/000<mix number>/..../<file number><internal name> ON <backup family>
```

In this title, the **BD* node indicates that this is a printer backup file. The prefix for punch backup files is **BP*. On a system running InfoGuard security software with the USERCODEDBACKUP option set to TRUE, the backup file titles are prefixed with the usercode of the process, rather than an asterisk (*).

The first node of the title is followed by one or more nodes that store mix numbers. The first of of these nodes contains the mix number of the job or the session. Other mix number nodes, if there are any, contain the mix numbers for other ancestors of the process, in order, from eldest to youngest. The last of the mix number nodes contains the mix number of the process itself. If there is only one mix number node, the backup file was created directly by a job or session. Each mix number node begins with three or four zeros: three zeros if the mix number is 4 digits long, and four zeros if the mix number is 3 digits long.

The last node of the file name stores a file number and an internal name. The internal name is the value of the INTNAME file attribute, which can be assigned by the process.

If INTNAME is not assigned, its value defaults to the file identifier used in the file declaration.

The file number is a 3-digit number that indicates the chronological order of this backup file compared to other backup files produced by the same process. For example, suppose a process declares a backup file with an INTNAME of A and another with an INTNAME of B. If the process opens and closes A three times, the system creates multiple backup files whose titles end with 000A, 001A, and 002A. If the process then opens B, the system creates a backup file whose title ends with 003B.

The backup family is the family determined by the rules discussed under "Storing Printer Backup Files Temporarily" earlier in this section.

Note that the backup file title can be affected by the task attributes discussed under "Other Print-Related Task Attributes" later in this section.

Submitting Print Requests

When a job terminates, the system groups the backup files produced by the job and its tasks into *print requests*. These print requests are groups of all the backup files that can be printed on the same device. The system then queues all the print requests for printing. In this context, a session is treated like a job, and backup files produced by tasks of the session are queued for printing when you end the session.

For WFL jobs submitted through a MARC or CANDE *WFL* command, the backup files are associated with the session and are queued for printing when the session ends. However, for WFL jobs submitted through a MARC or CANDE *START* command, the backup files are associated with the job and are queued for printing when the job terminates.

Selecting Print Requests

When one of the default printers becomes available, the system chooses one of the queued print requests to be the next print request printed. By default, short print requests are chosen before longer print requests. However, if the BACKUPBYJOBNR operating system option is set, then backup files are printed in order according to job number. The operator can set or reset this system option by using the OP (Options) system command.

Normally, the system removes any backup file from disk once the backup files has been printed. However, the system does not delete the backup file in the following circumstances:

- The SAVEBACKUPFILE file attribute is assigned the value TRUE.
- The LOCKEDFILE file attribute is assigned the value TRUE.
- The file resides on a CD-ROM disk.
- The file resides on a disk that is write-protected.

Programmatic Control Over Printing

A program can control the handling of printer output by specifying print attributes and print modifiers. Using these attributes and modifiers, the program can control such issues as

- The location and the device kind of the backup file
- The printer used
- The time the print request is considered for printing
- The number of copies that are printed
- The portions of the backup file to be printed
- The formatting and translation of printed output

The final values of the print attributes and print modifiers for a backup file are the result of several different factors, most of which are controlled by a programmer. To begin with, each print attribute and modifier has an ultimate default value that is used if no other factor affects the value. The ultimate defaults can be overridden by process defaults. The process defaults are established by the PRINTDEFAULTS task attribute. The PRINTDEFAULTS value consists of a list of print attributes and modifiers and their associated values. The system applies these values to all backup files produced by the process, unless the values are overridden for particular backup files.

The PRINTDEFAULTS value is itself the outcome of several layers of possible assignments. These sources of these assignments include the PRINTDEFAULTS usercode attribute in the USERDATAFILE, the PRINTDEFAULTS attribute of a session, inheritance from a parent process, assignments to the object code file, run-time task equations, and assignments to an active process.

You can override the process defaults for particular backup files by assigning print attributes to the backup file. *Print attributes* is the name given to file attributes that are related to printing, and they are assigned in the same way as any other file attribute. Using print attributes, a process can cause each backup file to be handled differently.

Another option for printing files is to use the WFL *PRINT* statement. You can enter this statement in WFL jobs, in MARC or CANDE sessions, or at an ODT. The PRINT statement is used mainly to print permanent backup files that were created on an earlier occasion. The backup files remain on disk when printing is completed, so they can be reused later.

The PRINT statement can assign print attributes and modifiers for any or all of the backup files printed. These assignments override all previous assignments for the backup files.

A process can affect the print handling for another process by making assignments to the PRINTDEFAULTS task attribute of the process. Where more specific control is needed, you can use the FILECARDS task attribute to specify print attributes for each backup file.

However, the PRINTDEFAULTS and FILECARDS values that are assigned externally can be overridden internally. A process can assign a different value to its own PRINTDEFAULTS task attribute after initiation. Also, file attribute assignments made by the process outside the file declaration override any conflicting assignments made by way of the FILECARDS task attribute.

Other Print-Related Task Attributes

Aside from PRINTDEFAULTS and FILECARDS, the following task attributes are related to printing: BACKUPFAMILY, BDNAME, DESTNAME, DESTSTATION, and OPTION (BACKUP, BDBASE, and NOSUMMARY options only). However, these task attributes were implemented before the current Print System. You can now use various print attributes to achieve effects similar to the effects of most of these task attributes. Print attributes are the preferred method for achieving such print control.

The BDNAME task attribute, if assigned, prevents a backup file from being automatically printed; instead, the file is saved on disk. In addition, BDNAME causes the backup file to be stored under the usercode of the process. The BDNAME value replaces **BD* as the beginning of the file name. However, the remainder of the file name follows the standard backup-file naming conventions.

As we have seen, BDNAME has several effects. You can achieve some of the same effects through the use of several print attributes. You can prevent automatic printing by setting the PRINTDISPOSITION attribute to DONTPRINT. You can assign a file name by setting USERBACKUPNAME to TRUE and assigning the desired name to FILENAME. The following example shows what these assignments look like in WFL:

```
FILE OUT (PRINTDISPOSITION=DONTPRINT, USERBACKUPNAME=TRUE,
          FILENAME= <file name>)
```

An advantage to using print attributes instead of BDNAME is that the print attributes give you complete control over the backup file name, whereas BDNAME only affects the prefix. On the other hand, this method is admittedly somewhat more complex than using BDNAME. A single BDNAME assignment affects all backup files used by a process, whereas when print attributes are used, separate FILENAME assignments must be made for each backup file. For example, if a process creates multiple backup files by opening and closing the same logical file repeatedly, then the FILENAME value should be changed before each file open operation; otherwise, each time the file is opened, the previous backup file with the same FILENAME is removed.

If BDNAME is assigned a non-null value, the backup file is saved and not printed, regardless of the PRINTDISPOSITION and SAVEBACKUPFILE values.

If BDNAME has a non-null value and USERBACKUPNAME is FALSE, then the FILENAME value is ignored. However, if both BDNAME and USERBACKUPNAME are TRUE, then the FILENAME value is used as the file title. If FILENAME was not assigned, then the INTNAME file attribute value is used as the title. If INTNAME was not assigned, then the file identifier is used as the title.

You can use the BACKUPFAMILY task attribute to specify the family where backup files produced by a process are to be stored. Only a WFL job or a message control system (MCS) can assign this task attribute. You can also assign the family for a backup file by using the FAMILYNAME print attribute. If there is a conflict between FAMILYNAME and BACKUPFAMILY, the FAMILYNAME value takes precedence over the BACKUPFAMILY value.

The DESTNAME task attribute specifies that output is to be printed at a particular station where a remote printer is attached. The DESTSTATION task attribute has the same effect as DESTNAME, but specifies the station by number instead of name. You can also specify a destination station by using the DESTINATION print attribute. If there is a conflict, the DESTINATION value takes precedence over the DESTNAME or DESTSTATION value.

You can use the BDBASE option of the OPTION task attribute to cause the task to assume some of the characteristics of a job. One of the effects of this option is to cause task backup files to be submitted for printing when the task terminates. If

BDBASE is not set, the backup files are not submitted for printing until the task's job terminates. Another method of controlling the timing of print requests is to use the PRINTDISPOSITION print attribute. Assigning PRINTDISPOSITION a value of EOT has the same effect on printing as setting the BDBASE option.

If you set BDBASE, then the PRINTDISPOSITION value is treated as it would be for a job. PRINTDISPOSITION values of EOT and EOJ are synonyms in this case, and both cause backup files to be printed when the task terminates. PRINTDISPOSITION values of CLOSE, DIRECT, and DONTPRINT have their usual effect, regardless of whether BDBASE is set.

The BACKUP option of the OPTION task attribute is discussed earlier in this section under "Storing Printer Backup Files Temporarily." The NOSUMMARY option of the OPTION task attribute is discussed under "Controlling Job Summary Printing" in the "Determining Process History" section.

Controlling Data Communications and Messages

You can use task attributes to help control the handling of remote files, to suppress unwanted messages, or to specify the language in which messages are to be displayed.

In addition to the topics discussed here, you can find helpful information in the discussions of CANDE, MARC, and ODT terminal communications in Section 3, "Tasking from Interactive Sources."

Controlling Message Tanking

Processes can communicate with terminals by way of remote files. *Tanking* is a method the system can use to temporarily store messages that a process writes to a remote file. You can use the TANKING task attribute to specify the default tanking mode for all remote files used by a process. The effects of this task attribute vary, depending on whether or not the terminal that the process writes to is controlled by COMS.

When a process writes a message to a remote file, the system inserts the message in an output queue. The system transfers messages from the output queue to the remote device as fast as the remote device is able to accept them. If the process writes messages to the remote file faster than the remote device can receive them, then the output queue can become full. If the output queue is full, and the process writes another message to the remote file, then the system can respond by tanking the output. Tanked output is stored in a file called the *tank file* on disk. The system retrieves messages from the tank file and places them in the output queue when space becomes available.

If the output queue is full and tanking is not enabled for the remote file, and the process attempts to write to the remote file, then the process must wait for room to become available in the output queue before the write operation can complete. The result can be a delay in the execution of the process. However, the process does not actually become suspended and does not appear in the W (Waiting Mix Entries) system command display.

The tanking mode for a particular remote file is determined primarily by the file attribute TANKING. To prevent tanking from occurring, you can assign TANKING

a value of NONE. To enable tanking, you can assign TANKING a value of SYNC. To enable a process to close the remote file and continue execution while tanked output still exists, you can assign TANKING a value of ASYNC. When ASYNC is used and the process closes the remote file, the system continues to transfer messages from the tank file to the output queue until the tank file is empty. For details about the TANKING file attribute, refer to the *A Series File Attributes Programming Reference Manual.*

The default value of the TANKING file attribute is UNSPECIFIED. If the file attribute has this value, then tanking is determined by the TANKING task attribute and the MCS. The TANKING task attribute has the same possible range of values as the TANKING file attribute. Thus, setting the TANKING task attribute to NONE, SYNC, or ASYNC causes these values to be applied to all remote files whose TANKING file attribute is UNSPECIFIED.

If the TANKING file attribute and the TANKING task attribute are both UNSPECIFIED, the MCS controlling the station can set the tanking mode for the remote file. The MCS can do this by way of a parameter to the Station Assignment to File *DCWRITE*. The DCWRITE statement is described in the *A Series DCALGOL Programming Reference Manual.*

For remote files that communicate with terminals controlled by the CANDE MCS, an operator can use the ?TANKING network control command to specify the default tanking mode. The ?TANKING command can specify default values of UNSPECIFIED, NONE, SYNC, or ASYNC.

For remote files that communicate with terminals controlled by COMS, the effects of the TANKING file and task attributes vary depending on the type of program involved. Three types of application programs can run under COMS: direct window programs, remote-file programs, and MCS window programs.

Direct window programs communicate with terminals through special COMS structures rather than through remote files. Consequently, the TANKING file attribute and task attribute have no meaning for these programs.

Remote-file programs are programs that communicate through declared or dynamic remote-file windows. Declared remote-file windows are windows that appear in the COMS configuration file and have particular programs associated with them. Dynamic remote-file windows are created by COMS at run time when a program initiated from a MARC session opens a remote file.

For remote-file programs with a TANKING value of NONE, the system does not perform tanking for the remote file. If the TANKING value is UNSPECIFIED, SYNC, or ASYNC, the system performs tanking as if the TANKING value were ASYNC.

Unisys recommends that you enable tanking for a remote-file program unless the program services only a single terminal. If a remote-file program services multiple terminals and uses a TANKING value of NONE, the program can go into a waiting state when writing output to a terminal. While the program is in a waiting state, it is unable to service input from other terminals. On the other hand, if a remote-file program services a single terminal, it can be reasonable for the program to wait for all output to be displayed before accepting any further input.

An MCS window program is a program that you initiate from a COMS window devoted to a subsidiary MCS. For example, any programs you initiate by entering a RUN command in a CANDE window are considered MCS window programs. For such programs, the system supports the full range of TANKING file attribute and task attribute values: NONE, SYNC, ASYNC, and UNSPECIFIED. The subsidiary MCS, such as CANDE, can specify a tanking mode if the TANKING file and task attribute are both UNSPECIFIED.

In addition to the system-level tanking that has been described up to this point, COMS-level tanking is provided for the programs that run in a COMS environment. COMS-level tanking affects direct window programs, remote-file programs, and MCS window programs. COMS places output messages in the COMS tank file if the messages are being written faster than the station can receive them, or if the messages are sent to a window dialogue that is suspended.

By default, only messages generated for the user's current window dialogue are displayed at the terminal, and all other window dialogues are considered suspended. The user can resume another dialogue by using an ON command to transfer to the dialogue, or by entering a RESUME command that specifies the dialogue. When the window dialogue is resumed, COMS retrieves tanked messages and sends them to the station.

COMS-level tanking is a necessary feature in the COMS windowing environment and is performed regardless of the value of the TANKING file and task attributes.

Suppressing Unwanted Messages

Although system messages are intended to be helpful, there can be situations where you might find it more convenient to suppress the display of certain messages.

Deimplementation warning messages are a good example of this principle. The system issues a deimplementation warning message for a process when the process uses a feature that has been scheduled for future deimplementation. These warning messages can be very valuable because they help you to identify programs that need to be modified before you can migrate your system to a new Mark release.

However, the system displays these deimplementation warnings each time the program is run. If you run the program frequently, you may see the warning messages more often than you care to be reminded of the pending deimplementation. You can suppress the messages by using the SUPPRESSWARNING task attribute. This attribute enables you to specify a list of warning message numbers or number ranges, as in the following example:

```
RUN OBJECT/PROG;SUPPRESSWARNING = "1,4,8-10";
```

You can learn the identifying number for a message in either of two ways. First, you can note the warning number when it appears in the message itself. For example, after seeing the following message, you might assign SUPPRESSWARNING a value of "13".

```
WARNING 13: DISK FILE HEADER CHANGES.  SEE 3.7 MCP D-NOTE 6638
```

Second, you can interrogate the TASKWARNINGS task attribute. This task attribute returns the value of the WARNINGS file attribute of the object code file that is being executed. The WARNINGS file attribute, in turn, stores the message numbers for all the warning messages that the system has ever displayed for processes executing code from that object code file. The following is an example of a declaration and statements you can use in an ALGOL program to suppress all previously displayed warning messages:

```
EBCDIC ARRAY WARN[0:999];
  REPLACE WARN BY MYSELF.TASKWARNINGS;
  REPLACE MYSELF.SUPPRESSWARNING BY WARN;
```

You might also find it useful to suppress DISPLAY messages. A process issues a DISPLAY message by executing a DISPLAY statement. DISPLAY messages are used to enable a process to communicate information to the user without actually opening a remote file or ODT file. DISPLAY messages appear in the MSG (System Messages) system command display, at the terminal of the user that initiated the process, and in the system log.

If a process is initiated by a user at a data comm terminal, the DISPLAY messages issued by the process are probably of interest only to that user. The appearance of these messages in the MSG display can be a needless distraction to the system operator. You can eliminate this distraction by setting the DISPLAYONLYTOMCS task attribute to TRUE for the process. When this attribute is TRUE, if the process is initiated from a data comm terminal, DISPLAY messages appear at the originating terminal but do not appear in the MSG display at the ODT.

Localization

Localization is the process of tailoring the user interface of a program to users of a particular nation or culture. Two task attributes can assist you in the localization process: the LANGUAGE task attribute and the CONVENTION task attribute.

You can use the LANGUAGE task attribute to specify the language that is used for a process. This task attribute has effects on two levels:

- The system attempts to use the specified language when displaying any system messages generated for the process, such as BOT, EOT, and RSVP messages. The LANGUAGE value has effect only if system messages in the specified language have been installed on your system.
- The specified language becomes the default language for any messages that are displayed by MESSAGESEARCHER statements in an ALGOL or NEWP program. The LANGUAGE value has effect only if a version of the OUTPUTMESSAGEARRAY using the specified language has been bound to the object code file.

You can use the CONVENTION task attribute to specify the conventions for dates, times, and currency used by a process. This task attribute affects processes that use the CENTRALSUPPORT library to format data according to requested conventions.

The CONVENTION task attribute specifies a default convention to be used for CENTRALSUPPORT procedure calls. The user process can selectively override this default through parameters to the CENTRALSUPPORT procedures.

For further information about localization, refer to the *A Series MultiLingual System (MLS) Administration, Operations, and Programming Guide.*

Limiting I/O Usage

When a process executes an I/O statement, the central processor must execute some operating system code to initiate the I/O operation. Thereafter, I/O processors (IOPs), data link processors (DLPs), and various peripheral devices such as disk drives might all devote varying amounts of time to executing the I/O operation. At the completion of the I/O operation, the central processor executes some I/O finish code.

Of all the system resource usage caused by I/O operations, only the I/O initiation time is recorded by the system for individual processes. The accumulated I/O initiation time for a process is stored in the ACCUMIOTIME task attribute. The I/O initiation time for a process is also visible in the output from the TI (Times) system command and in the Major Type 1, Minor Types 2 and 4 (EOJ and EOT) system log entries.

You can use the MAXIOTIME task attribute to set a limit on the amount of I/O initiation time that a process can use. When the ACCUMIOTIME task attribute reaches a value equal to that of MAXIOTIME, the system discontinues the process and displays the error message "EXC I/O TIME".

The main use of the MAXIOTIME task attribute is to ensure that WFL jobs are placed in the proper job queues. For example, suppose there is a high-priority job queue that is intended for jobs that are not very I/O intensive. The system administrator can use the IOTIME job queue attribute to provide default and limiting values for the MAXIOTIME task attribute of all WFL jobs that use the job queue. If you submit an extremely I/O intensive job through the job queue, the system discontinues the job when it exceeds the MAXIOTIME value. This enforcement of the MAXIOTIME value gives you an incentive to resubmit the job through a different job queue. For an introduction to the subject of job queues, refer to the discussion of WFL in Section 4, "Tasking from Programming Languages."

It is also possible for the system administrator to limit each person's usage of disk space. Refer to "Limiting Disk Usage" earlier in this section.

Section 10
Determining Process History

Process history consists of information about how a process terminated, the accumulated resource usage of the process, and what actions the process took while it was active. Process history information can help you determine if a program is running as intended, and can help you to locate the source of any problems that arise.

This section describes the uses of various sources of process history information, including termination messages, job summaries, system log entries, history-related task attributes, and program dumps.

Understanding Termination Messages

You can quickly find out how a process terminated by examining the C (Completed Mix Entries) system command display. The following is an example of this display:

```
---Job-Task-Time--Hist---------- COMPLETED ENTRIES ----------------
* 1962\3430 11:43 EOT  (LANJ) *LIBRARY/MAINTENANCE
* 2619\3368 11:43 EOT  (ELMER) *OBJECT/MAIL ON PACK
* 3353\3354 11:43 SNTX (ORDS) *BINDER ON SYS37 MCP/FIXSBP ON DPMAST
  3384\3422 11:42 O-DS (JAS) (JAS)MARC WFL
  3384\3423 11:42 P-DS (JAS) (JAS)WFLCODE
  3327\3327 11:42 EOJ  (RALPH) JOB (RALPH)OBJECT/BNATEST ON DPMAST
```

For each entry, the following information is displayed: the job number, the mix number, the time the process terminated, the type of termination, the usercode of the process, and the name of the process (which is usually the object code file title).

If the process was initiated from a Menu-Assisted Resource Control (MARC) session, then a similar termination message is automatically displayed on the TASKSTATUS screen. The following is an example:

```
12:10 3384\3718 EOT  (ROLLINS)MARC  WFL
```

For a process initiated from a Command and Edit (CANDE) session, abnormal terminations result in a display of the termination type and other process history information. The following is an example:

```
#2316 OPERATOR DSED @ (00000120)*
#O-DS @ 00000120.
#ET=3.2 PT=0.1 IO=0.1
```

The first two lines shown in the preceding example would be displayed only for an abnormal termination. These lines give the mix number, the cause of the termination, and the sequence number of the statement the process was executing when it terminated. (The sequence number is replaced by a code address if the program was compiled without the LINEINFO compiler option being set. For information about how to interpret the code address, refer to "Determining Where a Fault Occurred" later in this section.)

All terminations, whether normal or abnormal, result in the display of a line similar to the third line shown in the preceding example. This line gives statistics on the elapsed time, accumulated processor time, and accumulated I/O time for the process.

The CANDE, MARC, and ODT termination messages make use of the same termination type abbreviations. Of these, the following indicate normal terminations:

EOJ	The process was a job that terminated normally.
EOT	The process was a task that terminated normally.
SNTX	The process was a compilation that encountered syntax errors. The process terminated normally, but no object code file was created.

Table 10–1 lists the abnormal termination messages, their meanings, and the corresponding values for history-related task attributes. For an introduction to history-related task attributes, refer to "Determining the Type of Termination" later in this section.

Table 10–1. Abnormal Termination Messages

Message	HISTORYTYPE	HISTORYCAUSE	Meaning
A-DS	8	0	The process was a Work Flow Language (WFL) job whose initiation failed because the job attribute list included an invalid task attribute assignment; or, the process was discontinued but is now executing an EPILOG procedure.
D-DS	4	6	The process encountered a data comm error.
E-DS			The process encountered a Data Management System II (DMSII) error.
F-DS	4	4	The process requested a machine operation that could not be executed. Examples are dividing by zero or reading past the end of an array.
I-DS	4	7-9	The process encountered an I/O error.

continued

Table 10–1. Abnormal Termination Messages (cont.)

Message	HISTORYTYPE	HISTORYCAUSE	Meaning
N-DS	4	13	The process encountered a BNA error.
O-DS	4	1	The process was discontinued by an operator command.
P-DS	4	2	The process attempted an illegal action or deliberately set its STATUS task attribute to TERMINATED, or was terminated because its parent terminated.
Q-DS	7	0	The process was a job that did not qualify for any job queue, or was discontinued by an operator command while it was in a job queue.
R-DS	4	3	The process exceeded a resource limit, such as MAXPROCTIME.
S-DS	4	5	The process violated system parameters.
U-DS	4	10-11	The process was discontinued by an unknown cause.
Unn-DS	Not specified	nn	The process was discontinued with an unrecognized HISTORYCAUSE value. In the actual message, the digits *nn* are replaced by the HISTORYCAUSE value.
?-DS	4	0	The process was discontinued by an unknown cause.

Using Log Information

The system records the activity of each process in two types of logs:

- The system log (SUMLOG)

  This is a central log that stores information about all kinds of actions on the system.

- Job logs

  A separate job log is created for each job on the system and is stored in the job's job file. The system creates a job log for WFL jobs and other independent processes, as well as for CANDE and MARC sessions. The job log contains information about the job (or session) and its descendant tasks. Depending on the values of various task attributes and system options, the system might create a printout of the job log, called the *job summary*.

The following subsections explain how the programmer and system operator can control the contents of these logs and the generation of reports from these logs.

Specifying the Information to Be Logged

An operator can use the LOGGING (Logging Options) system command to select the major and minor log entry types that are to be logged. You can specify that a particular type of log entry is to appear in the job log, in the system log, in both, or in neither. The following LOGGING command causes Major Type 1, Minor Type 5 (File Open) entries to appear in job logs, and Major Type 1, Minor Type 6 (File Close) entries to appear in the system log:

```
LOGGING 1,5 JOBFILE ALL;1,6 SUMLOG ALL;
```

You can use the DEPTASKACCOUNTING task attribute and the FILEACCOUNTING task attribute to control certain types of logging. These task attributes affect the system log and the job log equally. You can use DEPTASKACCOUNTING to prevent the system from generating log entries to record the initiation and termination of a dependent process. You can use the FILEACCOUNTING task attribute to prevent the system from generating log entries to record file open and close actions. You can create defaults for these task attributes on a systemwide basis with the ACCOUNTING (Resource Accounting) system command. You can create defaults for these task attributes on a usercode basis through assignments to the usercode attributes with the same names in the USERDATAFILE.

You can use either of two system commands to log comments about the history of a particular process. The LC (Log Comment) system command enters a comment in the system log only. The LJ (Log to Job) system command enters a comment in both the system log and the job log of a particular job. The following is an example of this command:

```
3335 LJ JOB RAN NORMALLY
```

You can use the NOJOBSUMMARYIO task attribute to suppress the logging of information in the job log. If NOJOBSUMMARYIO is set, no entries are written to the job log, except for the Major Type 1, Minor Type 1 (BOJ) entry or the Major Type 4, Minor Type 1 (Log-on) entry. NOJOBSUMMARYIO can also be set and reset throughout a job to prevent selected parts of the job from appearing in the job log. Using NOJOBSUMMARYIO saves I/O time and thus allows the job to run more efficiently.

You can use the LG (Log for Mix Number) system command and the LOGSELECT usercode attribute to enable logging of selected types of events for a particular usercode. These features enable the system administrator to monitor the activities of a particular user who might be committing some type of security breach. These features affect the system log only.

Controlling Job Summary Printing

The printing of job summaries is controlled primarily by the JOBSUMMARY task attribute. To cause job summary printing, you can assign a value of UNCONDITIONAL, and to prevent job summary printing, you can assign a value of SUPPRESSED. To cause conditional printing of job summaries, you can use either of two values: ABORTONLY or CONDITIONAL. Either of these values causes job summary printing if the job or any of its tasks terminate abnormally. The difference between the two values is that the CONDITIONAL value also causes job summary printing if the job has any printer backup files associated with it or if a compiler task encounters syntax errors.

If the JOBSUMMARY task attribute has a value of DEFAULT, then job summary printing is controlled by either of two types of defaults.

- If the NOSUMMARY option of the OPTION task attribute is set, then a JOBSUMMARY task attribute of DEFAULT is interpreted as CONDITIONAL.
- If the NOSUMMARY option of the OPTION task attribute is reset, then the Print System JOBSUMMARY option controls the job summary printing. The Print System JOBSUMMARY option is set or reset through the PS DEFAULT system command. The JOBSUMMARY option can specify any of the following values: CONDITIONAL, UNCONDITIONAL, SUPPRESSED, or ABORTONLY.

Note that the Print System JOBSUMMARY option replaces the operating system option NOSUMMARY, which is no longer supported.

A job summary can be printed for any WFL job that compiled successfully. This is true even if the job never ran because no job queue would accept it or because an operator discontinued the job while it was queued.

Saving the Job Summary File

You can use the JOBSUMMARYTITLE task attribute to cause the job summary file to be saved as a permanent disk file.

If the JOBSUMMARYTITLE value is a null string (the default value), then the system creates a job summary file only if a job summary is to be printed. If the system does create a job summary file, the system removes the file once it is printed. The job summary file title usually has the following form:

```
*BD/ØØØ<job number>/ØØØSUMMARY
```

If you assign a file title to JOBSUMMARYTITLE, then the system creates a job summary file with the specified file title. The job summary file remains on disk, whether or not the system prints the job summary. You can use a Command and Edit (CANDE), Menu-Assisted Resource Control (MARC), or WFL *PRINT* command to print out the job summary file later. For a description of the PRINT command, refer to the *A Series Print System (PrintS/ReprintS) Administration, Operations, Programming Guide*.

Analyzing the System Log

You can use the LOGANALYZER utility to obtain a detailed report of the history of a particular process. You can invoke LOGANALYZER by using the LOG command from CANDE, MARC, WFL, or an ODT. The LOG command causes the current system log to be searched, unless the title of an old system log is specified.

The following command displays all log entries for the job with mix number 7483 and for all descendants of that job:

```
LOG JOB 7483
```

The following command displays all log entries for the process with mix number 8923:

```
LOG MIX 8923
```

You can specify various options to limit the types of entries that are displayed for the process and to direct the output to an ODT, a remote terminal, or a printer. For details, refer to the discussion of LOGANALYZER in the *A Series System Software Support Reference Manual*.

Programmatically Interrogating Process History

After a task terminates, the task variable associated with it continues to exist until the parent exits the block that contains the task variable declaration. As long as the task variable exists, the parent can use it to interrogate the final task attribute values of the task. By interrogating history-related task attributes, the parent can find out whether the task terminated normally.

On the other hand, the history of a job cannot be interrogated through task attributes. The task variable of a job can be accessed only by the job itself and its descendants, and the descendants cannot survive the termination of the job.

Determining the Type of Termination

Several task attributes and a special WFL expression are available for determining how a task terminated. The relevant WFL expression is the *task state inquiry*. This expression can be used to determine whether termination was normal or abnormal. For example, the following WFL statement causes a specified action to be taken if the task terminated abnormally:

```
IF TSK ISNT COMPLETEDOK THEN
```

Another way to determine whether a task terminated normally is to inspect the HISTORYTYPE task attribute. A HISTORYTYPE value of NORMALEOTV indicates that the termination was normal. A value of DSEDV indicates that termination was abnormal. Most of the other values indicate that the process has not yet terminated and give an indication of its current state.

If termination was abnormal, the HISTORYCAUSE task attribute can be interrogated to determine the general type of abnormal termination that occurred. For example, a value of OPERATORCAUSEV indicates that an operator command discontinued the process and a value of DCERRV means that the process was discontinued because of a data comm error.

A more detailed account of why a task terminated abnormally is stored in the HISTORYREASON task attribute. For example, suppose the HISTORYCAUSE value is RESOURCECAUSE, meaning that a resource limit was exceeded. The HISTORYREASON value might be PROCESSEXCEEDEDV, which means specifically that the processor time limit was exceeded.

The system uses the HISTORYTYPE and HISTORYCAUSE values to determine what termination message to display for a process. The correspondence between these task attributes and the termination messages is shown in Table 10–1.

Determining Whether a Compilation Was Successful

You can use any of several methods to determine programmatically whether a particular compilation uncovered syntax errors in the source program.

For a compilation initiated from WFL, the task state expression can be used to determine whether the compilation was successful. To use this expression, a task variable must first be associated with the compilation in the COMPILE statement. The following is an example:

```
COMPILE OBJECT/PROG WITH ALGOL [COMPILETASK] LIBRARY;
IF COMPILETASK IS COMPILEDOK THEN
   RUN SYSTEM/XREFANALYZER (Ø);
```

Another way to determine whether the compilation was successful is to interrogate the HISTORYTYPE task attribute of the compilation. A value of NORMALEOTV means

that the compilation was successful, but a value of SYNTAXERRORV means that syntax errors were found.

Another method that can be used is to interrogate the TASKVALUE task attribute. TASKVALUE has a value of 0 (zero) if the compilation was successful or 1 if syntax errors were found.

Responding to Task Failures

WFL includes a special statement that specifies actions to be taken if any offspring terminates abnormally. This is the *ON TASKFAULT* form of the ON statement. The *ON TASKFAULT* statement remains in effect for the remainder of the job unless overridden by another *ON TASKFAULT* statement. For example, a WFL job could include the statement *ON TASKFAULT, ABORT*. This statement causes the job to terminate abnormally when any of its offspring terminates abnormally.

Determining Where a Fault Occurred

You can use the STACKHISTORY task attribute to determine the statement that was being executed and the procedures that had been entered when a process terminated abnormally. To understand the value returned by this attribute, you must have compiled the program that was being executed with one or both of the following compiler options set: LINEINFO and LIST.

Setting LINEINFO causes the STACKHISTORY value to include the sequence number for each of the relevant statements in the source program. Setting LIST causes the compiler to produce a printout of the source program that includes code addresses for each line. The code addresses are needed to interpret the STACKHISTORY value if LINEINFO was not set.

The following is an example of the source program printout for a program that was compiled with the LIST and LINEINFO options set. (The example has been condensed horizontally to fit on the page.)

```
BEGIN                                          00000200  000:0000:0
                                             BLOCK#1 IS SEGMENT 0003
                                          1  00000300  003:0000:1
                                             00000400  003:0000:1
REAL X, Y;                                   00000500  003:0000:1
PROCEDURE ONE;                               00000600  003:0000:1
BEGIN                                        00000700  003:0000:1
  PROCEDURE TWO;                             00000800  003:0000:1
                                               ONE IS SEGMENT 0004
                                          2  00000900  004:0000:1
  BEGIN                                      00001000  004:0000:1
     Y := X DIV 0;                           00001100  004:0000:1
  END;                                    3  00001200  004:0001:2
                                          3  00001300  004:0001:3
  TWO;                                       00001400  004:0001:3
END;                                         00001500  004:0002:1
                                    ONE(004) LENGTH IN WORDS IS 0005
                                          2  00001600  003:0000:1
                                             00001700  003:0000:1
ONE;                                         00001800  003:0000:1
                                             00001900  003:0000:5
END.                                         00002000  003:0000:5
                                  BLOCK#1(003) LENGTH IN WORDS IS 0006
```

In this example, each line of source code ends with the sequence number and code address of the line. The code address is divided into three parts by colons; the first part is the code segment number, the second is the word number, and the third is the syllable number. The numbers in the code address are in hexadecimal format.

If a process terminates normally, the STACKHISTORY value is a null string. However, if the process terminates abnormally, STACKHISTORY returns a value such as the following:

```
004:0000:5 (00001100), 004:0002:1 (00001400), 003:0000:5 (00001800).
```

This value gives the code address and sequence number for the statement that was being executed when the process terminated and for each procedure invocation statement that was in effect when the process terminated. Thus, the value in this example indicates that the statement at line 1100 was being executed when the process terminated, and that procedure invocation statements at lines 1400 and 1800 were in effect.

If LINEINFO is set, STACKHISTORY returns the following value:

```
004:0000:5, 004:0002:1, 003:0000:5.
```

The numbers in this example give a somewhat less exact idea of the locations of the statements that were being executed when the process terminated. Each statement usually occurs on the line preceding the specified code address. The address 004:0000:5 does not appear in the printout, but the statement occurs on the next lower-numbered line: 004:0000:1.

Note that if the object code file was produced by the Binder, you must use some extra care in interpreting the code addresses or sequence numbers returned in the STACKHISTORY value. When the Binder produces a bound object code file, the Binder changes the code segment numbers for statements in the subprogram. Fortunately, if you use the Binder option LIST, the Binder produces a printout that lists such changes. The following is an example of such a printout:

```
                    O B J E C T / A L G O L / B I N D   O N   D I S K
                    = = = = = = = = = = = = = = = = = = = = = = = = =

HOST IS OBJECT/ALGOL/HOST;                                          00001270
BIND PRINTIT FROM OBJECT/ALGOL/SUB;                                 00001272
STOP;                                                               00001274
BEGIN BINDING PRINTIT OF BLOCK#1 FROM OBJECT/ALGOL/SUB
      PRINTIT  (02,0006) CHANGED TO (02,0006)
K  <---- NEW GLOBAL ADDED TO HOST -- WARNING ONLY
      K  (02,0004) CHANGED TO (02,0008)
      LINE  (02,0005) CHANGED TO (02,0003)
      J  (02,0003) CHANGED TO (02,0005)
      BUFFER  (02,0002) CHANGED TO (02,0004)
      ?010  (01,0006) CHANGED TO (01,0006)
      <SEG DICT ITEM>   (01,0002) CHANGED TO (01,0005) = 03 000001300001
      <SEG DICT ITEM>   (01,0003) CHANGED TO (01,0007) = 05 07000000005F
      <SEG DICT ITEM>   (01,0004) CHANGED TO (01,0008) = 05 08000054 0002
END OF BINDING PRINTIT
```

Note the three lines near the bottom of the list that begin with "<SEG DICT ITEM>". These list changes to the address couples for code segments in the subprogram. The second number in each address couple is the offset, which is the same as the code segment number for that code segment. The list informs us that code segment number 2 was changed to 5, 3 was changed to 7, and 4 was changed to 8. Therefore, you should look at the STACKHISTORY value for code addresses that begin with 5, 7, or 8, and make a note that they really begin with 2, 3, or 4, respectively. Then you can look for the code addresses in the compiler listing that was created when you originally compiled the subprogram. Suppose that the STACKHISTORY value is as follows:

```
005:000F:1, 003:0017:3.
```

You should translate the first address into 002:000F:1, and then look at the compiler listing of the subprogram to determine which statement had that code address. The second address does not begin with 5, 7, or 8, so you don't need to translate it. You can find the procedure invocation referred to by the second address at 003:0017:3 in the compiler listing for the host program.

If the host program and the subprogram were compiled with the LINEINFO compiler option set, and the Binder was run with the LINEINFO Binder option set, then the bound object code file contains sequence numbers that appear in the STACKHISTORY value. The Binder does not change the sequence numbers of the host program or the subprogram. When interpreting the sequence number, beware of the possibility that the same sequence number occurred in both the host program and the subprogram file. For example, suppose that the following is the STACKHISTORY value:

```
005:000F:1 (00000750), 003:0017:3 (00001350).
```

The subprogram and the host program both might contain lines with the sequence number 750, and they also both might contain lines with the sequence number 1350. However, the last address in the STACKHISTORY value always refers to a statement in the host program. At line 1350 in the host program listing is a procedure invocation statement. If this statement invokes the bound-in procedure, then line 750 is found in the subprogram listing. Otherwise, line 750 is found in the host program listing.

Another task attribute that provides information related to process history is the STOPPOINT task attribute. This real-valued attribute has fields defined that store the fault reason and the code address. The fault reason is the same as the value returned by the HISTORYREASON task attribute, and the code address is the same as the first code address of the STACKHISTORY value.

Designing a Program to Survive Faults

A fault is an illegal action that is detected by the hardware, such as an attempt to divide by zero. In general, a process is discontinued if it encounters a fault. However, ALGOL provides a unique feature that can be used to allow the process to continue normal execution after a fault. The ALGOL *ON* statement specifies actions to be taken if a fault occurs. In addition, the ON statement can be used to interrogate the type of fault and the stack history. The stack history value returned is identical in format to that returned by the STACKHISTORY task attribute.

If any fault occurs, the following ON statement stores the stack history into array FAULTARRAY and the fault type into FAULTNO. The statement then invokes the procedure HANDLEFAULTS, passing the fault type to it as a parameter:

```
ON ANYFAULT [FAULTARRAY:FAULTNO], HANDLEFAULTS(FAULTNO);
```

Another method of responding to faults is to use the RESTART task attribute. For details, refer to Section 11, "Restarting Jobs and Tasks."

Controlling Program Dumps

A program dump is a printout of information about the current state of a process. You can use this information to help debug a defective program. The following subsections explain how to specify when program dumps are to occur, and how to specify which types of information should be included in the dump.

On a system running InfoGuard security enhancement software, some security options can restrict the contents of program dumps and the ability to copy program dumps. Refer to the *A Series Security Administration Guide* for details.

Using Program Statements to Control Program Dumps

You can initiate and control program dumps in either of two ways: by using program dump statements, or by using the OPTION task attribute.

The program dump statements that are available are the ALGOL *PROGRAMDUMP* statement, the COBOL(68) *CALL PROGRAM DUMP* statement, the COBOL74 and COBOL85 *CALL SYSTEM DUMP* statement, the FORTRAN or FORTRAN77 *DEBUG PROGRAMDUMP* statement, and the Pascal *Programdump* procedure. Some languages provide other statements to dump process information, but these are language-specific features. The preceding statements call an operating system feature that is available from a variety of sources.

Alternatively, you can enable a program dump by setting certain options of the OPTION task attribute. If the FAULT option is set, then the process generates a program dump if it terminates abnormally because of an internal cause. If the DSED option is set, then the process generates a program dump if the process terminates abnormally because of an external cause. For a definition of internal and external causes, refer to "Understanding Internal and External Causes" later in this section.

You can also specify various dump options, which determine the types of information that are included in the program dump. These dump options can be accessed through assignments to the OPTION task attribute. In ALGOL, FORTRAN, FORTRAN77, and Pascal, these options can also be set by parameters in a program dump statement.

If a program dump statement specifies dump options, then the dump options specified in that statement are used, and the value of the OPTION task attribute is ignored. If the program dump is caused by the DSED or FAULT option of the OPTION task attribute, or by a program dump statement that does not specify any dump options, then the dump options specified by the OPTION task attribute are used for the dump.

The possible dump options are ARRAY, BASE, CODE, DBS, FILE, LIBRARIES, PRESENTARRAYS, PRIVATELIBRARIES, SIBS, TODISK, and TOPRINTER. The effects of these options are explained in the discussion of the OPTION task attribute in the *A Series Task Attributes Programming Reference Manual*. The effects of the TODISK and TOPRINTER options are also discussed under "Controlling the Program Dump Destination" later in this section.

Using Operator Commands to Control Program Dumps

If a process is behaving abnormally, you might want to invoke a program dump for the process. You can use the dump later to help debug the process.

One way you can invoke a dump is by using the DUMP (Dump Memory) system command. The *<mix number> DUMP* form of this command initiates a program dump for the specified process. The *<mix number> DUMP <option list>* form of this command assigns dump-related options to the OPTION task attribute and then initiates a program dump. The OPTION values are retained and affect any later program dumps for the process, unless overridden by later assignments. The following example dumps information about arrays and files for a process with the mix number 3457:

```
3457 DUMP ARRAYS, FILES
```

It is possible to view the program dump while the process is still running. Refer to "Analyzing a Program Dump from a Running Process" later in this section.

You can also trigger a dump by way of the DS (Discontinue) system command. The *<mix number> DS <option list>* form of this command initiates a program dump and discontinues the process. The option list in this command controls the contents of the dump by assigning options to the OPTION task attribute. The following example dumps arrays and code segments and discontinues the process with mix number 3457:

```
3457 DS ARRAYS, CODE
```

Note that the simple form of the DS command, *<mix number> DS*, causes a program dump if the DSED option of the OPTION task attribute was previously set through object code file assignments, task equations, or task attribute assignments executed by the process. You can prevent such a dump from occurring by using the *<mix number> DS NONE* form of the DS command.

Controlling the Program Dump Destination

You can direct a program dump to a printer backup file for printing, to a disk file for later analysis and printing, or both. You can control the program dump destination through two dump options: TOPRINTER and TODISK. These options are available in ALGOL, FORTRAN77, and Pascal through program dump statements. Languages that provide access to task attributes can also assign these options by way of the OPTION task attribute. Additionally, these options can be assigned in a DS (Discontinue) or DUMP (Dump Memory) system command.

The TOPRINTER option causes any program dumps generated by the process to be directed to a printer backup file called the *task file*. For details about the task file, refer to "Using the Task File" later in this section.

The TODISK option causes any program dumps generated by the process to be directed to a disk file. The contents of the program dump are determined by the other dump options, except for the BASE option. Whenever TODISK is set, the BASE option is treated as if it is also set. The following are advantages to using the TODISK option instead of the TOPRINTER option:

- The dump is performed more rapidly, and produces less printer output at the time of the dump. This factor makes it convenient for you to set the dump options to dump all possible information. By setting all the dump options, you reduce the likelihood of having to try to reproduce the problem later to obtain more information.
- The disk file stores dump information in a format that can be analyzed by the DUMPANALYZER utility. DUMPANALYZER also enables you to decide at analysis time what information to include in the report. You can even run DUMPANALYZER repeatedly to produce reports on different information from the same dump. Another benefit is that DUMPANALYZER provides a detailed analysis of the process information block (PIB).

You can also use DUMPANALYZER to produce a report similar to one created by the TOPRINTER option. Like TOPRINTER reports, DUMPANALYZER reports include the names of all the identifiers used by the process. (However, identifiers are included in the report only if all object code files used by the process are present when DUMPANALYZER is run.) For information about running the DUMPANALYZER utility, refer to the *A Series System Software Support Reference Manual.*

When the TODISK option is used, the default title for the resulting disk file has the following format:

```
(<usercode>)PDUMP/<process name>/<date>/<time>/<mix number> ON <family>
```

The values of the various elements of this title are as follows:

Title Element	Value
<usercode>	The value of the USERCODE task attribute of the process.
<process name>	The value of the NAME task attribute of the process, except that any usercode or family name is omitted. If the resulting process name is more than eight nodes long, then only the first eight nodes are included.
<date>	The current date, in the form YYMMDD.
<time>	The current time, in the form HHMMSS.
<mix number>	The value of the MIXNUMBER task attribute of the process.
<family>	DISK, unless the FAMILY task attribute provides a primary family to be used in place of DISK.

The following is an example title:

```
(SMITH)PDUMP/OBJECT/TEST/X/890105/155015/9210 ON STAFFPK
```

You can use file equations to specify a different file name or family name for a dump to disk. You can file-equate the FILENAME, FAMILYNAME, and TITLE file attributes. The file equations must specify PDUMP as the internal name of the file. For example, a WFL job can use the following statement to initiate a program and specify the title of any program dumps generated by the program. Note that the file equation has effect only if the TODISK option is specified, either in the OPTION task attribute or in the statement that invokes the program dump.

```
RUN OBJECT/JADCON;
    FILE PDUMP(TITLE = JADCON/DUMP ON PACK);
    OPTION = (FAULT, TODISK);
```

If a program dump occurs, the system adds a suffix to the file-equated title. The suffix is a 3-digit integer ranging from 000 to 999. The suffix is incremented by one for each program dump generated by the process. Thus, in the previous example, if OBJECT/JADCON runs under usercode BLAKE and generates three program dumps in a single run, the program dumps receive the following titles:

```
(BLAKE)JADCON/DUMP/000 ON PACK;
(BLAKE)JADCON/DUMP/001 ON PACK;
(BLAKE)JADCON/DUMP/002 ON PACK;
```

You can include a usercode in the PDUMP file equation, but only a privileged process can assign the program dump a usercode different from that of the process. If the process is nonprivileged, and PDUMP is equated to a different usercode, then a security violation results when a program dump occurs. The system deletes the program dump file rather than saving it under the requested usercode.

If neither the TODISK nor the TOPRINTER option is set, the operating system option PDTODISK determines whether the program dump is directed to a disk file or to the task file. If the PDTODISK option is set, program dumps are written by default to a disk file; otherwise, program dumps are directed by default to the task file. An operator can use the OP (Options) system command to set or reset the PDTODISK option.

If either the TODISK or TOPRINTER option is set for a process, the program dump is directed only to the destination specified by the option: a disk file for TODISK, or the task file for TOPRINTER. If both of these options are set, then two program dumps occur: the first is directed to disk and the second is directed to the task file.

If the TODISK and TOPRINTER options are both used, the two dumps that result can differ slightly. This is because the act of directing a program dump to disk can cause some arrays used by the process to be made present or overlayed. The contents of the arrays are not affected, and both present and overlayed arrays are included in the dump. However, if you compare both of the dumps that were produced, you might see the same array indicated as present in one dump, and overlayed in the other dump.

Using the Task File

The task file is a predeclared printer backup file that is associated with each process. If a program dump is directed to a task file, the task file is automatically queued for printing, in the same way as other printer backup files produced by a process. If a process generates multiple program dumps, then by default, they are all stored in the same task file.

You can use the TASKFILE task attribute to write comments to the task file or interrogate the file attributes of the task file. You can also use this task attribute in a program to force multiple program dumps to be stored in separate backup files. The program can achieve this effect by closing the task file after each dump and then writing a comment to the task file. An example of this method is given in the TASKFILE task attribute description in the *A Series Task Attributes Programming Reference Manual*.

A program can also use the TASKFILE task attribute to access the task file of an ancestor process.

You can assign file attributes to the task file through file equation. This task attribute can be assigned only before process initiation. The following is a WFL example of such an assignment:

```
RUN OBJECT/PROG;
  FILE TASKFILE (PRINTDISPOSITION=DONTPRINT,USERBACKUPNAME=TRUE,
                 FILENAME=PROG/DUMP);
```

You can also use the BDNAME task attribute to save the task file and assign a prefix other than *BD to the file title.

Some security restrictions apply if file equations or a BDNAME task attribute assignment is used to prefix the task file title with a usercode other than that of the process. The following are WFL examples of such statements:

```
RUN OBJECT/PROG;
  BDNAME = (FRAN)PROGDUMP;

RUN OBJECT/PROG;
  FILE TASKFILE (PRINTDISPOSITION=DONTPRINT,USERBACKUPNAME=TRUE,
                 FILENAME=(FRAN)PROGDUMP);
```

In general, a process must have privileged status to open a file under another usercode. The system enforces this rule even more strictly for task files by requiring that the process have a privileged usercode rather than merely being a privileged program. The purpose of this restriction is to prevent nonprivileged users of privileged programs from using a program dump to overwrite files under another usercode.

This restriction is not foolproof, however. If a privileged program is running under a nonprivileged usercode, and the program opens the task file with a write statement before the dump takes place, the program can successfully open the task file under another usercode. The following is an ALGOL example of such a write statement:

```
WRITE (MYSELF.TASKFILE,//,"DUMP NUMBER ONE");
```

When the program dump takes place later, the dump is directed to the already-opened task file. For this reason, if you are designing a privileged program intended for use by nonprivileged users, you should not include any statements that would cause the task file to be opened before the dump.

Analyzing a Program Dump from a Running Process

Some program dumps occur when a program is terminated, either by a fault or by a DS (Discontinue) system command. However, there can also be situations when it is useful to generate a program dump for a process while it is still running. Such a dump can be initiated by the DUMP (Dump Memory) system command or by a PROGRAMDUMP statement in the program.

By default, program dumps are directed to printer and do not print until the process and its job have terminated. The following paragraphs explain how you can gain access to the program dump while the process is still running.

One method of gaining immediate access to a program dump is by directing the program dump to disk. For information on directing dumps to disk, refer to "Controlling the Program Dump Destination" earlier in this section. If the program dump is directed to disk, then the dump file becomes available as soon as the dump is completed. You can then run the DUMPANALYZER utility to analyze the disk file. For a description of DUMPANALYZER, refer to the *A Series System Software Support Reference Manual*.

If the program dump is directed to printer, you can enable immediate printing by setting the PRINTDISPOSITION attribute of the task file to CLOSE. You can accomplish this assignment with a task equation in the statement that runs the program. The following is a WFL example:

```
RUN OBJECT/TEST/ALGOL/TASK;FILE TASKFILE(PRINTDISPOSITION= CLOSE)
```

Alternatively, you can assign the task file PRINTDISPOSITION through a FILECARDS task attribute assignment within the program. The following is an ALGOL example:

```
REPLACE MYSELF.FILECARDS BY
   "FILE TASKFILE(PRINTDISPOSITION = CLOSE);" 48"ØØ";
```

If the program dump is initiated by the DUMP command, the system closes the task file at the end of the program dump. The PRINTDISPOSITION attribute then causes the program dump to be queued for printing.

If the program dump is initiated by a PROGRAMDUMP statement in the program, the task file is *not* closed automatically at the end of the dump. To cause immediate printing, the program must follow the PROGRAMDUMP statement with a statement that closes the task file. The following is an ALGOL example:

```
CLOSE(MYSELF.TASKFILE);
```

Causing Symbolic Dumps for RPG Processes

The task file of an RPG process can store a *symbolic dump* instead of, or in addition to, a program dump. A symbolic dump provides much of the same information as a program dump, but is shorter and simpler to read. A symbolic dump can be produced in any of the following ways:

- The RPG process can execute a DUMP operation code. This operation produces a symbolic dump, but no program dump. By default, the symbolic dump is written to the task file. However, the RPG process can specify that the symbolic dump is to be written to another file previously declared by the process.
- The operator can enter the *AX DUMP* form of the AX (Accept) system command in response to a halted RPG program. This action produces a symbolic dump, but no program dump. The symbolic dump is always written to the task file.
- The RPG process can generate a program dump when the process terminates abnormally and dump options were specified in a DS (Discontinue) system command or the DSED or FAULT option was set in the OPTION task attribute. If an abnormal termination results in a program dump, a symbolic dump appears in the task file after the program dump. If there is no program dump, then no symbolic dump is produced either.

The DUMP (Dump Memory) system command, when applied to an RPG process, produces a program dump, but no symbolic dump.

For further information, refer to the discussion of the DUMP operation code in the *A Series Report Program Generator (RPG) Programming Reference Manual, Volume 1: Basic Implementation.*

Effect of Resource Limits on Program Dumps

Resource limits imposed by task attributes are deliberately overridden by the system for a process that is generating a program dump. Task attributes that might be overridden include DISKLIMIT, ELAPSEDLIMIT, MAXCARDS, MAXIOTIME, MAXLINES, MAXPROCTIME, MAXWAIT, SAVEMEMORYLIMIT, STACKLIMIT, TEMPFILELIMIT, and WAITLIMIT. This policy ensures that a process can generate a complete program dump, even when the termination is caused by the process exceeding one of these limits.

Understanding Internal and External Causes

The causes of abnormal terminations are divided into two categories: internal and external. The difference between these two types of causes can determine whether a process generates a program dump, and whether the process restarts automatically. To be more specific, the FAULT option of the OPTION task attribute causes a program dump if a process is discontinued by an internal cause. The DSED option of the OPTION task attribute causes a program dump if a process is discontinued by an external cause. The RESTART task attribute causes a process to restart only if it is terminated by an internal cause.

An abnormal termination is considered to be due to an external cause if the HISTORYCAUSE and HISTORYREASON task attributes have any of the following combinations of values:

HISTORYCAUSE	HISTORYREASON
OPERATORCAUSE	Any
FAULTCAUSE	DISKPARITYV
RESOURCECAUSE	Any
PROGRAMCAUSE	DEATHINFAMILYV, INFANTICIDEV, CLIENTDIEDINACRV, or LIBMAINTV

An abnormal termination is considered to be due to an internal cause if the HISTORYCAUSE and HISTORYREASON values are not any of the combinations listed in the preceding table.

Section 11
Restarting Jobs and Tasks

A process can be discontinued by any of a variety of causes, including system commands, program faults, or resource limits. In most situations, this discontinuation is permanent. The system does not attempt to continue execution of the process or restart it from the beginning. The system removes the process stack, process information block (PIB), task attribute block (TAB), and any temporary files that the process was using. Only the permanent files used by the process are preserved and reflect all changes made by the process before it terminated.

However, in certain cases you can cause a process to be saved for later restarting. Work Flow Language (WFL) jobs automatically restart after being terminated by a system halt/load. Processes written in other languages can invoke a checkpoint, which stores information that allows the process to be re-created later from a given point in its execution. Also, any process can be designed to restart from the beginning after encountering a program fault.

This section explains how to restart processes and how to design processes so they can be restarted successfully.

Designing WFL Jobs for Automatic Restarts

A WFL job is the only type of user process that automatically restarts if interrupted by a halt/load. If a halt/load occurs while a WFL job is executing, then the WFL job and its offspring are terminated. After the halt/load, the job recovers in one of two ways.

If the restarted WFL job was executing a checkpointed task at the time of the halt/load, then a process called JOBRESTART appears in the W (Waiting Entries) system command display. For information about how to respond to this waiting entry, refer to "Restarting a Checkpointed Task" later in this section.

If the job was not executing a checkpointed task at the time of the halt/load, the system begins execution of the job from the last point at which no offspring were in use. The following examples illustrate this point:

- Suppose that at the time of the halt/load the WFL job is waiting for a single synchronous task to complete. After the halt/load, the WFL job resumes by executing the task initiation statement again. This creates a new task that is an instance of the same program.
- Suppose that the WFL job initiates a total of three asynchronous tasks before the halt/load, and all of these tasks are still in-use when the halt/load occurs. After the halt/load, execution resumes with the first of the three task initiation statements.

- Suppose the WFL job initiated an asynchronous task called *A* and then another asynchronous task called *B*. Then task *A* terminates. Then a halt/load occurs while task *B* is still in use. After the halt/load, execution of the job resumes with the statement that initiated task *A*. This was the last point at which no in-use task existed, because task *A* still existed when task *B* was initiated.

Preventing Job Side Effects

The values of string variables declared in the WFL job are not retained across a halt/load.

The values of the task attributes of the job are also lost, except for the MIXNUMBER task attribute and any task attributes assigned in the job attribute list. Thus, for example, values assigned to any task attribute using the MYJOB task variable are not retained. Further, the effects of the ST (Stop) system command are not retained across a halt/load, because the ST system command simply assigns the STATUS task attribute a value of SUSPENDED.

The ON RESTART statement can be used to specify actions that are taken immediately after a halt/load. Typically, the ON RESTART statement is used to restore the values of string, file, and task variables before job execution continues.

Task equations included in task initiation statements are reexecuted when the task initiation statement is reexecuted. Therefore, the ON RESTART statement does not need to restore attributes specified in task equations.

The job can determine whether it has been restarted by interrogating the RESTARTED task attribute. This task attribute returns a value of TRUE if the job has been restarted.

Preventing Task Side Effects

When the WFL job reinitiates a task, the physical files used by the new task reflect any changes made by the old task before the halt/load. When designing a program that is to be initiated by a WFL job, you must plan ahead for this possibility and provide a way for the program to produce appropriate audit trails.

A WFL task that opens a remote file might not be able to do so after a halt/load. Normally, a task equation such as the following is used to enable a WFL task to open a remote file:

```
RUN OBJECT/PROG;
   STATION = MYSELF(SOURCESTATION);
```

This task equation directs the task to open any remote files at the station that initiated the WFL job. However, the requested station might not exist after a halt/load. This is the case, for example, if the job was initiated from a pseudostation, such as a Command and Edit (CANDE) dialogue opened through the Communications Management System (COMS). This pseudostation is discarded during a halt/load and is not reestablished until

you log on to the same CANDE dialogue again. The task terminates abnormally if it attempts to open a remote file at a nonexistent pseudostation.

Understanding Job Restart Failure

Any of the following circumstances can prevent a WFL job from restarting after a halt/load:

- The system switches to using a different job description file after the halt/load. The operator can use system commands to cause the switch to a different job description file. For further information, refer to the discussion of the job description file in the *A Series System Administration Guide*.
- The operator physically transfers the pack containing the job description file to a different type of system and attempts to make it the new job description file for that system.

  If the operator uses the *DL JOBS ON <family>* command to mark the pack as the location of the next job description file, then after the next halt/load, the system attempts to restart the jobs from the specified job description file. The jobs should restart successfully, provided that the pack was transferred to the same type of system with the same type of memory architecture; for example, from one A 15 system with Actual Segment Descriptor (ASD) memory to another.

  However, transfers from an A 3 to an A 10 system, and so on, are not supported and might cause the system to halt/load again.
- The operating system option AUTORECOVERY is reset. The operator can reset this option using the OP (Options) system command. Resetting AUTORECOVERY causes the mix limit for each job queue to be set to zero after a halt/load. Any job that would have restarted will instead remain in a job queue until the operator uses the MQ (Make or Modify Queue) system command to assign a new mix limit to the job queue.

  Resetting AUTORECOVERY also prevents automatic halt/loads in some situations. For details, refer to the *A Series System Commands Operations Reference Manual*.
- An operator changes the job queue definitions after the job is initiated, but before the halt/load. For example, the job attribute list of a job might set CLASS = 10 and MAXPROCTIME = 60. The definition of job queue 10 might include a PROCESSTIME limit of 120. The job is submitted through job queue 10 originally. While the job is executing, an operator might use the MQ (Make or Modify Queue) system command to lower the PROCESSTIME limit for that job queue to 30. Then a halt/load might occur. After the halt/load, the job cannot restart because its MAXPROCTIME value is greater than the PROCESSTIME limit that is now defined for job queue 10. The job terminates abnormally with a queue violation.
- A task of the job executed a checkpoint and then was terminated by the halt/load. In this case, the job is suspended after the halt/load and appears in the W (Waiting Entries) system command display. For information about operator responses to this situation, refer to "Restarting a Checkpointed Task" later in this section.

Understanding Disk Resource Control Effects

On systems using the Disk Resource Control (DRC) system, the system normally delays restarting WFL jobs until the DRC system becomes active. The WFL jobs remain in their job queues, and the system displays an RSVP message notifying the operator that DRC initialization is underway. You can use the FS (Force Schedule) system command to force a queued WFL job to restart before DRC is active. However, be aware that the following statements in a WFL job can have unexpected effects if they execute before DRC is active:

- CHANGE and REMOVE statements

  If these statements specify usercoded files, they are ignored.

- COPY statement

  If this statement creates a usercoded copy of a file, the copy proceeds normally, but DRC is not notified of the increased disk usage for that usercode. Therefore, it might become possible for the actual disk usage of that usercode to exceed the limit set in DRC.

For further information about DRC, refer to the *A Series Disk Subsystem Administration and Operations Guide*.

Manually Restarting WFL Jobs

You restart a running WFL job with the RESTART (Restart Jobs) system command. This command first discontinues the current job and its tasks, and then restarts the new job as though a halt/load had occurred. For example, the job resumes execution from the last point at which no offspring were in use. For further information about the restart point, as well as about job and task side effects that you should plan for, refer to "Designing WFL Jobs for Automatic Restarts" earlier in this section.

If the DSED program dump option is set for any task of the job, the RESTART command causes the task to generate a program dump.

You can use the RESTART command to achieve some of the effects of a halt/load without interrupting the system. For example, if you need to perform maintenance on a disk unit, you must terminate any jobs that have files open on that disk unit. You can set the mix limit for the relevant job queue to 0, then apply the RESTART command to such a job (rather than using the DS command). You can then hold the restarted job in the job queue until pack maintenance is completed. When you increase the mix limit, the job restarts from the last point where it had no tasks active.

You can also use the RESTART command to test ON RESTART statements in WFL jobs without having to halt/load the machine.

If the WFL job had no checkpointed task in progress at the time of the RESTART command, then the system automatically submits the job to a job queue. The job resumes execution whenever it is selected from that job queue.

If the WFL job was executing a checkpointed task at the time the RESTART command was entered, the job does not restart immediately after the command. Instead, the independent runner JOBRESTART appears in the W (Waiting Mix Entries) system command display. For information on how to respond to this waiting entry, refer to "Restarting Checkpointed Tasks Automatically" later in this section.

Checkpoint Facility

The checkpoint facility provides the ability to restart a terminated task from any selected point in its execution. Invoking a checkpoint causes the creation of a checkpoint file, which records the state of the task when the checkpoint was invoked. Either a statement in the task or a BR (Breakout) system command can invoke a checkpoint. Later, you can use the RERUN statement to restart the task from the point at which the checkpoint was invoked.

The main application of the checkpoint facility is the restarting of tasks that were terminated by a system halt/load. The unique advantage of the checkpoint facility is the ability to restart a task from a selected point during the task's execution. You can invoke repeated checkpoints for the same task and restart the task from any of these checkpoints.

ALGOL and COBOL(68) each provide a CHECKPOINT statement that enables a program to invoke a checkpoint during its execution. Additionally, programs can invoke a checkpoint by calling the exported MCP procedure CALLCHECKPOINT. The CALLCHECKPOINT procedure can be invoked from any of the languages that support libraries, including C, COBOL74, COBOL85, FORTRAN, FORTRAN77, NEWP, and Pascal. You can also use operator commands to initiate a checkpoint for ALGOL, COBOL(68), and COBOL74 tasks. However, checkpoints cannot be initiated for BASIC, WFL, or RPG tasks.

There are several restrictions on the circumstances in which a checkpoint can be invoked. One restriction is that the task must have been initiated from WFL, rather than from a session or a user program. Another is that the task must not have any offspring. These restrictions, and others, are discussed in detail in the following subsections.

Programmatically Invoked Checkpoints

Designing a program to be checkpointed and successfully restarted involves more than simply including a checkpoint invocation statement. You must verify that the program is not using features that are disallowed for checkpointing. You must also plan for recovery of data file contents and libraries.

Storing Information with a Checkpoint

Invoking a checkpoint causes the following types of information about the task to be stored:

- The structure of the process stack, including information about the procedures that have been entered, but have not yet exited, and the statement that is currently being executed
- The current values of all objects declared by the task
- The current values of the task attributes of the task

Planning for File Recovery

The checkpoint facility does not store a record of the contents of files used by a task. Instead, information is stored about the attributes of the files; that is, whether each file is open and the current position of the record pointer for each file. You must plan for the fact that file contents might have been modified, or files might have been removed or replaced, between the time the task was checkpointed and the time it is restarted.

When the task is restarted, each data file must be on the same type of medium as it was when the checkpoint was invoked. They do not have to be on the same physical units or at the same locations on disk. They must retain the same basic characteristics, such as blocking.

If a temporary disk file is open when the checkpoint is invoked, the file is locked and assigned a title that begins with the letters CP. However, the system does not assign this title to the TITLE attribute of the logical file; instead, the TITLE attribute retains whatever value it was assigned by the program. If this file is later locked by the program, the system enters the file in the disk directory under the title specified in the TITLE file attribute. At restart, the process looks for the file only under the CP directory, and the task is suspended with a NO FILE condition.

To prevent this situation, all files that will eventually be locked can be opened as permanent files. That is, the file attribute PROTECTION can be set to SAVE. You can design the task to remove this file later by closing the file with the PURGE option set. Another method of avoiding this problem is never to lock a temporary file.

Planning for Library Recovery

It is possible to checkpoint a user task that is linked to a library, but only if the task is not currently executing a library procedure. When a user task linked to a library is checkpointed, the checkpoint records the values of the library attributes. However, the checkpoint does not store any information about the state of the library or its contents.

When the user task restarts, the task is not immediately relinked to the library. The library link is reestablished the first time the user task calls a library procedure.

You must be aware that the values of global objects in the library might have changed since the user task was checkpointed. Global objects in the library might have changed for any of the following reasons:

- The user task might have invoked a library procedure after the checkpoint and before the user task terminated. This library procedure might have included statements that modified the values of global objects in the library.
- If the library was frozen with a duration of TEMPORARY and a sharing option of PRIVATE, then the library thaws when the user task terminates, and the system removes the library process. The values of all global objects in the library are lost when the library terminates. When the user task restarts, its first attempt to use the library causes the creation of a new instance of the library.
- If the library has a sharing option of SHAREDBYALL, then other tasks have access to the library and might make changes to global objects in the library after the original user task is checkpointed.

Invoking the Checkpoint

The task invokes a checkpoint by executing a CHECKPOINT statement or by invoking the exported MCP procedure CALLCHECKPOINT. These methods are discussed separately in the following pages.

Using a CHECKPOINT Statement

ALGOL and COBOL(68) each provide a *CHECKPOINT* statement. You can create multiple checkpoints by including a CHECKPOINT statement at several points in the program. Later, you can restart the task from any of these checkpoints.

Each CHECKPOINT statement can specify the following options:

- Device option

  Determines the family where the checkpoint-related files are to be created. A value of DISK causes checkpoint files to be created on the family named DISK. A value of DISKPACK causes checkpoint files to be created on the family named PACK. In ALGOL, but not in COBOL(68), you can specify PACK as a synonym for DISKPACK.

- Disposition option

  Determines whether checkpoint files are saved. If the value is PURGE, then the checkpoint files are removed if the task terminates normally. If the value is LOCK, then checkpoint files are saved indefinitely. Later, you can use the checkpoint files to restart the task even if it terminated normally.

  The disposition option also determines if a checkpoint removes any previous checkpoint files created by the same task. If the disposition is PURGE, then any previous checkpoints that were invoked with a disposition of PURGE are removed. If the disposition is LOCK, then no previous checkpoints are removed.

- Exception action option

  This option specifies that some particular statement is to be executed if the checkpoint is not successful. This option is available only in COBOL(68). In ALGOL, other means are used to determine if the checkpoint was successful.

Examples

The following is an ALGOL example:

```
CHECKPOINT (DISK,PURGE);
```

The following is a COBOL(68) example:

```
CHECKPOINT TO DISKPACK WITH LOCK; ON EXCEPTION GO P2.
```

Using the CALLCHECKPOINT Procedure

A program can invoke a checkpoint by calling the MCP exported procedure CALLCHECKPOINT. You can create multiple checkpoints by invoking CALLCHECKPOINT at several points in the program. Later, you can restart the task from any of these checkpoints.

CALLCHECKPOINT is an integer procedure that receives four integer parameters, in the following order: UTYP, CPTYP, CCODE, CPNUM, and RSFLAG. The following table explains these parameters.

Parameter Name	Type	Input/Output	Meaning
UTYP	Integer	Input	Similar to the device option in a CHECKPOINT statement. The UTYP parameter determines the family where the checkpoint-related files are to be created. A value of 1 causes checkpoint files to be created on the family named DISK. A value of 17 causes checkpoint files to be created on the family named PACK. These values can also be represented by the VALUE function in ALGOL as VALUE(DISK) and VALUE(PACK).
CPTYP	Integer	Input	Similar to the disposition option in a CHECKPOINT statement. The CPTYP parameter determines whether checkpoint files are saved. A value of 0 is the same as a disposition of PURGE: the checkpoint files are removed if the task terminates normally. A value of 1 is the same as a disposition of LOCK: the checkpoint files are always saved indefinitely. Later, you can use the checkpoint files to restart the task even if it terminated normally.

continued

continued

Parameter Name	Type	Input/Output	Meaning
			The CPTYP parameter also determines if a checkpoint removes any previous checkpoint files created by the same task. If the disposition is 0 (PURGE), then any previous checkpoints that were invoked with a disposition of 0 or PURGE are removed. If the disposition is 1 (LOCK), then no previous checkpoints are removed.
CCODE	Integer	Output	If the checkpoint is unsuccessful, the CCODE parameter stores one of the values listed in Table 11-1, "Checkpoint Completion Codes."
CPNUM	Integer	Output	CPNUM returns the number that the system assigned to this checkpoint. The numbering scheme is explained under "Creating Output Disk Files with a Checkpoint" later in this section.
RSFLAG	Integer	Output	If the task was restarted from a checkpoint, then RSFLAG returns a value of 1 the next time the task invokes the CALLCHECKPOINT procedure. In this case, CALLCHECKPOINT actually does not invoke a checkpoint for the task. If the task invokes CALLCHECKPOINT a second time, RSFLAG returns a value of 0 and the checkpoint is actually invoked.
Procedure result	Integer	Output	A value of 0 indicates a successful checkpoint. A value of 1 indicates that the checkpoint was not taken, in which case either the CCODE parameter or the RSFLAG parameter should be nonzero.

The following are ALGOL statements that declare the CALLCHECKPOINT procedure and invoke it:

```
LIBRARY MCPSUPPORT(LIBACCESS=BYFUNCTION,FUNCTIONNAME="MCPSUPPORT.");

INTEGER PROCEDURE CALLCHECKPOINT(UTYP, CPTYP, CCODE, CPNUM, RSFLAG);
  INTEGER UTYP, CPTYP, CCODE, CPNUM, RSFLAG;
  LIBRARY MCPSUPPORT;

INTEGER CCODE_ACTUAL, CPNUM_ACTUAL, RSFLAG_ACTUAL, CPRESULT;

CPRESULT := CALLCHECKPOINT(VALUE(DISK),1, CCODE_ACTUAL, CPNUM_ACTUAL,
  RSFLAG_ACTUAL);
```

The following COBOL85 program uses the explicit library interface to invoke the CALLCHECKPOINT procedure. The invocation specifies a device option of PACK and a disposition of PURGE.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CHECK-POINT.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT ATTR-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD ATTR-FILE.
Ø1 ATTR-REC PIC X(8Ø).

WORKING-STORAGE SECTION.
77 CHECKPOINTDEVICE PIC S9(11) USAGE BINARY.
77 CHECKPOINTTYPE   PIC S9(11) USAGE BINARY.
77 COMPLETIONCODE   PIC S9(11) USAGE BINARY.
77 CHECKPOINTNUMBER PIC S9(11) USAGE BINARY.
77 RESTARTFLAG      PIC S9(11) USAGE BINARY.
77 RSLT             PIC S9(11) USAGE BINARY.
77 VALUE-OF-PACK    PIC S9(11) USAGE BINARY.
77 VALUE-OF-PURGE   PIC S9(11) USAGE BINARY VALUE Ø.

LOCAL-STORAGE SECTION.
LD LD-CALLCHECKPOINT.
77 CHECKPOINTDEVICE PIC S9(11) USAGE BINARY.
77 CHECKPOINTTYPE   PIC S9(11) USAGE BINARY.
77 COMPLETIONCODE   PIC S9(11) USAGE BINARY.
77 CHECKPOINTNUMBER PIC S9(11) USAGE BINARY.
77 RESTARTFLAG      PIC S9(11) USAGE BINARY.
77 RSLT             PIC S9(11) USAGE BINARY.

PROGRAM-LIBRARY SECTION.
LB MCPSUPPORT IMPORT
   ATTRIBUTE
   FUNCTIONNAME IS "MCPSUPPORT"
   LIBACCESS    IS BYFUNCTION.
ENTRY PROCEDURE CALLCHECKPOINT
   WITH   LD-CALLCHECKPOINT
   USING
          CHECKPOINTDEVICE
          CHECKPOINTTYPE
          COMPLETIONCODE
          CHECKPOINTNUMBER
          RESTARTFLAG
   GIVING
          RSLT.
```

```
PROCEDURE DIVISION.
INIT-PARA.
       CHANGE ATTRIBUTE KIND         OF ATTR-FILE
                                     TO PACK.
       MOVE   ATTRIBUTE KIND         OF ATTR-FILE
                                     TO VALUE-OF-PACK.
       PERFORM CHECKPOINT-PARA.
       STOP RUN.

CHECKPOINT-PARA.
       MOVE VALUE-OF-PACK  TO CHECKPOINTDEVICE.
       MOVE VALUE-OF-PURGE TO CHECKPOINTTYPE.
       CALL CALLCHECKPOINT
       USING
              CHECKPOINTDEVICE
              CHECKPOINTTYPE
              COMPLETIONCODE
              CHECKPOINTNUMBER
              RESTARTFLAG
       GIVING
              RSLT.
```

Creating Output Disk Files with a Checkpoint

Invoking a checkpoint causes the creation of one or more of the following checkpoint-related disk files: a checkpoint file, a checkpoint job file, and checkpoint temporary files. The following paragraphs explain what these files are, and the factors that determine whether they are created.

A checkpoint file is always created if the checkpoint is successful. This file stores a complete description of the checkpointed task and is titled according to the following convention:

```
(<usercode>)CP/<job number>/<checkpoint number> ON <family name>
```

The usercode in the checkpoint file title is the usercode of the task. The job number is the 4-digit mix number of the job that initiated the task. The checkpoint number is a 3-digit number used to distinguish this checkpoint from any other checkpoints executed by the same task. The family name is taken from the value of the device option in the CHECKPOINT statement.

If the disposition option is set to PURGE, the checkpoint number is always 0 (zero) and each succeeding checkpoint with PURGE set removes the previous checkpoint file. If the disposition option is set to LOCK, the checkpoint number starts at a value of 1 for the first checkpoint, and is incremented by 1 for each succeeding checkpoint that is invoked with LOCK. If a task invokes some checkpoints with LOCK and some with PURGE, then the "locked" checkpoints use ascending checkpoint numbers and the "purged" checkpoints use a checkpoint number of 0.

A checkpoint job file is produced by the first checkpoint in the task that is invoked with LOCK. This checkpoint job file makes it possible to restart the checkpointed task after its original job has terminated. Later checkpoints with LOCK do not produce a job file, nor do any checkpoints invoked with PURGE. The checkpoint job file is titled according to the following convention:

```
(<usercode>)CP/<job number>/JOBFILE ON <family name>
```

Checkpoint temporary files store the contents of temporary files in use by the checkpointed task. Checkpoint temporary files are created only in certain circumstances, which are discussed under "Planning for File Recovery" earlier in this section. These files are titled according to the following convention:

```
(<usercode>)CP/<job number>/F<file number> ON <family name>
```

In the checkpoint temporary file title, the file number is a 3-digit file number that starts at 1 and is incremented by 1 for each temporary disk file.

The way the checkpointed task terminates can have an effect on the checkpoint-related files. If the checkpointed task terminates abnormally and the last checkpoint has a disposition of PURGE, the system retitles the checkpoint file to have the next sequential checkpoint number and creates a checkpoint job file if none exists. If the checkpointed

task terminates normally and all checkpoints have a disposition of PURGE, then the system removes all checkpoint-related files that were created for the task.

For tasks that invoke a large number of checkpoints with the LOCK disposition, the checkpoint number is incremented up to 999 and then is recycled to 1 (leaving 0 undisturbed). When this happens, the checkpoint files previously numbered 1, 2, and so on are lost as new ones using those numbers are created.

When a task restarts at a checkpoint that was not the last, subsequent checkpoints invoked from the restarted task continue in numerical sequence from the one used for the restart. Old high-numbered checkpoints are thus lost.

If a rerun is initiated and the original job number is in use by another task, then a new job number is assigned to the job. The titles of all checkpoint-related files for the task are changed to reflect the new job number.

Restrictions on the Use of Checkpoints

There are certain restrictions on the features that can be in use by a task when it is checkpointed. These restrictions apply to both programmatically initiated checkpoints and operator initiated checkpoints. If any of these features are in use, they prevent a successful checkpoint. If a checkpoint fails, the task continues normally, but no checkpoint files are created.

The following restrictions apply to the tasking environment of the checkpointed task:

- The task must have have been initiated by a RUN statement in a WFL job. The checkpointed task must be the only in-use offspring of the WFL job at the time the checkpoint is invoked.
- The task cannot be a remote task. That is, it must not be initiated on a BNA host system other than that on which the job is running.
- The task must not have any offspring at the time of the checkpoint. However, the task can have offspring at earlier, or later, points in its execution.

A checkpoint cannot be invoked from within the following types of procedures:

- An imported library procedure. The checkpoint cannot take place if an imported library procedure is anywhere in the process stack. However, the checkpoint can be invoked if the user process is merely linked to a library.
- A SORT input or output procedure. (SORT provides its own restart capability; refer to the *A Series System Software Utilities Operations Reference Manual*.)
- A USE procedure in COBOL(68).

Several types of files cannot be open at the time of the checkpoint. However, the process can close these files, take a checkpoint, and then reopen the files and continue to use them. The following are the restricted types of files:

- Direct files
- Duplicated files

- Files whose FILESTRUCTURE attribute value is not ALIGNED180
- ISAM files
- Multireel unlabeled tape files
- ODT files
- Remote files
- Paper tape files
- Port files
- Reversed tape files

Some restrictions also apply to printer output. No backup files that have a PRINTDISPOSITION file attribute value of CLOSE, DIRECT, or EOT can be open. In addition, the BDBASE option of the OPTION task attribute cannot be set.

The process cannot have an open Data Management System II (DMSII) set at the time of the checkpoint.

No direct arrays can be in the process stack at the time of the checkpoint. A direct array can be declared in a procedure in the program. However, the procedure must not have been entered, or must have been entered and exited, before the checkpoint.

The checkpoint file cannot be created if doing so would cause the user's file usage on a family to exceed the limits enforced for the user by the disk resource control (DRC) system. For information about the disk resource control system, refer to the *A Series Disk Subsystem Administration and Operations Guide.*

Determining Eligibility for Checkpoints

A task can determine whether it is probably eligible for checkpoints by interrogating the CHECKPOINTABLE task attribute. This read-only Boolean attribute is assigned by the system. The system assigns a value of FALSE if the task does not meet certain basic requirements for a checkpointed task. However, a CHECKPOINTABLE value of TRUE does not guarantee that a checkpoint will succeed. For details, refer to the discussion of CHECKPOINTABLE in the *A Series Task Attributes Programming Reference Manual.*

Determining Whether the Checkpoint Succeeded

The checkpoint facility returns a value indicating the result of the attempted checkpoint. This value is divided into the following fields:

[46:01]	If this bit is set, then the current task was restarted from this checkpoint.
[25:12]	If the checkpoint succeeded, this field stores the checkpoint number assigned to the checkpoint files.
[10:10]	If the checkpoint failed, this field stores the completion code that indicates why the checkpoint failed. For a list of the possible completion codes, refer to Table 11–1.
[00:01]	This is the exception bit. If this bit is set, then either the checkpoint did not succeed or the process was restarted from this checkpoint.

In ALGOL, you can store the result value in a Boolean variable by invoking the checkpoint facility as a function. This Boolean value can then be stored in a real variable, and the various fields of the real variable can be conveniently interrogated.

In the following ALGOL example, BOOL is a Boolean variable and the other variables are real.

```
BOOL := CHECKPOINT(PACK,LOCK);
REALRSLT := REAL(BOOL);
CPRESTART := REALRSLT.[46:Ø1];
CPNUMBER := REALRSLT.[25:12];
CPCOMPLETION := REALRSLT.[1Ø:1Ø];
CPEXCEPTION := REALRSLT.[ØØ.Ø1];
```

In COBOL(68), the result value is automatically stored in a special register called CHECKPOINT-STATUS. This is a predefined Level-2 variable that stores the result value for the most recent checkpoint statement. If the task was restarted from this checkpoint, then CHECKPOINT-STATUS stores a negative value. You can use MOVE statements to extract the values of the checkpoint number, completion code, and exception fields.

The following example copies the value of CHECKPOINT-STATUS into a variable called CPSTATUS. The example then extracts the values from various fields of CPSTATUS and stores them in four separate variables. The variables CPSTATUS, CPRESTART, CPNUMBER, CPCOMPLETION, and CPEXCEPTION were all declared as 77-level COMP PIC 9(11).

```
CHECKPOINT TO DISK WITH LOCK.
MOVE CHECKPOINT-STATUS TO CPSTATUS.
MOVE CPSTATUS TO CPRESTART [ 46:00:01 ].
MOVE CPSTATUS TO CPNUMBER [ 25:11:12 ].
MOVE CPSTATUS TO CPCOMPLETION [ 10:09:10 ].
MOVE CPSTATUS TO CPEXCEPTION [ 00:00:01 ].
```

You can tell whether a checkpoint was successful by observing the completion message that is displayed. The following is an example of a successful completion message:

```
#1082 CHECKPOINT #1080/001 TAKEN  @ F54:004E:0 @ (029000)*
```

The following is an example of the completion message for a checkpoint that failed:

```
#1111 CHECKPOINT ABORTED: BAD IPC ENVIRONMENT @ (029000)*
```

Each completion message corresponds to one of the completion codes from field [10:10] of the checkpoint result. The completion messages are listed in Table 11–1.

Table 11–1. Checkpoint Completion Codes

Completion Code	Completion Message and Meaning
0	CHECKPOINT #<mix number>/<checkpoint number> TAKEN
	The checkpoint was executed successfully.
1	INVALID AREA IN STACK
2	SYSTEM ERROR
	Completion errors 1 and 2 both mean that a system error occurred.
3	BAD IPC ENVIRONMENT
	The process has offspring or was not initiated by a WFL *RUN* statement.
4	NO USER DISK FOR CP FILE
	The family requested by the device option in the checkpoint statement is not available.
5	IO ERROR DURING CHECKPOINT
	An I/O error occurred.

continued

Table 11–1. Checkpoint Completion Codes (cont.)

Completion Code	Completion Message and Meaning
6	# ROWS IN CP FILE > 1024 The process is too large to be successfully checkpointed.
7	DIRECT FILE NOT ALLOWED The process has a direct file that is open.
8	TOO MANY TEMPORARY DISK FILES The process has more than 998 temporary files.
9	ILLEGAL FILEKIND The process is using a file for writing directly to a line printer or a card punch.
10	DUPLICATED FILE NOT ALLOWED The process is using a duplicated file.
11	ILLEGAL FILE ORGANIZATION The process is using an Index Sequential Access Method (ISAM) file.
12	INSUFFICIENT MEMORY TO CHECKPOINT Not enough memory is available to checkpoint the process.
13	OPEN REVERSED TAPE FILE NOT ALLOWED The process is using a reversed tape file.
14	ICM AREA IN STACK The process is using a BNA Version 2 port file.
15	DMS AREA IN STACK The process is using a DMSII data set.
16	DIRECT ARRAY IN STACK The process has entered, and not yet exited, a block that includes a direct array declaration.
17	SECURITY ERROR SAVING TEMPORARY DISK FILE The process has a temporary file open under another usercode. This situation can occur, for example, if both the following are true: • The process opened a permanent disk file that resided under someone else's usercode. • While the process had the file open, another process attempted to remove the file, thus changing it to a temporary file.

continued

Table 11–1. Checkpoint Completion Codes (cont.)

Completion Code	Completion Message and Meaning
19	STACKMARK
	A system error occurred.
20	SORT AREA IN STACK
	The process is using the SORT function.
21	IN USE ROUTINE NOT ALLOWED
	The process has entered a USE procedure.
22	ILLEGAL CONSTRUCT
	Either the process has opened a port file or there is operating system code in the process stack. The latter can occur, for example, when a fault causes the execution of an ALGOL *ON* statement.
23	BDBASE ILLEGAL
	The BDBASE option of the OPTION task attribute has been set.
24	ILLEGAL FILESTRUCTURE
	The process has an open file with a FILESTRUCTURE value for which checkpointing is not implemented.
25	MULTI-REEL UNLABELED TAPE NOT ALLOWED
	The process has opened a multireel unlabeled tape file.
26	SURROGATE TASK NOT ALLOWED
	The task was initiated on a BNA host other than that on which the job is running.
28	PROGRAM USES LIBRARIES
	The process is executing an imported library procedure.
30	ROW SIZE TOO SMALL FOR CP FILE
	The process stack is too large to fit in a row of the checkpoint file. The maximum size of a process stack that can be checkpointed is approximately 22700 words.
32	OPERATOR CHECKPOINT REQUEST CANCELED
	A checkpoint or restart was already underway.
34	BR REQUEST REJECTED
	The BRCLASS task attribute value is NOBR.
36	OPEN BACKUP FILE WITH PRINTDISPOSITION = EOT NOT ALLOWED
37	OPEN BACKUP FILE WITH PRINTDISPOSITION = CLOSE NOT ALLOWED

continued

Table 11–1. Checkpoint Completion Codes (cont.)

Completion Code	Completion Message and Meaning
38	OPEN BACKUP FILE WITH PRINTDISPOSITION = DIRECT NOT ALLOWED Each of these three values means that the PRINTDISPOSITION file attribute has a value not allowed for checkpoints.
40	ATTEMPT TO EXCEED TEMPORARY FILE LIMIT ON CP FILE
41	ATTEMPT TO EXCEED FAMILY LIMIT ON CP FILE
42	FAMILY INTEGRAL LIMIT EXCEEDED ON CP FILE These three values mean that the checkpoint file cannot be created because doing so would cause the user's file usage on a particular family to exceed the limits set by the disk resource control system. For information about the disk resource control (DRC) system, refer to the *A Series Disk Subsystem Administration and Operations Guide*.
43	CHECKPOINT ABORTED: INVALID ENVIRONMENT IN STACK The process has invoked and not yet exited a library procedure. Either the process is currently executing code from that procedure, or it is executing code from some other procedure that was invoked from the library procedure.
44	CHECKPOINT ABORTED: DISK TYPE MUST BE DISK OR PACK A process invoked the exported MCP procedure CALLCHECKPOINT and passed a value of other than 1 (disk) or 17 (pack) to the UTYP parameter.
45	CHECKPOINT ABORTED: CHECKPOINT TYPE MUST BE ZERO OR ONE A process invoked the exported MCP procedure CALLCHECKPOINT and passed a value of other than 0 (purge) or 1 (lock) to the CPTYP parameter.

Operator-Invoked Checkpoints

You can initiate checkpoints for a task by using the BR (Breakout) system command. This feature is designed to allow you to checkpoint tasks when an external condition prevents execution from continuing. For example, checkpointing a task just before halt/loading the system preserves the work done up to that point.

The BR command, if it is completed successfully, has the same effect as a CHECKPOINT statement in a program. All restrictions that apply to a programmed checkpoint also apply to an operator-initiated checkpoint. For example, the task must have been initiated from a WFL job and must not have any offspring. The task must be written in ALGOL, COBOL(68), or COBOL74. For details about these restrictions, refer to "Restrictions on the Use of Checkpoints" earlier in this section.

The programmer is responsible for designing a task to recover data file contents and libraries after a restart. It can be difficult to design such recovery mechanisms without

knowing exactly when the checkpoint will take place. You can overcome this difficulty through the use of the BRCLASS task attribute. This task attribute specifies whether the task currently allows an operator-invoked checkpoint.

Programmatically Preventing Operator Checkpoints

You can use the BRCLASS task attribute to specify how a task will respond to an operator-invoked checkpoint. Through the use of repeated assignments to BRCLASS, you can specify that operator checkpoints are allowed at some points in the task and not at others.

You can disallow operator checkpoints by assigning BRCLASS a value of NOBR (the default). You can allow a single checkpoint, and cause the task to be discontinued automatically after the checkpoint, by assigning BRCLASS a value of ONCEONLY. You can allow multiple checkpoints, and allow the task to continue normally after each checkpoint, by assigning BRCLASS a value of MULTIPLE.

Note: Multiple operator checkpoints are possible only if BRCLASS is set to MULTIPLE for both the job and the task. It is not sufficient to assign MULTIPLE to the task alone.

The BRCLASS task attribute has effect only if the CHECKPOINTABLE task attribute is TRUE.

Displaying the Checkpoint Status

You can use the *<mix number> BR* system command to determine whether a task is eligible for an operator checkpoint and whether the task is currently being checkpointed or restarted. The response has the following form:

```
TASK <mix number> <checkpoint status>
```

The following are possible responses if the task is not currently being checkpointed or restarted. The phrase "CANNOT CONTINUE AFTER BR" indicates that the BRCLASS task attribute has a value of ONCEONLY. When BRCLASS = ONCEONLY, the system automatically discontinues the process after the checkpoint completes. However, the process can continue if the operator cancels the checkpoint with an OF (Optional File) system command, as discussed under "Operator Actions after the Checkpoint" later in this section.

```
TASK <mix number> IS NOT CHECKPOINTABLE BY THE OPERATOR
TASK <mix number> IS CHECKPOINTABLE
TASK <mix number> IS CHECKPOINTABLE (CANNOT CONTINUE AFTER BR)
```

The following are responses that indicate that a checkpoint has been requested or is underway:

```
TASK <mix number> CHECKPOINT REQUESTED
TASK <mix number> CHECKPOINT REQUESTED (CANNOT CONTINUE AFTER BR)
TASK <mix number> CHECKPOINT RUNNING
TASK <mix number> CHECKPOINT RUNNING (CANNOT CONTINUE AFTER BR)
```

The following responses indicate that a restart is underway. In these responses, the phrase "PROGRAM" means that the checkpoint was initiated by a CHECKPOINT statement in the task. The phrases "ONCEONLY" and "MULTIPLE" specify the value of the BRCLASS task attribute in cases where the checkpoint was operator initiated.

```
TASK <mix number> : RESTARTING
TASK <mix number> : RESTARTING (PROGRAM)
TASK <mix number> : RESTARTING (ONCEONLY)
TASK <mix number> : RESTARTING (MULTIPLE)
```

The Y (Status Interrogate) system command displays more limited information about the checkpoint status of a task. If a checkpoint or restart action has been requested for a task, a line of the following form appears in the Y display:

```
CHECKPOINT STATUS : <status>
```

The possible <status> values are REQUESTED, RUNNING, or RESTARTING. These values have the same meaning they do in the BR display.

Invoking a Checkpoint Interactively

You can invoke a checkpoint for a task by entering the *<mix number> BR +* form of the BR command. If the checkpoint is accepted, it is executed with DISK as the device option and PURGE as the disposition option. Checkpoint files are therefore created on DISK family, with a checkpoint number of 0.

If the BRCLASS task attribute value is ONCEONLY, the task is discontinued after the checkpoint.

The system might delay execution of the checkpoint request if it is not immediately able to save the task stack correctly. For example, a checkpoint request cannot be completed while the task is waiting on an event. If the checkpoint request is being delayed, a BR command shows a checkpoint status of REQUESTED.

Canceling a Checkpoint Interactively

If the checkpoint status is REQUESTED, you can cancel the checkpoint request by entering a BR command of the form *<mix number> BR –*. This command cancels the request immediately.

Operator Actions after the Checkpoint

As soon as the checkpoint has been taken successfully and the checkpoint file is entered into the directory, the checkpoint function waits for an operator action. The following is an example of the W (Waiting Mix Entries) system commands display for such a process:

```
---Job-Task-Pri---Elapsed------- 5 WAITING ENTRIES --------------------
  6927\6928  50      1:44 (JASMITH) (JASMITH)OBJECT/ALGOL/CP ON SYSPK
       OPERATOR CHECKPOINT #6927/000 TAKEN  @ 112F:00EA:1 @ (00000500)
```

You can determine the possible responses to this waiting state by entering a *<mix number> Y* command. The REPLY line of the Y command display lists one or more of the following possible responses:

- <mix number> DS

  This command immediately discontinues the checkpointed task and its job. Any user protection, such as EPILOG procedures, will not be considered during the DS operation. This restriction ensures that neither the job nor the task will change the state of any of its files after the checkpoint has been taken. This response is always available after an operator-initiated checkpoint.

- <mix number> OF

  This command cancels the checkpoint, removes the files created by the checkpoint, and causes the checkpointed task and its job to continue their normal execution. This response can be used if you decide that the checkpoint was not needed.

- <mix number> OK

  This command causes the system to complete the checkpoint, and causes the checkpointed task and its job to continue execution normally after the checkpoint. Any files created by the checkpoint are saved. This response is allowed only if the BRCLASS task attribute value was MULTIPLE.

Additionally, if the task was checkpointed in preparation for a halt/load, the ??PHL (Programmatic Halt Load) system command can be used to initiate the halt/load. After the halt/load, you can enter a command to restart the task.

Restarting a Checkpointed Task

A checkpointed task can be restarted automatically after a halt/load or explicitly with a WFL *RERUN* statement.

Restarting Checkpointed Tasks Automatically

If a WFL job was executing a checkpointed task when a halt/load occurred, the job does not immediately restart after the halt/load. Instead, an independent runner called JOBRESTART appears in the W (Waiting Entries) system command display. The following is an example of the entry:

```
---Mix-Pri---Elapsed------------ 2 WAITING ENTRIES -----------------
  7Ø82  5Ø       :55  JOB  JOBRESTART
       RESTART PENDING 7119 DAILY/RUNNIT
```

In this example, DAILY/RUNNIT is the job that is pending restart, and its mix number is 7119. The checkpointed task does not appear in the display. JOBRESTART is a job that was initiated by the system software to do the restarting, and its mix number is 7082. The following are the possible operator responses to this example and the effects of the responses:

- 7082 OK

  This command restarts the job and restarts the task at the last checkpoint.

- 7082 DS

  This command discontinues the job and the task. Any checkpoint files are saved, regardless of whether the disposition was LOCK or PURGE. The checkpoint number of PURGE files is left as 0 (zero).

- 7082 QT

  In this context, QT has the same effect as DS.

Initiating a Restart Explicitly

You can use a WFL *RERUN* statement to restart a checkpointed task. The checkpoint files of the task to be restarted must have been permanently saved. Checkpoint files are permanently saved if the checkpoint disposition is LOCK, if the job terminates abnormally, or if the checkpoint is initiated by an operator BR command.

The RERUN statement can be included in a WFL job. Also, you can enter the RERUN statement directly at the ODT, in which case the RERUN statement causes the creation of a WFL job that does the restart. The RERUN statement has the following form:

```
RERUN <job number> / <checkpoint number>
```

In the RERUN statement, the job number is the mix number of the job that initiated the checkpointed task. The checkpoint number identifies the checkpoint that is to be used.

If the checkpointed task had a usercode, the checkpoint files are stored under that usercode. To restart such a task, you must enter the RERUN command in a job that specifies the usercode. The following can then be entered at an ODT:

```
?BEGIN JOB;USERCODE = <usercode> / <password>;
  RERUN <job number> / <checkpoint number>
```

Following are some of the conditions that can prevent a successful restart:

- The usercode of the checkpointed task or its job is no longer valid.
- The program has been recompiled since the checkpoint was created.
- The system is now running on a different MCP release level than it was when the checkpoint was created. For example, the system is now running a 4.0 MCP, and the checkpoint was created on a system running a 3.8 MCP.
- The system is now using different intrinsics from when the checkpoint was taken.
- The checkpoint files are not present on DISK family or PACK family. The files must be on one of these two families, regardless of any FAMILY equations entered with the RERUN statement.
- The process was restarted on a different type of machine from the one where the checkpoint was taken. For example, the process was checkpointed on an A 3 and restarted on an A 9.

If a rerun is initiated and the job number is in use by another job, a new job number is assigned and the checkpoint files are automatically retitled to reflect the new job number.

Table 11–2 lists the messages that can be displayed to show the result of the restart attempt.

Table 11–2. Restart Messages

Message Text
RESTART PENDING
RESTART INITIATED
RESTART ABORTED: MISSING CHECKPOINT FILE
RESTART ABORTED: IO ERROR DURING RESTART
RESTART ABORTED: USERCODE NO LONGER VALID
RESTART ABORTED: OPERATOR DSED RESTART
RESTART ABORTED: OPERATOR QTED RESTART
RESTART ABORTED: MISSING CODE FILE
RESTART ABORTED: NOT ABLE TO RESTART

continued

Table 11–2. Restart Messages (cont.)

Message Text
RESTART ABORTED: INVALID JOB FILE
RESTART ABORTED: ERR COPYING JOB FILE
RESTART ABORTED: MISSING JOB FILE
RESTART ABORTED: FILE POSITIONING ERROR
RESTART ABORTED: WRONG JOB FILE
RESTART ABORTED: WRONG CODE FILE
RESTART ABORTED: BAD CHECKPOINT FILE
RESTART ABORTED: BAD STACK NUMBER
RESTART ABORTED: WRONG MCP
RESTART ABORTED: MISSING FAMILY MEMBER
RESTART ABORTED: MACHINE TYPES DIFFER
RESTART ABORTED: PAGED ARRAY PAGE SIZE HAS CHANGED
RESTART ABORTED: FILE IS RESTRICTED
RESTART ABORTED: FILE IS ON A RESTRICTED FAMILY
RESTART ABORTED: TAPE LABELKIND CONFLICTS WITH FILEUSE

Automatic Retries

You can design a process to be restarted automatically if it is terminated because of an error. This effect is achieved by assigning a value to the RESTART task attribute. The value of this task attribute specifies the number of times the process is to be restarted following an error termination. Execution of the restarted process begins with the first statement in the outer block. After each restart, the RESTART task attribute value is decremented by one. When the RESTART value is zero, the next error termination is final.

When the process is restarted, no "EOJ" or "EOT" messages are displayed. Some elements of the process survive the error termination and are reused. These elements are the PIB, the code segment dictionary, and the base of the process stack. All task attribute values of the original process are retained, including the mix number. In addition, the values of any parameters the process received from its initiator are saved.

However, the values of all objects declared by the process are lost. These include all variables, arrays, and so on, that are declared in the process. These objects are re-created and reinitialized after the process restarts.

If the process has tasks, they are discontinued with a "PARENT PROCESS TERMINATED" error each time the process has an error termination. However, each time the process is restarted, it can execute the task initiation statements again and create new tasks.

A process that was discontinued by an operator command does not restart, regardless of the value of the RESTART attribute. The RESTART value also does not cause a process to be restarted after a halt/load.

The RESTART task attribute is primarily useful in situations where the process might be discontinued by a temporary hardware fault or where the process will receive different input data after it restarts. If the process is attempting to do something invalid or contradictory, repeated restarts are not helpful. The process terminates abnormally each time.

If the process includes a statement that assigns a value to RESTART, make sure that the statement is not reexecuted after each restart. If the statement is always reexecuted, then the value of RESTART can never reach zero and the process restarts infinitely. The following is an example of an ALGOL program that would enter such a loop:

```
100 BEGIN
200   REAL X;
300   MYSELF.RESTART := 4;
400   X := X DIV 0;
500 END.
```

The following example shows how the program could be modified so that it would not enter an infinite loop. The SW1 task attribute is used as a flag to indicate whether the process has been executed at least once.

```
100 BEGIN
200   REAL X;
300   IF NOT MYSELF.SW1 THEN MYSELF.RESTART := 4;
400   MYSELF.SW1 := TRUE;
500   X := X DIV Ø;
600 END.
```

In ALGOL, you can use the ON statement to prevent an abnormal termination from occurring after a program fault. The ON statement has fewer applications than the RESTART task attribute because it applies only to errors that would otherwise cause the process to be discontinued with HISTORYCAUSE = FAULTCAUSE. However, for these cases, the ON statement provides more flexible error handling than the RESTART task attribute.

For more information about the ALGOL *ON* statement, refer to "Designing a Program to Survive Faults" in Section 10, "Determining Process History."

Section 12
Tasking across Multihost Networks

The linking of systems into a multihost network provides the capability for a type of distributed processing. Each process executes on a single host system. However, the various members of a process family can run on different host systems and can communicate with each other in most of the same ways they could if they were all running on the same system.

This type of distributed processing is referred to as *remote tasking* and is provided by Host Services software. Remote tasking is supported across BNA Version 1, BNA Version 2, and Open Systems Interconnection (OSI) networks.

This section uses some specialized terminology to discuss remote tasking. The term *remote process* is used to refer to a process that is initiated from one host system, but runs on another host system. The host from which the remote process is initiated is referred to as the *local host.* The host at which the remote process runs is referred to as the *remote host.*

In the same way, the *local operator* is an operator at the system from which the remote process is initiated. The *remote operator* is an operator at the system where the remote process runs.

A remote process can be initiated from programs or from interactive sources such as the operator display terminal (ODT), a Command and Edit (CANDE) session, or a Menu-Assisted Resource Control (MARC) session. Any messages generated by the process are routed back to the local ODT and originating terminal. You can monitor and control the remote process by transmitting ODT commands to the remote host system.

The following are reasons why you might want to initiate a remote process:

- To equalize the processor load on the various systems at an installation. If the local system is overloaded, a process may be able to run more quickly at a remote host.
- To make use of a program that is stored at a remote host. A process must run on the same system where the object code file is stored. Therefore, initiating a program that is stored at another system implies the creation of a remote process.
- To more efficiently access files that are stored on a remote host. A remote process running on the remote host can access these files more efficiently than a local process that accesses the files using Host Services Logical I/O. The result can be savings in I/O time and elapsed time.

For further information about Host Services, other than remote tasking, refer to the *A Series Distributed Systems Service (DSS) Operations Guide.* For additional information about Host Services Logical I/O, refer to the *A Series I/O Subsystem Programming Guide.*

Submitting Remote WFL Jobs

Any Work Flow Language (WFL) job can be designed to run on a remote host. Additionally, a local operator can initiate jobs that are stored on remote hosts.

Running a Local WFL Job on a Remote Host

In some cases, it might be convenient to store a WFL job source program on the local host, even though the job is to be run on a remote host. For these cases, you can include an *AT <hostname>* specification at the start of the job.

You can submit the WFL job for execution by entering a START command at the local host. If the *AT hostname* specification in the job requests a hostname that is not currently available, the system rejects the job and displays the message "UNKNOWN HOST SPECIFIED". If the requested hostname is available, the system transfers the job to the remote host. The entry "JOB/HANDLER/<local hostname>" appears in the mix at the remote host and indicates that a job has been transferred to the remote host. The job compilation, job queuing, and job execution all take place at the remote host.

If the *AT <hostname>* phrase is used, the job cannot include a job parameter list, any BINARY data specifications, or a null character within a quoted string. Also, if the WFL source program is stored in a disk file, a question mark must be included before the END JOB statement. If the WFL source program is submitted in array form, it should not include any strings with embedded null characters; otherwise, the job receives a syntax error at the remote host.

The following is an example of the job heading for a job that is to run on a remote host named CHICAGO:

```
?AT CHICAGO BEGIN JOB REMOTE/RUNNER;
```

Submitting a WFL Job Stored on a Remote Host

If a WFL source program resides on a remote host, you can submit the WFL program for execution with the command *AT <hostname> START <file title>*. The following is an example of this command:

```
AT CHICAGO START (SMITH)REMOTE/RUNNER ON DPMAST
```

The WFL program is compiled and executed on the remote host where it resides.

A WFL job initiated in this way runs without a usercode in some circumstances. For a discussion of these circumstances, refer to "Usercode Identity" later in this section.

Meeting Remote Job Queue Requirements

You must be aware of the possibility that the job queue definitions on the remote host might be different from those on the local host. The job is enqueued on the remote host just as it would be if it were a local job submitted on that host. If the job does not qualify for any of the queues, it is discontinued.

The job queuing algorithm is outlined in "Selecting the Queue for a Job" in Section 4, "Tasking from Programming Languages."

Initiating Non-WFL Remote Processes

You can initiate remote processes from a local session or a local process. Several restrictions apply to the features that can be used by the remote process.

Specifying the Remote Host

The HOSTNAME task attribute can be used to specify the remote host at which the process is to run. This task attribute can be assigned through a task equation or an assignment to the task variable before initiation.

The HOSTNAME task attribute can be accessed from ALGOL, COBOL74, and WFL. Therefore, remote processes can be initiated from any of these languages. For example, the following ALGOL statements initiate a remote process:

```
PROCEDURE RUNNER;
  EXTERNAL;
REPLACE T.NAME BY "OBJECT/RUNNIT ON DPPACK.";
REPLACE T.HOSTNAME BY "SEATTLE.";
CALL RUNNER [T];
```

CANDE and MARC also enable HOSTNAME to be included as a task equation following a RUN statement. The following is an example of a CANDE command that initiates a remote process:

```
RUN RUNNIT;HOSTNAME=MIAMI
```

The equivalent statement in MARC is as follows:

```
RUN OBJECT/RUNNIT;HOSTNAME=MIAMI
```

Limitations on a Non-WFL Remote Process

The following restrictions apply to a remote process that is not a WFL job:

- The remote process must be an external process whose object code file is stored on the remote host.
- The remote process can be passed no more than one parameter. The parameter must be a real array of one dimension. The actual parameter must have a zero lower bound. The system automatically chooses a passing mode of call-by-value for the parameter.
- The WFL *COMPILE* statement cannot cause the resulting object code file to be executed as a remote process. For example, suppose the compiler equation COMPILER HOSTNAME = SFA15C is used. The compilation can run successfully on the foreign host with a disposition of LIBRARY or SYNTAX, but is rejected if the disposition is GO or LIBRARY GO.

If one of the preceding restrictions is violated, the initiating process is discontinued with HISTORYCAUSE = 2 (PROGRAMCAUSEV) and HISTORYREASON = 31 (ILLEGALTASKXFERV). The following error message is displayed:

```
ILLEGAL HOST-TO-HOST TRANSFER OF TASK
```

Another restriction is that a WFL job cannot use global file assignments for remote tasks initiated by the job. For example, the following sequence of statements is illegal:

```
FILE IN(KIND=DISK,TITLE=NEW/INPUT/DATA);
RUN OBJECT/UPDATE;
   HOSTNAME = ALBANY;
   FILE CARD := IN;
```

Global file assignments have no effect when applied to remote tasks initiated from WFL. The remote task executes normally, but the file used by the task does not receive any of the file attributes specified for the global file in the WFL job. When the remote task opens the file, the following nonfatal attribute error message is displayed:

```
[<hostname>] <mixno> ATTRIBUTE ERROR:<file internal name>.GLOBALFILESIRW
```

A remote task initiated from a local WFL job cannot read from any data specifications in the WFL job. When the remote task attempts to read from a data specification, it is suspended with a "NO FILE" condition and waits for a card reader file with the requested title to appear. An RSVP message such as the following is routed back to the local host:

```
[ALBANY] 2Ø79 RSVP (JASMITH)OBJECT/UPDATE ON USERPK. NO FILE CARD (CR)
```

A coroutine cannot use a continue statement to transfer control to a coroutine on a remote host. By default, the PARTNER task attribute of a remote task is treated as

MYSELF and the PARTNEREXISTS task attribute of a remote task returns a value of FALSE. In this case, any continue statement executed by the remote task has no effect. Execution simply proceeds to the next statement in the remote task.

The MYJOB task variable of a remote task is treated as a reference to the DSSSUPPORT library on the remote host. Any references to MYSELF.EXCEPTIONTASK in the remote task are treated as references to TASKING/MESSAGE/HANDLER, a task that is initiated by DSSSUPPORT on the remote host. TASKING/MESSAGE/HANDLER is discussed under "Displaying TASKING/MESSAGE/HANDLER and TASKING/STATE/CONTROLLER" later in this section. Any references to MYJOB.EXCEPTIONTASK in the remote task are treated as references to the remote task itself.

Any files accessed by a remote process are searched for on the remote host by default. If the remote process uses a file on the local host, the HOSTNAME file attribute must be assigned. For the remote process to open a remote file on the local host, the process must also set its STATION task attribute to zero and assign the desired station name to the FILENAME task attribute.

Host Availability

If a process attempts to initiate a task at a remote host that is nonexistent or currently unavailable, the initiation fails, but the initiating process continues normally. The task variable of the task stores a STATUS value of -2 (BADINITIATE), a HISTORYTYPE of 4 (DSEDV), and a HISTORYCAUSE of 13 (NETWORKCAUSEV). The HISTORYREASON value varies depending on the exact reason the host is unreachable.

A pair of messages such as the following are displayed when this error occurs:

```
6749 TASK NOT INITIATED AT TESTSYS : ERROR - HOST NOT REACHABLE
6749 FOREIGN TASK INITIATION FAILED @ 103A:0001:4 @ (00000234)*
```

Initiating Processes from a Remote Session

An alternate method of initiating a process on a remote host is to initiate it from a remote CANDE or MARC session. You can establish a remote session by using the Station Transfer feature provided by Host Services. A process initiated from such a session is considered a local process because the session is under the direct control of the remote host. The process is therefore not limited by any of the restrictions previously discussed under "Limitations on a Non-WFL Remote Process" in this section.

For a detailed discussion of Station Transfer, refer to the *A Series Distributed Systems Services (DSS) Operations Guide.*

Interrogating the Remote Ancestry of a Process

A process can find out which host system it is running on by interrogating its own HOSTNAME task attribute. This feature makes it possible to write a single program that will take different actions when it is run on different systems.

A process can interrogate its remote ancestry by inspecting the ITINERARY task attribute. This task attribute stores the hostnames of the host systems where each of the ancestors of the process are running. This task attribute can be useful for cases where the process needs to transmit information back to the user and thus needs to know where the user is located.

Preventing User Identity Problems

The user identity of a process consists of several related task attributes, including USERCODE, ACCESSCODE, CHARGE, and FAMILY. Each system in a multihost network has its own USERDATAFILE, which stores definitions of the users that are allowed on the system. These definitions can be different on different host systems. For a remote process to run successfully, it must be assigned an identity that is recognized on the remote host.

Usercode Identity

The most basic user identity requirement is that a remote process must run with a usercode that is allowed at the remote host, or it must run without a usercode.

If the remote process has a usercode, then the usercode must be one that is permitted as a remote user at the remote host. Remote users are defined by REMOTEUSER entries in the USERDATAFILE of the remote host. REMOTEUSER entries can specify in detail the hosts that can submit a process with a particular usercode. The following is an example of a REMOTEUSER entry at a remote host that allows processes with usercode JASMITH to be initiated from the host named CHICAGO:

```
RU JASMITH OF CHICAGO
```

In addition to the REMOTEUSER entry, there must be a USER entry for the usercode of the process in the USERDATAFILE at the remote host. A USER entry defines a usercode and assigns usercode attributes to the usercode.

A system administrator at the remote host can cause remote processes that request a particular usercode to be run under a different usercode instead. The substitute usercode is referred to as a *local alias usercode.* A remote process assumes a local alias usercode if the REMOTEUSER entry at the remote host specifies a local alias for the requested usercode. The local alias usercode must also be defined by a USER entry in the USERDATAFILE at the remote host. The following is an example of a REMOTEUSER entry that specifies a local alias usercode:

```
RU JASMITH OF CHICAGO LOCALALIAS=JOHNSMITH
```

The following is an example of a USER entry for the local alias:

```
USER = JOHNSMITH
     MAXPW = 1
     PASSWORD = ?
     FAMILY DISK = SYSPK OTHERWISE DISK
     IDENTITY = "ALIAS FOR JASMITH FROM CHICAGO"
```

Local alias usercodes are intended for use in cases where two different users on two different systems happen to have the same usercode. Establishing local alias usercodes allows these users to run processes on each other's systems, but prevents them from accessing each other's files.

Alternatively, the system administrator can use a local alias usercode to cause many usercodes from remote systems to be mapped to a single usercode at the local system. This mechanism can be useful if the users need to have access to the same set of files.

If no local alias usercode is defined for the requested usercode, then the requested usercode must itself be defined by a USER entry in the USERDATAFILE at the remote host. Note that one or more of the usercode attributes can have different values on the remote host than they have on the local host. These differences do not prevent remote process initiation.

In addition, the usercode can have passwords on the remote host that are different from those defined for that usercode on the local host. If the remote process inherits the usercode of the local process, the password is implicitly changed to a password that is accepted at the remote host. However, if the remote process is explicitly assigned a usercode at initiation time, the password specified should be one of the passwords defined for the usercode at the local host. If the remote process changes its usercode after it is initiated, the process must specify a password that is allowed on the remote host.

A WFL job is the only type of remote process that can run without a usercode. The following are sources from which a remote WFL job can receive a usercode, listed in order of precedence:

1. Assignments to the USERCODE task attribute in the job attribute list.
2. The usercode of a session, if the job is submitted from a CANDE or MARC session.
3. The terminal usercode, if the job is submitted from an ODT. An operator can assign a terminal usercode to an ODT by using the TERM (Terminal) system command.
4. The host usercode of the system, if the job is submitted from an ODT or a nonusercoded MARC session. The host usercode is assigned by the HU (Host Usercode) system command. You can create a nonusercoded MARC session by logging on with an asterisk (*) at a SUPERUSER-capable station.

If a process does not receive a usercode from any of the first three sources listed, then the host usercode is evaluated for SYSTEMUSER status. If the USER entry for the usercode at the remote host assigns SYSTEMUSER status, then the job runs without a usercode. Otherwise, the host usercode is used as the usercode for the job.

A local process cannot initiate a nonusercoded remote process. In the first place, any null usercode explicitly assigned to a process is overridden at initiation time by inheritance from the parent. For example, if the local process assigns a null USERCODE value to a task variable, and then initiates a remote process with the task variable, then the null USERCODE assignment is ignored. The task is initiated successfully, but inherits the usercode of its parent. In the second place, if the local process is nonusercoded, and it does not explicitly assign a usercode to the remote process, then the remote process inherits a null usercode. However, the system cannot initiate a remote process that has a null usercode, so the system displays error messages such as the following:

```
DISPLAY: 1807000 HOST SERVICES ERROR 17: USER ERROR - NO USERCODE
TASK OBJECT/ALGOL/TASK ON PACK NOT INITIATED AT PARIS : USER ERROR
   - NO USERCODE
FOREIGN TASK INITIATION FAILED @ 109E:0001:4 @ (00012500)*
```

For more information about usercode definitions and the REMOTEUSER command, refer to the *A Series Security Administration Guide*.

Accesscode and Charge Validation

A remote process does not inherit the CHARGE task attribute value of its parent. If no CHARGE value is explicitly assigned to the remote process, it runs without a charge code. However, the remote process runs with a CHARGE value if one is explicitly assigned by a statement in the parent process. Note that usercode definitions in the USERDATAFILE can specify the range of CHARGE values that are valid for processes with a given usercode. If the CHARGEREQ attribute is set in the usercode definition at the remote host, then the remote task must have one of the CHARGE values defined by the CHARGECODE usercode attribute at the remote host.

The ACCESSCODE task attribute also is not inherited by remote processes, but it can be assigned. However, if it is assigned, it must be assigned a value that is allowed for the remote process usercode on the remote host.

FAMILY Identity

A remote task does not inherit the FAMILY task attribute value of its parent. Instead, the remote task inherits the FAMILY specification in the usercode definition at the remote host, if there is such FAMILY specification. If there is no FAMILY specification in the usercode definition at the remote host, then by default the remote task runs with a null FAMILY value.

The parent process can override the FAMILY specification at the remote host by explicitly assigning a FAMILY value to the remote task, either with a task equation or with an assignment to the task variable of the remote task.

Logging of Remote Processes

The system logging and job logging responsibilities for remote process families are divided between the local host and the remote host.

System Log Entries

When a local process or session initiates a remote process, the local system log does not contain a log entry to record the event. However, if no other remote processes have been active recently, then the system makes a log entry showing that a port file called TASKPORT has been opened. The following is an example of an entry that can appear in the local log if a local process initiates a remote process. The initiating process in this example had a mix number of 1799:

```
14:10:15   OPEN 1799  EXT NAME: TASKPORT.
                      INT NAME: TASKPORTS.
                      FILE ACCESS RULE = UNION      (ACTOR = STACK 0352,
                        JOB 6717, TASK 6733)
                                                 (DECLARER = STACK 0165,
                        JOB 4677, TASK 4677)
                      USE = IN KIND=PORT    UNIT NUMBER 0
                      OPENTYPE: OFFER, POSITION: AT FRONT, MOTION:
                        NONESUBFILE: 0009
```

All system log entries for the remote process are made in the system log at the remote host. The remote process receives BOJ and EOJ log entries, even if it is actually a task. The BOJ log entry shows the originating unit as 0 (zero) and also includes a line called ITINERARY that specifies the host that initiated the process. The following is an example of this log entry:

```
14:10:26   BOJ  9295  (WHSMITH)OBJECT/REPORTER ON DPMAST.
                      CODE COMPILED: FEB 25, 1987 15:07:00 BY ALGOL 37.
                        165
                      QUEUE: 0
                      ORIGINATING UNIT: 0
                      PRIORITY: 50
                      USERCODE: WHSMITH.
                      ITINERARY: SANTAFE
```

Job Summaries for Remote Processes

The job summary for a process family is printed on the system where the job runs. Thus, for WFL jobs that include an *AT <hostname>* specification, the job summary is printed at the remote host. The same is true for WFL jobs started by an *AT <hostname> START* command. Any other independent remote processes, such as programs initiated by an ALGOL *RUN* statement, also print job summaries on the remote host.

When a local job initiates a remote task, a single entry is made in the job summary indicating that the remote task was initiated. No other job summary entries are made for the remote task. The following is an example of this entry:

```
17:55:18      1708  DISPLAY: [MIAMI] 3382 BOT (JAS)OBJECT/UPDATE/FILES.
```

Resource Limits for Remote Processes

Remote processes do not inherit resource limits from their local parents. For example, if a local job has a MAXPROCTIME limit, remote tasks do not inherit that limit. Furthermore, the local job cannot be discontinued because of excessive resource usage by its remote tasks.

The only way resource limits are propagated across networks is by explicit assignment. Thus, a local process can initiate a remote task and assign it a MAXPROCTIME value. If the remote task uses more processor time on the remote host than MAXPROCTIME allows, the remote task is discontinued.

For information about how resource limits are propagated in a local process family, refer to Section 2, "Understanding Interprocess Relationships."

Interacting with Remote Processes

An operator at the local host system can use system commands to monitor or interact with processes running on remote host systems. A user at a MARC or CANDE session can also use MARC or CANDE commands to monitor or interact with processes running on remote host systems.

Viewing Remote Process Messages

In general, any messages generated by a remote process are routed back to the local host. These include "BOT" and "EOT" messages, display messages, accept messages, and RSVP messages. The following are the only exceptions to this rule:

- WFL jobs initiated by an *AT <hostname> START* command. No messages are returned to the local host for such a job. (On the other hand, messages are returned for WFL jobs that use an *AT <hostname>* specification in the job header.)
- Non-WFL independent processes. These include any remote processes initiated by an ALGOL or COBOL74 *RUN* statement. Only a single "BOJ" message is routed back to the local host, and the mix number displayed is always 0000.

Remote process messages appear in the MSG (Display Messages) system command display at the local host, prefixed by the hostname of the remote host, as in the following example:

```
---Mix-Time---------------------- MESSAGES ------------------------
*  **  19:33 [PARIS] 1057 EOT (JASMITH)OBJECT/REPORTER ON DPMAST.
*  **  19:25 [PARIS] 1057 BOT (JASMITH)OBJECT/REPORTER ON DPMAST.
```

If the remote process was initiated from a CANDE or MARC session, the process messages are also routed back to the CANDE or MARC session. The following is an example of a CANDE command that initiates a remote task and the messages that are returned:

```
RUN REPORTER ON DPMAST;HOSTNAME=PORTLAND
#RUNNING 6881 AT PORTLAND.
#[PORTLAND] 6881 BOT (JASMITH)OBJECT/REPORTER ON DPMAST.
#ET=1:ØØ.2 PT=Ø.Ø IO=Ø.2
#[PORTLAND] 6881 EOT (JASMITH)OBJECT/REPORTER ON DPMAST.
```

The message "#ET = 1:00.2 PT = 0.0 IO = 0.2" is the termination message displayed for the process by CANDE. The elapsed time, processor time, and I/O time displayed in this message summarize the resource usage accumulated by the process on the remote host.

Note that the local termination message for the process appears before the EOT message from the remote host. This occurs because there is a slight delay in the forwarding of messages from the remote host.

Local Operator Control of Remote Processes

At the local host, you can control and interrogate remote processes by using system commands prefixed with the phrase *AT <hostname>*. You can direct any system command to a remote host in this way. However, security restrictions can be implemented at the remote host to limit or prevent the execution of such commands.

The system provides a usercode for each system command that is directed to a remote host. If the ODT has a terminal usercode, the terminal usercode is used. You can assign a terminal usercode with the TERM (Terminal) system command. If there is no terminal usercode, the host usercode is used. You can assign a host usercode with the HU (Host Usercode) system command.

If a process uses the DCKEYIN statement to submit a system command with an *AT <hostname>* prefix, the system command is submitted under the usercode of the process that executed the DCKEYIN statement.

For the command to be accepted at the remote host, the associated usercode must have a USER entry and a REMOTEUSER entry in the USERDATAFILE at the remote host. Otherwise, an error occurs at the remote host and the command is not executed. The command is also rejected if no usercode is associated with it. (The command might not have a usercode if neither a terminal usercode nor a host usercode is defined.)

If the REMOTEUSER entry defines a local alias usercode, the system command becomes associated with the local alias usercode. In this case, the USERDATAFILE must include a USER entry for the local alias usercode.

The remote host inspects the USER entry of the associated usercode to find out whether SYSTEMUSER status is set for the usercode. If SYSTEMUSER status is set for the usercode, then the system command is always allowed. If the usercode is not a SYSTEMUSER, then only a limited subset of the system commands can be used.

If the command usercode does not have SYSTEMUSER status, then the output of mix display commands is filtered so that only processes running under the command usercode are displayed. Likewise, commands that specify a particular process can only be applied to processes running under the command usercode. The following are the tasking-related commands that are available:

- AX (Accept)
- C (Completed Mix Entries)
- CU (Core Usage)
- DBS (Database Stack Entries)
- DS (Discontinue)
- DUMP (Dump Memory)
- FA (File Attribute)
- FR (Final Reel)
- HI (Cause EXCEPTIONEVENT)
- J (Job and Task Structure Display)
- LIBS (Library Task Entries)
- MSG (Display Messages)
- MX (Mix Entries)
- OF (Optional File)
- OK (Reactivate)
- OT (Inspect Stack Cell)
- RM (Remove)
- SL (Support Library)
- SQ (Show Queue)
- ST (Stop)
- THAW (Thaw Frozen Library)
- TI (Times)
- Y (Status Interrogate)

MARC Control of Remote Processes

You can enter system commands in MARC and direct them to a remote host by including the *AT <hostname>* prefix. The security checking that is done is the same as that done for commands entered at the ODT. However, the usercode of the MARC session is used as the command usercode. If the MARC session has no usercode, then the system uses the host usercode.

CANDE Control of Remote Processes

You can direct system commands to a remote host from a CANDE session by prefixing the command with *?AT <hostname>*. These system commands are subject to the same restrictions as commands entered using *AT <hostname>* at an ODT. The usercode of the CANDE session, or its local alias, is inspected for SYSTEMUSER status at the remote host and the commands are handled according to the results of this test.

Visibility of Remote Processes to Remote Operators

A remote process is visible to an operator at the remote host in the same way as it would if it were a local process. However, the remote process appears to be a job, even if it is actually a task. The following is an example of the Y (Status Interrogate) system command display for a remote task:

```
STATUS OF JOB 145Ø/145Ø AT 15:58:Ø1
PRIORITY = 8Ø
ORIGINATION: UNIT Ø
USERCODE: CYNTHIA
CHARGECODE: 6825
STACK STATE: WAITING ON AN EVENT
PROGRAM NAME: (CYNTHIA)OBJECT/UPDATER ON SYSPK
```

The "ORIGINATION" displayed is always "UNIT 0" if the process was initiated from a remote host. (However, there are other circumstances that can also cause an origination of "UNIT 0" to be displayed.)

The following is an example of how such a task might appear in the J (Job and Task Structure) system command display. No job is displayed for the task.

```
145Ø  5Ø  ..(CYNTHIA) (CYNTHIA)OBJECT/ALGOL/TASK ON SYS37
```

Displaying TASKING/MESSAGE/HANDLER and TASKING/STATE/CONTROLLER

The networking software creates two special processes that handle initiation of remote processes and communication between the remote processes and their local parents. These processes are tasks initiated by the DSSSUPPORT library on the local host and the remote host. Their names are TASKING/MESSAGE/HANDLER and TASKING/STATE/CONTROLLER. One instance of each of these tasks appears in the mix at a host as long as any remote processes or parents of remote processes are running at the host. These tasks also continue to appear in the mix for a few minutes after all remote processes have terminated.

Using Host Services-Supported Task Attributes

Host Services supports the majority of the task attributes discussed in this guide. However, there are some task attributes that cannot be used across multihost networks. If Host Services does not support a task attribute, then that task attribute cannot be accessed by a process running on a different host system. For example, if a parent process is running on one system and its task is running on another system, the parent cannot access some of the task's task attributes.

The following list shows all the task attributes. Each task attribute supported by Host Services is marked with an asterisk (*).

ACCEPTEVENT	* ACCESSCODE	* ACCUMIOTIME
* ACCUMPROCTIME	APPLYLIST	AUTORESTORE
AUTOSWITCHTOMARC	BACKUPFAMILY	* BDNAME
BRCLASS	* CHARGE	CHECKPOINTABLE
* CLASS	* COMPILETYPE	* CONVENTION
CORE	DATABASE	DECKGROUPNO
DEPTASKACCOUNTING	* DESTNAME	* DESTSTATION
DISKLIMIT	DISPLAYONLYTOMCS	* ELAPSEDLIMIT
* ELAPSEDTIME	ERROR	EXCEPTIONEVENT
EXCEPTIONTASK	* FAMILY	FETCH
* FILEACCESSRULE	FILEACCOUNTING	* FILECARDS
HISTORY	* HISTORYCAUSE	* HISTORYREASON
* HISTORYTYPE	* HOSTNAME	HSPARAMSIZE
INHERITMCSSTATUS	INITPBITCOUNT	INITPBITTIME
* ITINERARY	* JOBNUMBER	* JOBSUMMARY
JOBSUMMARYTITLE	* LANGUAGE	LIBRARY
LIBRARYSTATE	LIBRARYUSERS	LOCKED
* MAXCARDS	* MAXIOTIME	* MAXLINES
* MAXPROCTIME	* MAXWAIT	* MCSNAME
* MIXNUMBER	MYPPB	* NAME
NOJOBSUMMARYIO	* OPTION	ORGUNIT
OTHERPBITCOUNT	OTHERPBITTIME	PARTNER
PARTNEREXISTS	PRINTDEFAULTS	* PRIORITY
* RESOURCE	RESTART	RESTARTED
* SAVEMEMORYLIMIT	* SOURCEKIND	SOURCENAME
* SOURCESTATION	* STACKHISTORY	* STACKLIMIT
* STACKSIZE	STARTTIME	* STATION
* STATUS	* STOPPOINT	SUPPRESSWARNING
* SW1	* SW2	* SW3

continued

continued

* SW4	* SW5	* SW6
* SW7	* SW8	TADS
TANKING	TARGET	TASKERROR
TARGET	TASKERROR	TASKFILE
TASKLIMIT	TASKSTRING	* TASKVALUE
TASKWARNINGS	* TEMPFILELIMIT	* TEMPFILEMBYTES
* TYPE	* USERCODE	VALIDITYBITS
* WAITLIMIT		

In most cases, if Host Services does not support a task attribute, then any attempt to access the task attribute through Host Services is ignored. No error results, but the task attribute remains unchanged. A warning message is displayed if the DSSSUPPORT library on the system was compiled with the DIAGNOSTICS option set. However, if a process attempts to access the ACCEPTEVENT, EXCEPTIONEVENT, or TASKFILE task attribute of another process across a multihost network, the accessing process is discontinued.

It is possible for hosts running different software release levels to be linked in the same network. When accessing the task attributes of a remote task, be aware of the possibility that the remote host may be running an old version of Host Services that does not support all the task attributes that the current version of Host Services does.

Part II
Interprocess Communication

Section 13
Understanding Interprocess Communication

Interprocess communication, or IPC, is a voluntary exchange of information between two or more processes. Interprocess communication is sometimes referred to as *interprogram communication*. The latter term is avoided in this guide for two reasons.

First, a program is an artifact stored in a file that doesn't do anything. When the program is initiated, a process is created, and the process can communicate with other processes.

Second, the processes involved in interprocess communication are not necessarily instances of different programs. They might be two different instances of the same program, or they might be internal processes created by initiating procedures within the same program.

If the distinction between programs and processes seems unclear to you, read Section 1, "Understanding Basic Tasking Concepts," before proceeding any further in this section.

Information in a computer system is always stored in a particular form. For example, to store information about whether a given condition is true or false, a process might declare a Boolean variable. To store numeric data, the process might declare an integer variable or real variable. To record a set of instructions that can be invoked repeatedly, the process might declare a procedure. All of the things that can be declared in processes can be thought of, in a general way, as "objects."

With this point in mind, you can see that IPC consists of processes making use of objects declared by other processes. For example, if one process assigns a value of 3 to an integer variable declared in another process, this assignment is an example of interprocess communication. If a process invokes a procedure declared by another process, this procedure invocation is another example of interprocess communication.

Why should two processes need to have access to the same objects? The following are some examples:

- In an electronic mail system. One way for such a system to work would be for each user to initiate his or her own mail process. The mail processes could then use IPC techniques to send messages back and forth.
- For transaction processing. For example, you might have a file that is updated by many different online users. You can use IPC techniques to ensure that different users' updates do not overwrite each other.
- To promote reuse of code. You might write a procedure that is useful in many different applications. You can place the procedure in a library where it can be used by many different applications.

A Series systems provide a variety of IPC techniques. When you design applications that use IPC techniques, there are three main decisions you need to make:

- The type of object that is to be shared
- The method by which the object is to be shared
- The means of synchronizing access to the shared object

A factor contributing to all of these decisions is programming language restrictions. Not all types of objects, sharing methods, or synchronization methods are available in all programming languages.

Objects Used in Interprocess Communication

The types of objects you use for IPC depends on the types and quantity of data or code that are to be shared.

To exchange Boolean, numeric, or text data, it can be convenient to use the types of variables in which you would normally store such information in a program. For example, if the information to be communicated is an integer, you could store it in a level 77 BINARY elementary item in COBOL74, or in an INTEGER variable in ALGOL or WFL. Also, arrays generally can be used for IPC.

For certain simple data types, you also have the option of using task attributes to store the value. The operating system provides several task attributes as storage areas for use by the application. Refer to Section 14, "Using Task Attributes."

For communicating large volumes of textual data, you might want to use a port file. Processes can read from and write to port files much as if they were physical files. However, the port file exists only as a communication path between two or more processes. Port files help to prevent ambiguity and timing problems by providing separate input and output queues for each pair of communicating processes. For an overview of port file capabilities, refer to Section 19, "Using Shared Files."

For code that is to be shared between processes, you can use a procedure declaration. Information about sharing procedures is given in Section 15, "Using Global Objects;" Section 17, "Using Parameters;" and Section 18, "Using Libraries."

Methods of Sharing Objects

A Series systems provide several methods of sharing objects among processes. Your choice among these methods is determined to a large extent by the way the communicating processes are related.

For example, suppose that a WFL or ALGOL program initiates an internal procedure as a task. Statements in the procedure are able to access any variables declared globally to the procedure within the program. The parent of the task also has access to these globally-declared objects, which therefore can be used to pass information between the parent and one or more tasks. This method of sharing objects is discussed in Section 15, "Using Global Objects."

Another IPC method exists for communication between a process and its initiator. This method is called tasking parameter passing. WFL, ALGOL, and COBOL74 are able to specify most types of variables as parameters in process-initiation statements. Depending on the passing mode used, the parameter can provide one-way or two-way communication between the process and its initiator. ALGOL is also capable of passing a procedure as a parameter to another ALGOL program or a Pascal program. This makes parameter passing a viable method for sharing code as well as data. For further information about parameter passing, refer to Section 17, "Using Parameters."

Processes belonging to the same process family can conveniently access each other's task attributes by way of certain predefined task variables. Thus, a task can access the attributes of its job by way of the MYJOB task variable. The task can access the attributes of its parent and other ancestors by way of the EXCEPTIONTASK task attribute. Any two tasks in the same process family can exchange information through the task attributes of a common ancestor. Predefined task variables are discussed in Section 2, "Understanding Interprocess Relationships."

One IPC method that does not require the communicating processes to be related in any way is the library mechanism. A library is a type of process that stores procedures for use by other processes. Libraries can be written in ALGOL, C, COBOL(68), COBOL74, COBOL85, FORTRAN, FORTRAN77, NEWP, Pascal, and PL/I.

You can design libraries to have any of several levels of sharing properties. In the case of SHAREDBYALL libraries, each process using the library accesses the same instance of the library program. Each procedure exported by the library can access variables in the library that are declared globally to the procedure. Processes using a shared library can communicate by assigning and reading values of global variables in the library. For further information about libraries, refer to Section 18, "Using Libraries."

Port files are another IPC tool that is available between unrelated processes. In the case of port files, the sharing method is part of the design of the object itself. When one process attempts to open a port file, the system searches for a matching process that is attempting to open the other end of the port file. When the system finds a correct match it establishes the port file link between the processes. Port files can be used in all A Series programming languages except WFL. WFL is excepted because it is not capable of reading from or writing to any kind of file. For further information about port files, refer to Section 19, "Using Shared Files".

Methods of Synchronizing Access

When two or more processes are able to update the value of a common data item, the possibility arises that the updates can interfere with and overwrite each other. An example is that of a variable that records the current balance of a customer account. Suppose the account has a current balance of $100. One process might have responsibility for subtracting $10 from the account. Another process, running simultaneously, might have responsibility for adding $15 to the account. The net result should be a balance of $105. However, the actual results can be quite different.

The problem arises because this type of update involves building on the value that is already present. If more than one process updates the account, a sequence like the following can occur:

1. Process A reads the account balance ($100) into variable A1.
2. Process B reads the account balance ($100) into variable B1.
3. Process A subtracts $10 from variable A1, leaving $90.
4. Process B adds $15 to variable B1, leaving $115.
5. Process A assigns the value from A1 to the account balance, leaving a balance of $90.
6. Process B assigns the value of B1 to the account balance, leaving a balance of $115.

In other words, process B can unintentionally delete the effect of the update performed by process A. The result is that the customer balance is left at $115 instead of the correct $105.

To prevent such situations from occurring, it is sometimes necessary that a process be able to secure exclusive access to an object for the duration of the transaction. A Series systems provide a special type of variable called an *event* for handling these and other types of timing problems. You must use some means, such as global declarations, parameters, or SHAREDBYALL libraries, to provide the communicating processes with access to the event. You can then design the processes to use the event as a sort of flag to signal the availability of another object, such as a variable or file.

Events can be declared and used in ALGOL, COBOL(68), and COBOL74. Certain implicitly declared events can also be accessed by WFL. For further information about events, refer to Section 16, "Using Events."

Section 14
Using Task Attributes

Certain task attributes exist only for the purpose of transmitting information between different members of a process family. These attributes have no meaning to the system, and thus can be used only for storing values to be read later. The following are the task attributes that fall into this category:

- LOCKED

  This Boolean-valued task attribute accesses the availability state of an event. For further information, refer to "Using Implicitly Declared Events" in Section 16, "Using Events."

- SW1 through SW8

  Each of these attributes stores a Boolean value.

- TARGET

  This attribute stores an integer value.

- TASKSTRING

  This attribute stores a string value.

- TASKVALUE

  This attribute stores a real value.

In a more general way, all task attributes are instruments for interprocess communication (IPC). After all, each task attribute stores information about the process it applies to, and this information is visible to any other process that can access the task variable. What distinguishes the task attributes in the preceding list is that they have no meaning at all, except what is established by convention between two processes.

These task attributes provide the simplest means of IPC. There is no need to create and define complex data structures, as all task attributes are predeclared.

Each of the attributes involved stores only a single Boolean or arithmetic value. However, the values can be changed and read repeatedly during process execution.

A disadvantage to using these task attributes is that the task attribute names are fixed and thus do not convey any information about what is being stored in the attribute. Someone reading the program might have trouble understanding why the attribute is being used. By contrast, a variable can always be assigned a meaningful name.

Another disadvantage is that it generally takes more processor time to read or write a task attribute than to read or write variables declared by the process.

For two processes to communicate using task attributes, one or both must have access to a common task variable. If two processes belong to the same process family, they

can always communicate by way of the MYJOB task variable. If two processes have a common parent, they can communicate by way of their own EXCEPTIONTASK task attribute. For further information about the task variables a process can access, refer to Section 2, "Understanding Interprocess Relationships."

The task attribute most commonly used for IPC is TASKVALUE, and its most common use is in task equations. For example, you could use TASKVALUE to instruct a program whether to produce a printout. The program could contain the following statement:

```
IF MYSELF.TASKVALUE = 1 THEN F.KIND := VALUE(PRINTER)
   ELSE F.KIND := VALUE(REMOTE);
```

If TASKVALUE has a value of 1, the program produces a printout; otherwise the program displays its output at the user's terminal. You might use a statement like the following to initiate the program and cause the program to produce a printout:

```
RUN REPORT/GENERATOR;TASKVALUE = 1
```

Section 15
Using Global Objects

In Section 1, "Understanding Basic Tasking Concepts," the concept of an *internal task* was introduced. An internal task is created by a statement that initiates a single procedure within a program. The capability of initiating internal tasks exists only in WFL and ALGOL.

WFL and ALGOL share a similar type of program structure. Both languages allow you to create blocks that can include declarations of objects for use within the block. Both languages allow you to nest blocks within other blocks. Both languages allow nested blocks to use objects declared in the blocks they are nested within. These objects are referred to as *global objects*.

Globally declared objects can be used to allow an internal task to communicate with its parent or with other internal tasks of the same parent. Even widely separated members of a process family can communicate with each other by way of global objects. For example, sibling or cousin tasks could communicate, or a task could communicate with an ancestor. For an introduction to the possible relationships in a process family, refer to Section 2, "Understanding Interprocess Relationships."

Processes can communicate through a particular global object only if the processes meet both the following rules:

- Each process is one of the following: the process that executed the declaration of the global object, or an internal task of that process, or an internal task of one of these internal tasks, and so on.
- Each process must have been created by initiating a procedure that falls within the scope of the declaration of the global object.

The *scope* of a declaration consists of all the blocks that have access to the object declared. Conversely, the *addressing environment* of a block consists of all the objects that can be used by statements in the block. The following subsections discuss the scope of declarations in WFL and ALGOL, and give examples of related processes that communicate through global objects.

Global objects can also be used in SHAREDBYALL libraries to provide communication between unrelated processes. The use of global objects in libraries is discussed in Section 18, "Using Libraries."

Communication through Global Objects in WFL

The types of blocks that can occur in a WFL job are the outer block and any SUBROUTINE declarations in the job. The scope of a declaration in WFL is limited to the following blocks:

- The block in which the declaration occurs
- Any blocks that are nested in the declaration block and that occur after the declaration

The following WFL example illustrates the effects of these scope rules:

```
100 ?BEGIN JOB;
110 INTEGER OUTERINT1;
120 SUBROUTINE FIRSTSUB;
130   BEGIN
140     INTEGER FIRSTINT;
150     SUBROUTINE NESTEDSUB;
160       BEGIN
170         INTEGER NESTEDINT;
180         OUTERINT1 := 3;
190         FIRSTINT  := 3;
200         NESTEDINT := 3;
210       END NESTEDSUB;
220     OUTERINT1 := 2;
230     FIRSTINT  := 2;
240   END FIRSTSUB;
250 INTEGER OUTERINT2;
260 OUTERINT1 := 1;
270 OUTERINT2 := 1;
280 ?END JOB
```

This example includes three procedures: the outer block of the job and two subroutines, of which NESTEDSUB is nested within FIRSTSUB. Each procedure includes integer variable declarations. Additionally, each procedure that is within the scope of an integer variable declaration includes a statement making an assignment to the integer variable.

Thus, the integer variable OUTERINT1 can be used by statements in the outer block, the FIRSTSUB subroutine, and the NESTEDSUB subroutine. This is because the scope of a declaration includes the procedure it is declared in and all nested procedures. By contrast, the integer variable OUTERINT2 cannot be used by statements in FIRSTSUB or NESTEDSUB, because these subroutines are declared prior to OUTERINT2.

The integer variable FIRSTINT can be used by statements in FIRSTSUB, because FIRSTINT is declared in FIRSTSUB; and by statements in NESTEDSUB, because it is nested in FIRSTSUB. However, FIRSTINT cannot be used by statements in the outer block, because the outer block is not nested inside FIRSTINT.

The integer variable NESTEDINT can be used only by statements in NESTEDSUB, because no other procedures are nested in NESTEDSUB.

The next example shows the use of global objects in WFL to provide an elementary type of IPC.

```
100 ?BEGIN JOB GLOBAL/DISPLAY;
110   CLASS = 0;
120 STRING MSG;
130 TASK S1, S2;
140 SUBROUTINE SUBONE;
150   BEGIN
160     WHILE S2(STATUS) ISNT SUSPENDED DO
170       WAIT(1);
180     MSG := ACCEPT("ENTER A MESSAGE PLEASE");
190     S2(STATUS = ACTIVE);
200   END SUBONE;
210 SUBROUTINE SUBTWO;
220   BEGIN
230     MYSELF(STATUS = SUSPENDED);
240     DISPLAY(MSG);
250   END SUBTWO;
260
270 PROCESS SUBONE [S1];
280 PROCESS SUBTWO [S2];
290
300 ?END JOB
```

In this example, two subroutines, SUBONE and SUBTWO, are initiated as asynchronous tasks. Both subroutines fall within the scope of the string MSG, which is declared in the outer block. SUBONE waits for SUBTWO to become suspended. SUBTWO executes a statement that suspends itself. At this point, SUBONE resumes execution and assigns an operator ACCEPT message to the MSG string. SUBONE then changes the status of SUBTWO to ACTIVE. When SUBTWO resumes execution, it displays the value of the MSG string.

This is a simple example, but even in this example it was necessary to take measures to regulate the timing of the asynchronous tasks. For example, the statement at line 180 should execute before the statement at line 240; otherwise, the DISPLAY statement at line 240 displays an empty value. This example uses assignments to the STATUS task attribute to suspend and restart execution of the asynchronous tasks. Other timing methods available in WFL include the LOCKED task attribute and various forms of the WAIT statement. These timing methods are discussed under "Using Implicitly Declared Events" in Section 16, "Using Events".

Section 16
Using Events

Shared objects and task attributes provide a relatively simple means of communicating information if all the tasks involved are synchronous tasks. If the tasks in a process family are all synchronous, then only one process is executing at a time. The order in which processes access shared objects is therefore fixed.

However, in cases where asynchronous processes access the same object, the order in which they access the shared objects is not fixed. This fact can create many unexpected side effects in communication. For example, suppose two processes communicate a vital bit of information by way of a shared integer variable. How is one process to know that the other process has updated the variable, so that it is now ready to be read?

The answer is that a programmer must implement flags to indicate whether a particular variable is to be accessed at this time. You can implement many types of flags. For example, a process could reset the value of a variable to zero after reading it. Another process could be designed to write a new value to the variable whenever the variable contains a zero. In this example, the zero value is being used as a flag to show that the variable has been read and is ready to have a new value written into it.

One problem with these types of flags is that the processes involved have to keep checking the flag periodically to see if it has been set. These repeated checks waste processor time. Another problem is that, between the time that one process reads the flag and the time it sets the flag, another process might have written to or read the flag. The flag is, therefore, not completely reliable.

You can avoid both of these problems by using *events*. An event is a special type of object that is used only for regulating the timing of asynchronous processes. A process can wait for an event to assume a certain state, without using any processor time while it waits. When the event assumes the desired state, the process resumes execution automatically.

Events can be declared in ALGOL and COBOL74 programs, but not in other languages. Work Flow Language (WFL) jobs can wait on certain predeclared and implicitly declared events.

Events can be made available to tasks in the same way as other objects can. That is, internal tasks can access events declared globally in their parents. An internal or external task can be passed an event as a parameter.

An event consists of an identifier that has two states associated with it: the available state and the happened state. The available state can be AVAILABLE or NOT AVAILABLE. The happened state can be HAPPENED or NOT HAPPENED. These values can be inspected or changed by any of several event-related statements that are described in the following pages.

The available state of an event is typically used to temporarily restrict access to a particular object, so that only one process can access the object during a given period of

time. The happened state is used to allow one or more processes to wait without using any processor time while waiting.

The initial available state of an event is AVAILABLE. The initial happened state of an event is NOT HAPPENED.

Declaring Events

In ALGOL, an event declaration is similar to a simple variable declaration. The following statement declares two events:

```
EVENT EDATA, EACCESS;
```

Events can be grouped in ALGOL as a one-dimensional event array. The following example declares an event array:

```
EVENT ARRAY EVNT[1:12];
```

The elements of this array can be used wherever an event is allowed. For example, EVNT[3] accesses the third event in the previous array declaration.

Events can be declared in COBOL74 as elementary or group items. The following example declares an event as an elementary item:

```
77 E1 USAGE IS EVENT.
```

The following example declares a group item that contains two events and a two-dimensional event array:

```
Ø1 EGROUP      USAGE IS EVENT.
   Ø3 E-1.
   Ø3 E-2.
   Ø3 E-3 OCCURS 5.
      Ø5 E-4 OCCURS 1Ø.
```

Accessing the Available State

The available state of an event records whether the event is currently assigned to a process. An event can only be assigned to one process at a time. If the event is currently assigned to a process, the available state is NOT AVAILABLE. If the event is not assigned to a process, the available state is AVAILABLE. A *procure* statement is one that changes the available state from AVAILABLE to NOT AVAILABLE. A *liberate* statement is one that changes the available state to AVAILABLE.

To prevent two processes in a process family from accessing the same object at the same time, you declare an event that can be used by all the processes that access the object. The processes should be designed according to a common convention so that each

attempts to procure the declared event before accessing the shared object. If the event cannot be procured immediately, the process should either wait for the event to become AVAILABLE or proceed with other business until the event becomes AVAILABLE. When a process is finished using the shared object, it should liberate the event and thus make the shared object AVAILABLE for use by other processes.

This mechanism of protecting a shared object depends on the cooperation of all the processes that access the object. The system is not aware of any link between the event and the object it protects.

Furthermore, procuring an event does not prevent other processes from accessing the event. It simply prevents other processes from directly procuring the event. These other processes could execute statements to liberate the event and then procure it, or execute statements that access the happened state. This fact allows considerable flexibility in the use of events.

Procuring an Event Unconditionally

An unconditional procure statement is one that stops execution of the process until the requested event becomes AVAILABLE. When the event becomes available, the unconditional procure statement immediately changes the event back to NOT AVAILABLE and allows the process to resume executing. If the requested event is already AVAILABLE, then the unconditional procure does not stop execution of the process; instead, the unconditional procure immediately changes the event to NOT AVAILABLE and allows the process to continue executing.

The following ALGOL statement unconditionally procures the event E1:

```
PROCURE (E1);
```

The following COBOL74 statement has the same effect:

```
LOCK (E1).
```

There is one situation that can cause an unconditional procure to continue waiting even after an event becomes AVAILABLE. For details, refer to "Partially Liberating an Event" later in this section.

Procuring an Event Conditionally

A conditional procure statement allows the process to continue execution if the requested event cannot be immediately procured. The process makes one attempt to procure the event and, if the event is AVAILABLE, changes the available state to NOT AVAILABLE. The conditional procure statement returns information that enables the process to tell whether the conditional procure action was successful.

The following ALGOL statement conditionally procures the event E1 and stores the result in the Boolean variable BOOL. If the conditional procure succeeds, BOOL receives a value of FALSE. If the conditional procure fails, BOOL receives a value of TRUE.

```
BOOL := FIX (E1);
```

The following COBOL74 statement conditionally procures the event E1. The AT LOCKED clause specifies an action to be taken if the procure fails.

```
LOCK (E1) AT LOCKED GO P2.
```

Liberating an Event

A liberate statement sets the available state of the event to AVAILABLE and sets the happened state to HAPPENED. (For information about the happened state, refer to "Accessing the Happened State" later in this section.) The process then continues normally.

If another process was waiting to procure the event, that process procures the event and continues execution. The available state returns to NOT AVAILABLE. If more than one process was waiting to procure the event, then only one of the processes succeeds, and the other processes continue to wait until the event is liberated again. The programmer cannot predict which of the contending processes will procure the event. However, the highest priority process is the one with the best chance of succeeding.

The following ALGOL statement liberates event E1:

```
LIBERATE (E1);
```

The following COBOL74 statement has the same effect:

```
UNLOCK (E1).
```

Partially Liberating an Event

A partial liberate statement sets the available state of an event to AVAILABLE, but leaves the happened state unchanged. The process that performs the partial liberate statement continues execution normally. The partial liberate statement differs from a liberate statement in that it does not cause waiting processes to resume execution. Any processes that had previously executed a wait statement or an unconditional procure statement will continue to wait indefinitely. However, because the partial liberate statement changes the event to AVAILABLE, the event can be procured by procure statements executed after the partial liberate statement.

You should be very careful when using the partial liberate statement. You need to either cause or liberate the event eventually so that the processes that are waiting on the event can resume. (The cause statement is discussed under "Causing an Event"

later in this section.) However, because the partial liberate statement changes the state to AVAILABLE, another process could procure the event before the first process fully liberates it. Unless you design the code carefully, two different processes might accidentally use the resource flagged by the event at the same time.

In ALGOL, the partial liberate statement is called FREE. The following is an example of this statement:

```
FREE (E1);
```

The partial liberate statement can also be used as a function that returns a Boolean value. If the event is already AVAILABLE, a value of FALSE is returned. If the event is NOT AVAILABLE, a value of TRUE is returned, and the event is set to AVAILABLE. The following ALGOL statement partially liberates event E1 and stores the result in BOOL:

```
BOOL := FREE (E1);
```

The partial liberate statement is not available in COBOL74.

Testing the Availability of an Event

An availability test returns a Boolean value that indicates whether the event is AVAILABLE. If the available state is AVAILABLE, the test returns TRUE. If the available state is NOT AVAILABLE, the test returns FALSE. The process continues normal execution in either case; it does not wait for the event to become AVAILABLE. The availability test does not make any change to the event and does not affect processes waiting on the event.

The following ALGOL statement tests the available state of the event E1:

```
WHILE AVAILABLE (E1) DO . . .
```

The availability test is not available in COBOL74.

Note that the availability test is not an adequate substitute for the conditional procure statement. Thus, the effects of the following two statements are quite different:

```
FIX (E1);
IF AVAILABLE (E1) THEN PROCURE (E1);
```

Suppose these statements are executed by a process called *A*. The first statement, FIX, causes a conditional procure. This statement procures event E1 if it is AVAILABLE, but abandons the procure and allows process A to continue running if E1 is NOT AVAILABLE. The second statement attempts an unconditional procure if E1 is AVAILABLE. However, there might be another process, hereafter referred to as *B*. Process B might procure E1 after process A executes the availability test, but before

process A executes the unconditional procure. In that case, process A would cease execution until process B eventually liberated the event.

The lesson to be learned from this example is that the availability test should be used only in cases where the process does not need to procure the event, but only needs to determine whether the event is currently in use by another process. However, even this use can cause efficiency problems if done with excessive frequency. Refer to "Buzz Loops" later in this section for details.

Determining the Ownership of an Event

A process becomes the owner of an event when the process successfully procures that event, and remains the owner until the event is liberated. A process can use the MCP procedure EVENT_STATUS to determine whether that process is the current owner of the event.

The EVENT_STATUS procedure is primarily useful in fault-handling code, EPILOG procedures, and EXCEPTION procedures. In these contexts, the EVENT_STATUS result enables the process to determine whether it should liberate the event before exiting a procedure, to make the event available to other processes. For further information about EPILOG and EXCEPTION procedures, refer to "Using EPILOG and EXCEPTION Procedures" later in this section.

Note: *The EVENT_STATUS procedure is the only safe method of determining the owner of an event. Unsafe NEWP programs that manipulate events directly should be avoided, because the format of events differs among A Series systems and is subject to change without notice. Use of the EVENT_STATUS procedure makes it unnecessary to modify application programs when the event format changes.*

The following declarations can be included in an ALGOL program to enable the EVENT_STATUS procedure to be used:

```
LIBRARY MCPSUPPORT (LIBACCESS = BYFUNCTION);

REAL PROCEDURE EVENT_STATUS(EV);
     EVENT EV;
     LIBRARY MCPSUPPORT;

DEFINE LOCKOWNERF = [42:39] #; % Lock owner field in EVENT_STATUS result
```

The program passes the event in question to the EV parameter of the EVENT_STATUS procedure. This procedure returns information about the event in the procedure result. The format of this result is as follows:

Field	Meaning
[42:39]	Stack number of the process that owns this event
[3: 2]	Event usage
	0 = Normal event
	2 = Interrupt attached
[1: 1]	Availability state
	0 = AVAILABLE
	1 = NOT AVAILABLE
[0: 1]	Happened state
	0 = NOT HAPPENED
	1 = HAPPENED

Note: *The information in the EVENT_STATUS result reflects the state of the event at a single moment in time. If other processes have access to the event, ownership of the event could change between the time a process calls EVENT_STATUS and the time the process reads the result.*

Assuming that EVENT_STATUS, the MCPSUPPORT library, and the LOCKOWNERF field have been declared as shown previously, the following ALGOL statement can be used to determine whether the current process owns an event:

```
IF PROCESSID = EVENT_STATUS(EVENT1).LOCKOWNERF THEN
BEGIN
   %  Take various appropriate actions
END;
```

Accessing the Happened State

A process can use the happened state of an event to inform another process that some expected condition has been fulfilled. A statement that sets the happened state to HAPPENED is said to *cause* the event. A statement that sets the happened state to NOT HAPPENED is said to *reset* the event. Every process that has visibility to the event also has the right to cause or reset the event.

A process can also *wait on* an event, in which case execution of the process is suspended until another process causes the event. A process that is waiting on an event does not use any processor time. The waiting process cannot resume execution until the event is caused by some other process. Any number of processes can wait on the same event.

Processes that wait on the happened state and processes that wait on an unconditional procure statement are in a similar situation. In both cases, the process can take no further action until another process modifies the event. However, the following differences might make it more convenient to use the happened state in some cases and the availability state in others:

- Causing the happened state reactivates all the processes that are waiting on the happened state. However, liberating the available state reactivates, at most, one process. Other processes attempting unconditional procures will continue to wait.
- A single process can wait on the happened state of more than one event simultaneously. If any of the events are caused, the process resumes execution. By contrast, a single process can attempt to procure only one event at a time.
- The functions that wait on the happened state and functions that reset the happened state can be used separately or together. By contrast, a function that waits on the availability state of an event always resets the availability state at the same time that it reactivates the process.

Causing an Event

The cause statement sets the happened state to HAPPENED and reactivates all processes that are waiting on the happened state of an event. Causing an event has no effect on the available state. The process that performs the cause continues without interruption.

Reactivating a process simply makes that process eligible for processor time. The priority of the process, compared to other processes in the mix, determines how soon the process resumes execution.

The happened state can be reset as soon as it is caused, if another process is waiting on the event with a wait and reset statement. Refer to "Waiting On and Resetting an Event" later in this section.

An ALGOL statement can cause only one event at a time. The following ALGOL statement causes the event EVNT:

```
CAUSE (EVNT);
```

A single COBOL74 statement can cause one or more events, as in the following example:

```
CAUSE EVNT1, EVNT2, EVNT3.
```

Implicitly Causing an Event

A secondary effect of liberating an event is that the happened state is set to HAPPENED, and processes waiting on the happened state are reactivated. For more information about the liberate statement, refer to "Liberating an Event" earlier in this section.

Causing and Resetting an Event

The cause and reset statement reactivates waiting processes and then returns the event to NOT HAPPENED. If the event is already HAPPENED when this function is applied, the effect is to reset the event to NOT HAPPENED.

The following ALGOL statement causes and resets the event EVNT1:

```
CAUSEANDRESET (EVNT1);
```

The following COBOL74 statement causes and resets three events:

```
CAUSE AND RESET EVNT1, EVNT2, EVNT3.
```

Partially Causing an Event

The partial cause statement sets the happened state of an event without reactivating any processes that are waiting on the event. The waiting processes cannot reactivate until a later statement causes the event. In addition, any new processes that attempt to wait on the event will immediately continue because the event is already HAPPENED.

The following ALGOL statement partially causes the event EVNT:

```
SET (EVNT);
```

The partial cause statement is not available in COBOL74.

Resetting an Event

The reset statement changes the happened state of an event to NOT HAPPENED. This statement makes it possible to reuse an event after it has been caused. If the event is not reset after it is caused, then any processes that try to wait on the event will continue immediately instead of waiting.

The following ALGOL statement resets the event EDATA:

```
RESET (EDATA);
```

The following COBOL74 statement resets two events:

```
RESET EDATA, ECONTROL.
```

You can also reset the happened state with the statements discussed under "Causing and Resetting an Event" earlier in this section or "Waiting on and Resetting an Event" later in this section.

Waiting on an Event

The wait statement suppresses execution of the process until another process causes the event. If the happened state is already HAPPENED, then the wait statement has no effect and the process proceeds immediately. The wait statement does not change the happened or available states of the event.

The waiting process does not use any processor time. Nevertheless, the waiting process is considered active, rather than suspended, and does not appear in the W (Waiting Entries) system command display.

A waiting process is discontinued if it exceeds the time limit specified by the WAITLIMIT task attribute.

The following is an ALGOL statement that waits on the event EVNT1. The COBOL74 syntax is identical, except that it terminates with a period rather than a semicolon:

```
WAIT (EVNT1);
```

Waiting on Time

A wait statement can also cause the process to wait for a specified number of seconds. The wait statement implicitly causes the system to create an event. The system causes the event after the specified time period. The actual time can be somewhat longer than the requested time, depending on the priority of the process and how busy the processor is.

The maximum time delay that a process can request is 164926 seconds (about 46 hours). If a wait statement specifies a longer period of time, the system reduces it to this maximum value.

The following ALGOL statement waits for 123 seconds:

```
WAIT ((123));
```

The WFL syntax is the same, except that only one set of parentheses is used.

In COBOL74, the statement appears as follows:

```
WAIT UNTIL 123.
```

Waiting on and Resetting an Event

The wait and reset statement has the same effect as the wait statement, except that the happened state of the event is reset to NOT HAPPENED after the process reactivates.

The following ALGOL statement waits and resets the event EWAIT:

```
WAITANDRESET (EWAIT);
```

The following COBOL74 statement has the same effect:

```
WAIT AND RESET UNTIL EWAIT.
```

Waiting on Multiple Events

The wait statement can specify a list of events. The process waits until any one of the events is caused. If any one of the events is already HAPPENED when the wait statement is executed, the process does not wait at all. The wait statement can return a value that specifies which one of the events was caused. If more than one of the events was caused, the value returned indicates the leftmost of the caused events in the list.

A wait and reset statement can also wait on multiple events. This statement resets to NOT HAPPENED the single event that reactivates the process.

The following ALGOL statement waits for 10 seconds, or until event E1 or E2 is caused, whichever comes first. The relative position of the event that reactivates the process is stored in T. For example, if E1 reactivates the process, T receives a value of 2.

```
T := WAIT ((10), E1, E2);
```

The following COBOL74 statement has the same effect:

```
WAIT UNTIL 10, E1, E2 GIVING T.
```

Testing the Happened State

The happened test inspects the happened state of an event. This test returns a value of TRUE if the event is HAPPENED and FALSE if the event is NOT HAPPENED.

Note that repeated happened tests are not the most efficient method of waiting on an event. Refer to "Efficiency Considerations" later in this section.

The following ALGOL statement invokes the procedure PFILE if the event E1 is HAPPENED:

```
IF HAPPENED (E1) THEN PFILE;
```

The following COBOL74 statement has the same effect:

```
IF E1 THEN GO PFILE.
```

Duration of the Happened State

You can use an event to flag either a momentary condition or an elapsed condition. A momentary condition is one that is relevant only to the particular process or processes that are already waiting for the condition. An elapsed condition is one that continues to be relevant to other processes in the future.

You can flag a momentary condition by using statements that cause the event and then immediately reset it. An event is immediately reset after being caused if either of the following conditions are true:

- The event was caused by a cause and reset statement.
- At least one of the processes waiting on the event used a wait and reset statement.

The program will be easier to understand and maintain if these methods of resetting the event are not mixed. If you use a wait and reset statement, you should use a simple cause statement. If you use a simple wait statement, you should use a cause and reset statement.

You can flag an elapsed condition by using simple cause and wait statements. After the event is caused, it remains in the HAPPENED state. When the elapsed condition ends, a reset statement returns the event to the NOT HAPPENED state.

Note that the use of separate reset statements automatically implies an elapsed condition. Even if the reset statement is the first action executed after a cause or wait statement, a significant interval of time can elapse before the reset statement is executed. Only through the use of wait and reset or cause and reset statements can you flag a truly momentary condition.

Using Implicitly Declared Events

A process can access a number of types of events that are never explicitly declared. Some of these are predeclared and always available. Others are created by the system in response to certain forms of the wait statement.

Two predeclared events that are associated with every process are the exception event and the accept event. You can access these events by using the EXCEPTIONEVENT and ACCEPTEVENT task attributes. A process can wait on, cause, or reset these events by way of their associated task attributes. The following ALGOL statement waits on the exception event of the process:

```
WAIT (MYSELF.EXCEPTIONEVENT);
```

The following COBOL74 statement has the same effect:

```
WAIT UNTIL EXCEPTIONEVENT OF MYSELF.
```

The following WFL statement has the same effect:

```
WAIT;
```

The accept event cannot be accessed in WFL. The ALGOL and COBOL74 syntax for accessing the accept event parallels that used for the exception event. In addition, COBOL74 allows the following special syntax for waiting on the accept event:

```
WAIT UNTIL ODT-INPUT-PRESENT.
```

You can access another predeclared event by using the LOCKED task attribute. This attribute translates Boolean assignments into procure and liberate statements. Thus, a statement that sets the LOCKED attribute of a process to TRUE has the effect of unconditionally procuring the predeclared event. Setting LOCKED to FALSE liberates the predeclared event. If LOCKED is already TRUE, then any processes that attempt to set LOCKED to TRUE are queued until another process sets LOCKED to FALSE. The main virtue of this task attribute is that it provides WFL jobs with an easy way of protecting a resource, even though WFL jobs cannot access events directly.

Certain types of objects have event-valued attributes associated with them. These objects include DCALGOL queues, Direct I/O buffers, port files, and remote files. Processes can wait on these event-valued attributes just as if they were explicitly declared events. For information about DCALGOL queues, refer to the *A Series DCALGOL Programming Reference Manual*. For information about Direct I/O buffers, port files, and remote files, refer to the *A Series I/O Subsystem Programming Guide*.

The WAIT statement in WFL can also include clauses that cause the job to wait until specified task attribute values or file attribute values are attained. Refer to the *A Series Work Flow Language (WFL) Programming Reference Manual* for full details.

Using Interrupts

An interrupt is a procedure that is associated with an event. Specifying an interrupt allows a process to continue executing other statements at the same time that it waits on an event. When the event is caused, control passes directly to the interrupt procedure. When the interrupt procedure finishes, the process resumes execution where it left off.

An interrupt cannot be invoked using any of the standard procedure invocation statements. An interrupt is entered only when the associated event is caused. Causing an event invokes the interrupt even if the event is already in a HAPPENED state. Therefore, there is no effective difference between using a cause statement or a cause and reset statement to invoke the interrupt.

You can use attach and detach statements to specify with the event an interrupt is associated with. Execution of the interrupt can be selectively allowed or suppressed through the use of enable and disable statements. The statements that attach or detach and enable or disable an interrupt can occur in any order, and do not affect each other. For example, detaching an interrupt does not also cause it to be disabled. The initial state of an interrupt is detached and enabled.

An interrupt might not always execute immediately when its event is caused. Any of the following three circumstances can delay execution of an interrupt:

- The interrupt is disabled.
- The process is waiting on an event. Any interrupts that are caused are queued and executed when the process resumes.
- The processor is engaged in executing a higher-priority process.

Declaring Interrupts

The purpose of an interrupt declaration is to assign an identifier to the interrupt and specify the statements that are to be executed when the associated event is caused.

An interrupt cannot be passed any parameters. Otherwise, it has the same addressing environment as a procedure would have if it were declared at the same point in the program. That is, in ALGOL the interrupt can access objects declared within the interrupt and within any procedures that are declared globally to the interrupt.

In rare instances, you might want to restart the process at a point other than the point at which the process was interrupted. You can achieve this effect in ALGOL with a bad GO TO statement (that is, a GO TO statement that transfers control to a statement outside the interrupt). However, COBOL74 does not allow a GO TO statement to transfer control outside of the interrupt.

You should be aware of a side effect that arises from using a bad GO TO to exit an interrupt. During execution of an interrupt, the system automatically executes a general disable on all other interrupts used by the process. A bad GO TO out of an interrupt leaves the process with all interrupts disabled. You should include a general enable statement to correct this situation. (Refer to "Using General Disable and Enable Statements" later in this section.)

The following is an ALGOL example of an interrupt declaration:

```
INTERRUPT BLOCK1;
  BEGIN
    DISPLAY("ERROR");
    DISPLAY("INTERRUPT BLOCK1 OCCURRED");
  END;
```

In COBOL74, an interrupt declaration can occur only in the DECLARATIVES section of the procedure division. The following is an example of a DECLARATIVES section that includes an interrupt called INT-1:

```
DECLARATIVES.
INT SECTION.
    USE AS INTERRUPT PROCEDURE.
INT-1.
    DISPLAY "ERROR".
    DISPLAY "INTERRUPT 1 OCCURRED".
END DECLARATIVES.
```

Attaching or Detaching an Interrupt

The attach statement associates an interrupt with an event. If the interrupt is already attached to another event, it is automatically detached from the old event and then attached to the new event.

You can attach each interrupt to only one event. However, you can attach more than one interrupt to the same event. When the event is caused, the associated interrupts are queued for execution in the reverse of the order that they were attached to the event.

It is possible to attach an interrupt to an event that is declared in a different process. The interrupt executes as part of the process that declared it, even if it is associated with an event in a different process. The interrupt declaration cannot be more global than the event declaration, or an "UP LEVEL ATTACH" error results. This error occurs at compile time if the compiler detects the problem. Otherwise, it occurs at run time.

The detach statement removes the association of an interrupt with an event. If the interrupt is not currently associated with an event, the detach statement has no effect and execution continues normally.

Note that if the interrupt is disabled, queued instances of the interrupt might have accumulated. Detaching the interrupt, or attaching the interrupt to a different event, causes these queued instances to be deleted. You can prevent this problem by enabling the interrupt before detaching it from an event or attaching it to a different event.

The following are ALGOL statements that attach and detach an interrupt. The first statement attaches the interrupt INT1 to the event E1. The second statement implicitly detaches the interrupt and then attaches it to the event E2. The third statement then detaches the interrupt and leaves it detached.

```
ATTACH INT1 TO E1;
ATTACH INT1 TO E2;
DETACH INT1;
```

The following COBOL74 statements attach two interrupts to the same event and then detach them:

```
ATTACH INT-1 TO E1.
ATTACH INT-2 TO E1.
DETACH INT-1, INT-2.
```

Enabling or Disabling an Interrupt

There might be periods during process execution when it would be undesirable for the interrupt to occur. These are generally periods when the process is accessing objects that are also modified by the interrupt. A programmer can selectively suppress execution of interrupts through the use of enable and disable statements.

If an interrupt's event is caused while the interrupt is disabled, the interrupt is queued for later execution. If the event is caused more than once, then multiple instances of the interrupt are queued for execution. When a later statement enables the interrupt, the queued interrupts are executed one at a time in reverse chronological order.

All interrupts are implicitly disabled while any interrupt is executing. That is, any interrupts that are caused while an interrupt is executing are queued for later execution. When the interrupt completes, the queued interrupts are executed one at a time in reverse chronological order.

Because the queuing of interrupts creates substantial overhead for a process, you should leave the interrupt in the enabled state whenever possible.

The following are examples of ALGOL statements that enable and disable an interrupt. Each statement can specify only one interrupt:

```
ENABLE INT1;
DISABLE INT1;
```

The following are examples of COBOL74 statements that enable and disable multiple interrupts:

```
ALLOW INT1, INT2.
DISALLOW INT1, INT2.
```

Using General Disable and Enable Statements

You can use a general disable statement to disable all the interrupts declared by the process. Interrupts declared in other related processes, such as a parent or offspring, are not affected. While a general disable is in effect, any interrupts whose events are caused are queued for later execution.

To again enable the interrupts that were disabled by the general disable statement, use a general enable statement. For the most part, the general enable statement does not

enable interrupts that were already disabled when the general disable statement was entered. However, if a statement enables a specific interrupt while a general disable statement is in effect, then the general enable statement also enables that interrupt.

The following ALGOL statements illustrate the interaction of specific and general enables and disables for three interrupts, INT1, INT2, and INT3:

```
ENABLE INT1;   % Enables INT1.
DISABLE INT2;  % Disables INT2.
DISABLE INT3;  % Disables INT3.
DISABLE;       % Disables INT1.  INT2 and INT3 remain disabled.
ENABLE INT2;   % All three events remain disabled.
ENABLE;        % Enables INT1 and INT2.  INT3 remains disabled.
```

The following are the general disable and enable statements in COBOL74:

```
DISALLOW INTERRUPT.
ALLOW INTERRUPT.
```

Waiting for Interrupts

You can use a special form of the wait statement to make the process wait for interrupts. While the process is waiting for interrupts, any interrupt can execute; as soon as the interrupt completes, the process returns to its waiting state. The only way the process can proceed any further is if an interrupt executes a bad GO TO statement that transfers control to a different statement outside the interrupt.

Waiting for interrupts can be useful for processes, such as message control systems (MCSs), that are driven by input received over time from a variety of sources. However, waiting on multiple events might be more efficient in these cases; refer to "Efficiency Considerations" later in this section.

In ALGOL, the following wait statement causes the process to wait for interrupts:

```
WAIT;
```

The COBOL74 equivalent is the following statement:

```
WAIT UNTIL INTERRUPT.
```

Efficiency Considerations

The event and interrupt features provide a very efficient method of synchronizing processes, provided that they are used as intended. However, some misuses of these features can cause performance problems. The following subsections describe some possible problems and ways to avoid them.

Buzz Loops

Several of the event-related statements allow a process to test the state of an event without causing the process to wait. These are the happened test, the availability test, the conditional procure statement, and the partial liberate statement.

These statements are designed for occasional, rather than frequent, use because each execution of the statement uses processor time. In particular, looping continuously on these statements is a very inefficient way of making a process wait. Such a loop is called a buzz loop. The following is an ALGOL example of such a loop:

```
WHILE NOT HAPPENED (E1) DO;
```

This loop repeats the happened test over and over until the event E1 attains a state of HAPPENED. This loop causes two problems:

- It wastes processor time that could be devoted to executing other processes, including the process that will eventually cause the event.
- On a single-processor system, it could become an infinite loop. Assume that another process is supposed to cause event E1. If the looping process has higher priority, it will completely monopolize the processor. The second process never executes and thus never causes event E1.

You should replace the buzz loop with some form of the wait statement, which does not use any processor time. The ALGOL statement *WAIT (E1)* could replace the loop shown in the preceding example.

Preventing Excessive Interrupt Overhead

Use of interrupts increases the processor usage of a process. The processor overhead is small if only one interrupt is used and the interrupt is not often caused. However, the overhead is much greater when multiple interrupts are used and greater still when interrupts are queued because an interrupt was disabled.

By contrast, a wait statement does not cause any continuing drain on processor resources. A process that executes a wait statement is simply ignored until the associated event is caused.

Because of these facts, wait statements should be used in preference to interrupts where possible. This is particularly true where the process needs to wait on several events simultaneously. In these cases, a statement that waits on multiple events is more efficient than a statement that waits on multiple interrupts.

Preventing Starvation Problems

A process that waits on multiple events must be carefully designed or there is a possibility that some events might be overlooked. This possibility arises because the value returned by the wait statement always indicates the leftmost of the events in the event list that have been caused. For example, consider the following ALGOL statement:

```
ENUM := WAIT (E1, E2, E3);
```

If E1 is caused, ENUM receives a value of 1. If E1 and E2 are caused, ENUM still receives a value of 1. Now, suppose that E1 is an event that happens very frequently. Each time the wait statement is executed and E1 has already happened, the wait statement returns 1 as a value; thus, the process might never be notified that event E2 or E3 has happened. This situation is referred to as a *starvation problem.*

Strictly speaking, a starvation problem exists only if the repeated wait statement is not fulfilling the needs of the particular application. The effect of the wait statement is to give preference to the leftmost events in the event list. But if the leftmost events occur infrequently, there will be no starvation. If you order the list so that the most important events are on the left, then the starvation condition might even be desirable.

However, if you want to ensure that no event can be overlooked, then a simple solution is to use happened tests after each execution of the wait statement. You could apply a happened test to each event that is to the right of the event that was returned by the wait statement. The following is an ALGOL example of a procedure that uses this technique:

```
PROCEDURE EVENTWAIT;
BEGIN
   BOOLEAN BOOL;
   INTEGER ENUM;
   DO BEGIN
      ENUM := WAIT (E1, E2, E3);
      CASE ENUM OF
        BEGIN
          1: BOOL := INPUTHANDLER (TRUE, HAPPENED(E2), HAPPENED(E3));
          2: BOOL := INPUTHANDLER (FALSE, TRUE, HAPPENED(E3));
          3: BOOL := INPUTHANDLER (FALSE, FALSE, TRUE);
        END;
   END
   UNTIL BOOL;
END EVENTWAIT;
```

The procedure EVENTWAIT is responsible for waiting on three events, E1, E2, and E3, which were declared globally. When at least one of these events is caused, EVENTWAIT invokes another procedure called INPUTHANDLER and passes it Boolean values indicating whether each of the three events has been caused. The ENUM value indicates the leftmost event that has happened. The happened test is used for each of

the events to the right of that event. You increase efficiency by minimizing the number of happened tests.

INPUTHANDLER is expected to make whatever response is appropriate for each event. INPUTHANDLER returns a Boolean value of TRUE if there is no need to wait on any more events. INPUTHANDLER is also expected to reset the events that were caused, so that it will be meaningful to wait on them again.

Note that the INPUTHANDLER invocation is used in this example for the sake of simplicity. From an efficiency standpoint, such repeated procedure invocations are rather expensive. It would be better to include the code that handles each event in the EVENTWAIT procedure.

Discontinued Processes and Events

When a number of processes are being synchronized through the use of events, unexpected problems can occur if one of the processes is discontinued. A process might be discontinued by the system because of an error, or by an operator using a DS (Discontinue) system command.

If the process has procured an event, but has not yet liberated it, then the event remains procured when the process is discontinued. Any other processes attempting to unconditionally procure the event will wait indefinitely.

Similarly, if the process was supposed to execute a cause statement, but was discontinued first, then the event is never caused. Other processes waiting on the event will wait indefinitely.

The programmer can ignore these problems if none of the processes using an event is ever likely to be discontinued. However, in environments such as a SHAREDBYALL library, where a large number of user processes from various sources can access the same event, the programmer might want to take special precautions. The following subsections describe methods of dealing with these problems.

Using EPILOG and EXCEPTION Procedures

An EPILOG procedure is a special type of procedure that is available only in DCALGOL. An EPILOG procedure is executed whenever the block that declares it is exited, even if the block exit was caused by the process being discontinued. The EPILOG procedure can be designed to perform cleanup actions, such as liberating or causing an event.

An EPILOG procedure can determine whether the block exit is normal or whether the process is being discontinued, by inspecting the STATUS, HISTORYTYPE, HISTORYCAUSE, and HISTORYREASON task attributes of the MYSELF task variable. You can design the EPILOG procedure to take different actions, depending on whether the block exit is normal.

Note that the EPILOG procedure can itself be discontinued and, thus, prevented from completing all its cleanup functions. For example, if you enter two DS commands for a process, the first causes the EPILOG procedure to be entered. The second DS command

discontinues the EPILOG procedure if it has not yet finished executing. This problem should rarely occur if the EPILOG procedure is kept brief.

If you need to ensure that certain actions are always performed when a procedure is exited abnormally, you can use an EXCEPTION procedure instead of an EPILOG procedure. EXCEPTION procedures are available in DCALGOL, DMALGOL, and NEWP. These procedures serve a similar function to EPILOG procedures. However, an EXCEPTION procedure is executed only if the block that declares it is exited abnormally, whereas an EPILOG procedure is executed even if the block exit is normal. Block exits are considered abnormal in either of the following cases:

- If the block is exited because of a bad GO TO statement. This is a GO TO statement that transfers control to a label outside the block.
- If the block is exited because the process was discontinued, either because of an operator DS (Discontinue) system command or an internal fault.

Another important feature of EXCEPTION procedures is that you can prevent them from being interrupted. To do this, you simply add the PROTECTED clause to the EXCEPTION procedure declaration. The PROTECTED clause is available in DMALGOL and NEWP, but not in DCALGOL. If the block that declares a protected EXCEPTION procedure is exited abnormally, then the EXCEPTION procedure executes in protected mode. A protected EXCEPTION procedure cannot be interrupted by the DS (Discontinue) or ST (Stop) system commands, or by stack stretches. Note, however, that the system marks an object code file as nonexecutable if it contains a protected EXCEPTION procedure. An operator must use the *MP <file title> + EXECUTABLE* form of the MP (Mark Program) system command or the SL (Support Library) system command before the object code file can be executed.

If you want an EXCEPTION procedure to be executed before any block exit, normal or abnormal, you can include an explicit call on the EXCEPTION procedure in the block. The following is an example:

```
700 PROCEDURE P1;
710 BEGIN
720   FILE MYFILE(KIND=DISK);
730   PROTECTED EXCEPTION PROCEDURE CLEANUP;
740   BEGIN
750     CLOSE(MYFILE,LOCK);
760   END;
        .
        .
        .
900   CLEANUP;
910 END;
```

The vertical ellipsis points in this example denote lines that are omitted because they are not essential to the point being illustrated. If P1 exits normally, then the EXCEPTION procedure CLEANUP is explicitly invoked by the statement at line 900. Note that in this case, CLEANUP executes without protected status. If P1 exits abnormally, the system automatically invokes CLEANUP and executes it with protected status.

Using Timed Wait Statements

By including a time limit on a wait statement, you can make it possible for a process to recover if a particular important event is not caused. For example, the following statement could be used in ALGOL:

```
ENUM := WAIT ((120),E1);
```

This statement waits for 120 seconds or until event E1 is caused, whichever comes first. For example, you might know that if E1 is not caused within 120 seconds, then something has gone wrong. The process could check the value of ENUM to determine if the wait timed out. If so, the process could check the STATUS task attribute of the process that was supposed to cause the event and find out whether that process was discontinued. (This type of checking is possible only if the process has access to the task variable of the process that was supposed to cause the event.)

Using Conditional Procure Statements

There is no direct way to set a time limit on an unconditional procure statement. One alternative is to use a conditional procure statement, such as the FIX statement in ALGOL or a LOCK statement with an AT LOCKED clause in COBOL74. If the conditional procure fails, the process could attempt it again after a specified time period. (Note that the process should not execute conditional procures in rapid succession, as this causes the problem discussed under "Buzz Loops" earlier in this section.) If several conditional procures fail, the process could check the status of other processes that might have procured the event.

Determining Whether to Liberate an Event

If the state of an event is NOT AVAILABLE, then the process that most recently procured the event can be referred to as the *owner* of that event. A process can use the MCP procedure EVENT_STATUS to determine whether that process is the current owner of an event. The EVENT_STATUS procedure is especially useful in fault-handling code and in EPILOG and EXCEPTION procedures. Refer to "Determining the Ownership of an Event" earlier in this section.

Example of Event Usage

The following is a simplified example of an online application that has one driver process and three servers. The driver process reads input from users and passes it on to whichever server is not currently busy. The underlying assumption is that the user is capable of submitting input faster than any single server can process it; this could be the case if the server has to perform many time-consuming actions, such as disk I/Os, to process the input. However, this example concentrates on the timing and resource control aspects of this situation, and so the servers in the example do not really do any useful work.

```
100 BEGIN
110 FILE TERM(KIND=REMOTE);
120 BOOLEAN FINISHED;
130 EBCDIC ARRAY MSG[0:71];
140 EVENT INMSG_EVENT, MSG_READ;
150 INTEGER I, READNUM;
160 TASK T1, T2, T3;
170
180 PROCEDURE SERVER;
190 BEGIN
200   BOOLEAN DONE;
210   EBCDIC ARRAY MSGCOPY[0:71];
220   WHILE NOT DONE DO
230   BEGIN
240     PROCURE(INMSG_EVENT);
250     REPLACE MSGCOPY BY MSG FOR 72;
260     CAUSE(MSG_READ);
270     IF MSGCOPY = "QUIT" THEN
280        DONE := TRUE
290     ELSE BEGIN
300            REPLACE MSGCOPY[68] BY MYSELF.MIXNUMBER FOR 4 DIGITS;
310            WRITE(TERM,72,MSGCOPY);
320          END;
330   END;
340 END;
350
360 PROCURE(INMSG_EVENT);
370 PROCESS SERVER [T1];
380 PROCESS SERVER [T2];
390 PROCESS SERVER [T3];
400
410 OPEN(TERM);
420 WHILE NOT FINISHED DO
430 BEGIN
440   WAIT(TERM.INPUTEVENT);
450   READ(TERM,72,MSG);
460   IF MSG = "QUIT" THEN
470      BEGIN
480        FINISHED := TRUE;
490        READNUM := 3;
500      END
```

```
510   ELSE READNUM := 1;
520   I := 1;
530   WHILE I LEQ READNUM DO
540     BEGIN
550       LIBERATE(INMSG_EVENT);
560       WAITANDRESET(MSG_READ);
570       I := * + 1;
580     END;
590 END;
600
610 WHILE T1.STATUS GTR VALUE(TERMINATED) OR
620       T2.STATUS GTR VALUE(TERMINATED) OR
630       T3.STATUS GTR VALUE(TERMINATED) DO
640       WAITANDRESET(MYSELF.EXCEPTIONEVENT);
650
660 END.
```

The communication in this example takes place between the parent process and three asynchronous tasks that are instances of procedure SERVER. The communication takes place by way of the array MSG and the events INMSG_EVENT and MSG_READ. Of these, MSG is used to convey messages from the parent process to the servers. The parent process uses INMSG_EVENT to inform the servers that there is a message waiting to be read. A server uses MSG_READ to inform the parent that it has successfully read the message, so the parent can now reuse the MSG array.

When this program is initiated, the driver process procures INMSG_EVENT and initiates three instances of the SERVER procedure. Each of these servers begins by attempting to procure INMSG_EVENT; since the driver has already procured this event, all the servers wait.

The driver process then enters the loop on lines 420-590. Within this loop, the driver waits for input from a user to appear in the remote file, and then reads the input into MSG. In most cases, the driver then liberates INMSG_EVENT and waits on the MSG_READ event. When the driver liberates INMSG_EVENT, one of the servers succeeds in procuring the event and copies the contents of MSG to the local array MSGCOPY. The server then causes MSG_READ, informing the driver that MSG is again available for use as a buffer. The server then performs some processing on the input in MSGCOPY and notifies the user of the result by writing a message to the remote file.

If the input received from the user is the command QUIT, then the driver takes some special actions. It liberates INMSG_EVENT and waits on MSG_READ three times without performing any more read operations. This allows the contents of the MSG array to be read by each of the three servers. Each server recognizes the QUIT command and terminates gracefully. Then the driver terminates as well.

Note that this program uses the available state of INMSG_EVENT, but uses the happened state of MSG_READ. This difference reflects the different purposes for which these events are used. The program alternates between two phases: a phase in which the driver uses the MSG array, and a phase in which any single one of the servers can use the MSG array. Causing MSG_READ initiates the phase in which the driver uses MSG; liberating INMSG_EVENT initiates the phase in which one of the waiting servers is allowed to use MSG.

Section 17
Using Parameters

A *parameter* is an object passed to a procedure by the procedure invocation statement. Note that the term "procedure" is used here, as it is throughout this guide, to refer to complete programs as well as to subroutines within a program. Most A Series programming languages can pass parameters to procedures. Parameters can be of many types, and in each language, most or all of the types of available variables can be passed as parameters.

Each parameter has two aspects: an *actual parameter* and a *formal parameter*. The actual parameter is the parameter specified in the procedure invocation statement. The formal parameter is the parameter as it is declared in the procedure that is being invoked.

Parameters that are used in a process initiation statement provide an avenue of communication between the initiating process and the new process. Such parameters are referred to hereafter as *tasking parameters*.

Parameters that are used in a library procedure invocation statement provide another type of interprocess communication. Such parameters are hereafter referred to as *library parameters*.

The "Determining the Scope of Parameters" and "Parameter Passing Modes" subsections of this section provide information that is relevant to both tasking parameters and library parameters. The remainder of this section addresses only tasking parameters. For further information about library parameters, refer to Section 18, "Using Libraries."

Determining the Scope of Parameters

Section 15, "Using Global Objects," defined the *scope* of a declaration as all the blocks in a program that have access to an object declared in the program. That section explained how the scope of a declaration extends through nested blocks in a program.

You can use parameters to pass an object to a procedure that does not fall within the scope of the declaration of that object. This fact makes parameters a more general tool for IPC than global objects. A parameter can increase the scope of an object in the following ways:

- An object can be passed as a parameter to a procedure that is not nested within the block that declares the object.
- A parameter can be passed to an external procedure, whether the procedure is a passed external procedure, a library procedure, or a separate program.

The objects a procedure can access, because the objects are declared in the procedure or are declared globally to the procedure, are referred to as the *direct addressing environment* of the procedure. The objects in the direct addressing environment, together with any objects passed as parameters to the procedure, comprise the *extended addressing environment* of the procedure.

If a procedure is passed as a parameter to another procedure, the invoked procedure gains access to the passed procedure. However, the scope is extended only one way. The passed procedure does not automatically gain access to objects declared in the invoked procedure.

The following ALGOL example includes two cases where the scope of a declaration has been increased by the effects of parameter passing:

Example

```
100 BEGIN
110
120 PROCEDURE P(Q);
130   PROCEDURE Q (R);
140     REAL R;
150     FORMAL;
160 BEGIN
170   REAL A;
180   A := 2;
190   Q(A);
200   DISPLAY (STRING(A,*));
210 END;
220
230 PROCEDURE Y;
240 BEGIN
250   PROCEDURE X(Z);
260   REAL Z;
270   BEGIN
280     Z := Z * 2;
290   END;
300
310   P(X);
320 END;
330
340 Y;
350
360 END.
```

Case 1

Procedure X cannot be directly invoked by a statement in procedure P, because the declaration of procedure X occurs within procedure Y. However, the statement at line 310 that invokes procedure P passes procedure X as an actual parameter to the formal parameter Q. Thus, the statement at line 190, which invokes the formal parameter Q,

actually results in an invocation of procedure X. In this way, a statement in procedure P is able to invoke a procedure outside the direct addressing environment of procedure P.

Case 2

Real variable A cannot be directly accessed by a statement in procedure X because A is declared within procedure P. However, the statement at line 310 passes procedure X as an actual parameter to formal parameter Q of procedure P. The statement in procedure P at line 190 then passes A as a parameter to procedure Q, thus making it possible for procedure X to access A.

Even after being passed to P, X does not automatically have access to objects declared in P. Thus, X could not have accessed A if A had not been passed as a parameter to X.

Parameter Passing Modes

There are several different *passing modes* that govern the relationship between the actual parameter and the formal parameter. The passing mode determines, for example, whether assignments made to the formal parameter are reflected by the actual parameter. The passing mode can also make a large difference in program performance in cases where the actual parameter is an expression. The three types of passing modes available on A Series systems are call-by-value, call-by-name, and call-by-reference.

The following subsections describe the three types of passing modes and explain how you can specify which passing mode is used.

Call-by-Value Parameters

If a parameter is passed by value, the system evaluates the actual parameter when the procedure is invoked and assigns the value to the formal parameter. Changes made to the value of the formal parameter do not affect the value of the actual parameter. Similarly, any changes made to the value of the actual parameter after procedure invocation do not affect the value of the formal parameter.

An advantage to using call-by-value parameters is that they never result in the accidental creation of a *thunk*. (Thunks are defined in the discussion of call-by-name parameters that follows.) Another advantage is that they simplify program structure. Because the actual parameter and the formal parameter do not affect each other, new values can be assigned to either without creating unexpected side effects.

Call-by-Name Parameters

When a parameter is passed by name, the system never creates the formal parameter. Instead, the system substitutes the actual parameter for the formal parameter wherever the formal parameter is mentioned in the procedure.

The effect of passing by name is simplest in cases where the actual parameter is a simple variable. When the procedure accesses the formal parameter, the effect is as if the

procedure were using a global variable. Any changes made to the value of the formal parameter immediately affect the value of the actual parameter and vice versa. This feature makes call-by-name parameters a useful means of communicating information between an asynchronous process and its initiator.

When an actual parameter that is a constant or an expression is passed by name, the compiler generates a *thunk*. A thunk (also known as an *accidental entry*) is a piece of code that evaluates the actual parameter and assigns the resulting value to the formal parameter. The system substitutes the thunk for the formal parameter wherever the formal parameter is mentioned in the procedure.

Thunks can be undesirable because they slow execution of the program and affect the definition of the critical block. (Critical blocks are discussed in Section 2, "Understanding Interprocess Relationships.") The programmer can prevent the creation of a thunk by passing each element of the expression as a separate parameter.

If a constant is passed by name, then whenever the value of the formal parameter is read, the formal parameter returns the value of the constant. The value of the formal parameter cannot change. An attempt to assign a value to the formal parameter results in a run-time error.

The effect of passing an expression by name varies, depending on whether the expression evaluates as a reference to a single object. For example, A[I] evaluates into a reference to a single element of array A. In this guide, such an expression is referred to as a *simple expression*. Other examples of simple expressions are the POINTER function in ALGOL and references to character-based record fields. On the other hand, an expression such as A + B does not evaluate as a reference to a single element. Such an expression is referred to as a *complex expression*.

For a simple expression, the system passes a thunk that reevaluates the expression each time the the parameter is used in the procedure. For example, suppose the actual parameter A[I] is passed to the formal parameter F. At the time of the procedure invocation, I has a value of 5. The formal parameter F becomes a reference to element 5 of array A. When F is read, it reflects the most recent value of A[5]. When F is assigned, it changes the value of A[5]. If I is then assigned a value of 10, F becomes a reference to A[10]. Thereafter, reading or assigning F really accesses the value stored in A[10].

For a complex expression, the system passes a thunk that reevaluates the expression each time the formal parameter is read in the procedure. However, it is impossible to assign a value to the formal parameter; any attempt to do so results in a run-time error.

Call-by-Reference Parameters

When a parameter is passed by reference, the system passes the formal parameter a reference to the place where the actual parameter is stored in memory. Passing a parameter by reference is essentially the same as passing it by name, except that the compiler does not create a thunk for a call-by-reference parameter. Any expressions passed by reference are, therefore, evaluated immediately and changed into simple values or pointers to simple values.

The effects of passing a parameter by reference are somewhat different in FORTRAN77 and FORTRAN than in other languages. In non-FORTRAN languages, the effects of passing by reference are as follows:

- When a simple variable is passed by reference, the effect is the same as if it had been passed by name. Changes made to the value of the formal parameter immediately affect the value of the actual parameter and vice versa.
- In most languages, constants and complex expressions cannot be passed by reference; a syntax error results from an attempt to do so. However, simple expressions can be passed by reference. For a simple expression, the system passes a reference to the location of the element. This location never changes, even if the value of the subscript later changes. For example, suppose the actual parameter A[I] is passed to the formal parameter F and I has a value of 5. Formal parameter F becomes a reference to array element A[5]. Even if I is later assigned a different value, F remains a reference to A[5]. When F is read, it reflects the most recent value of A[5]. When F is assigned, it changes the value of A[5].

In FORTRAN77 and FORTRAN, the effects of passing by reference are as follows:

- For simple variables of type integer, real, double precision, complex, or logical, two different kinds of call-by-reference passing are available. The default method is known as *call-by-value-result*. With this method, the value of the actual parameter is assigned to the formal parameter. Thereafter, assignments to the actual parameter have no effect on the formal parameter. Assignments to the formal parameter have no immediate effect on the actual parameter; however, when the procedure is exited, the value of the formal parameter is assigned to the actual parameter. The alternate method is true *call-by-reference* passing, in which the formal parameter receives a reference to the actual parameter itself; changes to the actual parameter are immediately visible to the formal parameter and vice versa. The programmer can request true call-by-reference passing by enclosing the formal parameter in slashes (/).
- For parameters that are character variables, arrays, or subprograms, the parameter is always treated as a true call-by-reference parameter. Any changes to the actual parameter are immediately visible to the formal parameter and vice versa.
- Constants of type integer, real, double precision, complex, or logical can be passed by reference, but character or array constants cannot. The receiving procedure can make assignments that change the value of the formal parameter, but the value of the actual parameter is never updated to reflect the change.
- For an actual parameter that is a simple expression, the parameter is treated as either call-by-value-result, or true call-by-reference, depending on the way the formal parameter is declared. If the formal parameter is a character variable or array, then the parameter is treated as a true call-by-reference parameter. If the formal parameter is an integer, real, double precision, complex, or logical variable, then by default the parameter is treated as call-by-value-result; however, if the formal parameter is enclosed in slashes, the parameter is treated as true call-by-reference.
- For an actual parameter that is a complex expression, the system evaluates the expression and passes the value to the formal parameter. The receiving procedure can make assignments that change the value of the formal parameter, but the value of the actual parameter remains unchanged.

Read-Only Parameters

A concept related to parameter passing modes is that of *read-only parameters*. The term "read-only" refers, not to a passing mode, but to a restriction on the ways a parameter can be used.

Formal parameter declarations in a Pascal program can include a CONST clause, which causes a parameter to be treated as a *read-only parameter*. The CONST clause prevents the receiving Pascal program or procedure from making any changes to the value of the formal parameter. However, the CONST clause does not guarantee that the formal parameter has a constant value. The formal parameter value can change because the CONST clause does not affect the passing mode. If the actual parameter is passed by name or by reference, then any changes made by the initiator to the value of the actual parameter are immediately reflected in the value of the formal parameter.

Specifying the Passing Mode

You will seldom have the opportunity to choose among all three of these passing modes for a particular parameter. The choice of passing modes is restricted on the basis of several different considerations, including parameter type, language, and process type.

Though there are many different parameter types, these types fall into two basic categories: *word* and *descriptor*. Boolean variables, integer variables, and real variables are examples of word types. Strings, arrays, files, and other complex data structures are descriptor types.

Word-type parameters can be passed by value, by name, or by reference.

In most languages, descriptor-type parameters must be passed by name or by reference. Exceptions are Pascal, which allows descriptor type parameters to be passed by value, and WFL, which can pass strings by value. Also, message control systems (MCSs) and Host Services tasking can pass descriptor type parameters by value. Host Services tasking makes it possible to write a program that passes an array to a remote process by value. (Remote processes are discussed in Section 12, "Tasking across Multihost Networks.")

Each language imposes a different set of restrictions on the passing mode. For example, ALGOL passes descriptor types by name or by reference and word types by name, by reference, or by value. COBOL(68) and COBOL74 pass all library parameters by reference, and parameters to tasks or bound-in procedures by reference or by value. WFL passes parameters either by reference or by value. For details about these language restrictions, refer to the programming language reference manuals.

One additional restriction is based on the process type. A statement that initiates an independent process can pass parameters only by value, not by name or by reference.

Using Tasking Parameters

A Series software provides the application programmer with the ability to design a program in one language that initiates a program written in a different language. The

initiating program can even pass parameters to the initiated program. However, because each language provides a different set of parameter types, the programmer needs to understand which types of parameters are compatible.

The languages that can initiate a process and pass it parameters are ALGOL, COBOL(68), COBOL74, and Work Flow Language (WFL).

The languages that can receive tasking parameters from another program are ALGOL, C, COBOL(68), COBOL74, COBOL85, Pascal, and PL/I.

WFL jobs can also receive parameters. However, strictly speaking, these are compile-time rather than tasking parameters because a WFL job is recompiled each time it is submitted. ALGOL, COBOL(68), COBOL74, FORTRAN, and RPG can all submit WFL jobs, but none of them can pass a parameter to the WFL job. Parameters can be passed to a WFL job only by a START statement. START statements can be submitted in Command and Edit (CANDE) or Menu-Assisted Resource Control (MARC) sessions or at an operator display terminal (ODT). START statements can also be submitted by DCALGOL programs using the DCKEYIN function or by WFL jobs.

The remainder of this section discusses only tasking parameters and not WFL compile-time parameters.

Whenever a process passes a tasking parameter, the system software checks that the number of actual parameters the calling program passes matches the number of formal parameters declared in the receiving program.

The system also compares each actual parameter with the matching formal parameter to determine if they are of compatible types. The matching is done based on parameter order rather than parameter names. It is permissible for the actual and formal parameters to have different names.

In many cases, the system allows matches between similar, though not identical, parameter types. For instance, an integer actual parameter can generally be passed to a real formal parameter. Also, types that are, in effect, identical might have different names in different languages. Details about which parameter matches are allowed by the system software are given under "Matching Each Parameter Type" later in this section.

Information about how the passing mode is determined for tasking parameters is given under "Resolving Passing Mode Conflicts" later in this section.

Special considerations for arrays passed as tasking parameters are discussed under "Passing Arrays" later in this section.

Matching Each Parameter Type

By using Tables 17–1 and 17–2 at the end of this subsection, you can find out what parameter types in a given language match particular parameter types in any other given language. In the following discussion, the term *original parameter* refers to the parameter you want to find a match for. The original parameter might be either an actual parameter or a formal parameter. The term *matching parameter* refers to the

parameter about which you are uncertain. The tables can help you decide what type the matching parameter should be.

Note: *The tables in this section document the tasking parameter-type-matching rules enforced by the system at process initiation time. If you are initiating an imported library procedure, you should also be aware of the library parameter matching rules discussed in Section 18, "Using Libraries." These rules are enforced by the operating system at library linkage time. In general, the library parameter matching rules are much stricter than the tasking parameter matching rules.*

Further, if you are initiated a bound-in procedure, you should be aware of the binding parameter matching rules discussed in the A Series Binder Programming Reference Manual. *These rules are enforced by the Binder during its run, and in general are still more limiting than the library parameter matching rules.*

To use the parameter matching tables, you must start out knowing the following characteristics of the original parameter: the language, the name of the parameter type, and whether it is a formal or an actual parameter. For the matching parameter, you must know the language in which it will be specified.

Begin by looking at Table 17–1. Table 17–1 is separated into three columns labeled Language, Parameter Type, and General Type. Look down the Language column until you find the language of your original parameter. Next, scan down the Parameter Type column until you find the type of your original parameter. Next, look immediately to the right, in the General Type column, and make a note of the general type that is listed there.

In some cases, the general type shown is "(Unique)" instead of a word or a phrase. This means that your original parameter is of a unique type that does not match any other parameter type. For example, an ALGOL Boolean direct array can be passed only to another ALGOL Boolean direct array. In this case, you can skip the rest of these directions, because there are no other matching parameter types to be found.

Next, look at Table 17–2. This table extends over several pages and each page includes one or more boxes; each box is a separate entry. A General Type heading appears at the upper left of each box. The boxes appear in alphabetical order based on the General Type headings. Look for the box whose General Type heading corresponds to the general type you noted earlier.

Within the box you selected, scan down the Language and Parameter Type columns. Make a note of the parameter types that are in the language you want to find out about.

At this stage, you can consider yourself finished if you want to be. You can take the parameter types you noted and look in the appropriate programming language reference manual for the detailed syntax of the parameter types. However, if this initial search did not uncover any parameters in the language you want, or if you want a more complete list of the possible parameter types for the matching parameter, then the information in the Special Matches column of the box can help you to extend your search.

The Special Matches column of each box can include up to three subentries that list general types that match your original parameter, but only in some limited circumstances. Examine each of the subentries that appears in the Special Matches column of the box. The following are the possible subentries and their meanings:

- Matching Actuals

  This is a list of general types that can match your original parameter, provided that your original parameter is a formal parameter and you are looking for an actual parameter to match it. If you are looking for an actual parameter, then make a note of each of these general types. Then, for each of these general types, do the following:

  - Go to the box that is labeled with the name of the specific general type.
  - Look at the main parameter group in the box and note any parameter types shown that are in the language you want for your matching parameter.
  - Ignore any Matching Actuals, Matching Formals, or COBOL Matches subentries that appear in the box.

- Matching Formals

  This is a list of general types that can match your original parameter, provided that your original parameter is an actual parameter and you are looking for a formal parameter to match it. To translate these general types into specific parameter types, follow the same steps that you did for the Matching Actuals subentry.

- COBOL(68 & 74) Matches

  This is a list of general-type matches that are allowed if the calling program or the receiving program is written in COBOL(68) or COBOL74. If this is the case, then note the general types shown. For each general type, do the following:

  - Go to the box that is labeled with the name of the specific general type.
  - If the original parameter is in COBOL(68) or COBOL74, then note any parameters shown in the Parameter Types column that are in the language you want for your matching parameter. If the original parameter is not in COBOL(68) or COBOL74, then note only the COBOL(68) and COBOL74 types that appear in the Parameter Types column.
  - Ignore any Matching Actuals, Matching Formals, or COBOL Matches subentries that appear in the box.

You now have a complete list of the possible parameter types for your matching parameter. Refer to the various programming language reference manuals for the syntax used to declare the parameter types you have listed.

Note that the programming languages restrict some parameter types so that they can be used only as formal parameters or only as actual parameters. The syntax given in the programming language reference manuals should explain any such restrictions.

The following examples illustrate the method for finding matching parameter types.

Example 1

Suppose you want to pass a string value from a WFL job to an ALGOL program. Look at the last line of Table 17–1. The parameter type shown is WFL STRING. The general type shown is Real Array.

Now look through Table 17–2 until you find the box labeled Real Array. The main parameter group in the box includes two ALGOL types: REAL ARRAY and REAL VALUE ARRAY.

In addition, the Matching Formals list specifies the general type integer array. Look at the box labeled Integer Array. The main parameter group in the box includes the following ALGOL types: INTEGER ARRAY and INTEGER VALUE ARRAY.

You now have a list of four different ALGOL parameter types. However, if you refer to the ALGOL manual, you will find that value arrays are not allowed as formal parameters (although they can be actual parameters). Therefore, a WFL STRING parameter can be passed to two ALGOL types: REAL ARRAY or INTEGER ARRAY.

Example 2

Suppose you want to pass an 01 DISPLAY Group Item from a COBOL74 program to an ALGOL program. Look through Table 17–1 until you find the line that says COBOL74 01 DISPLAY Group Item. The general type shown is EBCDIC Array.

Now look through Table 17–2 until you find the box labeled EBCDIC Array. The ALGOL parameter types shown in the box are EBCDIC ARRAY and EBCDIC VALUE ARRAY. Note these.

The Special Matches column of the box includes three general types as COBOL(68 & 74) Matches: Hex Array, Integer Array, and Real Array. Go to the box for Hex Array. The Parameter Types column includes two ALGOL types: HEX ARRAY and HEX VALUE ARRAY. Note these. Repeat this process for each of the general types that you noted.

When you have finished this process, you have the following list of ALGOL formal parameter types:

```
EBCDIC ARRAY
EBCDIC VALUE ARRAY
HEX ARRAY
HEX VALUE ARRAY
INTEGER ARRAY
INTEGER VALUE ARRAY
REAL ARRAY
REAL VALUE ARRAY
```

Of these, you should discard the value arrays because they cannot be used as formal parameters. The matching parameter could be any of the remaining ALGOL types from this list.

Example 3

Suppose you want to pass a real array from an ALGOL program to a COBOL74 program. Scan through the ALGOL parameters in Table 17–1 until you find REAL ARRAY. The General Type shown is also Real Array.

Now look through Table 17–2 until you find the box labeled Real Array. One COBOL74 parameter is shown in the Parameter Types column: 01 BINARY Group Item. Note this. Additionally, the box contains entries in the Special Matches column for Matching Actuals, Matching Formals, and COBOL(68 & 74) Matches. You can interpret these entries as follows:

- Matching Actuals

  Ignore this entry, as your original parameter is the actual parameter. The matching parameter you are looking for is a formal parameter.

- Matching Formals

  The entry shown under this heading is Integer Array. Go to the box for Integer Array. In the Parameter Type column, you find one COBOL74 parameter: 77 BINARY Elementary Item. Make a note of this.

- COBOL(68 & 74) Matches

  The entries shown under this heading are EBCDIC Array and Hex Array.

  – Go to the box for EBCDIC Array. The COBOL74 parameters shown in the Parameter Type column are 01 DISPLAY Group Item and 01 KANJI Group Item. Make a note of these.

  – Go to the box for Hex Array. The COBOL74 parameters shown in the Parameter Type column are 01 COMP Group Item and 01 INDEX Group Item. Make a note of these.

When you finish this process, you find that the COBOL74 formal parameter can be of any of the following types:

```
Ø1 BINARY Group Item
Ø1 COMP Group Item
Ø1 DISPLAY Group Item
Ø1 INDEX Group Item
Ø1 KANJI Group Item
77 BINARY Elementary Item
```

Example 4

Suppose you want to pass a HEX DIRECT ARRAY from an ALGOL program to a COBOL74 program. Look through Table 17–1 until you find the line that lists ALGOL HEX DIRECT ARRAY. The General Type column lists "(Unique)" instead of the general type. This means that HEX DIRECT ARRAY is a unique parameter type that can match only a parameter of exactly the same type. In this case, there is no need for you to look at Table 17–2.

Table 17–1. Programming Language Parameter Types

Language	Parameter Type	General Type
ALGOL	ASCII PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	ASCII ARRAY	ASCII Array
ALGOL	ASCII DIRECT ARRAY	(Unique)
ALGOL	ASCII PROCEDURE	(Unique)
ALGOL	ASCII STRING	(Unique)
ALGOL	ASCII STRING ARRAY	(Unique)
ALGOL	ASCII VALUE ARRAY	ASCII Array
ALGOL	BOOLEAN	Boolean
ALGOL	BOOLEAN ARRAY	Boolean Array
ALGOL	BOOLEAN DIRECT ARRAY	(Unique)
ALGOL	BOOLEAN PROCEDURE	Boolean Procedure
ALGOL	BOOLEAN PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	BOOLEAN VALUE ARRAY	Boolean Array
ALGOL	COMPLEX	(Unique)
ALGOL	COMPLEX ARRAY	Complex Array
ALGOL	COMPLEX PROCEDURE	(Unique)
ALGOL	COMPLEX PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	COMPLEX VALUE ARRAY	Complex Array
ALGOL	DIRECT FILE	(Unique)
ALGOL	DIRECT SWITCH FILE	(Unique)
ALGOL	DOUBLE	Double
ALGOL	DOUBLE ARRAY	Double Array
ALGOL	DOUBLE DIRECT ARRAY	(Unique)
ALGOL	DOUBLE PROCEDURE	Double Procedure
ALGOL	DOUBLE PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	DOUBLE VALUE ARRAY	Double Array
ALGOL	EBCDIC ARRAY	EBCDIC Array
ALGOL	EBCDIC DIRECT ARRAY	(Unique)

continued

Table 17–1. Programming Language Parameter Types (cont.)

Language	Parameter Type	General Type
ALGOL	EBCDIC PROCEDURE ARRAY	(Unique)
ALGOL	EBCDIC STRING	(Unique)
ALGOL	EBCDIC STRING ARRAY	(Unique)
ALGOL	EBCDIC VALUE ARRAY	(Unique)
ALGOL	EBCDIC PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	ENTITY REFERENCE	(Unique)
ALGOL	ENTITY REFERENCE ARRAY	(Unique)
ALGOL	EPILOG PROCEDURE	(Unique)
ALGOL	EVENT	Event
ALGOL	EVENT ARRAY	Event Array
ALGOL	FILE	File
ALGOL	FORMAT	(Unique)
ALGOL	HEX ARRAY	Hex Array
ALGOL	HEX DIRECT ARRAY	(Unique)
ALGOL	HEX PROCEDURE	(Unique)
ALGOL	HEX STRING	(Unique)
ALGOL	HEX STRING ARRAY	(Unique)
ALGOL	HEX VALUE ARRAY	Hex Array
ALGOL	HEX PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	INTEGER	Integer
ALGOL	INTEGER ARRAY	Integer Array
ALGOL	INTEGER DIRECT ARRAY	Integer Direct Array
ALGOL	INTEGER PROCEDURE	Integer Procedure
ALGOL	INTEGER PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	INTEGER VALUE ARRAY	Integer Array
ALGOL	LABEL	(Unique)
ALGOL	LIST	(Unique)
ALGOL	PICTURE	(Unique)

continued

Table 17–1. Programming Language Parameter Types (cont.)

Language	Parameter Type	General Type
ALGOL	PICTURE ARRAY	(Unique)
ALGOL	POINTER	Pointer
ALGOL	PROCEDURE (SUBROUTINE)	Procedure
ALGOL	QUERY VARIABLE	(Unique)
ALGOL	QUEUE	(Unique)
ALGOL	QUEUE ARRAY	(Unique)
ALGOL	REAL	Real
ALGOL	REAL ARRAY	Real Array
ALGOL	REAL DIRECT ARRAY	Real Direct Array
ALGOL	REAL PROCEDURE	Real Procedure
ALGOL	REAL PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	REAL VALUE ARRAY	Real Array
ALGOL	SWITCH	(Unique)
ALGOL	SWITCH FILE	(Unique)
ALGOL	SWITCH FORMAT	(Unique)
ALGOL	SWITCH LIST	(Unique)
ALGOL	TASK	Task
ALGOL	TASK ARRAY	Task Array
ALGOL	TRANSACTION RECORD	ALGOL Transaction Record
ALGOL	TRANSACTION RECORD ARRAY	ALGOL Transaction Record
ALGOL	UNTYPED PROCEDURE REFERENCE ARRAY	(Unique)
C	int argc, char* argv[]	Real Array (unbounded)
COBOL(68)	01 ASCII Group Item	EBCDIC Array
COBOL(68)	01 COMP Group Item	Real Array
COBOL(68)	01 COMP-2 Group Item	Hex Array
COBOL(68)	01 CONTROL-POINT Elementary Item	Task
COBOL(68)	01 CONTROL-POINT Group Item	Task Array

continued

Table 17–1. Programming Language Parameter Types (cont.)

Language	Parameter Type	General Type
COBOL(68)	01 DISPLAY Group Item	EBCDIC Array
COBOL(68)	01 EVENT Group Item	Event Array
COBOL(68)	01 INDEX Group Item	Hex Array
COBOL(68)	01 LOCK Group Item	Event Group Item
COBOL(68)	77 COMP-1 Elementary Item	Integer
COBOL(68)	77 COMP-4 Elementary Item	Real
COBOL(68)	77 COMP-5 Elementary Item	Double
COBOL(68)	77 CONTROL-POINT Elementary Item	Task
COBOL(68)	77 EVENT Elementary Item	Event
COBOL(68)	77 LOCK Elementary Item	Event
COBOL(68)	File	File
COBOL(68)	Transaction Record	Transaction Record
COBOL74	01 BINARY Group Item	Real Array
COBOL74	01 COMP Group Item	Hex Array
COBOL74	01 CONTROL-POINT Elementary Item	Task
COBOL74	01 CONTROL-POINT Group Item	Task Array
COBOL74	01 DISPLAY Group Item	EBCDIC Array
COBOL74	01 EVENT Group Item	Event Array
COBOL74	01 INDEX Group Item	Hex Array
COBOL74	01 KANJI Group Item	EBCDIC Array
COBOL74	01 LOCK Group Item	Event Array
COBOL74	77 BINARY Elementary Item	Integer
COBOL74	77 CONTROL-POINT Elementary Item	Task
COBOL74	77 DOUBLE Elementary Item	Double
COBOL74	77 EVENT Elementary Item	Event
COBOL74	77 LOCK Elementary Item	Event
COBOL74	77 REAL Elementary Item	Real
COBOL74	File	File

continued

Table 17–1. Programming Language Parameter Types (cont.)

Language	Parameter Type	General Type
COBOL74	Transaction Record	Transaction Record
COBOL85	01 BINARY Group Item	Integer Array
COBOL85	01 COMP Group Item	Hex Array
COBOL85	01 REAL Group Item	Real Array
COBOL85	01 DOUBLE Group Item	Double Array
COBOL85	01 DISPLAY Group Item	EBCDIC Array
COBOL85	77 REAL Elementary Item	Real
COBOL85	77 DOUBLE Elementary Item	Double
COBOL85	77 BINARY PIC 9(1-11) Elementary Item	Integer
COBOL85	77 BINARY PIC 9(11-23) Elementary Item	Double
COBOL85	77 File	File
Pascal	Array of Boolean	Boolean Array
Pascal	Array of Char	Integer Array
Pascal	Array of Char Subrange	Integer Array
Pascal	Array of Enumeration	Integer Array
Pascal	Array of Enumeration Subrange	Integer Array
Pascal	Array of Explicit Data Type	Real Array
Pascal	Array of Fixed (n < 12)	Integer Array
Pascal	Array of Fixed (n > 11)	Double Array
Pascal	Array of Integer	Integer Array
Pascal	Array of Integer Subrange	Integer Array
Pascal	Array of Packed Array	Real Array
Pascal	Array of Real	Real Array
Pascal	Array of Record	Real Array
Pascal	Array of Set	Real Array
Pascal	Array of Sfixed (n < 12)	Integer Array
Pascal	Array of Sfixed (n > 11)	Double Array

continued

Table 17–1. Programming Language Parameter Types (cont.)

Language	Parameter Type	General Type
Pascal	Array of Vlstring	Real Array
Pascal	Binary (n)	EBCDIC Array
Pascal	Bits (n)	EBCDIC Array
Pascal	Boolean	Boolean
Pascal	Boolean Subrange	Boolean
Pascal	Boolean1	Hex Array
Pascal	Boolean4	Hex Array
Pascal	Char	Integer
Pascal	Char Subrange	Integer
Pascal	Digits (n)	Hex Array
Pascal	Digits_s (n)	Hex Array
Pascal	Display_s (n)	EBCDIC Array
Pascal	Display_z (n)	EBCDIC Array
Pascal	Enumeration	Integer
Pascal	Enumeration Subrange	Integer
Pascal	Explicit Record (call-by-value)	Real Array
Pascal	Explicit Record (var)	EBCDIC Array
Pascal	Fixed (n < 12)	Integer
Pascal	Fixed (n > 11)	Double
Pascal	Function: Boolean	Boolean Procedure
Pascal	Function: Boolean Subrange	Boolean Procedure
Pascal	Function: Char	Integer Procedure
Pascal	Function: Char Subrange	Integer Procedure
Pascal	Function: Enumeration	Integer Procedure
Pascal	Function: Enumeration Subrange	Integer Procedure
Pascal	Function: Fixed (n < 12)	Integer Procedure
Pascal	Function: Fixed (n > 11)	Double Procedure
Pascal	Function: Integer	Integer Procedure
Pascal	Function: Integer Subrange	Integer Procedure

continued

Table 17–1. Programming Language Parameter Types (cont.)

Language	Parameter Type	General Type
Pascal	Function: Real	Real Procedure
Pascal	Function: Sfixed (n < 12)	Integer Procedure
Pascal	Function: Sfixed (n > 11)	Double Procedure
Pascal	Hex (n)	Hex Array
Pascal	Integer	Integer
Pascal	Integer Subrange	Integer
Pascal	Integer48 (n)	EBCDIC Array
Pascal	Integer96 (n)	EBCDIC Array
Pascal	Long Set (> 48 Elements in Set)	Real Array
Pascal	Packed Array of Boolean	Hex Array
Pascal	Packed Array of Char	EBCDIC Array
Pascal	Packed Array of Enumeration	
	(0-16 Elements)	Hex Array
	(17-256 Elements)	EBCDIC Array
	(> 256 Elements)	Integer Array
Pascal	Packed Array of Fixed (n < 12)	Integer Array
Pascal	Packed Array of Fixed (n > 11)	Double Array
Pascal	Packed Array of Integer	Integer Array
Pascal	Packed Array of Real	Real Array
Pascal	Packed Array of Record	Real Array
Pascal	Packed Array of Set	Real Array
Pascal	Packed Array of Sfixed (n < 12)	Integer Array
Pascal	Packed Array of Sfixed (n > 11)	Double Array
Pascal	Packed Array of Subrange	
	(0-16 Elements)	Hex Array
	(17-256 Elements)	EBCDIC Array
	(> 256 Elements)	Integer Array
Pascal	Packed Array Of Vlstring	Real Array
Pascal	Procedure	Procedure

continued

Table 17–1. Programming Language Parameter Types (cont.)

Language	Parameter Type	General Type
Pascal	Real	Real
Pascal	Real48 (n)	EBCDIC Array
Pascal	Record	Real Array
Pascal	Schema	Refer to "Passing Parameters to Pascal Schemata" later in this section.
Pascal	Sfixed (n < 12)	Integer
Pascal	Sfixed (n > 11)	Double
Pascal	Short Set (1-48 Elements In Set)	Real
Pascal	S_digits (n)	Hex Array
Pascal	S_display (n)	EBCDIC Array
Pascal	U_display (n)	EBCDIC Array
Pascal	Vlstring	Real Array
Pascal	Word48 (n)	EBCDIC Array
Pascal	Word96 (n)	EBCDIC Array
Pascal	Z_display (n)	EBCDIC Array
PL/I	Boolean (48-bit Op)	Boolean
PL/I	Character Array (8-bit)	EBCDIC Array
PL/I	Dimensions/Lower Bounds	(Unique)
PL/I	Dimensions/No Lower Bounds	(Unique)
PL/I	Double Array	Double Array
PL/I	Double (96-bit Op)	Double
PL/I	File	File
PL/I	Integer (48-bit Op)	Integer
PL/I	Pointer	Pointer
PL/I	Real (48-bit Op)	Real
PL/I	Single Array Boolean	Boolean Array
PL/I	Single Array Integer	Integer Array

continued

Table 17–1. Programming Language Parameter Types (cont.)

Language	Parameter Type	General Type
PL/I	Single Array Real	Real Array
WFL	BOOLEAN	Boolean
WFL	INTEGER	Integer
WFL	REAL	Real
WFL	STRING	Real Array

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
ASCII Array			
	ALGOL	ASCII ARRAY	
	ALGOL	ASCII VALUE ARRAY	
Boolean			
	ALGOL	BOOLEAN	Matching Actuals:
	Pascal	Boolean	Boolean Procedure (with no parameters)
	Pascal	Boolean Subrange	Integer
	PL/I	48-bit Op Boolean	Real
	WFL	BOOLEAN	
			Matching Formals:
			Integer
			Real
Boolean Array			
	ALGOL	BOOLEAN ARRAY	
	ALGOL	BOOLEAN VALUE ARRAY	
	Pascal	Array of Boolean	
	PL/I	Single Array Boolean	
Boolean Procedure			
	ALGOL	BOOLEAN PROCEDURE	
	ALGOL	BOOLEAN VALUE ARRAY	
	Pascal	Array of Boolean	
	Pascal	Function: Boolean	
	PL/I	Single Array Boolean	
Complex Array			
	ALGOL	COMPLEX ARRAY	
	ALGOL	COMPLEX VALUE ARRAY	
Direct File			
	ALGOL	DIRECT FILE	

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
Double			
	ALGOL	DOUBLE	
	COBOL(68)	77 COMP-5 Elementary Item	
	COBOL74	77 DOUBLE Elementary Item	
	COBOL85	77 BINARY PIC 9(11-23) Elementary Item	
	COBOL85	77 DOUBLE Elementary Item	
	Pascal	Fixed (n > 11)	
	Pascal	Sfixed (n > 11)	
	PL/I	96-bit Op Double	
Double Array			
	ALGOL	DOUBLE ARRAY	
	ALGOL	DOUBLE VALUE ARRAY	
	COBOL85	01 DOUBLE Group Item	
	Pascal	Array of Fixed (n > 11)	
	Pascal	Array of Sfixed (n > 11)	
	Pascal	Packed Array of Fixed (n > 11)	
	Pascal	Packed Array of Sfixed (n > 11)	
	PL/I	Double Array	
Double Procedure			
	ALGOL	DOUBLE PROCEDURE	
	Pascal	Function: Fixed (n>11)	
	Pascal	Function: Sfixed (n>11)	

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
EBCDIC Array			
	ALGOL	EBCDIC ARRAY	Matching Actuals:
	ALGOL	EBCDIC VALUE ARRAY	Integer Array
	COBOL(68)	01 ASCII Group Item	
	COBOL(68)	01 DISPLAY Group Item	Matching Formals:
	COBOL74	01 DISPLAY Group Item	Integer Array
	COBOL74	01 KANJI Group Item	
	COBOL85	01 DISPLAY Group Item	COBOL(68 & 74) Matches:
	Pascal	Binary (n)	Hex Array
	Pascal	Bits (n)	Integer Array
	Pascal	Display s (n)	Real Array
	Pascal	Display z (n)	
	Pascal	Explicit Record (var)	
	Pascal	Integer48 (n)	
	Pascal	Integer96 (n)	
	Pascal	Packed Array of Enumeration (17-256 Elements in Enumeration)	
	Pascal	Packed Array of Subrange (17-256 Elements in Subrange)	
	Pascal	Packed Array of Char	
	Pascal	Real48 (n)	
	Pascal	S display (n)	
	Pascal	U display (n)	
	Pascal	Word96 (n)	
	Pascal	Word48 (n)	
	Pascal	Z display (n)	
	PL/I	8-bit Character Array	

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
Event			
	ALGOL	EVENT	
	COBOL(68)	77 EVENT Elementary Item	
	COBOL(68)	77 LOCK Elementary Item	
	COBOL74	77 EVENT Elementary Item	
	COBOL74	77 LOCK Elementary Item	
Event Array			
	ALGOL	EVENT ARRAY	
	COBOL(68)	01 EVENT Group Item	
	COBOL(68)	01 LOCK Group Item	
	COBOL74	01 EVENT Group Item	
	COBOL74	01 LOCK Group Item	
File			
	ALGOL	FILE	
	COBOL(68)	File	
	COBOL74	File	
	COBOL85	77 File	
	PL/I	File	
Hex Array			
	ALGOL	HEX ARRAY	Matching Actuals:
	ALGOL	HEX VALUE ARRAY	Integer Array
	COBOL(68)	01 COMP-2 Group Item	
	COBOL(68)	01 INDEX Group Item	Matching Formals:
	COBOL74	01 COMP Group Item	Integer Array
	COBOL74	01 INDEX Group Item	
	COBOL85	01 COMP Group Item	COBOL(68 & 74) Matches:
	Pascal	Boolean1	EBCDIC Array
	Pascal	Boolean4	Integer Array
	Pascal	Digits s (n)	Real Array

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
Hex Array (cont.)	Pascal	Digits (n)	
	Pascal	Hex (n)	
	Pascal	Packed Array of Enumeration (0-16 Elements)	
	Pascal	Packed Array of Subrange (0-16 Elements)	
	Pascal	Packed Array of Boolean	
	Pascal	S digits (n)	
Integer			
	ALGOL	INTEGER	Matching Actuals:
	COBOL(68)	77 COMP-1 Elementary Item	Boolean
	COBOL74	77 BINARY Elementary Item	Integer Procedure (with no parameters)
	COBOL85	77 BINARY PIC 9(1-11) Elementary Item	Real
	Pascal	Char	Real Procedure (with no parameters)
	Pascal	Char Subrange	
	Pascal	Enumeration	Matching Formals:
	Pascal	Enumeration Subrange	Boolean
	Pascal	Fixed (n < 12)	Real
	Pascal	Integer	
	Pascal	Integer Subrange	
	Pascal	Sfixed (n < 12)	
	PL/I	48-bit Op Integer	
	WFL	INTEGER	

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
Integer Array			
	ALGOL	INTEGER ARRAY	Matching Actuals:
	ALGOL	INTEGER VALUE ARRAY	EBCDIC Array
	COBOL85	01 BINARY Group Item	Hex Array
	Pascal	Array of Char	Real Array
	Pascal	Array of Char Subrange	
	Pascal	Array of Enumeration	Matching Formals:
	Pascal	Array of Enumeration Subrange	EBCDIC Array
	Pascal	Array of Fixed (n < 12)	Hex Array
	Pascal	Array of Integer	Real Array
	Pascal	Array of Integer Subrange	
	Pascal	Array of Sfixed (n < 12)	
	Pascal	Packed Array of Enumeration (> 256 Elements)	
	Pascal	Packed Array of Fixed (n < 12)	
	Pascal	Packed Array of Integer	
	Pascal	Packed Array of Subrange (> 256 Elements)	
	Pascal	Packed Array of Sfixed (n < 12)	
	PL/I	Single Array Integer	
Integer Direct Array			
	ALGOL	INTEGER DIRECT ARRAY	Matching Actuals:
			Real Direct Array
			Matching Formals:
			Real Direct Array

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
Integer Procedure			
	ALGOL	INTEGER PROCEDURE	Matching Actuals:
	Pascal	Function: Char	Real Procedure
	Pascal	Function: Char Subrange	
	Pascal	Function: Enumeration	Matching Formals:
	Pascal	Function: Enumeration Subrange	Integer
	Pascal	Function: Fixed (n < 12)	Real
	Pascal	Function: Integer	Real Procedure
	Pascal	Function: Integer Subrange	
	Pascal	Function: Sfixed (n < 12)	
Pointer			
	ALGOL	POINTER	
	PL/I	Pointer	
Procedure			
	ALGOL	PROCEDURE (SUBROUTINE)	
	Pascal	Procedure	
Real			
	ALGOL	REAL	Matching Actuals:
	COBOL(68)	77 COMP-4 Elementary Item	Boolean
	COBOL74	77 REAL Elementary Item	Integer
	COBOL85	77 REAL Elementary Item	Integer Procedure
	Pascal	Real	Real Procedure (with no parameters)
	Pascal	Short Set (1-48 Elements in Set)	
	PL/I	48-bit Op Real	Matching Formals:
	WFL	REAL	Boolean
			Integer

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
Real Array			
	ALGOL	REAL ARRAY	Matching Actuals:
	ALGOL	REAL VALUE ARRAY	Integer Array
	C	Int argc, Char *argv []	
	COBOL(68)	01 COMP Group Item	Matching Formals:
	COBOL74	01 BINARY Group Item	Integer Array
	COBOL85	01 REAL Group Item	
	Pascal	Array of Explicit Data Type	COBOL(68 & 74) Matches:
	Pascal	Array of Packed Array	EBCDIC Array
	Pascal	Array of Real	Hex Array
	Pascal	Array of Record	
	Pascal	Array of Set	
	Pascal	Array of Vlstring	
	Pascal	Explicit Record (call-by-value)	
	Pascal	Long Set (> 48 Elements in Set)	
	Pascal	Packed Array of Real	
	Pascal	Packed Array of Record	
	Pascal	Packed Array of Set	
	Pascal	Packed Array of Vlstring	
	Pascal	Record	
	Pascal	Vlstring	
	PL/I	Single Array Real	
	WFL	STRING	

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
Real Direct Array			
	ALGOL	REAL DIRECT ARRAY	Matching Actuals: Integer Direct Array Matching Formals: Integer Direct Array
Real Procedure			
	ALGOL	REAL PROCEDURE	Matching Actuals:
	Pascal	Function: Real	Integer Procedure Matching Formals: Integer Integer Procedure Real
Task			
	ALGOL	TASK	
	COBOL(68)	01 CONTROL-POINT Elementary Item	
	COBOL(68)	77 CONTROL-POINT Elementary Item	
	COBOL74	01 CONTROL-POINT Elementary Item	
	COBOL74	77 CONTROL-POINT Elementary Item	
Task Array			
	ALGOL	TASK ARRAY	
	COBOL(68)	01 CONTROL-POINT Group Item	
	COBOL74	01 CONTROL-POINT Group Item	

continued

Table 17–2. Matching Parameter Types (cont.)

General Type	Language	Parameter Type	Special Matches
Transaction Record			
	ALGOL	TRANSACTION RECORD	
	ALGOL	TRANSACTION RECORD ARRAY	
	COBOL(68)	Transaction Record	
	COBOL74	Transaction Record	

Resolving Passing Mode Conflicts

Some programming languages, such as WFL and ALGOL, allow the initiating program to specify the passing mode for a tasking parameter. In addition, programming languages typically allow the receiving program to specify a passing mode for the formal parameter. Thus, it is possible for the calling program and the receiving program to request different passing modes for the same parameter.

The system is very forgiving of these types of mismatches and generally allows any combination of actual and formal passing modes without issuing an error. However, when the calling program and the receiving program request different passing modes, the system uses the passing mode requested by the calling program. For example, if a call-by-value actual parameter is passed to a call-by-reference formal parameter, the system passes the parameter by value.

Note that the system is less forgiving of passing mode mismatches for parameters passed to library procedures. For a discussion of the allowable passing mode combination for library procedures, refer to Section 18, "Using Libraries."

Be very careful when writing a program that is intended to be initiated, and passed a parameter, by calling programs written by other people. The calling program might use a different passing mode for the parameter than you expected. For example, you might design the receiving program to receive a parameter by value, and make assignments to the parameter. However, if the calling program actually passes an expression by name, then the receiving program terminates with an error when it attempts to assign a value to the formal parameter. This is true because the calling program implicitly passed a thunk, and it is not possible to store values into thunks. You can avoid these types of problems by not making assignments to the formal parameter.

There is one type of passing mode problem that can make it impossible even for the receiving program to read the value of the formal parameter. If the calling program specifies a constant or an expression as a call-by-name actual parameter, then the compiler creates a thunk. If the receiving program specifies the formal parameter as call-by-reference, then the formal parameter cannot receive a thunk. The calling

program can initiate the receiving program successfully. However, when the receiving program attempts to interrogate or modify the value of the formal parameter, the system issues an "INVALID OPERATOR" error and discontinues the receiving program.

Note that this error does not occur if the call-by-name actual parameter is a variable, rather than a constant or an expression. If a variable is used, then the compiler does not create a thunk. The receiving program can use the formal parameter without any problems.

Examples

Suppose that the following COBOL74 program is the receiving program. Note that the formal parameter specification indicates the parameter REAL-PARAM is to be received by reference:

```
100 IDENTIFICATION DIVISION.
110 ENVIRONMENT DIVISION.
120 DATA DIVISION.
130 WORKING-STORAGE SECTION.
140  77 REAL-PARAM BINARY PIC 9(11) RECEIVED BY REFERENCE.
150 PROCEDURE DIVISION USING REAL-PARAM.
160 START-HERE SECTION.
170 P1.
180     MOVE 15 TO REAL-PARAM.
190 STOP RUN.
```

The following ALGOL program invokes the preceding program and passes the real variable ACTUALREAL as the actual parameter. Note that the statement at line 150 in the following example specifies that the parameter is to be passed by value. This statement overrides the RECEIVED BY REFERENCE clause and causes the parameter to be passed by value. When the receiving program assigns a value of 15 to the formal parameter, the value of the actual parameter is not affected. Thus, the statement at line 210 displays a value of 5; but if the statement at line 150 were deleted, the statement at line 210 would display a value of 15.

```
100 BEGIN
110 FILE TERM (KIND=REMOTE);
120 TASK T;
130 REAL ACTUALREAL;
140 PROCEDURE COBOLTASK (RVAL);
150     VALUE RVAL;
160     REAL RVAL;
170     EXTERNAL;
180 ACTUALREAL := 5;
190 REPLACE T.NAME BY "OBJECT/COBOL/TASK.";
200 CALL COBOLTASK (ACTUALREAL) [T];
210 WRITE (TERM,*//,ACTUALREAL);
220 END.
```

Now suppose that the COBOL74 program was invoked by a WFL job instead. The following WFL job invokes the COBOL74 program and passes a real parameter by value (the default passing mode in WFL):

```
100 ?BEGIN JOB WFL/TEST;
110    CLASS = 2;
120    JOBSUMMARY = SUPPRESSED;
130    ELAPSEDLIMIT = 120;
140 REAL R := 5;
150 RUN OBJECT/COBOL/TASK (R);
160 DISPLAY STRING(R,*);
170 ?END JOB
```

The statement at line 160 displays a value of 5. However, if you change the RUN statement to read *RUN OBJECT/COBOL/TASK (R REFERENCE);* then the parameter is passed by reference and the statement at line 160 displays a value of 15.

Passing Arrays

When an array is passed as a parameter, the actual and formal arrays must be of compatible data types (such as integer, real, and so on). The actual and formal arrays must also be compatible structurally. That is, the number of dimensions and the lower bounds for each dimension must be compatible.

The following subsections discuss these types of compatibility issues for arrays that are passed in process initiation statements. Note that this discussion centers on the compatibility issues the system enforces at run time. If the parameter is passed between procedures in a single program, the compiler can enforce additional restrictions at compile time. For information about any such compile-time restrictions, refer to the appropriate programming language manuals.

Matching Dimensions and Elements

When the calling program passes arrays, the actual array and the formal array must have the same number of dimensions.

However, it is not necessary for the actual array and the formal array to have the same number of elements in each dimension. Some languages allow formal array parameters that do not specify the number of elements in each dimension. For example, ALGOL does not allow upper bounds to be specified for the dimensions in a formal array parameter specification; and Pascal allows formal array parameters, called *schemata,* that are incompletely specified. (Schemata are discussed under "Passing Parameters to Pascal Schemata" later in this section.) In these cases, the system assigns the formal parameter the same number of elements as the actual parameter at run time.

Even if the formal parameter specifies the number of elements in each dimension of an array, the actual parameter can have a different number of elements. The system does not issue an error or warning in these cases. If the actual parameter passes more elements than the formal parameter can receive, the system ignores the extra elements.

Matching Unbounded Arrays

Some languages, such as ALGOL, allow formal array parameters that do not specify the lower bounds for array dimensions. Such array parameters are referred to in this guide as *unbounded array parameters*. Array parameters that explicitly specify the lower bounds are referred to as *simple array parameters*.

Be aware that parameter mismatch errors can result from passing an actual array with an unspecified lower bound to a formal array with a specified lower bound, or vice versa. For example, WFL STRING parameters are passed as unbounded real arrays. If a WFL program passes a string parameter to an ALGOL program, the ALGOL program must declare the formal parameter as unbounded; otherwise, a PARAMETER MISMATCH error occurs at run time.

The following is an example of an ALGOL program that is passed a string parameter from a WFL job.

```
100 PROCEDURE OUTER(ARR);
110    REAL ARRAY   ARR[*];
120 BEGIN
130 FILE TERM(KIND=REMOTE);
140 INTEGER ARR_SIZE;
150 POINTER P;
160 P := ARR;
170 ARR_SIZE := SIZE(ARR) * 6;
180 WRITE(TERM,*//,P FOR ARR_SIZE);
190 END.
```

In the preceding program, the SIZE function at line 170 returns the size of the array parameter in words. This value is multiplied by 6 to give the length of the array parameter in characters.

COBOL74 is somewhat more forgiving than ALGOL in that formal array parameters in COBOL74 programs can receive either simple or unbounded actual parameters. Consider the following COBOL74 program:

```
100 IDENTIFICATION DIVISION.
110 ENVIRONMENT DIVISION.
120 DATA DIVISION.
130 WORKING-STORAGE SECTION.
140     01 PARAM PIC X(12) DISPLAY.
150 PROCEDURE DIVISION USING PARAM.
160 START-HERE SECTION.
170 P1.
180     DISPLAY PARAM.
190
200     STOP RUN.
```

The preceding COBOL74 program is initiated twice by the following ALGOL program. The first time, the ALGOL program passes an unbounded array parameter. The second time, the ALGOL program passes a simple array parameter. In each case, the actual parameter is received by the formal parameter PARAM in the COBOL74 program. The COBOL74 program runs normally and displays the same output in each case.

```
100 BEGIN
110 REAL ARRAY ARRIN[0:12];
120 TASK T;
130 PROCEDURE EX1(ARRACT);
140   REAL ARRAY ARRACT[*];
150   EXTERNAL;
160 PROCEDURE EX2(ARRACT);
170   REAL ARRAY ARRACT[0];
180   EXTERNAL;
190 REPLACE ARRIN BY "HI THERE";
200 REPLACE T.NAME BY "(JASMITH)OBJECT/TEST/COBOL/TASK.";
210   CALL EX1 (ARRIN) [T];
220   CALL EX2 (ARRIN) [T];
230 END.
```

Note that the preceding comments about COBOL74 hold true only for tasking parameters. COBOL74 programs display less flexible behavior when they are invoked as libraries. In this case, the programmer must know in advance whether the actual array parameter is simple or unbounded. If the actual parameter is unbounded, the programmer must use a LOWER-BOUNDS clause in the formal array declaration, or else declare an extra BINARY parameter to receive the lower bound. Of these two techniques, the LOWER-BOUNDS clause is equally compatible with tasking or library calls, whereas the extra BINARY parameter works only for library calls.

For further information about unbounded array parameters to library procedures, refer to Section 18, "Using Libraries."

Matching Pascal Arrays

Some special rules apply for passing parameters to a Pascal formal parameter that is either a multidimensional array or an incompletely defined array.

Passing Multidimensional Arrays

Pascal arrays are all stored internally as one-dimensional arrays. Declaring a Pascal array with multiple dimensions creates an indexing compiler scheme, which makes it appear that the array has multiple dimensions. Within the Pascal program, the fact that the array is really one dimensional is never visible. However, this fact is visible when parameters are passed to a Pascal program from a program written in another language.

Because Pascal formal array parameters are implicitly one dimensional, actual array parameters passed to Pascal programs must always be one dimensional. The elements of the actual array are mapped into the formal array according to an algorithm that

increments the indexes for the highest dimension, then the next highest dimension, and so on.

For example, suppose the actual parameter is an ALGOL EBCDIC array of one dimension, [1:27]. The initiating process could pass this parameter to a Pascal formal parameter that is a three-dimensional packed array of char. Suppose each dimension is declared with indexes [1..3]. The following table illustrates the mapping of elements from the ALGOL actual array into the Pascal formal array:

ALGOL Index	Pascal Index
1	1,1,1
2	1,1,2
3	1,1,3
4	1,2,1
5	1,2,2
6	1,2,3
7	2,1,1
8	2,1,2
9	2,1,3

The initiating process maps the remaining elements in a similar way.

Passing Parameters to Pascal Schemata

Before reading the rules for passing parameters to Pascal schemata, you should understand the following Pascal terms:

- Index

  An index specifies a location in a particular array dimension. If a dimension has indexes running from 1 to 5, then there are five indexes in that dimension.

- Discriminant

  A discriminant appears in an array declaration and specifies the highest-numbered or lowest-numbered index for a particular dimension. If the discriminant is an integer, it is called a constant discriminant. If the discriminant is a variable, it is called a dynamic discriminant.

- Element

  An element is a single location in an array. An element is identified by an index for each dimension stating the element's location in that dimension.

- Schema

  A schema is an array declaration that includes one or more dynamic discriminants. In other words, a schema is a type of incomplete array declaration. Using a schema as a formal parameter makes it possible to pass arrays with different bounds and different numbers of elements to the same formal parameter. The plural of *schema* is *schemata.*

When passing an array to a formal parameter that is a Pascal schema, the initiating process must pass one or more additional parameters. This is the only situation in which the system requires that the number of actual parameters be different from the number of formal parameters. The additional actual parameters provide information about the size of the actual array. Each of these additional parameters is a call-by-value integer.

The following are Pascal schemata types and the rules for passing parameters to each of these schemata types:

- A vlstring (variable-length string). This formal parameter receives two actual parameters: a parameter that contains the string value, followed by a call-by-value integer parameter that records the length of the string.
- A one-dimensional packed array of char whose upper discriminant is dynamic. This formal parameter receives the following two actual parameters: a one-dimensional array, followed by one call-by-value integer parameter that gives the value of the dynamic discriminant.
- Any other type of array or packed array whose declaration includes at least one dynamic discriminant. This type of formal parameter receives the following actual parameters, in the order listed:
  - A one-dimensional array of a compatible type.
  - For each dimension, a call-by-value integer parameter specifying the total number of elements in that dimension and all higher dimensions. For example, imagine an array with five indexes in the first dimension, three in the second dimension, and two in the third dimension. The first integer parameter is 30, which is the result of multiplying 5, 3, and 2 together. The second integer parameter is 6, which is the result of multiplying 3 and 2 together. The third integer parameter is 2.
  - For each dynamic discriminant, a call-by-value integer parameter giving the value of the discriminant. The order of the integer parameters is as follows: first-dimension lower discriminant, first-dimension upper discriminant, second-dimension lower discriminant, second-dimension upper discriminant, and so on. Any constant discriminants are omitted.

Examples

The following programs illustrate how an ALGOL program can pass an array to a Pascal two-dimensional packed array of char. The ALGOL program passes a one-dimensional EBCDIC array.

```
% ALGOL PROGRAM
BEGIN
   EBCDIC  ARRAY
      ALGOLARRAY[Ø:24];
   TASK T;

   PROCEDURE OUTSIDE(ACTUALARRAY);
      EBCDIC  ARRAY ACTUALARRAY[*];
   EXTERNAL;

REPLACE T.NAME BY "OBJECT/PASCAL/TWODIM/CHAR.";
REPLACE ALGOLARRAY[Ø] BY "ONETWOONETWOONETWOONETWO";
CALL OUTSIDE(ALGOLARRAY) [T];
END.


{ PASCAL PROGRAM }
program pascalarray( ( formalarray : formalarraytype ) );
   TYPE
      indexrange = 1..1Ø;
      formalarraytype = packed array [2..5, 2..7] of char;
   VAR
      arrayindex, arrayindex2 : indexrange;
BEGIN
   for arrayindex := 2 to 5 do
      for arrayindex2 := 2 to 7 do
         formalarray[ arrayindex, arrayindex2 ] := 'a';
END.
```

The following example shows what would happen if the formal parameter *formalarray* in the preceding example were changed from a fully-specified array to a schema. Because of this change, the ALGOL program must pass additional call-by-value integer parameters.

```
% ALGOL PROGRAM
BEGIN
EBCDIC ARRAY
   ALGOLARRAY [Ø:24];
INTEGER
   ONEDIM, TWODIM, DISC1, DISC2;
TASK T;
PROCEDURE OUTSIDE (ACTUALARRAY, ONEDIM, TWODIM, DISC1, DISC2);
   VALUE ONEDIM, TWODIM, DISC1, DISC2;
   EBCDIC ARRAY ACTUALARRAY [*];
   INTEGER ONEDIM, TWODIM, DISC1, DISC2;
EXTERNAL;

REPLACE T.NAME BY "OBJECT/TASK/SCHEMA/PASCAL/TWODIM/CHAR.";
ONEDIM := 24;
TWODIM :=  6;
DISC1 := 2;
DISC2 := 7;
REPLACE ALGOLARRAY [Ø] BY "ONETWOONETWOONETWOONETWO";
CALL OUTSIDE (ALGOLARRAY, ONEDIM, TWODIM, DISC1, DISC2) [T];
END.


{ PASCAL PROGRAM }
program pascal_twodim_schema( ( formalschema : formalschematype ) );
   TYPE
      indexrange = 1..1Ø;
      formalschematype( disc1, disc2 :indexrange ) =
                        packed array [disc1..5, 2..disc2] of char;
   VAR
      indexschema, indexschema2 : indexrange;
BEGIN
   for indexschema := formalschema.disc1 to 5 do
      for indexschema2 := 2 to formalschema.disc2 do
         formalschema[ indexschema, indexschema2 ] := 'a';
END.
```

Passing COBOL74 Arrays to Bound Procedures

A COBOL74 host program can initiate a bound subprogram as a task with a PROCESS statement or with the *CALL <task identifier> WITH <section name>* form of the CALL statement. If the host program passes an array parameter to the task, the subprogram can receive various run-time errors (such as INVALID OPERATOR or SEG

ARRAY ERROR) when it attempts to use the array. These errors can occur even if the arrays in the host and subprogram are the same type and length.

Specifically, arrays of usage BINARY, COMPUTATION, REAL, or DOUBLE always receive run-time errors when passed as parameters to a bound subprogram called as a task. EBCDIC arrays (01-level with usage DISPLAY) are the only type of array that can be passed successfully to such a subprogram. Nonarray items (77-level) can be passed without a problem.

If it is necessary for the bound subprogram to share a non-EBCDIC array with the host program, you can declare the array in the subprogram as a global array rather than a parameter. This method allows the same data to be shared between the subprogram and host, and does not cause run-time errors.

Section 18
Using Libraries

A library is a type of process that provides a set of objects that can be used by other processes, which are known as *user processes*. The objects provided by a library are called *library objects*. These objects are said to be *exported* by the library, and *imported* by the user process. Multiple user processes can import objects from the same library process.

You can write library programs in ALGOL, C, COBOL(68), COBOL74, COBOL85, FORTRAN, FORTRAN77, NEWP, Pascal, and PL/I. You can write user programs in all of these languages, as well as in RPG. A library written in one language can be used by programs written in other languages.

A procedure is the type of object most commonly exported by libraries. By consolidating related procedures into a library, you can avoid duplicating the procedures in all the programs that need to use them. Further, you can maintain and enhance the shared procedures more easily when they reside in a library, because you don't have to repeat your work in every program that uses the procedures.

A Series systems provide several other methods by which programs can make use of a shared procedure, including binding, installation intrinsics, and separate programs. Compared to binding, libraries offer the following advantages:

- Libraries export objects at run time, whereas the Binder adds procedures from one object code file to another for permanent storage. You have to run the Binder separately for each object code file to which a procedure is to be added. You have to run the Binder again for each of these object code files whenever you make changes to the shared procedure.
- Libraries allow procedures to be shared between programs in a wider variety of languages than the Binder permits.

Compared to installation intrinsics, libraries offer the following advantages:

- Libraries can include objects that are declared globally to the exported procedures. These could include files, databases, and so on.
- Libraries can contain initialization and termination code.
- Libraries can themselves call other libraries.
- Individual users can create their own libraries without possessing special privileges.
- Libraries can be written in more languages than can installation intrinsics.
- More than one version of a library can be in use at a time.

Another method for sharing procedures is to write each procedure as a separate program. Any other program that needs to make use of one of these shared procedures can initiate the appropriate program as a task. Compared to this method of sharing procedures, libraries offer the following advantages:

- The shared procedures can either be entered or initiated by the user program, whereas a separate program can only be initiated. Procedure entrance takes less time and system resource than process initiation.
- There are more programming languages that provide the ability to use libraries than there are programming languages that provide the ability to initiate programs.

In addition to their role in providing shared procedures, libraries can also provide data structures to user processes. FORTRAN and FORTRAN77 libraries can export files and arrays in much the same way as exported procedures. Additionally, libraries can provide user processes with indirect access to data objects that are declared in a library but not actually exported. The use of libraries to allow user processes to share data objects is discussed in "Providing Global Objects" later in this section.

This section gives an overview of library concepts that are common to most A Series programming languages. This section also gives examples of library programs and user programs written in most of the A Series programming languages. However, for a complete description of the syntax related to library implementation and usage in each language, refer to the manual for the specific language.

This section notes various restrictions on COBOL(68) and COBOL74 libraries that arise from the fact that these languages do not permit nested blocks. Note that COBOL85 does permit nested blocks, and consequently provides more complete access to A Series library features than do COBOL(68) or COBOL74.

Creating Library Programs

In most programming languages, library programs can contain all the features of any ordinary program. What distinguishes a library program is the inclusion of an export list and a FREEZE statement. A library program generally also includes features that specify the sharing and duration properties of the library.

The following subsections outline the features required of library programs in most languages, while also noting certain exceptions that apply to COBOL(68) and COBOL74 libraries. The features of COBOL(68) and COBOL74 libraries are most easily understood as a subset of the general library features supported by the A Series operating system.

Exporting Objects

A library can contain many declarations of objects, some of which are exported and some of which are not exported. There is nothing in the declaration of an exported object that distinguishes it from a nonexported object. Instead, a separate construct called an *export list* specifies all the objects in a given block that are to be exported. The export list is used in addition to, rather than instead of, the declarations of the exported objects.

Export lists are not used in COBOL(68) and COBOL74. Libraries in these languages always export exactly one object, which corresponds to the PROCEDURE DIVISION of the program.

Freezing the Library

When a library program is first initiated, it does not immediately become a library process. In most languages, a library program is executed as an ordinary process until a FREEZE statement in the program is encountered. The FREEZE statement changes an ordinary process into a library process. While the process is frozen, it typically does little or no work on its own; it simply remains present in memory so that user processes can link to it and use the exported procedures.

Eventually, a library process ceases to be a library and resumes execution as an ordinary process. The duration of the library state is specified by one of three FREEZE options. In most languages, you specify the FREEZE option in the FREEZE statement. The following are all the options supported by A Series systems. Not all options are available in all languages.

- TEMPORARY

  The library program ceases execution and remains available as long as users of the library remain. A temporary library that is no longer in use unfreezes and resumes execution. The export objects declared in the library process do not become available to user programs again. Using the TEMPORARY option prevents memory space from being occupied by a library that is not in use.

- PERMANENT

  The library program ceases execution and remains available unless interrupted by an operator command or action by another program. (Means of resuming a permanent library are discussed under "Thawing and Resuming Libraries" later in this section.) It can be desirable to make a library permanent if it is frequently used; the PERMANENT option prevents the system from having to re-create the library many times during the day. It can also be desirable to make a library permanent if it accesses a database or other files that need to be kept open.

- CONTROL

  The program is made available as a library, and control passes to a local procedure in the library called the control procedure, where execution continues. The control library changes into a temporary library when the control procedure is exited.

  The programmer typically includes statements in the control procedure to prevent it from being exited until certain conditions are met. Thus, the CONTROL option makes it possible for a library to decide when to resume itself. The CONTROL option is available in ALGOL and, implicitly, in FORTRAN77.

After a library unfreezes, it cannot execute another FREEZE statement in order to become a library again.

FREEZE statements are not used in COBOL(68) and COBOL74 libraries. Programs written in these languages freeze automatically if they are initiated through the library linkage mechanism. (This method of initiation is discussed under "Initiating Library

Processes" later in this section.) You can use the TEMPORARY compile control option to specify that the library freeze should be temporary. If you do not use the TEMPORARY option, the library sharing option determines the type of freeze. A sharing value of SHAREDBYALL results in a permanent freeze, and sharing values of PRIVATE or SHAREDBYRUNUNIT result in a temporary freeze. For further information about the sharing option, refer to "Controlling Library Sharing" later in this section.

The FREEZE statement is also not used in C libraries. A C library freezes automatically if it is initiated by the library linkage mechanism. The freeze occurs immediately after the function *main* executes. You can use the DURATION compiler option to specify whether it should be a temporary or permanent freeze.

You can use the LIBS (Library Task Entries) system command to list the frozen library processes on the system. The list includes permanent, temporary, and control libraries.

Controlling Library Sharing

Although multiple user programs can use the library at the same time, they are not necessarily using the same instance of the library. You can use the compiler control option SHARING to specify whether multiple user processes access the same instance of the library. The following are the possible values of this option, and their meanings.

- PRIVATE

  The operating system initiates a separate instance of the library program for each user process that links to the library. Values assigned to global objects in the library by a particular user process are visible only to that user process.

- SHAREDBYALL

  All user processes share the same instance of the library. If one user process changes the value of a global object in the library, the next user process that interrogates the global object receives the changed value. The SHAREDBYALL option can be useful if the service provided by the library involves combining information from several users or sharing resources among several users.

- SHAREDBYRUNUNIT

  The same instance of the library is shared by a user process and all frozen libraries that the user process imports objects from, either directly or indirectly. Note that this group of processes, known as a *run unit*, is not the same as a process family. For example, tasks initiated by the user process are not part of the run unit. Any other user processes linked to the library are also considered to be separate run units and receive their own instances of the library. Note that this definition of run unit should not be confused with the ANSI COBOL74 and COBOL85 definitions of run unit.

  Note that a library is its own run unit until it freezes. For example, suppose a user process named UP links to a library named LIB1. Suppose also that, before LIB1 freezes, it links to a library called LIB2. In this case, library LIB2 is in the run unit of library LIB1, but not the run unit of user process UP. If library LIB1 had frozen before linking to LIB2, then both LIB1 and LIB2 would be in the run unit of user process UP.

- DONTCARE

  In most programming languages, this option is a nonpreferred synonym for SHAREDBYALL. However, in C and COBOL85, this option is a nonpreferred synonym for SHAREDBYRUNUNIT.

If a library program does not include the SHARING compiler control option, then the compiler assigns a default SHARING option to the library. The default value of the SHARING option is SHAREDBYALL in ALGOL, COBOL(68), FORTRAN, and FORTRAN77; and SHAREDBYRUNUNIT in C, COBOL74, COBOL85, and Pascal. PL/I libraries are always PRIVATE. The SHAREDBYALL value is not available in C or COBOL85.

Sharing is handled in a special way for COBOL(68) and COBOL74 libraries. If a library written in these languages has a sharing value of SHAREDBYALL or SHAREDBYRUNUNIT, then multiple user processes can link to the same instance of the library. However, the operating system ensures that only one user process can be executing the procedure exported by this library at any given time. If another user process invokes the procedure while it is in use, the operating system causes this user process to wait until the procedure becomes available.

Note that the library sharing option affects only the relationship between library declarations and library instances. The sharing option cannot prevent multiple user processes from accessing the same library instance through a shared library declaration. For example, the outer block of an ALGOL program might include a library declaration. If this ALGOL program initiates two internal tasks, the library declaration is visible to both tasks. The two tasks could use this library declaration to access the same library instance, even if the library sharing option is PRIVATE.

Initiating Internal Library Processes

In ALGOL programs, an internal procedure can be initiated as a task and can later freeze as a library, provided that the procedure includes a FREEZE statement and an export list. NEWP programs marked with UNSAFE(TASKING) status also have this capability. However, to simplify this discussion, this section discusses library processes as if they were always executions of an entire library program.

Reinitialization of Local Variables

In most languages, any local variables declared in an exported procedure are reinitialized each time that procedure is invoked. COBOL85 is the exception to this rule. Variables declared in COBOL85 nested programs retain their values from one invocation to the next, unless the PROGRAM-ID paragraph of the nested program includes an *IS INITIAL* clause.

A library program can use global objects to store information between procedure invocations. Refer to "Providing Global Objects" later in this section.

Restrictions on OWN Objects

Declarations of arrays in an exported library procedure cannot include any OWN clause. The OWN clause, which is available only in ALGOL, causes the value of an object to be saved between invocations of the procedure in which that object is declared. If an exported procedure includes a array declaration with an OWN clause, then when a user process invokes the procedure, the system discontinues the user process with the error "ILLEGAL OWN ARRAY". The library process itself is not affected by this error.

Note that an exported procedure can declare simple variables or pointers with OWN clauses. If the library is a shared library, the OWN clause allows multiple user processes to access the same instance of the same object. For example, if two user processes are concurrently executing the same library procedure, and the library procedure declares an OWN object, then any changes made by one user process to the value of the object are immediately visible to the other user process. To prevent timing ambiguities, you can use techniques such as those discussed under "Providing Global Objects" later in this section.

Restrictions on COBOL(68) and COBOL74 Libraries

COBOL(68) and COBOL74 object code files are structured in a special way that allows them to be executed either as libraries or as ordinary processes. Every program written in these languages is available for use as a library, except for programs that

- Specify in the data division that a parameter is RECEIVED BY CONTENT (that is, received as a call-by-value parameter).
- Specify an 01-level data item with an OCCURS clause. COBOL74 does not permit 01-level data items to be declared with an OCCURS clause. COBOL(68) allows such declarations, but does not permit programs with such declarations to be executed as libraries.
- Specify parameters in the USING clause of the PROCEDURE division that are not allowed for libraries. Each of the data items in the USING phrase must be defined as level 01 or level 77, and must not redefine another data item. Further, the parameters must be of data types that are allowed for library parameters. For a list of the allowed library parameter types for COBOL(68) and COBOL74, refer to Table 18–2, "COBOL(68) Parameters," and Table 18–3, "COBOL74 Parameters."
- Are compiled with a LEVEL compiler control option that specifies a lexical level greater than 2.

A program that does not use any of these restricted features is said to be *library-capable*. A library-capable COBOL(68) or COBOL74 program freezes if it is initiated through the library linkage mechanism. If the program is initiated by a process-initiation statement, then the program runs as an ordinary process and does not freeze. For a discussion of the library linkage mechanism, refer to "Initiating Library Processes" later in this section.

Circular library linkages are not allowed for COBOL(68) or COBOL74 libraries. If COBOL(68) or COBOL74 library invokes itself, the operating system discontinues the library with a run-time error. If a COBOL(68) or COBOL74 library invokes a procedure in another library that in turn invokes the original library, both libraries freeze successfully but the user process hangs indefinitely in the state WAITING ON AN EVENT.

An EXIT PROGRAM statement must be used to exit a COBOL(68) or COBOL74 library and to return to the calling program. By contrast, a STOP RUN statement causes the user process to terminate at the point of the statement that invoked the library procedure.

Because only the PROCEDURE DIVISION of a COBOL(68) or COBOL74 library is exported, most objects declared in the DATA DIVISION or ENVIRONMENT DIVISION are considered to be global to the exported procedure. Within each library instance, these objects retain their values from one procedure call to the next. In a shared library, any changes made to the values of these objects by a user process are visible to all other user processes.

One exception to this behavior occurs for objects that are referred to in the USING clause of the PROCEDURE DIVISION. This clause causes the objects to be treated as parameters to the PROCEDURE DIVISION. Each user process receives a separate instance of these objects, and the values of the objects are not retained from one procedure call to the next.

Another limitation arises from the fact that the entire PROCEDURE DIVISION is exited. There is no place in a COBOL(68) or COBOL74 library to specify initialization or termination code. By contrast, the outer block of an ALGOL library can contain statements that execute before the FREEZE statement or after the library unfreezes.

Creating User Programs

In general, user programs can include all of the features of an ordinary program. Further, a user program can itself be a library that invokes procedures imported from other libraries.

In most programming languages, user programs are distinguished by the inclusion of import declarations and library declarations. These declarations, and their equivalents in COBOL(68) and COBOL74, are described in the following subsections.

Importing Objects

Import declarations include information such as the name of the imported object, the type of object, and the library it is imported from. Declarations of imported procedures must also include parameter specifications for any parameters accepted by the procedure. However, the main body of the procedure (including all local declarations and statements) is omitted.

In languages that include library declarations and import declarations, the user program can use imported objects just as if they were local objects of the user program.

In COBOL(68) and COBOL74, import objects are not explicitly declared. Instead, when invoking an imported procedure, you can use a special form of the CALL statement that specifies both the name of the procedure and that of the library. If the CALL statement does not explicitly specify a procedure name, the procedure name is assumed to be PROCEDUREDIVISION.

In addition to import declarations, the user program can contain declarations of objects that are not imported from libraries.

Specifying Libraries

In most languages, the user program must include explicit declarations of all libraries used by the program. A library declaration specifies the identifier by which the library is known throughout the user program. A library declaration can also include library attribute assignments.

Library attributes should not be confused with task attributes or file attributes. The library attributes provide information that helps the operating system to link the user program to the correct library.

The user process can change the values of library attributes repeatedly, as long as the user process is not currently linked to the library in question. The operating system ignores any changes made to the attributes of a library while the user process is linked to the library.

In COBOL(68) and COBOL74, user programs do not include library declarations. However, these languages allow you to assign library attributes to a library.

The following subsections describe the library attributes. Each description begins by stating whether the attribute can be read or written or both, and gives the attribute type and the default value. Note that attribute types of EBCDIC string are pointer expressions in NEWP, but as true EBCDIC strings in ALGOL.

FUNCTIONNAME

Property	Value
Usage	Read/Write
Type	EBCDIC string
Default	Value of INTNAME library attribute

The FUNCTIONNAME library attribute identifies the function name of the library that is to be linked to. FUNCTIONNAME has meaning only if the LIBACCESS library attribute is set to BYFUNCTION.

The operating system stores the mappings between function names and library object code files and links to the appropriate code file if a function name is used. The function name makes it possible to change to a different library object code file without having to recompile all the user programs that use the library.

Not all libraries have associated function names, although any library can be assigned one or more function names by operator action. A library that has an associated function name is called a *support library*.

You can use the SL (Support Library) system command to display and control the mappings between function names and libraries. You can also submit SL commands programmatically through the DCALGOL *SETSTATUS* function. An SL command can do any of the following:

- Display the current function names and their associated object code files.
- Assign a particular library object code file to a function name, without affecting any programs that are currently using the library. In some cases, the change does not take effect until the current invocation of the library thaws and resumes execution. The associations between function names and library object code files survive a halt/load or a CM (Change MCP) system command.
- Create a new function name and assign the object code file that is initially associated with it.
- Delete an existing function name, without affecting any programs that are currently using the library. In some cases, the deletion is denied if a frozen invocation of the library currently exists.

User programs can access a library directly by object code file title even if a function name has been defined for the library. However, undesirable side effects can result if the first user program to access a library does so by object code file title instead of by function name.

For example, if a library has a SHARING value of SHAREDBYALL and the first user program accesses the library by object code file title, then the library process inherits several task attributes of the user program, including USERCODE, FAMILY, and SOURCESTATION. However, if the same library is first accessed by function name, then these task attributes are not inherited.

Also, system libraries must first be accessed by function name, so they receive their special privileges. For a discussion of this and other security issues related to the SL command, refer to "Security Considerations" later in this section.

INTNAME

Property	Value
Usage	Read/Write
Type	EBCDIC string
Default	Library identifier, except in COBOL(68) and COBOL74; see following discussion

The INTNAME library attribute specifies the internal name for the library.

One use of the internal name is in assignments to the LIBRARY task attribute. You can use the LIBRARY task attribute to alter the behavior of user programs. For example, suppose a user program declares a library with an internal name of LIB1, a LIBACCESS value of BYTITLE, and a TITLE value of OBJECT/LIB1. When you run the user program, you can assign the LIBRARY task attribute a value of LIBRARY LIB1(TITLE = OBJECT/OTHERLIB). This has the effect of causing the user program to link to a different library than it otherwise would.

INTNAME also serves as the default value for the TITLE and FUNCTIONNAME attributes.

Because libraries cannot be explicitly declared in COBOL(68) or COBOL74, the compiler constructs the default INTNAME for a library from the first reference to that library title in the user program. If the title includes multiple nodes separated by slashes (/), the INTNAME is formed from the final node of the title.

The first reference to the library title might be in the CALL statement that invokes the library, or in a CHANGE statement that assigns attributes to the library. If either of the following two statements is the first reference to a library in a COBOL74 user program, the library receives an INTNAME of LIB1:

```
CHANGE ATTRIBUTE FUNCTIONNAME OF "OBJECT/LIB1" TO "TESTSUPPORT.".
CALL "FACT IN OBJECT/LIB1" USING PARAM.
```

To prevent any two libraries in a COBOL(68) or COBOL74 program from receiving the same INTNAME, you should assign each library a TITLE attribute value that has a different value in the final node.

LIBACCESS

Property	Value
Usage	Read/Write
Type	Mnemonic
Default	BYTITLE

The LIBACCESS task attribute specifies the method the operating system should use to identify the appropriate library to link the user program to. LIBACCESS has one of the following mnemonic values: BYFUNCTION, BYINITIATOR, or BYTITLE. The following are the effects of these values:

- BYFUNCTION

  The operating system links the user program to the library with the function name specified by the FUNCTIONNAME library attribute. The value of the TITLE library attribute is ignored.

- BYINITIATOR

  This value has meaning only if the user program was, itself, initiated by a library in one of the following ways:

  - The user program was initiated as a dependent process by a library.
  - The user program is a library, and was initiated by the library linkage mechanism after being invoked by a library.

  In this case, a value of BYINITIATOR causes the user program to link to the initiating library process. When the BYINITIATOR value is used, the values of the FUNCTIONNAME and TITLE library attributes are ignored.

  Note: *If the user program was initiated by a library as a dependent process, and has since become a library itself with a freeze type of CONTROL, then the system does not allow that user program to use the BYINITIATOR value. If such a user program attempts to use BYINITIATOR to link to a library, the system suspends the user program. The operator can cause the user program to resume by specifying a library code file title with the FA (File Attributes) system command. Alternately, the operator can use the DS (Discontinue) command to terminate the user program.*

- BYTITLE

  The operating system links the user program to the library whose object code file title matches the value of the TITLE library attribute. The value of the FUNCTIONNAME library attribute is ignored.

LIBPARAMETER

Property	Value
Usage	Read/Write
Type	EBCDIC string
Default	Null string

The LIBPARAMETER library attribute is used to transmit information from the user program to the selection procedures of libraries that provide dynamic linkage to export objects. For further information, refer to "Dynamic Linkage" later in this section.

TITLE

Property	Value
Usage	Read/Write
Type	EBCDIC string
Default	Value of INTNAME library attribute, except in COBOL(68) or COBOL74; see following discussion

The TITLE library attribute specifies the object code file title of the library. The TITLE attribute has meaning only if the LIBACCESS library attribute is set to BYTITLE.

Because libraries cannot be explicitly declared in COBOL(68) or COBOL74, the compiler constructs the default title for a library from the first reference to that library in the user program. The first reference to the library title might be in the CALL statement that invokes the library, or in a CHANGE statement that assigns attributes to the library. If either of the following two statements is the first reference to a library in a COBOL74 user program, the library receives a title of OBJECT/LIB1:

```
CHANGE ATTRIBUTE FUNCTIONNAME OF "OBJECT/LIB1" TO "TESTSUPPORT.".
CALL "FACT IN OBJECT/LIB1" USING PARAM.
```

To prevent any two libraries in a COBOL(68) or COBOL74 program from receiving the same INTNAME, you should give each library a title that has a different value in the final node of the title. Refer to "INTNAME" earlier in this section.

Controlling Library Linkage

The following subsections explain dynamic aspects of the relationship between a user process and a library process, including the methods of initiating, linking to, and delinking from libraries. These subsections also explain how linkages are established between particular export objects and import objects.

Linking to Libraries

Library linkage is established at run time by the operating system, based on the values of the library attributes specified by the user process.

Library linkage typically occurs when the user process first invokes an object imported from a particular library. For example, suppose a user process has import declarations for three procedures, PROC1, PROC2, and PROC3, which all come from library LIB1. Note that the order in which these imports are declared might not be the same as the order in which they are invoked. Thus, PROC1 might be declared first, but the user process might invoke PROC3 first. In this case, the operating system establishes the linkage to library LIB1 when PROC3 is invoked.

If the requested library program cannot be found, the user process becomes suspended at this point. If the LIBACCESS value is BYTITLE, then a "NO LIBRARY" message is displayed as the RSVP message. If the LIBACCESS value is BYFUNCTION, and the FUNCTIONNAME library attribute requests a function name that does not exist, the RSVP message is "FUNCTION <function name> IS NOT DEFINED, SL, FA, OR DS".

If the requested library program is found, the system links the user program to an existing instance of the library or initiates a new instance as discussed under "Implicitly Initiating a Library" later in this section. The system then checks to see if the requested object is exported by the library, with one of the following results:

- If no object with the requested name is exported by the library, the system discontinues the user process with the message "MISSING OBJECT <object name> IN LIBRARY <library name>".
- If there is an export object with the requested name, the system compares the type and parameters of the import and export object. If the type and parameters match, the user process continues executing normally. If they do not match, the system discontinues the user process with the error message "OBJECT <object name> TYPE OR PARAMETER MISMATCH IN LIBRARY <library name>".

During the linkage process, the system attempts to establish a link between user program import objects and the corresponding library export objects. However, library linkage can proceed successfully even if the system does not find matches for all the import objects. A "MISSING OBJECT <object name> IN LIBRARY <library name>" or "OBJECT <object name> TYPE OR PARAMETER MISMATCH IN LIBRARY <library name>" error occurs only later, when the user process attempts to use the imported object.

Because of the possibilities for fatal errors in linking to libraries or using library objects, you might want to consider using the LINKLIBRARY function when linking to libraries. This function, which is available only in ALGOL, NEWP, and Pascal, makes a conditional attempt to establish linkage with the library. If the attempt fails, the function returns a value indicating the reason for the failure, but no error messages are displayed. If the attempt succeeds, the function returns a value indicating whether all the import objects defined in the user program for that library were really present in the library.

Initiating Library Processes

Library processes can be initiated in either of two ways: implicitly, through the library linkage mechanism, or explicitly, by a process-initiation statement in another process.

Implicitly Initiating a Library

If a user process attempts to link to a library, and no instance of the library is currently frozen, then the user process enters a waiting state. The STATUS task attribute is still ACTIVE, but the stack state in the Y (Status Interrogate) system command is WAITING ON AN EVENT. The library linkage mechanism of the operating system automatically initiates the library program, which executes normally until it freezes and becomes a library process. At this point, the system completes the linkage between the user process and the library process, and the user process resumes execution.

Even if an instance of the library is already frozen, the system might initiate a new instance of the library for the user process to link to. For example, if the sharing option of the library is PRIVATE, then the system initiates a new instance of the library for each user process. If the sharing option is SHAREDBYRUNUNIT, then the system initiates a new instance of the library for each run unit that uses the library. (For details, refer to the discussion of "Controlling Library Sharing" earlier in this section.)

If a PRIVATE or SHAREDBYRUNUNIT library is initiated through the library linkage mechanism, and the library requests a permanent freeze, the system actually freezes the library as a temporary library. The system does this because a PRIVATE or SHAREDBYRUNUNIT library instance can only be linked to by a user process once. Thus, no purpose would be served by allowing the library instance to linger after the original user delinks.

If the user process attempts to link to a program that is not capable of becoming a library, then the library linkage mechanism issues the error "LIBRARY WAS NOT INITIATED: <library name>" and discontinues the user process. This error can happen, for example, if the user process attempts to link to an ALGOL program that does not contain a FREEZE statement.

Because a library program initially runs as a regular program, the flow of execution can be such that the execution of a FREEZE statement is conditional and can occur anywhere in the program. If the library linkage mechanism initiates such a program, and the resulting process terminates without ever having executed a FREEZE statement, the system discontinues the user process with the error "LIBRARY DID NOT FREEZE: <library name>".

Explicitly Initiating a Library

A library process can be explicitly initiated by a process initiation statement in a program. However, the resulting process can freeze only if it is an independent process or an asynchronous dependent process. If the program is initiated as a synchronous dependent process, by a WFL *RUN* statement, for example, then when the process attempts to freeze, it is discontinued with the error "FREEZE FAILED, TASK TYPE NOT PROCESS OR RUN".

When a library process is intended to be explicitly initiated, the library should typically specify a freeze duration of PERMANENT or CONTROL. If the freeze is TEMPORARY, then the process can freeze successfully only if the process is an internal process initiated by a PROCESS statement.

For libraries that are explicitly initiated, some special considerations apply to the sharing option. If the sharing option is PRIVATE or SHAREDBYRUNUNIT, then the library instance is not directly available to user processes. However, such a library instance can serve as a secondary library in a dynamic linkage mechanism. (This type of linkage is discussed under "Dynamic Linkage" later in this section.) If the sharing option is SHAREDBYALL or DONTCARE, then any user process can link to that library instance.

A library program can determine whether it was initiated explicitly or by the library linkage mechanism by interrogating the LIBRARYSTATE task attribute. Bit [0:1] of the LIBRARYSTATE value stores a 1 if the process was initiated by the library linkage mechanism.

Linking Export and Import Objects

Import objects are linked to corresponding export objects in one of three ways: directly, indirectly, and dynamically. The declaration of each export object in the library program specifies which of these linkage methods is used. The method chosen depends on whether the export object originates in the library program, or if the library itself imports the object from another library.

Direct Linkage

Direct linkage occurs when the library program contains the complete declaration of the object that is named in the export list of the library. For example, if a procedure is exported, the library contains all the statements that make up the procedure.

Direct linkage is the only type of linkage that is provided by COBOL(68) and COBOL74 libraries.

Indirect Linkage

Indirect linkage occurs when the library program exports an object that is, in turn, imported from another library. The system then attempts to link the user process to this second library, which can provide the exported object directly, indirectly, or dynamically. A chain of indirect or dynamic linkages must eventually end in a library that provides the object directly.

Dynamic Linkage

Dynamic linkage is similar to indirect linkage in that it allows a library to export an object that was, in turn, imported from another library. However, dynamic linkage allows the primary library to import objects with the same name from multiple secondary libraries. Whenever a user process attempts to import an object with that name, the

primary library can dynamically select the version of the procedure to provide to the user process.

The dynamic linkage feature is available only in ALGOL libraries. Procedures that are exported dynamically include a BY CALLING clause, as in the following example:

```
PROCEDURE READFILE;
    BY CALLING SELECTION;
```

The BY CALLING clause specifies the name of a *selection procedure*, which you must declare elsewhere in the library program. Whenever a user process first links to a library, the system checks to see if any procedures imported by that user process are declared in the library with a BY CALLING clause. If so, the system invokes the selection procedure. The selection procedure must accept the following two parameters from the system:

- A parameter of type EBCDIC string, which the system uses to pass in the value of the LIBPARAMETER library attribute as specified by the user process. This parameter allows the user process to convey information to the library that might help the library to decide which secondary library to import the procedure from.
- A parameter of type procedure, which the system uses to pass in an MCP procedure. This procedure itself has a parameter, which is a task variable. The selection procedure must invoke the MCP procedure before exiting, and must pass to it the task variable of the secondary library that has been selected.

The secondary library that is selected can provide the requested object through direct, indirect, or dynamic linkage. A chain of indirect or dynamic linkages must eventually end in a library that provides the object directly.

The selection procedure is invoked only once, at library linkage. All links to exported procedures in the library are resolved during linkage. To cause the selection of a different secondary library after linkage, the user process must first delink from the dynamic library. The user process can then modify the LIBPARAMETER library attribute to request a different secondary library, and relink to the dynamic library.

For an example of a library that provides dynamic linkage, refer to "ALGOL Library: OBJECT/SAMPLE/DYNAMICLIB" later in this section.

Circular Linkage

It is possible for a chain of library linkages to be circular; thus, a library can reference itself indirectly through a chain of library references. A circular linkage can only be made if all the libraries involved are frozen and at least one of them was already frozen at the time it linked to one of the other libraries in the circle.

The system imposes the following restrictions on circular linkages:

- A given object exported by a library cannot be provided by direct or indirect linkage to the same object in the same library. That is, if library L exports procedure X, the chain of linkages that provide that procedure cannot lead back to procedure X in library L. However, the linkages could lead back to some other procedure, for example Y, in library L.

  For an example of libraries that violate this restriction, refer to "Example 1: Indirect Self Referencing," under "ALGOL Incorrect Circular Libraries" later in this section.

- No more than one of the libraries in a circular linkage can have a freeze type of CONTROL.

If the user process makes a procedure call that results in a circular linkage of two or more libraries that violates one of these restrictions, the system discontinues the user process and displays the error message "CURRENT CIRCULAR LIBRARY REFERENCE STRUCTURE IS NOT ALLOWED: <library name>". The library name identifies the library at the point in the chain where the linkage became circular. If the user process initiated the chain of circular linkages with a LINKLIBRARY function, the linkage fails and the function returns a value of –7.

Additionally, some incorrect types of circular linkage can result in the user process hanging indefinitely. This situation occurs if

- A library provides an object by direct linkage to the same object in the same library. Refer to "Example 2: Direct Self Referencing," under "ALGOL Incorrect Circular Libraries" later in this section.
- Two libraries are waiting on each other to freeze. Refer to "Example 3: Libraries that Wait on Each Other," under "ALGOL Incorrect Circular Libraries" later in this section.

In either of these cases, the Y (Status Interrogate) system command shows the user process to have a STACK STATE of WAITING ON AN EVENT. However, the STATUS task attribute value remains ACTIVE, and the user process does not appear in the W (Waiting Mix Entries) system command display. This situation continues until an operator enters a DS (Discontinue) system command or until the system is halt/loaded.

For a correct example of circular library linkage, refer to "ALGOL Circular User Programs" later in this section.

Matching the Object Name

Under "Linking to Libraries" earlier in this section, it was explained that the system matches import objects to export objects only if they have the same name. In general, the name matching is based on the identifier specified in the import or export declaration. However, there are a couple of exceptions to this rule.

ALGOL user programs can declare import objects under one name, and cause them to match export projects with a different name, by including an ACTUALNAME clause in the import declaration. For example:

```
PROCEDURE READIT;
  LIBRARY LIB1(ACTUALNAME = "READLINE");
```

Because of the ACTUALNAME assignment, the system looks for a matching export object named READLINE instead of READIT. You can also change the actual name of an import object outside the declaration, by using the SETACTUALNAME function. The SETACTUALNAME function makes the requested change, if possible; otherwise, SETACTUALNAME returns a value indicating why the actual name could not be changed. One reason SETACTUALNAME can fail is that the actual name of an import object cannot be changed while the user process is linked to the library the object is imported from.

Similarly, ALGOL library programs can declare export objects under one name, and cause them to match import objects with a different name, by including an AS clause in the export declaration. For example, the following export declaration causes an object named PROC_READ to be exported under the name READLINE:

```
EXPORT PROC_READ AS "READLINE";
```

One of the main uses of the ACTUALNAME clause, SETACTUALNAME function, and AS clause is to facilitate interlanguage communication. For example, identifiers in COBOL74 can include hyphens (-), whereas identifiers in ALGOL cannot. If a COBOL74 library exports an object with a name that includes a hyphen, an ALGOL user program cannot declare an import object with exactly the same name. Instead, the ALGOL user program can declare the import object with an identifier that is legal in ALGOL, and use an ACTUALNAME clause to specify the name used in the COBOL74 library.

As discussed under "Creating Library Programs" earlier in this section, most COBOL(68) and COBOL74 programs can be called as libraries. However, these programs do not include export lists or declarations of export objects. Instead, the PROCEDURE DIVISION of the program is always the single export object. The name of this export object is determined by the following rules:

- In COBOL(68), if the program contains a PROGRAM-ID comment, the first word of the comment is used as the name of the library export object; if no PROGRAM-ID comment appears, the name of the export object is PROCEDUREDIVISION.
- In COBOL74, if the program contains a PROGRAM-ID comment and the CCI option FEDLEVEL is equal to 5, the first word of this comment is used as the name of the library export object; if no PROGRAM-ID comment appears, or if the FEDLEVEL is not equal to 5, the name of the export object is PROCEDUREDIVISION.

In COBOL85 library programs, which export nested programs as library procedures, the export name is specified by the PROGRAM-ID paragraph in the IDENTIFICATION DIVISION of each nested program.

Type Matching

Under "Linking to Libraries" in this section, it was pointed out that the system compares import and export objects with matching names during library linkage. If the objects are procedures, the system has to compare several aspects of the import and export procedures to ensure that they match. The factors considered include the procedure type, the number and type of parameters, and the passing mode used for each parameter.

Matching Procedure Types

Procedures in ALGOL and some other languages can be invoked as functions that return values. Such procedures are referred to as *typed procedures*. For example, an ALGOL procedure can be of any of the following types:

- Untyped
- ASCII STRING
- BOOLEAN
- COMPLEX
- DOUBLE
- EBCDIC STRING
- HEX STRING
- INTEGER
- REAL

A user program written in COBOL(68), COBOL74, or COBOL85 can use the GIVING clause of the CALL statement to receive the value returned by a typed library procedure.

A FORTRAN or FORTRAN77 library procedure can be any of the following:

- SUBROUTINE
- REAL FUNCTION
- INTEGER FUNCTION
- DOUBLE PRECISION FUNCTION
- LOGICAL FUNCTION
- COMPLEX FUNCTION
- CHARACTER FUNCTION (FORTRAN77 only)
- COMMON
- FILE

If an export procedure is typed, the matching import procedure must be of the same type or a compatible type. For information about the compatibility of data types in different languages, refer to "Matching Parameter Types" later in this section.

Matching Parameter Types

For an import procedure to match the corresponding export procedure successfully, both procedures must specify the same number of parameters. The parameters must be specified in the same order in both procedures. Further, each parameter specified by the import procedure must be of a type compatible with the equivalent parameter to the export procedure.

Because the system permits user programs to be written in different languages than the libraries they use, there are times when the actual and formal parameters to a library procedure are specified in different programming languages. Each programming language provides different names for the same or similar types of data. The following are the parameter types that can be specified in an ALGOL library procedure:

- BOOLEAN, BOOLEAN ARRAY, or DIRECT BOOLEAN ARRAY
- DOUBLE, DOUBLE ARRAY, or DIRECT DOUBLE ARRAY
- REAL, REAL ARRAY, or DIRECT REAL ARRAY
- INTEGER, INTEGER ARRAY, or DIRECT INTEGER ARRAY
- COMPLEX or COMPLEX ARRAY
- EBCDIC STRING or EBCDIC STRING ARRAY
- ASCII STRING or ASCII STRING ARRAY
- HEX STRING or HEX ARRAY
- EVENT or EVENT ARRAY
- TASK or TASK ARRAY
- EBCDIC ARRAY or DIRECT EBCDIC ARRAY
- ASCII ARRAY or DIRECT ASCII ARRAY
- HEX ARRAY or DIRECT HEX ARRAY
- FILE or DIRECT FILE
- POINTER
- QUEUE
- TRANSACTION RECORD or TRANSACTION RECORD ARRAY
- PROCEDURE, declared using the FORMAL clause. If the procedure has parameters, they must each be of one of the types listed previously. The procedure itself must be untyped or else of one of the ALGOL procedure types listed under "Matching Procedure Types" in this section.

The following subsections list the parameter types that are available for library procedures in each of the various programming languages. For each parameter type, the

equivalent ALGOL parameter type is listed. You can also use this information to deduce which parameter types in two non-ALGOL languages are equivalent.

For example, you will find that a COMP, level 77 1-11 digits parameter in COBOL(68) is equivalent to an ALGOL integer variable, and that a BINARY, level 77 1-11 digits parameter in COBOL74 is also equivalent to an ALGOL integer variable. It follows that the COBOL(68) parameter type can also match the COBOL74 parameter type.

C Parameter Types

All of the data types supported in C can be passed between a C user program and a C library, with a few exceptions that are documented in the *A Series C Programming Reference Manual*.

The data types that can be passed between C libraries and user programs in other languages, or between C user programs and libraries in other languages, are much more limited. Table 18–1, "C Parameters," lists the C parameter types available for interlanguage library calls, and the ALGOL equivalents of these C parameter types.

Table 18–1. C Parameters

ALGOL Parameter	Corresponding C Parameters
BOOLEAN	int†
INTEGER	char
	int†
	short†
	long†
	pointers (all types)
REAL	float
	double
DOUBLE	long double
INTEGER ARRAY	int []
	short []
	long []
EBCDIC ARRAY	char []
	__heap_t
REAL ARRAY	float []
	double []
	long double []
	struct
	union

Legend
† Both signed and unsigned types

continued

Table 18–1. C Parameters (cont.)

ALGOL Parameter	Corresponding C Parameters
FILE	_file_t
INTEGER PROCEDURE	char () int () short () long ()
REAL PROCEDURE	float () double ()
DOUBLE PROCEDURE	long double ()
PROCEDURE	void ()

Legend
† Both signed and unsigned types

The *__heap_t* parameter passes the value of the *heap*, which is a memory area where a C program stores arrays, structures, addressed objects, and dynamically allocated objects.

Pointers in C are integer types that indicate the location of an item within the heap. The exact meaning of C pointers varies according to the memory model used for the C program; the programmer can request a particular memory model with the MEMORY_MODEL compiler control option. If the heap is implemented with a MEMORY_MODEL value of TINY or SMALL, then a non-C program that calls a C library can use the C pointer as an indicator of the offset of the item within a heap. If the heap is implemented with a different MEMORY_MODEL value, then the non-C program must use the *__heap_to_ptr_t* procedure to convert the C pointer into a conventional pointer.

The *__heap_to_ptr_t* procedure is one of several export procedures that the C compiler automatically creates in each C library. Other procedures that aid in array handling include *__copy_to_ptr_t*, *__copy_from_ptr_t*, *__free_t*, and *__malloc_t*. These procedures can be accessed only by programs written in ALGOL (including any of the extended forms of ALGOL), NEWP, or Pascal. For examples of the use of some of these procedures, refer to "Library Examples" later in this section.

COBOL74 Parameter Types

Table 18–3, "COBOL74 Parameters," lists the allowable parameters to a COBOL74 library and the corresponding ALGOL parameters. For further information about COBOL74 parameters, refer to the *A Series COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation*.

Table 18–3. COBOL74 Parameters

ALGOL Parameter	Corresponding COBOL74 Parameters
DOUBLE	BINARY, 77 12-23 digits DOUBLE, 77
DOUBLE ARRAY [0]	DOUBLE, 01
EBCDIC ARRAY [0]	COMP, 01 group item DISPLAY, 01 INDEX, 01 group item
EBCDIC STRING (non-resizable)	DISPLAY item in STRING clause
EVENT	EVENT, 77
EVENT ARRAY [0]	EVENT, 01
FILE	FILE
HEX ARRAY [0]	COMP, elementary 01 and 77 INDEX, 01 elementary item
INTEGER	BINARY, 77 1-11 digits COMP item in INTEGER clause
INTEGER ARRAY [0]	BINARY, 01 (If $INTEGERBNRY = TRUE, the default)
PROCEDURE	TRANSACTION PROCEDURE
REAL	REAL, 77
REAL ARRAY [0]	BINARY, 01 (If $INTEGERBNRY = FALSE)

continued

Table 18–3. COBOL74 Parameters (cont.)

ALGOL Parameter	Corresponding COBOL74 Parameters
	REAL, 01
TRANSACTION RECORD	TRANSACTION RECORD
TRANSACTION RECORD ARRAY [0]	TRANSACTION RECORD ARRAY

In COBOL74 user programs, the GIVING clause of a CALL statement can specify a variable to receive the value returned by a typed procedure. If the item in the GIVING clause is a level 77 REAL, the procedure must be of type real. If the item in the GIVING clause is a level 77 DOUBLE, the procedure must be of type double. If the item in the GIVING clause is of any other type, the procedure must be of type integer, and the system converts the integer value returned by the procedure to the data type specified in the GIVING clause. If there is no GIVING clause, the procedure must be untyped.

COBOL85 Parameter Types

Table 18–4, "COBOL85 Parameters," lists the allowable parameters to a COBOL85 library and the corresponding ALGOL parameters. For further information about COBOL85 parameters, refer to the *A Series COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation.*

Table 18–4. COBOL85 Parameters

ALGOL Parameter	Corresponding COBOL85 Parameters
DOUBLE	DOUBLE level 77
DOUBLE ARRAY	DOUBLE level 01
(Double Integer)†	BINARY level 77, 12-23 digits Integer (COMPUTATIONAL) 12-23 digits
(Double Integer Array)†	BINARY level 01, 12-23 digits
EBCDIC ARRAY	DISPLAY
EBCDIC STRING	String (DISPLAY)
HEX ARRAY	COMPUTATIONAL and INDEX
INTEGER	BINARY level 77, 1-11 digits Integer (COMPUTATIONAL) 1-11 digits
INTEGER ARRAY	BINARY level 01, 1-11 digits
REAL	REAL level 77
REAL ARRAY	REAL level 01

Legend
† These data types are not implemented in ALGOL.

The BINARY and INTEGER items with 12-23 digits correspond to a data type that is not implemented in ALGOL, but which would be a double-precision integer. These parameter types can be passed only between a COBOL85 user program and a COBOL85 library.

In COBOL85 user programs, the GIVING clause of a CALL statement can specify a variable to receive the value returned by a typed procedure. If the item in the GIVING clause is a level 77 numeric, you can use Table 18–4 to determine the corresponding

procedure type. If the item in the GIVING clause is a level 01 numeric item, the item can match only a procedure of type Integer.

FORTRAN and FORTRAN77 Parameter Types

Table 18–5, "FORTRAN77 Parameters," lists the allowable parameters to a FORTRAN or FORTRAN77 library and their corresponding ALGOL equivalents. For further information about FORTRAN and FORTRAN77, refer to the *A Series FORTRAN Programming Reference Manual* and the *A Series FORTRAN77 Programming Reference Manual*.

Table 18–5. FORTRAN/FORTRAN77 Parameters

ALGOL Parameter	Corresponding FORTRAN Parameter	Corresponding FORTRAN77 Parameter
BOOLEAN	LOGICAL	LOGICAL
BOOLEAN ARRAY [*]	LOGICAL array	LOGICAL array
COMPLEX	COMPLEX	COMPLEX
COMPLEX ARRAY [*]	COMPLEX array	COMPLEX array†
DOUBLE	DOUBLE PRECISION	DOUBLE PRECISION
DOUBLE ARRAY [*]	DOUBLE PRECISION array	DOUBLE PRECISION array†
EBCDIC ARRAY [*]‡	None	CHARACTER
	None	CHARACTER array
INTEGER	INTEGER	INTEGER
INTEGER ARRAY [*]	INTEGER array	INTEGER array
REAL	REAL	REAL
REAL ARRAY [*]	REAL array	REAL array

† The DOUBLEARRAYS option must be set in the FORTRAN77 source.

‡ The EBCDIC ARRAY [*] parameter must be followed by an INTEGER parameter. The INTEGER parameter matches the hidden lower-bounds parameter that FORTRAN77 generates for a CHARACTER or CHARACTER ARRAY.

A FORTRAN or FORTRAN77 array with a lower bound of 1 is equivalent to an ALGOL array with a lower bound of 0.

NEWP Parameter Types

Table 18–6, "NEWP Parameters," lists the allowable parameters to a NEWP library. The corresponding ALGOL parameters are identical. For further information about NEWP, refer to the *A Series NEWP Programming Reference Manual.*

Table 18–6. NEWP Parameters

ALGOL Parameter	Corresponding NEWP Parameter
ASCII ARRAY [0]	ASCII ARRAY [0]
BOOLEAN	BOOLEAN
BOOLEAN PROCEDURE	BOOLEAN PROCEDURE
DIRECT ARRAY [0]	DIRECT ARRAY [0]
DIRECT EBCDIC ARRAY [0]	DIRECT EBCDIC ARRAY [0]
DIRECT FILE	DIRECT FILE
DIRECT HEX ARRAY [0]	DIRECT HEX ARRAY [0]
DIRECT REAL ARRAY [0]	DIRECT REAL ARRAY [0]
DOUBLE	DOUBLE
DOUBLE ARRAY [0]	DOUBLE ARRAY [0]
DOUBLE PROCEDURE	DOUBLE PROCEDURE
EBCDIC ARRAY [0]	EBCDIC ARRAY [0]
EVENT	EVENT
EVENT ARRAY [0]	EVENT ARRAY [0]
FILE	FILE
HEX ARRAY [0]	HEX ARRAY [0]
INTEGER	INTEGER
INTEGER ARRAY [0]	INTEGER ARRAY [0]
INTEGER PROCEDURE	INTEGER PROCEDURE
POINTER	POINTER
PROCEDURE (Untyped)	PROCEDURE (Untyped)
REAL	REAL
REAL	SHORT SET (MAX VALUE <= 47)
REAL ARRAY	LONG SET (MAX VALUE > 47)
REAL ARRAY [0]	REAL ARRAY [0]
REAL PROCEDURE	REAL PROCEDURE
TASK VARIABLE or ARRAY [0]	TASK VARIABLE or ARRAY [0]
TRANSLATE TABLE	TRANSLATE TABLE

continued

Table 18–6. NEWP Parameters (cont.)

ALGOL Parameter	Corresponding NEWP Parameter
TRUTHSET	TRUTHSET

Pascal Parameter Types

Table 18–7, "Pascal Parameters," lists the allowable parameters to a Pascal library and their corresponding ALGOL equivalents. For further information about Pascal, refer to the *A Series Pascal Programming Reference Manual, Volume 1: Basic Implementation.*

Table 18–7. Pascal Parameters

ALGOL Parameter	Corresponding Pascal Parameters
BOOLEAN	Boolean Boolean subrange
BOOLEAN ARRAY [*]	Array of Boolean
BOOLEAN PROCEDURE	Function : Boolean Function : Boolean subrange
DOUBLE	Fixed (n > 11) Sfixed (n > 11)
DOUBLE ARRAY [*]	Array of fixed (n > 11) Array of sfixed (n > 11) Packed array of fixed (n > 11) Packed array of sfixed (n > 11)
DOUBLE PROCEDURE	Function : fixed (n > 11) Function : sfixed (n > 11)
EBCDIC ARRAY [*]	Bits (n) Binary (n) U_display (n) Z_display (n) Display_z (n) S_display (n) Display_s (n) Word48 (n) Word96 (n)

continued

Table 18–7. Pascal Parameters (cont.)

ALGOL Parameter	Corresponding Pascal Parameters
	Integer48
	Integer96
	Real48
	Explicit record (var)
	Packed array of char
	Packed array of subrange (17-256 elements in subrange)
	Packed array of enumeration (17-256 elements in enumeration)
FILE	Systemfile
HEX ARRAY [*]	Hex (n)
	Digits (n)
	S_digits (n)
	Digits_s (n)
	Boolean1
	Boolean4
	Packed array of Boolean
	Packed array of subrange (0-16 elements in subrange)
	Packed array of enumeration (0-16 elements in enumeration)
INTEGER	Integer
	Char
	Enumeration
	Fixed (n < 12)
	Sfixed (n < 12)
	Integer subrange
	Char subrange
	Enumeration subrange

continued

Table 18–7. Pascal Parameters (cont.)

ALGOL Parameter	Corresponding Pascal Parameters
INTEGER ARRAY [*]	Array of integer Array of char Array of enumeration Array of fixed (n < 12) Array of sfixed (n < 12) Array of integer subrange Array of char subrange Array of enumeration subrange Packed array of integer Packed array of fixed (n < 12) Packed array of sfixed (n < 12); Packed array of subrange (> 256 elements in subrange) Packed array of enumeration (> 256 elements in enumeration)
INTEGER PROCEDURE	Function : integer Function : char Function : enumeration Function : fixed (n < 12) Function : sfixed (n < 12) Function : integer subrange Function : char subrange Function : enumeration subrange
PROCEDURE	Procedure
REAL	Real Short set (max value <= 47)
REAL ARRAY [*]	Array of real

continued

Table 18–7. Pascal Parameters (cont.)

ALGOL Parameter	Corresponding Pascal Parameters
	Array of record
	Array of set
	Array of vlstring
	Array of packed array
	Array of explicit type
	Long set (max value > 47)
	Record
	Vlstring
	Explicit record (by-value)
	Packed array of real
	Packed array of set
	Packed array of record
	Packed array of vlstring
REAL PROCEDURE	Function : real

Some types of Pascal parameters can cause extra parameters to be passed if a variable of the parameter is declared as a parameter for a procedure or function. The Pascal parameters affected by this are string schema, fixed length string schema, and any other schema.

Refer to the *A Series Pascal Programming Reference Manual, Volume 1: Basic Implementation* for detailed information about the Pascal parameters.

Subranges of types integer, Boolean, char, or enumeration are mapped as their host type, except when the subranges or the types are components of packed arrays, which are described below.

Each user-defined type identifier is resolved to one of the Pascal parameter types shown in Table 18–7 according to its general type characteristics. For example, type *color*, as it is usually defined, would be considered an *enumerated type* and would be mapped to the generic type *integer*. The following two array declarations are equivalent:

```
array [index-type1] of array [index-type2] of ...

array [index-type1, index-type2] of ...
```

Packed arrays of integers, reals, sets, records, variable-length strings, or other arrays are mapped as unpacked arrays of the same type. For packed arrays of Boolean, char, subrange, or enumeration types, the mapping depends on the number of bits it takes to represent the range of the type. If four bits or fewer are required, the mapping is to a hexadecimal array with lower bound. If five to eight bits are required, the mapping is to an EBCDIC array with lower bound. If nine bits or more are required, the mapping is to an integer array with lower bound. A more detailed description appears in the the data representation discussion in the *A Series Pascal Programming Reference Manual, Volume 1: Basic Implementation*.

If a Pascal library program declares a parameter to be received as a read-only parameter, the user program is allowed to pass the corresponding actual parameter either by call-by-name, call-by-reference, call-by-value, or read-only. This is allowed because the Pascal library program ensures that the parameter's value remains unchanged. A user program can only pass a read-only parameter to a library program that receives the parameter as a read-only parameter, to ensure that the value of the actual parameter is not changed.

PL/I Parameter Types

Table 18–8, "PL/I Parameters," lists the allowable parameters to a PL/I library and the corresponding ALGOL parameters. For further information about PL/I, refer to the *A Series PL/I Reference Manual*.

Table 18–8. PL/I Parameters

PL/I Parameter	Corresponding ALGOL Parameter
Binary Fixed(p, q),	
q = 0, p <= 39	INTEGER
q = 0, 39 < p <= 78	None
q neq 0, p <= 39	None
q neq 0, 39 < p <= 78	None
Binary Float(p),	
p <= 39	REAL
39 < p <= 78	DOUBLE
Bit(n), n <= 48	BOOLEAN
Character(n)	EBCDIC STRING
Character(*)	EBCDIC STRING
Character(n) Varying	EBCDIC STRING
Character(*) Varying	EBCDIC STRING
Decimal Fixed(p, q),	
q = 0, p <= 11	INTEGER
q = 0, 11 < p <= 23	None
q neq 0, p <= 11	None
q neq 0, 11 < p <= 23	None
Decimal Float(p)	
p <= 11	REAL
11 < p <= 23	DOUBLE
Transaction record	TRANSACTION RECORD
Transaction record array	TRANSACTION RECORD ARRAY [*]

Matching Array Lower Bounds

Array parameters can be declared either with an undeclared lower-bound specification or with a formal lower-bound specification. Arrays with undeclared lower bounds are hereafter referred to as *unbounded arrays*. Arrays with formal lower bounds are hereafter referred to as *simple arrays*.

In the case of unbounded arrays, the lower-bound value of the array is provided during execution by the program that calls the procedure. In the case of simple arrays, the lower-bound value of the array is fixed during compilation, typically to a value of 0 (zero). The actual value of the lower-bound parameter for the simple array is ignored during execution.

If an unbounded array appears as a parameter to an imported or exported procedure, the system generates one or more hidden parameters that pass the actual lower bound for each dimension of the array. These hidden parameters are integer parameters that are passed by value.

The syntax for array specifications in the various programming languages is described in the following table.

Table 18–9. Unbounded and Simple Array Declarations

Language	Unbounded Array	Simple Array
ALGOL	ARRAY A[*];	ARRAY A[0];
C	float (*) []	None
COBOL(68)	01 A COMP WITH LOWER-BOUNDS.	01 A COMP.
COBOL74	01 A BINARY. 77 B PIC S9(11) BINARY.	01 A BINARY.
COBOL85	01 A BINARY WITH LOWER-BOUNDS.	01 A BINARY.
FORTRAN	REAL A(X).	None
FORTRAN77	REAL A(*).	None
NEWP	ARRAY A[*];	ARRAY A[0];
Pascal	var a: ARRAYTYPE;	None

Libraries and user programs written in COBOL(68) or COBOL85 can receive unbounded array parameters by including a LOWER-BOUNDS clause in the formal array declaration. A COBOL(68) or COBOL85 library always treats such an array parameter as if it had a lower bound of 0 (zero), regardless of the actual lower bound passed by the user program.

COBOL74 libraries and user programs can specify unbounded array parameters by adding a numeric parameter that receives the lower bound. In a COBOL74 user program, this extra parameter must occur immediately after the unbounded array in the CALL statement parameter list. In a COBOL74 library program, this extra

parameter must occur immediately after the unbounded array in the USING clause of the PROCEDURE DIVISION. COBOL74 libraries always treat array parameters as if they had a 0 (zero) lower bound, regardless of the value passed in the extra parameter.

Pascal array parameters can reference space in a data pool or in the heap. As a result, if a Pascal user program passes an array parameter to a library, some constructs in the library might result in incorrect references or cause overwrite corruption of other arrays stored in the same data pool or heap.

For example, suppose a Pascal user program passes an array A to an ALGOL library. In the exported ALGOL procedure, the expression *POINTER(A)* references the first element in the data pool or heap. By contrast, the expression *POINTER(A[0])* correctly references the actual first element of the Pascal array A. Other constructs that can cause similar problems, if not carefully used, include the SIZE and REMAININGCHARS functions and the REPLACE, SCAN, and RESIZE statements. Note that there is no system function that can return the size of a Pascal array.

Additional problems can arise if a Pascal user program passes an array parameter to a library written in a COBOL language. Like ALGOL libraries, COBOL libraries have no way of determining the upper bound of an array in the heap. However, COBOL libraries have the additional limitation that the lower bound of the array is always treated as 0, regardless of where the array starts in the data pool.

Matching Parameter-Passing Mode

Library parameters can be passed by value, by reference, by name, or as read-only. The read-only property causes the compiler to select the most efficient parameter-passing mode, and prevents the receiving procedure from modifying the value of the parameter. For an introduction to these parameter-passing modes and read-only parameters, refer to "Parameter Passing Modes" in Section 17, "Using Parameters."

If the library program declares a parameter to be received as a read-only parameter, the user program can pass the parameter as call-by-name, call-by-reference, call-by-value, or read-only. If a library program declares a parameter to be received by name or by reference, the user program can pass the parameter by name, by reference or by value. If the library program declares a parameter to be received by value, the user program can only pass the parameter by value.

Table 18–10, "Parameter–Passing Modes," illustrates the parameter-passing rules. In the table, the legal combinations of parameter-passing modes are marked with an X.

Table 18–10. Parameter-Passing Modes

Library Program	User Program			
	Read-Only	Name	Reference	Value
Read-Only	X	X	X	X
Name		X	X	X
Reference		X	X	X
Value				X

In ALGOL programs, parameters are declared with or without a VALUE clause. In ALGOL library programs, parameters declared with a VALUE clause are received by value. Parameters declared without the VALUE clause are received by reference, except for formal procedures and integer, real, double, Boolean, and complex variables, which are received by name. In ALGOL user programs, library procedure parameters declared with a VALUE clause are passed by value. Parameters declared without a VALUE clause are passed by reference, except for integer, real, double, Boolean, and complex variables, which are passed by name.

In COBOL(68) and COBOL74, parameters are passed by reference; therefore, a COBOL(68) or COBOL74 user program cannot call a library that has declared its parameters by value. An ALGOL library must declare its parameters to be by name or by reference if the ALGOL library is to be called by a COBOL(68) or COBOL74 program.

A COBOL(68) library can declare its 77-level COMP or COMP-1 parameters to be received by content (value) or by reference (which is the default). A COBOL74 library must declare its 77-level BINARY parameters to be received by reference. If any BINARY parameter is received by content, the program is not library-capable. Due to the parameter-passing rules and the ALGOL, COBOL(68), and COBOL74 language restrictions, an ALGOL program that calls a COBOL(68) or COBOL74 library must declare its integer, real, or double parameters to be by value.

In COBOL85, parameters can be passed by reference or by value.

In FORTRAN and FORTRAN77, variable parameters are passed by name; array parameters are passed by reference.

In NEWP, parameters are declared to be by value or by reference. Integers, real, double, and Boolean variables that are not declared to be by value are passed by reference.

In Pascal, parameters are passed by reference, read-only, or value. Parameters that are not declared to be passed by reference ("VAR" parameters) or by read-only ("CONST" parameters) are passed by value.

In PL/I, parameters are passed by reference or by name (not by value). A PL/I library declares its parameters to be received by reference. A user program must pass parameters by reference to a PL/I library.

Delinking from Libraries

The system automatically delinks a user process from a library when the user process exits the block in which the library is declared. However, at times it can be useful to delink a user process from a library at an earlier point. For example, delinking a user process from a library enables the user process to modify one or more of the library attributes. The user process cannot modify these library attributes while it is linked to the library.

There are two features that allow a user process to explicitly delink from a library: the DELINKLIBRARY function and the CANCEL statement.

The DELINKLIBRARY function, which is available in ALGOL and Pascal, delinks the user process from the library process without affecting any of the other processes using the library. The library remains frozen, unless it is a temporary library and the delinked process was the only process using the library.

The CANCEL statement, which is available in ALGOL, COBOL74, and COBOL85, also delinks the user process from the library process, but has the additional effect of causing the library process to unfreeze and resume execution as a regular process. This is true whether or not the library has a permanent or temporary freeze.

However, only libraries with a sharing option of PRIVATE or SHAREDBYRUNUNIT can be canceled. An attempt to cancel a SHAREDBYALL library results in the warning message "CANCEL WARNING, SHARED LIBRARY WAS DELINKED". In this case, the user process is delinked as if it had performed a DELINKLIBRARY function, and the library process remains frozen unless it is a temporary library with no other users.

If a user process cancels a SHAREDBYRUNUNIT library, then any other user processes in the same run unit that are currently linked to the library process lose their linkage. The next time one of these processes invokes an object in the canceled library, the system initiates a new instance of the library and links the process to the new library instance.

Note that internal tasks of a user process can link to a library by way of a single library declaration in the user process. If such an internal task is executing a library procedure when the parent user process executes a DELINKLIBRARY function or a CANCEL statement, the internal task is discontinued.

Thawing and Resuming Libraries

Thawing a library is the act of changing the frozen library process from a permanent library into a temporary library. By contrast, the act of *resuming* a library causes the process to lose its library status and resume execution as an ordinary process. Execution of the process resumes with the first statement after the FREEZE statement.

You can thaw a library process programmatically through assignments to the STATUS task attribute of a process, or operationally through the THAW (Thaw Frozen Library) system command. For details, refer to Section 6, "Monitoring and Controlling Process Status."

Additionally, you can resume a library process programmatically with the CANCEL statement in ALGOL, as discussed under "Delinking from Libraries" earlier in this section.

Determining Which Users Are Linked to a Library

You can use the Y (Status Interrogate) system command to display the user processes that are linked to a library. The following is an example of this display:

```
Status of Job 4840/4840 at 18:40:35
Program name: *SYSTEM/GENERALSUPPORT
Priority: 50
Origination: Unit 0
Stack State: Frozen

This library is being used by 8 programs:
   The MCP
   9345: *SYSTEM/SDASUPPORT ON SYS38
   4972: *SYSTEM/TCPHOSTSERVICES
   4876: *SYSTEM/TCPIP/BNAV2/MANAGERS/12152
   6551: *OBJECT/MAIL
   2953: (SWDUNCAN)MCP/39/TRAP/MULTICOMP/AMLIP
   2915: (JASMITH)MCP/MM17A ON MCPS
   4983: *SYSTEM/PRINT/REMOTE/SERVER
```

A DCALGOL program can obtain the same information by using the GETSTATUS call with type 0 (Mix Entries), subclass 1, and mask bits 18 and 34 set. Mask bit 18 returns the number of user processes linked to the library. Mask bit 34 returns a list of the user processes.

Understanding Library Process Structure

In most respects, an imported object can be used just as if it were declared by the user process rather than the library. The following subsections explore the implications of this structure for scope of declarations, task attribute usage, and error handling.

Process Stacks

A user process can either enter or initiate an imported procedure.

If the procedure is entered, it is executed as part of the user process stack. An imported procedure is never executed in the library process stack.

If the procedure is initiated, it must be as a dependent process. The system creates a new process stack to execute the procedure. The resulting process is considered to be an external process. (For information about external processes, refer to Section 2, "Understanding Interprocess Relationships.")

Library Task Attributes

If a user process enters an imported procedure, the task attributes of the user process govern the execution of the procedure. If the MYSELF predeclared task variable is used in the imported procedure, MYSELF refers to the user process. If the MYJOB task variable is used in the imported procedure, MYJOB refers to the job of the user process (that is, the eldest ancestor of the user process).

If a user process initiates an imported procedure, the resulting process receives its own set of task attributes. In this case, the MYSELF task variable refers to the new process. The MYJOB task variable refers to the job of the user process and the new process. (Because the new process must be dependent, the user process and the new process always have the same job.)

Thus, the MYSELF task variable, when referenced in an imported procedure, never refers to the task attributes of the library process. There is no direct way for a user process to access the task attributes of the library process it is linked to.

Certain task attributes are particularly useful in the implementation of libraries and user programs. These include the following:

- LIBRARY

  This task attribute can be used to pass library equations to a user process at run time. A library equation modifies the library attributes of libraries declared in the user process. Each library equation is applied to the library declaration with the corresponding internal name, as discussed under "INTNAME" earlier in this section.

- LIBRARYSTATE

  A library process can use this task attribute to determine whether the library process was initiated through the library linkage mechanism.

- LIBRARYUSERS

  This task attribute records the number of user processes that are currently linked to a library. For example, a control library could interrogate its own LIBRARYUSERS attribute before determining whether to thaw itself.

- STATUS

  A frozen library process can be thawed by assigning this task attribute a value of GOINGAWAY, as discussed in Section 6, "Monitoring and Controlling Process Status." In addition to thawing the library, the GOINGAWAY assignment prevents further user processes from linking to this library instance.

Error Handling

If a user process encounters a fault while executing an imported procedure, the system treats this as a fault in the user process rather than in the library process. If neither the imported procedure nor the user program incorporates fault handling code, the fault causes the user process to be terminated. The fault has no effect on the status of the library process.

A permanent or temporary library process cannot incur any faults while it is frozen, because it is not executing any statements. However, an operator can terminate the process with a DS (Discontinue) system command. Further, a control library process can incur faults while executing the control procedure, even while the library is frozen. If a library process is terminated by a fault or operator action while user processes are linked to it, the system also discontinues all the user processes.

For information about errors that can occur when a user process links to a library process or invokes an imported procedure, refer to "Linking to Libraries" earlier in this section.

Providing Global Objects

Shared libraries provide perhaps the most sophisticated means for communicating information between processes. Any number of user processes can access the same object by way of a shared library. The user processes can belong to different process families and can be written in different languages.

The main benefit of using a library for IPC is the flexible control it provides over the interactions between user processes and shared objects. For example, if a data item is being made available to many different user processes, the library can act to protect the data from being corrupted by a wrongly designed user process. The library can also filter information, so that a particular piece of data in an object can be made visible to one user and not to others. Also, a library can provide a simple interface to information that has a complex structure.

The key to using libraries for IPC lies in the addressing environment of an exported library procedure. Such a procedure can access any objects declared globally to it in the library. The rules for determining if a declaration is global to a given ALGOL procedure are discussed in Section 15, "Using Global Objects." Objects declared in COBOL(68) or COBOL74 libraries are global to the exported PROCEDURE DIVISION unless they are specified as parameters to the PROCEDURE DIVISION. For information about the scope of declarations in other languages, refer to the appropriate programming language manuals.

A library procedure can also access objects that are passed to it as parameters by a user process. These parameters can be used to inform the library procedure of changes that need to be made to a global object.

If the library procedure is a typed procedure, then you can use the return value to transfer information about the global object back to the user process. You can also use parameters that are passed to the procedure, by name or by reference, to transfer information back to the user process.

For user processes to communicate through a library, they must access the same instance of the library. You can ensure this by following these steps:

1. Set the SHARING option to SHAREDBYALL. This prevents a separate instance of the library from being initiated each time a new user process links to the library.
2. Use a permanent freeze in some cases. A library with a temporary freeze suffices for most cases, because it does not resume until all user processes have terminated. However, if there will be periods of time when no user processes are linked to the library, and the communication information needs to be preserved, then a permanent freeze must be used. This preserves the library instance until it is thawed by an operator command or a programmatic change to the STATUS task attribute.

Note that objects declared within a library procedure cannot be used for IPC. Even if user processes access the same instance of a library, they receive different instances of each exported library procedure. Changes made to these local objects by one user process are not visible to other user processes.

A library can be written to ensure that only one user process can access a particular global object at a time. For example, the library procedure that accesses a particular global object could be written to first procure a globally declared event, then access the global object, and then liberate the event. If all the library procedures that access the global object are written this way, then the global object is protected from simultaneous access by different user processes.

Events can also be used to ensure that user processes access a global object in a certain order. For example, assume that user process A is supposed to access a particular global object before user process B does. User process B could invoke a library procedure that waits on a certain global event. This event might be one that is caused only at the end of the library procedure called by user process A.

A library procedure can use the MYSELF task variable to access the task attributes of the user process. (In a library procedure, MYSELF always refers to the user process, not the library.) For example, the library procedure could interrogate the USERCODE task attribute of the user process. The library procedure could be defined to provide different actions for different user processes.

When designing a library, you must be aware of the fact that any of the user processes might be discontinued while executing a procedure from the library. If the library procedure being executed has procured an event, but has not yet liberated the event, then the event remains procured. Other user processes waiting to procure the event wait indefinitely. For further information, refer to the discussion of discontinued processes and events in Section 16, "Using Events."

Another point to be aware of is that the information stored in a permanent library can be lost if a system halt/load terminates the library process. The library can protect against this possibility by writing data out to disk files.

For an simple example of a library that provides user processes with shared access to a disk file, refer to "File Sharing Examples" in Section 19, "Using Shared Files".

Security Considerations

Users of a library can be restricted through the use of the SECURITYTYPE file attribute of the library's object code file. If the SECURITYTYPE value is PRIVATE, then any nonprivileged process that uses the library must have the same usercode as the library. If the SECURITYTYPE value is GUARDED or CONTROLLED, then a guard file is used to restrict the nonprivileged library users.

When a user process enters an imported procedure, the procedure is executed under the usercode of the user process, with whatever privileges are defined for the usercode.

A process can also temporarily assume additional privileges while executing an imported procedure. This is the case because the MP (Mark Program) system command can be used to assign options to a library object code file. These options can confer compiler status, control program status, privileged status, security administrator status, or tasking status. A user process benefits from these added privileges only while executing procedures imported from that library; it is not enough simply to be linked to the library.

Alternatively, the MP command can assign privileged transparent status, security administrator transparent status, or tasking transparent status to the library object code file. In this case, library procedures inherit privileges from the object code file of the user process that invokes the procedures.

For more information about privileges inherited from usercodes and from object code files, refer to Section 5, "Establishing Process Identity and Privileges."

The operating system recognizes a special class of libraries called *system libraries*. System libraries receive a special security status that allows them to access protected objects in the operating system. Much of the system software is provided in the form of system libraries. Examples of system libraries are GENERALSUPPORT, which provides intrinsic functions; MARCSUPPORT, which provides Menu-Assisted Resource Control (MARC); and DSSSUPPORT, which supports distributed system services (DSSs).

System libraries are always support libraries; that is, function names must be specified for these libraries through the SL (Support Libraries) system command. However, not all support libraries are system libraries. System libraries receive their special privileges only if they are initiated through the library linkage mechanism on behalf of a user process that links to the library by function. (For an introduction to support libraries, refer to "FUNCTIONNAME" earlier in this section.)

You can also use the SL command to assign any or all of the following security-related attributes to a library: LINKCLASS, MCPINIT, ONEONLY, SYSTEMFILE, and TRUSTED.

Of these attributes, TRUSTED and LINKCLASS are both related to the concept of *linkage classes*. These are classes that provide an additional level of library access control, beyond that provided by the security-related file attributes. Even if the library program is a public file, a user program must have an appropriate linkage class in order to use that library.

If the TRUSTED attribute of a library is set, then the system evaluates the linkage class for each library procedure separately. In ALGOL and NEWP libraries, you can use the export list to specify the linkage class for each library procedure.

If the TRUSTED attribute of a library is reset, then the system treats all the library procedures as having the same linkage class. You can assign the linkage class for a support library with the LINKCLASS option of the SL system command.

The following are the linkage classes that can be assigned to a library or a library procedure:

Library Linkage Class	Meaning
0	Unprotected. Programs of any linkage class can link to the library. This is the default value.
1	Only programs of linkage class 1 can link to the library.
2 to 7	Reserved for use by system software.
8 to 15	Free for site-dependent definition and use.

The linkage class of a user process is assigned by the system at the time the process is initiated. The following are the predefined linkage classes that the system can assign to a user process:

User Process Linkage Class	Type of Process	Compatible Library Linkage Classes
0	Default	0
1	Master control program (MCP) and some system libraries such as COMSSUPPORT, DATACOMSUPPORT, and PRINTSUPPORT	Any
2	Message control systems (MCSs) and tasking programs	0, 2, 3, 4
3	Environment libraries	0, 3, 4
4	Programs marked as privileged by the PU option of the MP (Mark Program) system command or by the PP (Privileged Program) system command	0, 4
5	Programs marked as compilers by the COMPILER option of the MP (Mark Program) system command or by the MC (Make Compiler) system command	0, 5

A library can itself become a user process if it calls another library. If the library process was initiated BYFUNCTION, then the LINKCLASS value assigned by the SL command determines the rights of that library as a user process.

The following are the remaining library attributes assignable by the SL command, and their meanings:

Attribute Name	Meaning
MCPINIT	If set, the library object code file can be initiated only by the operating system.
ONEONLY	If set, only one version of the library code file is permitted to be in use on the system at any one time.
SYSTEMFILE	If set, the library code file is to be made a nonremovable system file when initiated or marked as a support library by an SL command.

Note that some system libraries have selected library attributes assigned to them automatically by the operating system. You can use the SL command to set any Boolean attribute of a system library. However, if the operating system has already set a Boolean attribute, you cannot use the SL command to reset that attribute. The Boolean attributes are MCPINIT, ONEONLY, SYSTEMFILE, and TRUSTED.

If the LINKCLASS value of a system library is set by the operating system, then you can use an SL command to change the LINKCLASS only to values that are able to link to the system-defined LINKCLASS. For example, you can use an SL command to change the LINKCLASS from 2 to 1, but not from 2 to 3.

If an SL command assigns attributes to a system library that violate the preceding rules, the system displays the error message "LIBRARY ATTRIBUTES NOT CHANGEABLE".

Library Debugging

If you are debugging a user program, and you are not sure whether an observed bug originates in the user program or in one of the libraries it uses, then it can be helpful to set the LIBRARIES option of the OPTION task attribute. The LIBRARIES option causes information related to libraries to be included in any program dumps generated by the user process. This information includes

- The contents of all library process stacks to which the user process is linked.
- The contents of the *library directory* in each library process stack. There is one library directory for each export list in a library. However, only one library directory is in effect at the time a library freezes. For each export object, the library directory stores the name, the type of object, and the type of linkage used (direct, indirect, or dynamic). For exported procedures, the library directory also stores a description of the parameters of the procedure.
- The contents of all *library templates* in the user process stack. One library template exists for each library declaration executed by the user process. A library template stores information about the library attributes. A library template also stores descriptions of all the objects imported from a particular library, including the name, type of object, and parameters.

Library Examples

The following subsections give examples of libraries and user programs that import procedures from these libraries.

ALGOL Library: OBJECT/FILEMANAGER/LIB

The following library, called OBJECT/FILEMANAGER/LIB, provides a set of file management routines through dynamic linkage. This library represents features of dynamic linkage but does not necessarily represent efficient programming.

```
$SHARING = PRIVATE
BEGIN                                  % FILEMANAGER/LIB.
TASK ARRAY LIBTASKS [0:10];            % PROVIDE UP TO 11 DIFFERENT LIBRARY
                                       % PROCESSES.
STRING ARRAY FILETITLES [0:10];        % LIBPARAMETER FOR EACH LIB PROCESS

PROCEDURE FILEMANAGER (TASKINDEX);
  VALUE   TASKINDEX;
  INTEGER TASKINDEX;

BEGIN
  PROCEDURE READFILE;
```

```
  BEGIN
    .
    .
    .
  END READFILE;

  PROCEDURE WRITEFILE;
  BEGIN
    .
    .
    .
  END WRITEFILE;

  EXPORT READFILE, WRITEFILE;

  FREEZE (TEMPORARY);
  FILETITLES [TASKINDEX] := ".";
END FILEMANAGER;

PROCEDURE SELECTION (USERSFILE, MCPCHECK);
  VALUE  USERSFILE;
  EBCDIC STRING USERSFILE;
  PROCEDURE MCPCHECK (T); TASK T; FORMAL;
BEGIN
  INTEGER TASKINDEX;
  BOOLEAN FOUND;

  % LOOK AT ALL THE FILETITLES CHECKING TO SEE IF A LIBRARY PROCESS
  % HAS ALREADY BEEN INITIATED FOR FILE TITLE USERSFILE.
  WHILE NOT FOUND AND (TASKINDEX LEQ 10) DO
  BEGIN
    IF FILETITLES [TASKINDEX] = USERSFILE THEN
       FOUND := TRUE
    ELSE
       TASKINDEX := * + 1;
  END;

  % IF NO LIBRARY PROCESS EXISTS FOR THIS FILE TITLE, THEN CREATE ONE
  IF NOT FOUND THEN
  BEGIN
    WHILE NOT FOUND DO   % FIND AN UNUSED TASK VARIABLE
    BEGIN
      TASKINDEX := 0;
      WHILE NOT FOUND AND (TASKINDEX LEQ 10) DO
        IF LIBTASKS [TASKINDEX].STATUS LEQ 0 THEN
          FOUND := TRUE
        ELSE
          TASKINDEX := * + 1;
      IF NOT FOUND THEN
        % WAIT A SECOND AND MAYBE
        % A LIBRARY PROCESS WILL GO TO EOT.
        WAIT ((1));
```

```
    END;
    PROCESS FILEMANAGER (TASKINDEX) [LIBTASKS [TASKINDEX]];
    WHILE LIBTASKS [TASKINDEX].STATUS NEQ VALUE (FROZEN) DO
      WAIT ((1));
    FILETITLES [TASKINDEX] := USERSFILE;
  END;

  MCPCHECK (LIBTASKS [TASKINDEX]);
END SELECTION;

PROCEDURE READFILE;
  BY CALLING SELECTION;
PROCEDURE WRITEFILE;
  BY CALLING SELECTION;

EXPORT READFILE, WRITEFILE;
FREEZE (TEMPORARY);
END LIBRARY.
```

Before attempting to understand this example, you should be familiar with the concepts discussed under "Dynamic Linkage" in this section.

OBJECT/FILEMANAGER/LIB exports two procedures: READFILE and WRITEFILE. OBJECT/FILEMANAGER/LIB provides these procedures dynamically. The procedures are ultimately provided by various library processes that are initiated by OBJECT/FILEMANAGER/LIB. Each of these offspring library processes is an instance of the procedure FILEMANAGER. Each library process is intended to provide read and write access to a different data file. Each user process is expected to use the LIBPARAMETER library attribute to indicate the name of the file to be read or written.

The system automatically invokes the SELECTION procedure whenever a user process first links to OBJECT/FILEMANAGER/LIB. The system passes an MCP procedure to the MCPCHECK parameter of the SELECTION procedure. The system also passes the LIBPARAMETER attribute specified by the user process to the USERSFILE parameter of the SELECTION procedure.

The SELECTION procedure searches the string array FILETITLES to see if a library process with the name specified by USERSFILE is already running. If so, the SELECTION procedure selects the task variable of the requested library process. If no library process with the requested name is yet running, SELECTION initiates a new library process and stores the name of the process in the FILETITLES array.

After SELECTION has selected a task variable, it invokes the procedure MCPCHECK, passing the selected task variable as a parameter. The MCPCHECK procedure informs the system to link the user process to the library process with the specified task variable. The actual library linkage is not performed until SELECTION has been exited.

ALGOL User Program #1

The following ALGOL user program invokes the ALGOL dynamic library in the previous example, OBJECT/FILEMANAGER/LIB. This user program reads information from the file MYFILE by assigning a value of "MYFILE" to the LIBPARAMETER library attribute and then invoking the procedure READFILE. The user program then writes information to a file called OTHERFILE by canceling the library, changing the LIBPARAMETER library attribute to "OTHERFILE", and invoking the WRITEFILE procedure.

```
BEGIN
LIBRARY L (TITLE = "OBJECT/FILEMANAGER/LIB.");
PROCEDURE READFILE;
  LIBRARY L;
PROCEDURE WRITEFILE;
  LIBRARY L;

L.LIBPARAMETER := "MYFILE";

READFILE;

CANCEL (L);

L.LIBPARAMETER := "OTHERFILE"; % LIBPARAMETER CAN BE CHANGED
                               % BECAUSE THE LIBRARY HAS BEEN
                               % CANCELED.

WRITEFILE;
END PROGRAM.
```

ALGOL Library: OBJECT/SAMPLE/LIBRARY

The following ALGOL library, compiled as OBJECT/SAMPLE/LIBRARY, uses direct library linkage:

```
BEGIN
ARRAY MSG[Ø:12Ø];

INTEGER PROCEDURE FACT(N);
     INTEGER N;

     BEGIN
     IF N LSS 1 THEN
          FACT := 1
     ELSE
          FACT := N * FACT(N - 1);
     END;  % OF FACT.


PROCEDURE DATEANDTIME(TOARRAY, WHERE);
     ARRAY TOARRAY[*];
     INTEGER WHERE;

     BEGIN
     REAL T;
     POINTER PTR;

     T := TIME(7);
     PTR := POINTER(TOARRAY, 8) + WHERE;
     CASE T.[5:6] OF
          BEGIN
           Ø: REPLACE PTR:PTR BY "SUNDAY, ";
           1: REPLACE PTR:PTR BY "MONDAY, ";
           2: REPLACE PTR:PTR BY "TUESDAY, ";
           3: REPLACE PTR:PTR BY "WEDNESDAY, ";
           4: REPLACE PTR:PTR BY "THURSDAY, ";
           5: REPLACE PTR:PTR BY "FRIDAY, ";
           6: REPLACE PTR:PTR BY "SATURDAY, ";
          END;
     REPLACE PTR BY T.[35:6] FOR 2 DIGITS, "-",
                    T.[29:6] FOR 2 DIGITS, "-",
                    T.[47:12] FOR 4 DIGITS, ", ",
                    T.[23:6] FOR 2 DIGITS, ":",
                    T.[17:6] FOR 2 DIGITS, ":",
                    T.[11:6] FOR 2 DIGITS;
     END; % OF DATEANDTIME.

EXPORT FACT, DATEANDTIME AS "DAYTIME";
REPLACE POINTER(MSG, 8) BY
     "  - SAMPLE LIBRARY STARTED",
     " " FOR 94;
DATEANDTIME(MSG, 6Ø);
```

```
DISPLAY(MSG);
FREEZE(TEMPORARY);
REPLACE POINTER(MSG, 8)+ 19 BY "ENDED  ";
DATEANDTIME(MSG, 60);
DISPLAY(MSG);
END.
```

ALGOL Library: OBJECT/SAMPLE/DYNAMICLIB

The following ALGOL library, compiled as OBJECT/SAMPLE/DYNAMICLIB, uses dynamic and indirect library linkage. This library references the library OBJECT/SAMPLE/LIBRARY in the preceding example:

```
BEGIN
TASK LIB1TASK, LIB2TASK;
LIBRARY SAMLIB(TITLE= "OBJECT/SAMPLE/LIBRARY.");
INTEGER PROCEDURE FACT(N);
     INTEGER N;
     LIBRARY SAMLIB;

PROCEDURE DYNLIB1;
% LIBRARY PROVIDED DYNAMICALLY AND INDIRECTLY
     BEGIN % PRINTS DATE WITH TIME.
     LIBRARY SAMLIB (TITLE="OBJECT/SAMPLE/LIBRARY.");
     PROCEDURE DAYTIME(TOARRAY, WHERE);
          ARRAY TOARRAY[*];
          INTEGER WHERE;
          LIBRARY SAMLIB;
     EXPORT DAYTIME;
     FREEZE(TEMPORARY);
     END; % OF DYNLIB1.

PROCEDURE DYNLIB2;
% LIBRARY PROVIDED DYNAMICALLY
     BEGIN % PRINTS DATE WITHOUT TIME.
     PROCEDURE DAYTIME(TOARRAY, WHERE);
          ARRAY TOARRAY[*];
          INTEGER WHERE;

          BEGIN
          REAL T;
          T := TIME(7);
          REPLACE POINTER(TOARRAY, 8) + WHERE
               BY T.[35:6] FOR 2 DIGITS, "-",
                  T.[29:6] FOR 2 DIGITS, "-",
                  T.[47:12] FOR 4 DIGITS;
          END; % OF DAYTIME.
     EXPORT DAYTIME;
     FREEZE(TEMPORARY);
     END; % OF DYNLIB2
```

```
   END;

EXPORT PLIB1_A, PLIB1_B;
FREEZE(TEMPORARY);
END.
```

The third program (MYLIB2.) is a library that uses another library (MYLIB1.).

```
BEGIN
LIBRARY L(TITLE="OBJECT/MYLIB1.");
REAL PROCEDURE PLIB1_B (A, B);
   VALUE A;
   REAL A, B;
   LIBRARY L;

REAL PROCEDURE PLIB2_A (R);
   VALUE R;
   REAL R;
   BEGIN
   REAL X, Y;
   .......
   PLIB1_B (X, Y)            %% Procedure in MYLIB1; circular linkage
   .......                   %% is allowed, because MYLIB1 is frozen.
   PLIB2_A :=Y;
   END;

BOOLEAN PROCEDURE PLIB2_B (X);
   VALUE X;
   REAL X;
   BEGIN
   .......
   END;

EXPORT PLIB2_A, PLIB2_B;
FREEZE(TEMPORARY);
END.
```

ALGOL Incorrect Circular Libraries

The following are examples of libraries and user programs that use circular linkage incorrectly.

Example 1: Indirect Self Referencing

The following is the user program OBJECT/INDIRECT/CALL. This program invokes procedure X in the library OBJECT/INDIRECT/LIB1.

```
BEGIN
  LIBRARY L(TITLE="OBJECT/INDIRECT/LIB1.");
  PROCEDURE X;
    LIBRARY L;
  X;
END.
```

The following is the library OBJECT/INDIRECT/LIB1. This library provides procedure X indirectly by importing it from another library, OBJECT/INDIRECT/LIB2.

```
$SHARING = SHAREDBYALL
BEGIN
  LIBRARY L(TITLE="OBJECT/INDIRECT/LIB2.");
  PROCEDURE X;
    LIBRARY L;
  EXPORT X;
  FREEZE(PERMANENT);
END.
```

The following is the library OBJECT/INDIRECT/LIB2. This library also provides procedure X indirectly, in this case by importing procedure X from OBJECT/INDIRECT/LIB1.

```
$SHARING = SHAREDBYALL
BEGIN
  LIBRARY L(TITLE="OBJECT/INDIRECT/LIB1.");
  PROCEDURE X;
    LIBRARY L;
  EXPORT X;
  FREEZE(PERMANENT);
END.
```

This chain of linkages is completely circular. That is, the chain leads back not just to the original library, but also to the original procedure. When the user program invokes procedure X, the system discontinues the program and displays the message "CURRENT CIRCULAR LIBRARY REFERENCE STRUCTURE IS NOT ALLOWED."

Example 2: Direct Self Referencing

The following ALGOL library, called OBJECT/ALGOL/SELF/LIB, attempts to provide a procedure by importing it from the same procedure in the same library. When a user process attempts to invoke procedure X in this library, the user process hangs. The Y (Status Interrogate) system command shows a STACK STATE of WAITING ON AN EVENT, but the process does not appear in the W (Waiting Mix Entries) system command display. This situation continues until an operator enters a DS (Discontinue) system command or until the system is halt/loaded.

```
$SHARING = SHAREDBYALL
BEGIN
  LIBRARY L(TITLE="OBJECT/ALGOL/SELF/LIB.");
  PROCEDURE X;
     LIBRARY L;
  EXPORT X;
  FREEZE(TEMPORARY);
END.
```

Example 3: Libraries that Wait on Each Other

The following user program invokes a procedure in the library OBJECT/LIB/WAIT1.

```
BEGIN
  LIBRARY L(TITLE="OBJECT/LIB/WAIT1.");
  PROCEDURE X;
     LIBRARY L;
  X;
END.
```

The following is the library OBJECT/LIB/WAIT1. Before freezing, this library invokes a procedure in the library OBJECT/LIB/WAIT2.

```
$SHARING = SHAREDBYALL
BEGIN
  LIBRARY L(TITLE="OBJECT/LIB/WAIT2.");
  PROCEDURE X;
     LIBRARY L;
  PROCEDURE Y;
     DISPLAY ("Y");
  EXPORT X, Y;
  X;
  FREEZE(TEMPORARY);
END.
```

The following is the library OBJECT/LIB/WAIT2. Before freezing, this library invokes a procedure in the library OBJECT/LIB/WAIT1.

```
$SHARING = SHAREDBYALL
BEGIN
  LIBRARY L(TITLE="OBJECT/LIB/WAIT1.");
  PROCEDURE Y;
     LIBRARY L;
  PROCEDURE X;
     DISPLAY ("X");
  EXPORT X, Y;
  Y;
  FREEZE(TEMPORARY);
END.
```

Because OBJECT/LIB/WAIT1 was initiated through the library linkage mechanism, is SHAREDBYALL, and has not yet frozen, OBJECT/LIB/WAIT2 waits for OBJECT/LIB/WAIT1 to freeze. Both libraries are then waiting for each other to freeze. The user process hangs indefinitely. The Y (Status Interrogate) system command shows the user process to have a STACK STATE of WAITING ON AN EVENT, but the user process does not appear in the W (Waiting Mix Entries) system command display. This situation continues until an operator enters a DS (Discontinue) system command or until the system is halt/loaded.

C Library and ALGOL User Program

The following pair of examples illustrate the ability of an ALGOL calling program to indirectly access character data in a C library.

The following C library exports a procedure named WRITELINE. This procedure accepts a parameter that is a pointer to a string of characters. The procedure writes the string to a file named TEST.

```
#include <stdio.h>
#include <stdlib.h>
FILE * pf;

asm WRITELINE(char * pc) {
        fputs(pc, pf);
        fputc('\n',pf);
}

void cleanup(void) {
        /* called after main exits, i.e., after thaw */
        fputs("all done\n", pf);
        fclose(pf);
}

main() {
        pf = fopen("TEST", "w");
        atexit(cleanup);
}
```

The C library is called by the following ALGOL user program:

```
BEGIN

LIBRARY CLIB(LIBACCESS = BYTITLE, TITLE="OBJECT/STREAM/C.");
INTEGER PROCEDURE MALLOC(BYTES);
   VALUE                BYTES;
   INTEGER              BYTES;
   LIBRARY CLIB;
INTEGER PROCEDURE FREE(CPTR);
   VALUE              CPTR;
   INTEGER            CPTR;
   LIBRARY CLIB;
INTEGER PROCEDURE HEAPTOPTR(CPTR, APTR);
   VALUE                   CPTR      ;
   INTEGER                 CPTR      ;
   POINTER                       APTR ;
   LIBRARY CLIB;
INTEGER PROCEDURE WRITELINE(CPTR);
   VALUE                   CPTR ;
   INTEGER                 CPTR ;
   LIBRARY CLIB;

PROCEDURE XFER(S);
   VALUE      S ;
   STRING     S ;
BEGIN
   POINTER APTR;
   INTEGER CPTR;
   CPTR := MALLOC (LENGTH(S) + 1);
   HEAPTOPTR(CPTR, APTR);
   REPLACE APTR BY S, 48 "ØØ";
   WRITELINE(CPTR);
   FREE     (CPTR);
END XFER;

XFER("HELLO WORLD");
XFER("THIS IS AN EXAMPLE");

END.
```

In addition to importing WRITELINE from the C library, this program also imports the procedures MALLOC, HEAPTOPTR, and FREE. These procedures are implicitly created by the *#include <stdlib.h>* statement in the C library, and are referred to in C as *__malloc_t*, *__heap_to_ptr_t*, and *__free_t*.

The ALGOL program includes the XFER procedure, which accepts a string parameter and writes it to the file TEST by making appropriate calls on the C library. First, XFER invokes the MALLOC procedure, which allocates memory space for the string. MALLOC returns an integer, CPTR, which indicates the position of the string in memory.

CPTR can be used as a pointer only within the C library itself. To write data into the memory area allocated by MALLOC, the ALGOL program must first assign an ALGOL-style pointer to that memory location. The program does this with the call on HEAPTOPTR. The ALGOL program then uses the REPLACE statement to write the string to the area allocated for it.

The ALGOL program then invokes the WRITELINE procedure, passing CPTR as a parameter. The WRITELINE procedure in the C library uses CPTR to locate the string that is to be written to the file.

C User Program Passing Array to ALGOL Library

The following examples illustrate the C syntax for calling a library. The examples also illustrate the ability to pass arrays between C and ALGOL.

The following is a file named FOO, which is specified by a #include statement in the C user program. This file contains the import declaration for a procedure called LC.

```
extern "ALGOL" void LC (char*, char (&)[], int, int&, __heap_t,
                        __errno_t);
```

In the FOO file, the use of the string "ALGOL" has the following effects on the declaration:

- The return type of void causes the declaration to be interpreted as an untyped procedure rather than an integer procedure.
- Normally, C passes all parameters by value. The string "ALGOL" identifies LC as a non-C procedure, thereby allowing parameters to be passed by reference. Call-by-reference parameters are denoted by the ampersand (&) operator. For example, *int&* matches a call-by-reference integer, and *char (&)[]* matches a call-by-reference unbounded EBCDIC array.
- The string "ALGOL" also allows some hidden parameter types to be included in the import procedure declaration. In FOO, the hidden parameters are *__heap_t* and *__errno_t*. These parameters are supplied by the compiler, and are not mentioned in the statement in the C program that invokes the imported procedure. The *__heap_t* parameter passes the C program's heap as an EBCDIC array with 0 lower bound. The *__errno_t* parameter passes the predeclared global variable *errno* as a call-by-name integer.

Notice that the import procedure name, LC, is given in all uppercase. This is because many languages, including ALGOL, do not allow library objects to be exported with names containing lowercase letters.

The following is the C user program FOO/C.

```
#include <stdio.h>
#include "foo" (bytitle="OBJECT/FOO/A", intname="FOO")

main (int argc, char *argv []) {
   char buf [10];
   int  i, len;

   for (i=1; i<argc; i++) {
      errno = 0;
      LC (argv [i], buf, sizeof (buf), len);
      if (errno != 0) {
         printf ("string too long: %d > %d\n", len-1, sizeof (buf)-1);
         errno = 0;
      } else {
         printf ("(%2d) \"%s\" -> \"%s\"\n", len-1, argv [i], buf);
      }
   }
}
```

The *#include "foo"* line in this user program serves as the library declaration. This program calls the imported procedure LC to convert some text to lowercase letters. Notice that the invocation of LC has only four parameters supplied. The compiler automatically supplies parameters for *__heap_t* and *__errno_t* as described previously.

After invoking LC, the program uses the variable *errno* to read and update the error value returned by the library in the *__errno_t* parameter. Notice that the C program initializes *errno* to 0 before calling LC. The library functions responsible for assigning *errno* only modify the *errno* value if an error occurs. These functions do not automatically assign a value of 0 to *errno* when a valid result occurs, as the 0 would overwrite any value left to record a previous error.

The following is the ALGOL library FOO/A, which is used by the C program FOO/C.

```
BEGIN
   PROCEDURE LC (PTR, BUF, BUFSIZE, LEN, HEAP, ERRNO);
      VALUE PTR, BUFSIZE;
      INTEGER PTR, BUFSIZE;
      EBCDIC ARRAY BUF [*], HEAP[Ø];
      INTEGER LEN, ERRNO;
    BEGIN
      DEFINE MAX_LEN = 65536 #;
      POINTER P;
      INTEGER L;
      TRANSLATETABLE DOWNCASE(EBCDIC TO EBCDIC,
         "ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO "abcdefghijklmnopqrstuvwxyz");
      SCAN P:HEAP [PTR] FOR L:MAX_LEN UNTIL = 48"ØØ";
      LEN := MAX_LEN - L + 1; % +1 for trailing null
      IF LEN > BUFSIZE THEN BEGIN
         ERRNO := 1;
      END ELSE BEGIN
          REPLACE BUF [Ø] BY HEAP [PTR] FOR LEN WITH DOWNCASE;
      END IF;
    END;

    EXPORT LC;
    FREEZE(TEMPORARY);
END.
```

In this example, the BUF parameter is specified as unbounded in order to match the *char (&) []* parameter in the FOO file. The HEAP parameter is specified with a lower bound of 0 because C passes the *__heap_t* parameter this way.

Note that the PTR parameter is declared as an integer. The program is able to use PTR as a pointer by including it in the pointer expression HEAP [PTR]. This technique works reliably only if the MEMORY_MODEL compiler control option in the C program specifies a one-dimensional memory model. If the C program had specified a two-dimensional memory model, this ALGOL program would have to use the HEAPTOPTR function to convert the integer value to a pointer.

The SCAN statement and the LEN assignment statement both rely on the fact that C terminates each string with a null character.

As noted in the description of the C program, this procedure assigns a value to ERRNO only if an error is encountered.

C User Program Passing File to ALGOL Library

The following is an ALGOL library called FOO/FILE/A:

```
BEGIN
   INTEGER PROCEDURE STATIONNAME (F, MALLOC, COPYTOPTR);
      FILE F;
      INTEGER PROCEDURE MALLOC (SIZE);
         VALUE SIZE;
         INTEGER SIZE;
         FORMAL;
      PROCEDURE COPYTOPTR (LEN, BUF, OFF, PTR);
         VALUE LEN, OFF, PTR;
         INTEGER LEN, OFF, PTR; EBCDIC ARRAY BUF [*];
         FORMAL;
   BEGIN
      EBCDIC ARRAY T [Ø:3ØØ];
      POINTER P;
      INTEGER PTR, LEN;

      REPLACE P:T BY F(1).STATIONNAME;
      IF F.ATTERR THEN BEGIN
         REPLACE P:T BY "<attribute error>.";
      END IF;
      REPLACE P-1 BY 48"ØØ";
      LEN := OFFSET (P);
      PTR := MALLOC (LEN);
      COPYTOPTR (LEN, T, Ø, PTR);
      STATIONNAME := PTR;
   END;
   EXPORT STATIONNAME;
   FREEZE (TEMPORARY);
END.
```

FOO/FILE/A exports a single procedure called STATIONNAME. The STATIONNAME procedure accepts three parameters: a remote file called F, the procedure MALLOC, and the procedure COPYTOPTR. The STATIONNAME procedure reads the STATIONNAME attribute of the remote file into array T. The STATIONNAME procedure uses the MALLOC function to allocate space in the C user program heap. The STATIONNAME procedure then uses the COPYTOPTR function to copy the contents of array T into the C user program heap. The STATIONNAME procedure return value stores the length of the STATIONNAME file attribute value.

The following is the file FOO/FILE/H, which is a header file used by the C user program to declare the imported procedure STATIONNAME:

```
extern "ALGOL" char *STATIONNAME (__file_t, __malloc_t,
                                  __copy_to_ptr_t);
```

Of the items in this header file, *__file_t* corresponds to parameter F in the ALGOL library; *__malloc_t* corresponds to MALLOC; and *__copy_to_ptr_t* corresponds to COPYTOPTR.

The following is the C user program, FOO/FILE/C:

```
#include <stdio.h>
#include "foo.file.h" (bytitle="OBJECT/FOO/FILE/A", intname="LIB")

main () {
   printf ("this printf implicitly opens stdout.\n");
   printf ("title=\"%s\"\n", STATIONNAME (stdout->_file_no));
}
```

The first *printf* statement in the C user program implicitly opens the file *stdout* as a remote file. The second *printf* statement implicitly invokes the STATIONNAME procedure and displays the STATIONNAME file attribute value stored in the C user program heap.

Note that the STATIONNAME invocation does not mention the *__malloc_t* and *__copy_to_ptr_t* parameters, because these are automatically passed by the C compiler. However, the STATIONNAME invocation explicitly passes *__file_t* a parameter of type *int*, which the C compiler changes into a parameter of type file, as required by the ALGOL library. The *int* value that is passed should always be extracted from a *FILE** pointer, as shown by the *->_file_no* clause in this example.

COBOL(68) Library: OBJECT/SAMPLE1

The following COBOL(68) library compiled as OBJECT/SAMPLE1 is referenced in various examples in this section. The procedure exported by this library is named PROCEDUREDIVISION. This library is PERMANENT by default.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 PARAM PIC 9(11) COMP-1 REF.
PROCEDURE DIVISION USING PARAM.
P1.
     DISPLAY "I AM SAMPLE1".
     DISPLAY "MY PARAMETER IS " PARAM.
     EXIT PROGRAM.
```

COBOL(68) Library: OBJECT/SAMPLE2

The following COBOL(68) library compiled as OBJECT/SAMPLE2 is referenced in various examples in this section. This library is TEMPORARY; therefore, when it is no longer in use it unfreezes and resumes running as a regular program. The procedure exported by this library is named ENTRYPOINT.

```
$ SET TEMPORARY
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ENTRYPOINT.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77 PARAM PIC 9(11) COMP REF.
 PROCEDURE DIVISION USING PARAM.
 P1.
      DISPLAY "I AM SAMPLE2".
      DISPLAY "MY PARAMETER IS " PARAM.
      EXIT PROGRAM.
```

COBOL74 Library: OBJECT/SAMPLE4

The following COBOL74 library compiled as OBJECT/SAMPLE4 is referenced in various examples in this section. The entry point to this library is named PROCEDUREDIVISION. This library is PERMANENT by default.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 PARAM PIC 9(11) BINARY.
PROCEDURE DIVISION USING PARAM.
P1.
     DISPLAY "I AM SAMPLE4".
     DISPLAY "MY PARAMETER IS " PARAM.
     EXIT PROGRAM.
```

COBOL74 Library: OBJECT/SAMPLE5

The following COBOL74 library is referenced in various examples in this section. This library is TEMPORARY; therefore, when it is no longer in use it unfreezes and resumes running as a regular program. The FEDLEVEL option is set to 5 to allow the PROGRAM-ID to be used as the entry point name.

```
$ SET FEDLEVEL = 5
$ SET TEMPORARY
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ENTRYPOINT.
 ENVIRONMENT DIVISION.
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77 PARAM PIC 9(11) COMP.
PROCEDURE DIVISION USING INTEGER (PARAM).
P1.
     DISPLAY "I AM SAMPLE5".
     DISPLAY "MY PARAMETER IS " PARAM.
     EXIT PROGRAM.
```

COBOL(68) User Program

The following program invokes various COBOL(68) and ALGOL libraries described in this section:

```
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  PARAM1     PIC 9(11) COMP.
 77  PARAM2     PIC 9(11) COMP-1.
 77  RETURNVAL1 PIC 9(11) COMP.
 77  RETURNVAL2 PIC 9(11) DISPLAY.
 Ø1  TOARRAY    COMP WITH LOWER-BOUNDS.
     Ø3 ELEMENT PIC 9(6) COMP OCCURS 13.
 77  WH         PIC 9(11) COMP.
 PROCEDURE DIVISION.
 P1.

* CALL COBOL(68) LIBRARY NAMED "OBJECT/SAMPLE1"

     CALL "PROCEDUREDIVISION OF OBJECT/SAMPLE1" USING PARAM1.

* CALL COBOL(68) LIBRARY NAMED "OBJECT/SAMPLE1" USING TITLE ATTRIBUTE

     CHANGE ATTRIBUTE TITLE OF "OBJECT/SAMPLE3" TO
          "OBJECT/SAMPLE1.".
     CALL "PROCEDUREDIVISION IN OBJECT/SAMPLE3" USING PARAM2

* CALL COBOL(68) LIBRARY NAMED "OBJECT/SAMPLE2" WHOSE ENTRYPOINT IS
* NAMED ENTRYPOINT

     CALL "ENTRYPOINT OF OBJECT/SAMPLE2" USING PARAM1.

* CALL COBOL(68) LIBRARY NAMED "OBJECT/SAMPLE1"
* USING ANSI74 IPC SYNTAX; CANCEL THAT COBOL(68) LIBRARY

     CALL "OBJECT/SAMPLE1" USING PARAM2.
     CANCEL "OBJECT/SAMPLE1".

* CALL DIRECT ALGOL LIBRARY
* INTERNAL NAME IS "INTLIB"; TITLE IS "OBJECT/SAMPLE/LIBRARY"
```

```
      CHANGE ATTRIBUTE TITLE OF "INTLIB"
         TO "OBJECT/SAMPLE/LIBRARY.".
      CALL "FACT OF INTLIB"
          USING PARAM1 GIVING RETURNVAL1.

* CALL DYNAMIC ALGOL LIBRARY
* TITLE IS "OBJECT/SAMPLE/DYNAMICLIB"
* SELECTION PROCEDURE PARAMETER IS "WITH TIME"

      CHANGE ATTRIBUTE LIBPARAMETER OF "OBJECT/SAMPLE/DYNAMICLIB"
          TO "WITH NAME".
      CALL "DAYTIME IN OBJECT/SAMPLE/DYNAMICLIB"
         USING TOARRAY, WH.

      STOP RUN.
```

COBOL74 User Program

The following COBOL74 program uses various ALGOL and COBOL74 libraries described in this section:

```
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  PARAM1     PIC 9(11) BINARY.
 77  PARAM2     PIC 9(11) COMP.
 77  RETURNVAL1 PIC 9(11) BINARY.
 77  RETURNVAL2 PIC 9(11) COMP.
 Ø1  TOARRAY    BINARY WITH LOWER-BOUNDS.
     Ø3 ELEMENT PIC 9(6)  BINARY OCCURS 13.
 77  WH         PIC 9(11) BINARY.
 PROCEDURE DIVISION.
 P1.

* CALL COBOL74 LIBRARY NAMED "OBJECT/SAMPLE4"

      CALL "PROCEDUREDIVISION OF OBJECT/SAMPLE4" USING PARAM1.

* CALL COBOL74  LIBRARY NAMED "OBJECT/SAMPLE4" USING TITLE ATTRIBUTE

      CHANGE ATTRIBUTE TITLE OF "OBJECT/SAMPLE6" TO
           "OBJECT/SAMPLE4.".
      CALL "PROCEDUREDIVISION IN OBJECT/SAMPLE6"
           USING INTEGER (PARAM2).

* CALL COBOL74 LIBRARY NAMED "OBJECT/SAMPLE5" WHOSE ENTRYPOINT IS
* NAMED ENTRYPOINT

      CALL "ENTRYPOINT OF OBJECT/SAMPLE5" USING PARAM1.
```

```
* CALL COBOL74 LIBRARY NAMED "OBJECT/SAMPLE4"
* USING ANSI74 IPC SYNTAX; CANCEL THAT COBOL74 LIBRARY

     CALL "OBJECT/SAMPLE4" USING INTEGER (PARAM2).
     CANCEL "OBJECT/SAMPLE4".

* CALL DIRECT ALGOL LIBRARY
* INTERNAL NAME IS "INTLIB"; TITLE IS "OBJECT/SAMPLE/LIBRARY"

     CHANGE ATTRIBUTE TITLE OF "INTLIB"
        TO "OBJECT/SAMPLE/LIBRARY".
     CALL "FACT OF INTLIB"
        USING PARAM1 GIVING RETURNVAL1.

* CALL DYNAMIC ALGOL LIBRARY
* TITLE IS "OBJECT/SAMPLE/DYNAMICLIB"
* SELECTION PROCEDURE PARAMETER IS "WITH TIME"

     CHANGE ATTRIBUTE LIBPARAMETER OF "OBJECT/SAMPLE/DYNAMICLIB"
         TO "WITH NAME".
     CALL "DAYTIME IN OBJECT/SAMPLE/DYNAMICLIB"
        USING TOARRAY, WH.

     STOP RUN.
```

ALGOL User Program #3

The following ALGOL program uses various COBOL(68) libraries described in this section:

```
BEGIN

INTEGER PARAM;

LIBRARY COBOLLIB (TITLE = "OBJECT/SAMPLE1.");
LIBRARY OTHERLIB (TITLE = "OBJECT/SAMPLE5.");

PROCEDURE PROCEDUREDIVISION (N);
   VALUE N;
   INTEGER N;
   LIBRARY COBOLLIB;

PROCEDURE ENTRYPOINT (N);
   VALUE N;
   INTEGER N;
   LIBRARY OTHERLIB;

   PARAM := 12345;
   PROCEDUREDIVISION (PARAM);
   DISPLAY (STRING(PARAM,*));
```

```
    PARAM := 12345;
    ENTRYPOINT (PARAM);
    DISPLAY (STRING(PARAM,*));

END.
```

COBOL85 Libraries and User Program

The following is a COBOL85 library named TASKM/COBOL85/LIBRARY:

```
000100$ RESET LIST SET ERRORLIST LINEINFO
000200  IDENTIFICATION DIVISION.
000300  PROGRAM-ID. EXPLICIT-LIBRARY LIBRARY.
000400  ENVIRONMENT DIVISION.
000500  INPUT-OUTPUT SECTION.
000600  FILE-CONTROL.
000700       SELECT FL-1 ASSIGN TO DISK.
000800  DATA DIVISION.
000900  FILE SECTION.
001000  FD FL-1 GLOBAL.
001100  01 FL-1-REC PIC X(80) GLOBAL.
001200  WORKING-STORAGE SECTION.
001300  PROGRAM-LIBRARY SECTION.
001400  LB EXPLICIT-LIBRARY EXPORT
001500                      ATTRIBUTE SHARING IS SHAREDBYRUNUNIT.
001600  ENTRY PROCEDURE USERCODE.
001700  ENTRY PROCEDURE WRITER.
001800  PROCEDURE DIVISION.
001900  P-1.
002000      OPEN OUTPUT FL-1.
002100      CALL "DISPLAYER".
002200      CALL SYSTEM FREEZE TEMPORARY.
002300      CLOSE       FL-1 SAVE.
002400      EXIT PROGRAM.
002500  IDENTIFICATION DIVISION.
002600  PROGRAM-ID. WRITER.
002700  DATA DIVISION.
002800  PROCEDURE DIVISION.
002900  P-1.
003000       MOVE "DATA WRITTEN FROM LIBRARY" TO FL-1-REC.
003100       WRITE FL-1-REC.
003200       EXIT PROGRAM.
003300  END PROGRAM WRITER.
003400  IDENTIFICATION DIVISION.
003500  PROGRAM-ID. DISPLAYER.
003600  PROCEDURE DIVISION.
003700  P-1.
003800       DISPLAY "THE EXPLICIT LIBRARY HAS BEEN ENTERED."
003900       EXIT PROGRAM.
004000  END PROGRAM DISPLAYER.
004100  IDENTIFICATION DIVISION.
```

```
004200  PROGRAM-ID. USERCODE.
004300  DATA DIVISION.
004400  WORKING-STORAGE SECTION.
004500  01 REC-1.
004600     02 BUFFER      PIC X(20).
004700     02 USERNAME    PIC X(20).
004800     02 DIRECTORIES PIC 9(4) COMP.
004900  01 MAX-DIRECTORIES PIC 9(4) COMP VALUE 17 GLOBAL.
005000  LINKAGE SECTION.
005100  01 STR       PIC X(80).
005200  01 USER-CODE PIC X(20).
005300  PROCEDURE DIVISION USING STR USER-CODE.
005400  P-1.
005500       UNSTRING STR DELIMITED BY "/" OR "(" OR ")"
005600       INTO BUFFER USERNAME TALLYING IN DIRECTORIES.
005700       MOVE USERNAME TO USER-CODE.
005800       IF DIRECTORIES IS GREATER THAN MAX-DIRECTORIES
005900         CALL "ERROR-MESSAGE" USING DIRECTORIES.
006000       EXIT PROGRAM.
006100  IDENTIFICATION DIVISION.
006200  PROGRAM-ID. ERROR-MESSAGE.
006300  DATA DIVISION.
006400  WORKING-STORAGE SECTION.
006500  77 EXTRA-DIRECTORIES PIC 9(4) COMP.
006600  LINKAGE SECTION.
006700  77 TOTAL-DIRECTORIES PIC 9(4) COMP.
006800  PROCEDURE DIVISION USING TOTAL-DIRECTORIES.
006900  P-1.
007000       SUBTRACT MAX-DIRECTORIES FROM   TOTAL-DIRECTORIES
007100                                GIVING EXTRA-DIRECTORIES.
007200       DISPLAY "THERE WERE " EXTRA-DIRECTORIES
007300                             "EXTRA DIRECTORIES".
007400       EXIT PROGRAM.
007500  END PROGRAM ERROR-MESSAGE.
007600  END PROGRAM USERCODE.
007700  END PROGRAM EXPLICIT-LIBRARY.
```

The program TASKM/COBOL85/LIBRARY illustrates several library features that distinguish COBOL85 libraries from libraries in earlier COBOL implementations. These features include:

- An explicit FREEZE statement at line 2200, which includes the freeze duration option of TEMPORARY.
- The PROGRAM-LIBRARY SECTION, which includes an explicit export declaration at lines 1400-1700. The declaration specifies a sharing option of SHAREDBYRUNUNIT and lists USERCODE and WRITER as the names of nested programs to be exported.
- Three nested programs, including the two specified in the export declaration: USERCODE, at lines 4100-7600; and WRITER, at lines 2500-3300.

- Local variables. The exported nested program USERCODE includes declarations of the data items REC-1 at line 4500 and MAX-DIRECTORIES at 4900. Note that these local variables are reinitialized each time the USERCODE nested program is invoked. You can cause the values of these variables to be preserved by adding an *IS INITIAL* clause to the PROGRAM-ID paragraph at line 4200.

The following is another COBOL85 library called TASKM/COBOL85/PROCEDURE:

```
000100$RESET LIST SET ERRORLIST LINEINFO
000200$SHARING = SHAREDBYRUNUNIT
000300$LIBRARYPROG
000400  IDENTIFICATION DIVISION.
000500  PROGRAM-ID. IMPLICIT-LIBRARY.
000600  ENVIRONMENT DIVISION.
000700  DATA DIVISION.
000800  WORKING-STORAGE SECTION.
000900  77 SWAP PIC X(10).
001000  LINKAGE SECTION.
001100  01 A-REC.
001200     02 FLD-1 PIC X(10).
001300     02 FLD-2 PIC X(10).
001400  PROCEDURE DIVISION USING A-REC.
001500  P-1.
001600        MOVE FLD-1 TO SWAP.
001700        MOVE FLD-2 TO FLD-1.
001800        MOVE SWAP  TO FLD-2.
001900        EXIT PROGRAM.
```

The library TASKM/COBOL85/PROCEDURE is designed to function much as a COBOL74 library. There is no explicit export declaration or freeze statement, and the entire PROCEDURE DIVISION is exported. However, unlike COBOL74, COBOL85 requires the LIBRARYPROG compiler option to be set in order to indicate that the program will function as a library.

The following is a COBOL85 user program called TASKM/COBOL85/PROGRAM. This program calls the two COBOL85 libraries described previously:

```
000100$ RESET LIST SET ERRORLIST LINEINFO
000200  IDENTIFICATION DIVISION.
000300  PROGRAM-ID. DRIVER.
000400  ENVIRONMENT DIVISION.
000900  DATA DIVISION.
001500  WORKING-STORAGE SECTION.
001600  01 THE-TITLE PIC X(80).
001700  01 USERCODES.
001800     02 USERCODE-1 PIC X(20).
001900     02 USERCODE-2 PIC X(20).
002000  LOCAL-STORAGE SECTION.
002100  LD PARAM-1.
002200  01 P-REC-1.
002300     02 A-FLD-1 PIC X(10).
002400     02 A-FLD-2 PIC X(10).
```

```
002500  LD PARAM-2.
002600  01 P-REC-2 PIC X(80).
002700  01 P-REC-3 PIC X(20).
003100  PROGRAM-LIBRARY SECTION.
003200  LB LIB-ONE IMPORT
003300             ATTRIBUTE FUNCTIONNAME IS "THELIBRARY"
003400                       LIBACCESS    IS BYTITLE.
003500  ENTRY PROCEDURE WRITER.
003600  ENTRY PROCEDURE USERCODE WITH  PARAM-2
003700                           USING P-REC-2 P-REC-3.
003800  LB LIB-TWO IMPORT
003900             ATTRIBUTE TITLE IS "OBJECT/TASKM/COBOL85/PROCEDURE".
004000  ENTRY PROCEDURE PROCEDUREDIVISION WITH  PARAM-1
004100                                    USING P-REC-1.
004500  PROCEDURE DIVISION.
004600  P-1.
004700      CHANGE ATTRIBUTE TITLE OF LIB-ONE
004800                             TO "OBJECT/TASKM/COBOL85/LIBRARY".
005000      CALL WRITER.
005100      CALL WRITER OF LIB-ONE.
005200      MOVE ATTRIBUTE TITLE OF LIB-TWO TO THE-TITLE.
005300      CALL USERCODE
005400           USING THE-TITLE USERCODE-1.
005500      MOVE ATTRIBUTE TITLE OF LIB-ONE TO THE-TITLE.
005600      CALL "USERCODE IN OBJECT/TASKM/COBOL85/LIBRARY"
005700           USING THE-TITLE USERCODE-2.
005800      CALL PROCEDUREDIVISION OF LIB-TWO
005900           USING USERCODES.
006000      CALL "OBJECT/TASKM/COBOL85/PROCEDURE"
006100           USING USERCODES.
006200      DISPLAY USERCODES.
006300      STOP RUN.
```

The program TASKM/COBOL85/PROGRAM illustrates the explicit library declarations and import declarations provided by COBOL85. Thus, the PROGRAM-LIBRARY SECTION at lines 3100-4400 includes declarations of the libraries LIB-ONE and LIB-TWO. These library declarations include library attribute assignments as well as import declarations for WRITER, USERCODE, and PROCEDUREDIVISION.

TASKM/COBOL85/PROGRAM includes examples of the following types of CALL statements:

- A CALL statement that invokes an explicitly declared import object. The statements at lines 5000 and 5300-5400 are examples that invoke import objects declared in the PROGRAM-LIBRARY SECTION.
- A CALL statement that invokes an explicitly declared import object in an explicitly specified library. The statements at lines 5100 and 5800 refer to the LIB-ONE and LIB-TWO declarations in the PROGRAM-LIBRARY SECTION.

- A CALL statement that uses a string literal to specify the library object code file title. The statement at line 6000 invokes the library TASKM/COBOL85/PROCEDURE. The CALL statement does not need to specify the name of a particular import object, as the library in question exports only the PROCEDUREDIVISION.
- A CALL statement that uses a string literal to specify both the library object code file title and the import object name. The statement at line 5600 invokes the object USERCODE in the library OBJECT/TASKM/COBOL85/LIBRARY.

FORTRAN Library and User Program

The following FORTRAN program, compiled as MATHINTRINSICS, creates a library:

```
$ SHARING = PRIVATE
      BLOCK GLOBALS
          FILE 6(KIND=PRINTER)
          EXPORT SINE="SIN",COSINE
      END
      REAL FUNCTION SINE(X)
C*          PERFORM SINE CALCULATION...
          SINE=SIN(X)
      END
      REAL FUNCTION COSINE(X)
C*          PERFORM COSINE CALCULATION...
          COSINE=COS(X)
      END
C*    MAIN PROGRAM
          WRITE(6,/)SINE(X),COSINE(X)
          CALL FREEZE(TEMPORARY)
      END
```

The following FORTRAN program invokes the FORTRAN library MATHINTRINSICS:

```
 BLOCK GLOBALS
      FILE 5(KIND=REMOTE)
      FILE 6(KIND=PRINTER)
      LIBRARY LIB1(TITLE="MATHINTRINSICS",
*                  INTNAME="MATHINTRINSICS")
 END
 REAL FUNCTION SIN(X)
      REAL X
      IN LIBRARY LIB1
 END
 REAL FUNCTION COS(X)
      REAL X
      IN LIBRARY LIB1(ACTUALNAME="COSINE")
 END
C*  MAIN PROGRAM
      READ(5,/)X
      WRITE(6,/)SIN(X),COS(X)
 END
```

FORTRAN77 Library and User Program

The following examples present the FORTRAN77 versions of the library program and user program previously described under "FORTRAN Library and User Program" in this section.

The following FORTRAN77 program, compiled as MATHINTRINSICS, creates a library:

```
      BLOCK GLOBALS
FILE  6(KIND="PRINTER")
         EXPORT (SINE="SIN",COSINE)
      END
      REAL FUNCTION SINE(X)
C*          PERFORM SINE CALCULATION ...
            SINE=SIN(X)
      END
      REAL FUNCTION COSINE(X)
C*          PERFORM COSINE CALCULATION ...
            COSINE=COS(X)
      END
C*    MAIN PROGRAM
            WRITE(6,*)SINE(X),COSINE(X)
            CALL FREEZE("TEMPORARY")
      END
```

The following FORTRAN77 program invokes the FORTRAN77 library MATHINTRINSICS previously described:

```
      BLOCK GLOBALS
FILE  5(KIND="REMOTE")
FILE  6(KIND="PRINTER")
            LIBRARY LIB1(TITLE="MATHINTRINSICS",
*                        INTNAME="MATHINTRINSICS")
      END
      REAL FUNCTION SIN(X)
            REAL X
            IN LIBRARY LIB1
      END
      REAL FUNCTION COS(X)
            REAL X
            IN LIBRARY LIB1(ACTUALNAME="COSINE")
      END
      PROGRAM MAIN_PROGRAM
            EXTERNAL SIN, COS
            READ(5,*)X
            WRITE(6,*)SIN(X),COS(X)
      END
```

Pascal Library

The following is an example of a Pascal library.

```
library lib; usage(sharing = sharedbyall);
   interface
      type vect = array [1..3Ø] of integer;
      procedure sum (vector1, vector2 : vect);
      function fact (n : integer) : integer;
      function sin (r : real) : real;
   end;

   library mylib (title = 'OBJECT/ARITHLIB');
   procedure sum;
      var i : integer;
      begin
      for i := 1 to 3Ø do
         vector[i] := vector1[i] + vector2[i];
      end;
   function sin; mylib;
   function fact;
      begin
      if n < 1 then
         fact := 1
      else
         fact := n * fact(n-1);
      end;

   begin
   freeze;
   end.
```

This library exports the procedure *sum* and the functions *fact* and *sin*, all of which appear in the interface part. *Sum* and *fact* are completely declared in the library block. *Sin* is imported from another library.

When an ALGOL library procedure uses an EBCDIC array that was passed from a Pascal user program, the library procedure should specify the starting index of the array. This precaution is necessary because ALGOL processes the following statements slightly differently:

```
REPLACE A BY "HI";
REPLACE A[Ø] BY "HI";
```

The following statement causes the value "HI" to be correctly assigned to element 0 of array A:

```
REPLACE A[Ø] BY "HI";
```

The following ALGOL library illustrates the use of these types of statements:

```
 $SHARING = PRIVATE
BEGIN

PROCEDURE ALGOLDISPLAY( S1, S2 );
  EBCDIC ARRAY S1[*], S2[*]
  BEGIN
  REPLACE S1[Ø] BY "HI"; % THESE TWO STATEMENTS ARE CORRECT
  DISPLAY( S1[Ø] );

  REPLACE S2 BY "HI"      % THESE TWO STATEMENTS ARE ALLOWED,
  DISPLAY( S2);           % BUT WILL NOT GIVE THE EXPECTED RESULTS.

  END;

  EXPORT ALGOLDISPLAY;
  FREEZE( TEMPORARY );
END.
```

The following example illustrates how an ALGOL user program should pass an unbounded array to a Pascal library. The ALGOL user program declares the imported procedure PASCPROC with an unbounded array parameter A. The ALGOL user program declares another array, called MYARRAY, for use as the actual parameter. MYARRAY is declared with a lower bound of 0 because a Pascal library always assumes the first array element to be at index 0.

```
BEGIN
   LIBRARY MYLIB;
   PROCEDURE PASCPROC(A);
      EBCDIC ARRAY A[*];
      LIBRARY MYLIB;
   EBCDIC ARRAY MYARRAY [Ø:5];

   REPLACE MYARRAY BY "DATA";
   PASCPROC(MYARRAY);

END.
```

The following Pascal program invokes the procedure ALGOLDISPLAY in an ALGOL library called OBJECT/ALGOLLIB.

```
program p;
  type
    stringbyte = packed array [1..20] of char;

  library mylib( title= 'OBJECT/ALGOLLIB' );
  procedure algoldisplay( s1, s2 : stringtype ); mylib;

  var
    s1, s2 : stringtype;

  begin
  s1:= 'abcdefghijklmnopqrst';
  s2:= '12345678901234567890';

  algoldisplay( s1, s2 );
  end.
```

PL/I Library and User Program

The following is an example of a PL/I library called OBJECT/PL/LIB/CALLEE:

```
CALLEE: PROC OPTIONS(MAIN, EXPORT(DIST));
DIST: PROC 8X1,Y1,X2,Y2) RETURNS (FLOAT);
    DCL (X1,Y1,X2,Y2) FLOAT;
    RETURN (SQRT ((X2-X1)**3   + (Y2-Y1)**3  ));
   END;
FREEZE OPTIONS (TEMPORARY);
END CALLEE;
```

The following PL/I program invokes the PL/I library OBJECT/PL/LIB/CALLEE. This program uses the library procedure DIST to calculate the distance between two points, point 1 and point 2. The library procedure DIST uses the intrinsic SQRT function and the Pythagorean theorem to calculate the distance between point 1 and point 2, which are defined in terms of their Cartesian coordinates.

```
CALLER:  PROC OPTIONS(MAIN);
     DCL LIB LIBRARY (TITLE='OBJECT/PL/LIB/CALLEE');
     DCL DIST ENTRY (FLOAT,FLOAT,FLOAT,FLOAT)OPTIONS(LIBRARY=LIB);
     DCL (X1,Y1,X2,Y2,D) FLOAT;


/* (X,Y) COORDINATES OF POINT 1 */
     X1 = 1;
     Y1 = 1;
/* (X,Y) COORDINATES OF POINT 2 */
     Y2 = 4;
     Y2 = 4;
```

```
/* CALL DIST FUNCTION IN THE LIBRARY TO RETURN THE DISTANCE FROM
   POINT 1 TO POINT 2 */
     D = DIST(X1,Y1,X2,Y2);
     PUT LIST('DISTANCE ');
     PUT DATA (D);
END CALLER;
```

Section 19
Using Shared Files

Files are relevant to interprocess communication (IPC) in two ways:

- Certain kinds of files are intended specifically for use in IPC, and present unique advantages when compared to other IPC techniques.
- Processes can share the same file, even if the file is not used as a medium for IPC. For example, two processes might have a shared responsibility for updating a single file. Even though the file is not used for IPC, the processes must use IPC techniques to ensure that their updates do not conflict.

This section gives an overview of both of these aspects of files and IPC. For more detailed information on many of the topics discussed in this section, refer to the *A Series I/O Subsystem Programming Guide*.

Sharing Communications Files

IPC files enable processes to communicate with other processes in a manner similar to reading or writing a file on a physical device. This makes IPC files an ideal method for transmitting large quantities of textual information between processes.

The processes that communicate through an IPC file do not have to belong to the same process family. The processes specify various file attributes that allow the system to establish a link between the correct pair of processes.

A Series systems support three types of files for use in IPC: port files, host control (HC) files, and HYPERchannel (HY) files. The following subsections briefly outline the capabilities of each of these types of IPC files.

Using Port Files

Port files enable communication between processes regardless of whether those processes reside on a single host or on separate hosts in a local area network or wide area network. The types of networks that support port files include BNA Version 1, BNA Version 2, A Series Open Systems Interconnection (OSI), and Transmission Control Protocol/Internet Protocol (TCP/IP).

Port files have been implemented to provide the applications programmer with a single interface that can be used to communicate across all the types of multihost networks supported by A Series systems. However, the programmer can choose among any of several different port services with slightly varying functionality. Each type of network supports one or more of these port services. A port service that is available on most networks is BASICSERVICE. Port file applications that use BASICSERVICE can run with little or no modification on BNA Version 1, BNA Version 2, and OSI networks.

Port files can be used by programs written in any of the following languages: ALGOL, COBOL(68), COBOL74, COBOL85, FORTRAN, FORTRAN77, Pascal, PL/I, and RPG. The programs that communicate with each other through a port file do not have to be written in the same language.

A port file consists of one or more distinct communication paths, called *subfiles*, that are grouped under a common name. Individual subfiles are identified by way of the port file name and a number called the *subfile index*. You can specify the number of subfiles associated with a port file by using the MAXSUBFILES file attribute. On systems running BNA Version 2, you can establish differing priorities for the subfiles through the use of the DIALOGPRIORITY file attribute.

Before opening a subfile, the application can assign several file attributes that help the system to identify the matching port file. These file attributes include FILENAME, which specifies the name of the port file; MYNAME, which must match the YOURNAME file attribute of the matching port file; and YOURHOST, which specifies the host where the matching process is running. Other attributes can be used to restrict access to processes having specified usercodes.

An application can use any of several OPEN statement options to specify whether the process waits for a matching process to appear or continues execution immediately. If the process continues execution, it can either abandon the open operation or leave the subfile in an offered state, ready for the matching process to link to it.

Each subfile consists of an *input queue*, from which the process reads, and an *output queue*, to which the process writes. Messages are processed through each queue on a first-in, first-out basis, so they always reflect the chronological order in which they were transmitted. The system provides the event-valued file attributes INPUTEVENT and OUTPUTEVENT to inform the process of activity in the input and output queues.

The process can write messages to individual subfiles, or can use a *broadcast write* statement, which sends the same message to all the subfiles in a port file. Similarly, a process can read messages from a specific subfile, or use a *nonselective read* statement, which reads a message from any one of the subfiles with waiting input.

For a complete explanation of how to use port files, refer to the *A Series I/O Subsystem Programming Guide*.

The following subsections provide simple examples of port file programs written in COBOL74 and ALGOL.

COBOL74 Port File Example

The following COBOL74 program declares a port file called MSSR with three subfiles. This program runs on host SFA15CD and opens the port file with MYNAME = MASTER and YOURHOST = SF59D. The program opens the port and broadcasts a message to all subfiles. The program then performs a nonselective operation read to determine which of the remote processes responded first. It sends a message to the remote process that responded first and a different message to the other remote processes. A "USE AFTER ERROR" procedure causes the program to display a

message after any I/O error. Note that this program should be initiated after the three matching processes have been initiated and have attempted to open their subfiles.

```
*COBOL74 READ/WRITE PROGRAM USING BNA OPTIONS.
*THIS PROGRAM RUNS ON HOST SFA15CD.
IDENTIFICATION DIVISION.
PROGRAM-ID. COBOL74-PORTFILE-DEMO2.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MSSR ASSIGN TO PORT
           ACTUAL KEY IS MSSR-KEY
           FILE STATUS IS MSSR-STATUS.
DATA DIVISION.
FILE SECTION.
FD MSSR.
Ø1 MSSR-REC              PIC X(17).
WORKING-STORAGE SECTION.
Ø1 MSSR-WS.
   Ø5 MSSR-STATUS        PIC XX.
   Ø5 MSSR-KEY           PIC 9 COMPUTATIONAL.
PROCEDURE DIVISION.
DECLARATIVES.
IO-ERROR SECTION.
USE AFTER ERROR PROCEDURE ON I-O.
IO-ERROR-HANDLER.
     DISPLAY "I/O ERROR ENCOUNTERED".
     DISPLAY "STATUS IS ", MSSR-STATUS.
END DECLARATIVES.
THE-PROGRAM SECTION.
INITIALIZATION.
     CHANGE ATTRIBUTE MAXSUBFILES OF MSSR TO 3.
     CHANGE ATTRIBUTE MYNAME OF MSSR TO "MASTER.".
*COBOL74 APPLIES THE FOLLOWING ASSIGNMENTS TO ALL SUBFILES.
     CHANGE ATTRIBUTE YOURNAME OF MSSR(Ø) TO "SERVANT.".
     CHANGE ATTRIBUTE YOURHOST OF MSSR(Ø) TO "SF59D.".
OPEN-THE-PORT.
     MOVE Ø TO MSSR-KEY.
     OPEN I-O MSSR.
BROADCAST-THE-MESSAGE.
     MOVE Ø TO MSSR-KEY.
     MOVE "SEND ME A MESSAGE" TO MSSR-REC.
     WRITE MSSR-REC.
GET-A-MESSAGE.
     MOVE Ø TO MSSR-KEY.
     READ MSSR.
     DISPLAY "MESSAGE RECEIVED FROM SUBFILE ", MSSR-KEY.
     DISPLAY MSSR-REC.
SEND-A-MESSAGE.
* THE ACTUAL KEY AT THIS POINT CONTAINS THE SUBFILE
* FROM WHICH THE MESSAGE WAS REMOVED IN THE LAST
* READ STATEMENT.
```

```
           MOVE "YOU WIN" TO MSSR-REC.
           WRITE MSSR-REC.
      CLOSE-THE-WINNING-SUBFILE.
           CLOSE MSSR.
      * READ THE MESSAGES FROM THE LOSING SUBFILES.
      PERFORM GET-A-MESSAGE.
      PERFORM GET-A-MESSAGE.
      BROADCAST-A-MESSAGE-TO-LOSERS.
           MOVE "YOU LOSE" TO MSSR-REC.
           MOVE Ø TO MSSR-KEY.
           WRITE MSSR-REC.
      CLOSE-THE-LOSERS.
           CLOSE MSSR.
           STOP RUN.
```

ALGOL Port File Example

The following ALGOL program is designed to communicate with the preceding COBOL74 program. If three instances of this program are initiated, they will each communicate with one of the subfiles declared in the COBOL74 program. This ALGOL program runs on host SF59D and opens a port file called MSSR with a single subfile, and with MYNAME = MASTER and YOURHOST = SFA15CD. The program reads the message broadcast from the COBOL74 program, sends a reply, and then reads another message from the COBOL74 program. The read and write statements are handled by the procedures READIT and WRITEIT, which use complex wait statements to monitor the status of the subfiles used.

```
% COMPLEMENTARY ALGOL PROGRAM FOR READ/WRITE TO A PORT FILE.
% THIS PROGRAM IS TO BE EXECUTED ON HOST "SF59D".
BEGIN
FILE MSSR (KIND = PORT, MAXSUBFILES = 1, MAXRECSIZE = 17,
           MYUSE = IO, UNITS = CHARACTERS, MYNAME = "SERVANT.",
           YOURNAME="MASTER.", YOURHOST = "SFA15CD.");
EBCDIC ARRAY INMSS[1:17],OUTMSS[1:14];
BOOLEAN RSLT;
INTEGER INT;
PROCEDURE ABORT(REASON);
STRING REASON;
BEGIN
  DISPLAY (REASON);
  MYSELF.STATUS := VALUE(TERMINATED);
END;
PROCEDURE READIT;
BEGIN
   INTEGER EVNT;
   EVNT := WAIT ((120),MSSR.CHANGEEVENT,MSSR.INPUTEVENT);
   CASE EVNT OF
   BEGIN
      1: ABORT ("TIME LIMIT ELAPSED - NO MESSAGE RECEIVED");
      2: CASE MSSR.FILESTATE OF
         BEGIN
```

```
            VALUE(BLOCKED): VALUE(DEACTIVATIONPENDING):
            VALUE(OPENED): VALUE(SHUTTINGDOWN):
                IF HAPPENED (MSSR.INPUTEVENT) THEN
                   RSLT := READ (MSSR,17,INMSS)
                ELSE ABORT ("NO MESSAGE RECEIVED");
            ELSE : ABORT ("BAD FILESTATE");
         END;
      3: RSLT := READ (MSSR,17,INMSS);
   END;
   IF RSLT THEN ABORT ("MESSAGE MISCARRIED")
          ELSE DISPLAY (INMSS);
END READIT;

PROCEDURE WRITEIT;
BEGIN
   INTEGER EVNT;
   EVNT := WAIT ((120),MSSR.CHANGEEVENT,MSSR.OUTPUTEVENT);
   CASE EVNT OF
   BEGIN
      1: ABORT ("TIME LIMIT ELAPSED - NO ROOM TO WRITE");
      2: CASE MSSR.FILESTATE OF
         BEGIN
            VALUE(BLOCKED): VALUE(OFFERED): WRITEIT;
            VALUE(OPENED): IF HAPPENED (MSSR.OUTPUTEVENT) THEN
                           RSLT := WRITE(MSSR,14,OUTMSS);
            ELSE : ABORT ("BAD FILESTATE");
         END;
      3: RSLT := WRITE (MSSR,14,OUTMSS);
   END;
   IF RSLT THEN ABORT ("WRITE MISCARRIED");
END WRITEIT;

IF (INT := OPEN (MSSR)) NEQ 1 THEN
    ABORT("UNABLE TO OPEN SUBFILE");
READIT;
REPLACE OUTMSS BY "REMOTE MESSAGE";
WRITEIT;
READIT;
CLOSE (MSSR);
END.
```

Using Host Control (HC) Files

Host control (HC) files provide a simple, high-speed method for transferring data between processes running on two different hosts in a local area network.

HC file communication takes place through intersystem control (ISC) hardware. Each ISC link consists of an ISC hub, which is connected by cables to a host control data link processor (HCDLP) at each host system.

The ISC link must be dedicated to the exclusive use of a pair of application processes, one on each host. Each process opens a file with a KIND file attribute value of HC and a FILENAME value equal to the ISC hub name. The processes must communicate using direct I/O. Therefore, the application must be written in one of the languages that support HC files and direct I/O. The only languages that support both these features are ALGOL and NEWP.

Compared to port files, HC files have the potential for offering faster data transfer. However, unlike port files, HC files can operate only across a single type of dedicated hardware link, and only in a local area network. Further, HC files lack helpful port file features such as multiple subfiles, subfile matching based on user-supplied names, message rerouting, and data compression.

For additional information, refer to the *A Series I/O Subsystem Programming Guide*.

Using HYPERchannel (HY) Files

HYPERchannel (HY) files are similar to HC files in that HY files allow application processes to communicate in a local area network over dedicated hardware links. In the case of HY files, the links are HYPERchannel coaxial trunks that are connected to an A Series host through an A222 or A223 adapter and a HYPERchannel data link processor (HYDLP). An application declares an HY file by specifying a KIND file attribute value of HY. Applications that communicate over the HYPERchannel link must use direct I/O. Therefore, the applications must be written in one of the languages that support HY files and direct I/O. The only languages that support both these features are ALGOL and NEWP.

Compared to port files, HY files offer much the same advantages and liabilities as HC files. That is, HY files can offer a faster data transfer rate than port files, but lack helpful port file features such as multiple subfiles, subfile matching based on user-supplied names, message rerouting, and data compression.

However, the underlying HYPERchannel hardware gives HY files the following advantages over HC files:

- HYPERchannel interfaces are supported by many non-A Series systems, including 1100 and 2200 series systems as well as IBM® and DEC® systems. Thus, HY files enable you to implement applications that communicate across networks of multivendor systems.
- HYPERchannel links can connect systems spread over a distance of several kilometers. By contrast, the ISC links that support HC files are limited to a few hundred feet in length. Thus, HY files can provide communication across a larger distance than HC files can.

For further information about HY files, including examples of programs that use HY files, refer to the *A Series I/O Subsystem Programming Guide*.

IBM is a registered trademark of International Business Machines Corporation.
DEC is a registered trademark of Digital Equipment Corporation.

Sharing Other Kinds of Files

Port files, HC files, and HY files were designed specifically for use in IPC. A Series systems also support a number of other file types that are associated with permanent storage media, such as disk or tape. These file types are not well suited for use in transmitting information between processes, for two reasons. First, an I/O operation to a peripheral device takes greater elapsed time than operations on port files, which take place largely within main memory. Second, these file types do not provide some of the convenient features of port files, such as separate input and output queues, or event-valued file attributes that notify processes when records have been received.

However, there are cases where it can useful for two or more processes to share the same file, even if the file itself is not being used for IPC. These are cases where several different processes running at the same time are responsible for reading and updating the same file. The file is being used as a permanent storage medium, rather than a method of passing control information between processes.

In this situation, you can use IPC techniques to accomplish two of the goals of file sharing:

- To provide two or more processes with access to the same file
- To regulate timing to prevent these processes from accidentally overwriting each other's changes to the file

For most file types, the only way to achieve these goals is to design the processes so they communicate with the file through the same logical file. The *logical file* is an access structure, created by a file declaration in the program, that exists in system memory. By contrast, the *physical file* is the file that exists on a peripheral storage device, such as a disk or tape drive. Opening a file causes the logical file to be linked to a physical file so that the process can read or write information in the file. The following subsections discuss the concept of the logical file and considerations that arise from sharing logical files.

For disk files, it is also possible for processes to use the same physical file without using the same logical file. Considerations for doing this are discussed under "Using Shared Physical Files" later in this section.

Using Shared Logical Files

Processes can use any of the following methods to share logical files:

- An internal task can use logical files declared globally in the parent program, as discussed in Section 15, "Using Global Objects."
- An initiating process can pass a file as a parameter to a process that it initiates, as discussed in Section 17, "Using Parameters."
- A WFL job can declare a file and use a global file equation to cause a task to use this file in place of one declared in the task itself. An example of global file equation is given under "File Sharing Examples" later in this section.

- A shared library can contain a file declaration that is global to the exported library procedures. You can design the exported procedures to allow user processes to access the file declared globally in the library. Refer to "Providing Global Objects" in Section 18, "Using Libraries."

If all the processes that use the file belong to the same process family, then you can use any of these sharing methods. If the processes belong to different process families, then using shared libraries is the only method that can enable the logical file to be shared.

Specifying the File Location

If the TITLE file attribute of the logical file was not assigned a usercode, then at file-open time the file is searched for under the usercode of the process that declared the file. If the file is not found under that usercode, it is searched for as a nonusercoded file. This search pattern is the same even if the file-open statement is executed by a process different from the process that declared the file.

Similarly, the family where the system searches for the file is affected by the FAMILY task attribute of the process that declares the file, rather than by that of the process that opens the file. If the TITLE file attribute specifies a family, the FAMILY task attribute can specify a substitute family. If the TITLE file attribute does not include a family, the FAMILY task attribute can supply a default family.

Synchronizing Access to a File

The design of the I/O hardware prevents any two I/O operations from accessing the same record at the same time. For example, there is no way for a record to be read when a write operation involving the same record is half completed. The read operation is delayed until the write operation finishes.

However, you can prevent other types of synchronization problems only through careful program design. Note that these synchronization concerns arise only in cases where at least one of the processes writes to the shared file. If all the processes simply read from the file, then the order in which they execute their read operations makes no difference. The following are the basic goals of synchronization:

- For cases where one process writes to a particular record and a second process is to read the updated record, the second process should wait until the record is updated before reading it.
- For cases where two processes read and update the same record, the processes should be prevented from accidentally overwriting each other's updates. Such overwriting can occur if both processes read from the record and then both processes write to the record. The second write operation erases the effects of the first, and information can therefore be lost. A mechanism must be established to ensure that the first process completes its update of the record before the second process reads it.

These types of synchronization can be achieved conveniently through the use of events. Events can be shared between processes in the same way that logical files are shared:

they can be accessed as global objects by internal tasks, passed as call-by-reference parameters, or accessed through a SHAREDBYALL library.

If it is only necessary to ensure that different processes do not update the file at the same time, the available state of an event can be used. Each process can be designed to procure the event before using the file, and liberate the event afterward.

If it is necessary to ensure that processes access a file in a certain order, the happened state of an event can be used. The second process that is to use the file can wait on an event. When the first process is finished using the file, the process can cause the event.

Establishing Access Rights

Some special security considerations arise from the ability of different processes to share a logical file. The use of a shared logical file can override the file access privileges of some processes. The use of a shared logical file can make a physical file available to processes that would otherwise not have been able to access it. A shared logical file can also prevent a process from using a physical file that it would normally have had access to. You can control the effects of the shared logical file through careful program design.

Before reading the following discussion, you should be familiar with the concepts introduced in Section 5, "Establishing Process Identity and Privileges."

The access rights allowed for a logical file are determined at file-open time. Once the logical file is opened, all the processes that share the logical file have equal access rights to the physical file. This is true even if the processes have different usercodes, accesscodes, names, and security statuses.

When a process attempts to open a file, the system examines the physical file to determine the access rights that can be granted to the logical file. The system evaluates these rights according to one of the following two rules:

- Actor rule

  File access rights are based on the security status and task attributes of the process that opens the file.

- Declarer rule

  File access rights are based on the security status and task attributes of the process that declares the file.

The actor rule was formerly the only method used for file security checking. The declarer rule was introduced to provide an alternative that gives more predictable behavior. By default, the declarer rule is now used to determine file access rights.

You can override the default method of file access by assigning the FILEACCESSRULE task attribute of the process that opens the file. The default value of DECLARER causes file access rights to be based on the declarer rule. A value of ACTOR causes file access rights to be based strictly on the actor rule. The value of ACTOR can be assigned only by a privileged process.

The following subsections give examples of how the actor and declarer rules apply to file access through libraries and file access restricted by guard files.

Example: Nonprivileged Library Program

Suppose there is a SHAREDBYALL library program. The library object code file is nonprivileged and has a SECURITYTYPE of PUBLIC. An instance of the library is running under a privileged usercode called U1. This library declares a file with usercode U2. The library also exports procedures that can be used to open, close, read from, and write to the file. It happens that a physical file with the specified title already exists and has a SECURITYTYPE of PRIVATE. There are also a number of user processes, including one called A that has usercode U3 and runs with nonprivileged status, and another user process B that has usercode U4 and runs with privileged status.

Under the actor rule, if the library process attempts to open the file before freezing, then the file open operation is successful and all user processes are able to access the file by way of the exported procedures. The file open operation succeeds because the library runs under a privileged usercode and therefore has the right to access a private file stored under a different usercode.

On the other hand, if the library process freezes before it opens the file, and user process A enters a library procedure that opens the file, then the file open operation fails. This is because user process A is nonprivileged and, therefore, does not have the right to access private files stored under different usercodes. However, if user process B enters the library procedure that opens the file, the file open succeeds. Once the file is open, all library user processes, including user process A, are able to access the file by way of library procedures.

Under the declarer rule, the file open will be successful regardless of whether the library process opens the file directly or exports a procedure to a user process that opens the file. In either case, file access is evaluated based on the declaring process, which is the library. The library, because it is privileged, has the ability to access a file stored under a different usercode.

Example: Privileged Transparent Library Program

The concept of privileged transparent status was introduced under "Transparent Object Code File Privileges" in Section 5, "Establishing Process Identity and Privileges." For library procedures that open files, the effects of privileged transparent status vary, depending on whether the file to be open is globally declared, locally declared, or passed as a parameter. The following ALGOL library, named FILELIB, illustrates these three possibilities.

```
100 $ SHARING = SHAREDBYALL
110 BEGIN
120 FILE GLOBALFILE;
130
140 PROCEDURE GLOBAL_OPEN;
150   OPEN(GLOBALFILE);
160
170 PROCEDURE LOCAL_OPEN;
180 BEGIN
190   FILE LOCALFILE;
200   OPEN(LOCALFILE);
210 END;
220
230 PROCEDURE USER_OPEN(PASSEDFILE);
240   FILE PASSEDFILE;
250 BEGIN
260   OPEN(PASSEDFILE);
270 END;
280
290 EXPORT GLOBAL_OPEN, LOCAL_OPEN, USER_OPEN;
300 FREEZE(PERMANENT);
310 END.
```

FILELIB is a permanent, SHAREDBYALL library. The object code file is marked with privileged transparent status. FILELIB exports three procedures: GLOBAL_OPEN, which opens a file declared globally in the library; LOCAL_OPEN, which declares and opens a file; and USER_OPEN, which opens a file passed in as a parameter from the user process.

FILELIB is used by the following user program, called USERPROC.

```
100 BEGIN
110 FILE USERFILE;
120 LIBRARY L(LIBACCESS=BYTITLE,TITLE="OBJECT/TEST/ALGOL/LIB.");
130
140 PROCEDURE GLOBAL_OPEN;
150   LIBRARY L;
160
170 PROCEDURE LOCAL_OPEN;
180   LIBRARY L;
190
200 PROCEDURE USER_OPEN(PASSEDFILE);
210   FILE PASSEDFILE;
220   LIBRARY L;
230
240 GLOBAL_OPEN;
250 LOCAL_OPEN;
260 USER_OPEN(USERFILE);
270 END.
```

USERPROC invokes all of the library procedures: GLOBAL_OPEN, LOCAL_OPEN, and USER_OPEN.

Suppose that USERPROC runs with a FILEACCESSRULE value of ACTOR. If USERPROC is privileged, then it succeeds in opening all three files: LOCALFILE, GLOBALFILE, and PASSEDFILE. If USERPROC is nonprivileged, then the procedures might or might not succeed in opening the files. The success of each file open operation depends on the TITLE and SECURITYTYPE attributes of the file.

Suppose instead that the user process USERPROC has a FILEACCESSRULE value of DECLARER. In this case, USERPROC has different file access rights with regard to PASSEDFILE than it does with regard to GLOBALFILE and LOCALFILE. The rules are as follows:

- Because PASSEDFILE is ultimately declared by USERPROC, the security status of USERPROC determines whether it has the right to open PASSEDFILE. If USERPROC runs under a privileged usercode, the file open is executed with privileged status. Similarly, if the object code file for USERPROC is privileged, the library procedures inherit this status because they are privileged transparent.
- Because GLOBALFILE and LOCALFILE are declared in the library program, the rights to open these files are determined solely by the security status of the library process. Even if the object code file of USERPROC is privileged, the privileges inherited by GLOBAL_OPEN and LOCAL_OPEN do not extend to files declared in the library program. This behavior is a special exception to the rule that privileged transparent procedures inherit the privileged status of the code that invokes them.

There is a good reason for this strict treatment of files declared in privileged transparent libraries. The file access rights for a file declared in a library are permanently established at file-open time. Thus, a privileged user process opening a file in a shared library can have the effect of granting file access to other, nonprivileged user processes.

The behavior under the default declarer rule prevents this file access from being granted accidentally.

You can overcome this restriction by assigning a FILEACCESSRULE value of ACTOR to the privileged user process that opens the file.

Example: Parent and Task Accessing a Guarded File

Suppose that a process with a NAME task attribute value of (SMITH)PROC1 declares a file titled (SMITH)FILEA ON DISK. The process (SMITH)PROC1 initiates a second process with a NAME of (SMITH)PROC2 and passes the file as a call-by-reference parameter. Both of these processes are nonprivileged. Suppose, further, that (SMITH)FILEA ON DISK has a SECURITYTYPE of CONTROLLED and a guard file that allows only processes named (SMITH)PROC1 to access this file.

Under the actor rule, if (SMITH)PROC1 opens the file, then both (SMITH)PROC1 and (SMITH)PROC2 are granted access to the file. However, if (SMITH)PROC2 attempts to open the file before (SMITH)PROC1 opens it, the file open operation fails. The process (SMITH)PROC2 cannot use the file until it has been opened by (SMITH)PROC1.

Under the declarer rule, both processes are granted access to the file, regardless of which one opens the file first. This is because (SMITH)PROC1, which declares the file, is allowed access rights by the guard file.

Understanding I/O Accounting

The system maintains several records of the I/O time accumulated by a process. More specifically, these are records of the time I/O devices devoted to executing I/Os for the process. This information is maintained in the ACCUMIOTIME task attribute. This information also appears in the Major Type 1, Minor Type 2 (EOJ) and Minor Type 4 (EOT) log entries.

If you are involved in writing billing systems or in evaluating the system workload, then you should be aware that the system logs all I/O time for shared logical files to the process that declared the file. The process that declared the file is not necessarily the process that is actually executing read and write statements that use the file. Consider the following ALGOL example:

```
BEGIN
FILE DATAFILE(KIND=DISK,NEWFILE=FALSE,DEPENDENTSPECS=TRUE);
TASK T;
PROCEDURE UPDATE;
BEGIN
  ARRAY LINE[Ø:79];
  WHILE NOT READ(DATAFILE,8Ø,LINE) DO
  BEGIN
    REPLACE LINE BY "NEW DATA";
    WRITE(DATAFILE,8Ø,LINE);
  END;
END;
CALL UPDATE [T];
END.
```

In this example, the parent process initiates the procedure UPDATE as a synchronous task. UPDATE then reads each line of the file, modifies the data, and writes it back out to the file. Because the parent process was the declarer of DATAFILE, the ACCUMIOTIME attribute of the parent reflects all the I/O initiation time logged by the UPDATE task.

Note that the system handles I/O accounting in a different manner for direct files. Suppose that one process declares a direct file, and that several other processes read from or write to that file using direct arrays. In this situation, the I/O time for each I/O operation is charged to the process that declares the direct array used for the I/O, rather than to the process that declares the direct file.

File Sharing Examples

The following is a simple example of a library that allows multiple user processes to access the same disk file. This particular library allows processes to access a file as if it were a stack. In other words, whenever a user process writes to the file, the line pointer is incremented by one. Whenever a user process reads from the file, the line pointer is decremented by one. The PROCURE and LIBERATE statements are used to ensure that only one process accesses the file at a time.

```
$SHARING = SHAREDBYALL
BEGIN
  FILE STK(KIND=DISK,MAXRECSIZE=12,BLOCKSIZE=1200);
  EVENT STACK_ACCESS;
  INTEGER TOP_OF_STACK;

  BOOLEAN PROCEDURE PUSH_STK(BUF);
  ARRAY BUF[0];
  BEGIN
    PROCURE(STACK_ACCESS);
    TOP_OF_STACK := * + 1;
    PUSH_STK := WRITE(STK[TOP_OF_STACK], 12, BUF);
    LIBERATE(STACK_ACCESS);
  END PUSH_STK;

  BOOLEAN PROCEDURE POP_STK(BUF);
  ARRAY BUF[0];
  BEGIN
    PROCURE(STACK_ACCESS);
    POP_STK := READ(STK[TOP_OF_STACK], 12, BUF);
    TOP_OF_STACK := * - 1;
    LIBERATE(STACK_ACCESS);
  END POP_STK;

  EXPORT PUSH_STK, POP_STK;
  OPEN (STK);
  TOP_OF_STACK := -1;
  FREEZE(PERMANENT);
END.
```

The following is an example of a WFL job that uses a global file equation to cause two tasks to use the same logical file. The logical file is declared in the job at lines 140-150. The global file equations occur at lines 190 and 210.

```
100 ?BEGIN JOB TEST/WFL;
110   JOBSUMMARY = SUPPRESSED;
120   DISPLAYONLYTOMCS = TRUE;
130   CLASS = 0;
140 FILE GBAL(KIND=REMOTE,NEWFILE=TRUE,TITLE="JUNK/ERRORLOG",
150           MAXRECSIZE=15,UNITS=WORDS);
160 MYSELF(STATION = MYSELF(SOURCESTATION));
170 OPEN(GBAL);
180 PROCESS RUN OBJECT/TEST/ALGOL/TASK;
190    FILE BALANCES := GBAL;
200 PROCESS RUN OBJECT/TEST/ALGOL/TASK;
210    FILE BALANCES := GBAL;
220 LOCK(ERR);
230 ?END JOB
```

Note that, in the preceding WFL job, it is the colon before the equal sign on lines 190 and 210 that informs WFL that this is a global file equation. If the statement were *FILE BALANCES = GBAL*, then WFL would interpret this as meaning that the file title is GBAL.

The statement at line 160 ensures that the STATION task attribute of the job reflects the LSN or physical unit number of the originating station. This STATION value is inherited by the tasks, and determines the station where the GBAL remote file is opened.

The following is the program that is initiated twice by this WFL job.

```
100 BEGIN
110 FILE BALANCES;
120 PROCEDURE ERRWRITE(ERR_ARRAY,DEPOSIT,SEQ);
130   EBCDIC ARRAY ERR_ARRAY[*];
140   INTEGER DEPOSIT,SEQ;
150 BEGIN
160   INTEGER CUST_BALANCE;
170   MYJOB.LOCKED := TRUE;
180   READ(BALANCES[SEQ],//,ERR_ARRAY);
190   CUST_BALANCE := INTEGER(ERR_ARRAY,8) + DEPOSIT;
200   REPLACE ERR_ARRAY BY CUST_BALANCE FOR 8 DIGITS;
210   WRITE(BALANCES[SEQ],//,ERR_ARRAY);
220   MYJOB.LOCKED := FALSE;
230 END;
240 % The outer block statements are omitted from this example
250 END.
```

Because events cannot be declared in WFL, this program is designed to make use of the LOCKED task attribute to regulate access to the file. Setting LOCKED to TRUE has

the same effect as procuring an event, and setting LOCKED to FALSE has the same effect as liberating an event. The program uses the MYJOB task variable because this task variable has visibility to all the tasks of the WFL job. This mechanism ensures that only one process is actively reading and writing the file at a time, though all processes continue to have the file open.

Using Shared Physical Files

It is possible for multiple processes to use the same physical disk file at the same time without sharing the same logical file. The sharing is accomplished by using identifying file attributes to cause the logical files in each process to link to the same physical file when opened.

The fact that the logical file is not shared creates some additional synchronization problems beyond those previously discussed. These synchronization concerns arise only in cases where at least one of the processes writes to the shared physical file. If all the processes simply read from the file, then the order in which they execute their read operations makes no difference.

However, if one or more of the processes writes to the shared physical file, then the processes can be adequately synchronized only if they take turns opening and closing the file. No two processes should have the file open at the same time.

Entering a File in the Directory

Multiple logical files can link to the same physical file only if the physical file has been entered into the disk directory. The concept of the disk directory is closely related to the concepts of *permanent* and *temporary* files. A permanent file appears in the disk directory, and by default is retained when it is closed. A temporary file does not appear in the disk directory, and by default is removed when it is closed.

A process can cause a file to be entered in the directory by any of the following methods, which are referred to as *directory entrance operations*:

- Opening the file with the NEWFILE file attribute set to TRUE and the PROTECTION file attribute set to SAVE or PROTECTED. Of these two values, SAVE is preferable in most cases because PROTECTED adds overhead. Opening the file creates a permanent file, which is entered immediately in the directory. The file continues to exist after the process terminates, unless the process specifies the PURGE option in the statement that closes the file.
- For a file that was opened with a PROTECTION value of TEMPORARY (the default), closing the file with either the LOCK or CRUNCH option specified in the close statement. In addition to closing the file, this action enters the file in the directory.
- Opening the file with the NEWFILE file attribute and the SENSITIVEDATA file attribute both set to TRUE. The main purpose of using the SENSITIVEDATA attribute is to protect sensitive information, but it also has the side effect of entering a file in the directory.

- Opening the file with the NEWFILE file attribute and the DUPLICATED file attribute both set to TRUE. The main purpose of the DUPLICATED file attribute is to prevent data from being corrupted by disk errors, but this attribute also has the side effect of entering a file in the directory.

If two files with the same title exist on the same family, they cannot both be permanent. An option called AUTORM specifies the action to be taken if a process attempts to enter a file in the directory, and a file with same title already exists. If the AUTORM option is set, the attempt to enter the file in the directory causes one of the following actions:

- If the old file is not in use by any process, it is removed and the new file is entered in the directory.
- If the old file is still in use by another process, the old file is removed from the directory and the new file is added to the directory. The old file remains open as a temporary file. Some unexpected effects can arise from this situation. For example, suppose the new file was opened as a temporary file by process B and the old file was originally opened as a permanent file by process A. When process B closes the new file with LOCK, the new file is changed into a permanent file, and the old file is changed into a temporary file. If process A then closes the old file with LOCK, the old file becomes permanent again and the new file is changed back into a temporary file.

  In general, the last file entered or reentered in the directory is the one that is saved permanently, regardless of whether the directory entry was caused by the file attribute values in force at file open time, or by the option used for the close operation.

If the AUTORM option is reset, and a permanent file with a particular title already exists, then a process that attempts to enter a file with the same title into the directory is suspended with a "DUP LIBRARY" RSVP message. An operator can restart the process by entering the RM (Remove) system command, which deletes the existing duplicate file.

The system treats the AUTORM option as set for a particular process if either or both of the following are true:

- The AUTORM option of the OPTION task attribute is set.
- The AUTORM operating system option is set. This option can be set or reset through the OP (Options) system command.

Matching Physical Files

Once a physical file is entered in the directory, processes can link other logical files to it by specifying appropriate values for the following file attributes:

- KIND

  Specifies the type of storage medium, such as DISK.

- TITLE

  Specifies the usercode, file name, and family.

- NEWFILE

  If set to FALSE, specifies that an existing file should be opened. Note that the process will be suspended with a "NO FILE" message if the file does not exist. A NEWFILE value of TRUE can be used for entering a new file in the directory, as described under "Entering a File in the Directory" earlier in this section.

- DEPENDENTSPECS

  Causes the logical file to assume all the file attributes of the physical file. This makes it unnecessary for all processes to repeat the file attribute assignments that determine the structure of the file.

Ensuring Exclusive Access to a Physical File

Two processes are unable to share the same logical file if they are unrelated processes that do not use a SHAREDBYALL library. In this case, the processes are also unable to share events. When events are not available, the next best method of synchronization is to use certain features of the I/O subsystem to ensure that only one process has the file open at a time.

The simplest method of ensuring exclusive access to a physical file is to create the file with the default PROTECTION value, which is TEMPORARY. The file is not entered in the directory and therefore is not visible to other processes. Later, the process can close the file with LOCK, thus entering the file in the directory and making it available to other processes. If another process attempts to access the file before it is locked, the process is suspended with a "NO FILE" condition. When the file is locked, the process resumes.

Another method of securing exclusive access to a file is by setting the EXCLUSIVE file attribute to TRUE before opening the file. The EXCLUSIVE file attribute specifies that no other process can have the physical file open at the same time as this process.

If a process sets EXCLUSIVE to TRUE and then opens a file, then any other process that attempts to open that physical file is suspended until this process closes it. If a process sets EXCLUSIVE and attempts to open a physical file that is already in use by another process, the process is suspended until the other process closes the file. In either case, the RSVP message displayed is "WAITING ON: <file title>".

It is possible for multiple processes to be waiting to open the same physical file with EXCLUSIVE = TRUE. When the file becomes available, one of the waiting processes opens the file and the other processes continue to wait. It is not possible to predict which of the waiting processes will succeed in opening the file first.

If it is not desirable for the program to be suspended until the file becomes available, the process can attempt a conditional open operation instead. This can be effected by using an open statement with the AVAILABLE option set or by interrogating the AVAILABLE file attribute. If another process is currently using the file with EXCLUSIVE = TRUE, the conditional open operation fails and returns a result reporting the reason for the failure. (The results are documented in the AVAILABLE file attribute description in the *A Series File Attributes Programming Reference Manual*.) The process then continues executing normally.

Exclusive files are best suited to situations where a single body of information is to be transmitted from one process to another. An extended dialogue between processes cannot be implemented efficiently by this method, because it requires repeated file open and close operations. Each file open or close operation is an expensive operation that consumes many times the resources required to access an event or perform a simple read or write operation.

Sharing Nonexclusive Files

If the EXCLUSIVE file attribute is not set, then any number of logical files can be linked to the same physical disk file at the same time. However, you should be aware that this type of disk file sharing involves complexities of synchronization that are extremely difficult to resolve. When different logical files are used, different buffers are also used, with the result that it is not possible to predict the order in which read and write operations submitted by different processes will be executed. What is more, the BLOCKSIZE file attribute can cause multiple records to be read into or written from a buffer, so that changes by other processes to nearby records can be accidentally overwritten. (For more information about file blocking and file buffers, refer to the *A Series I/O Subsystem Programming Guide*.)

Because of these and other problems, this method of sharing disk files between processes is not recommended. Where unrelated processes need to use the same disk file concurrently, a shared library should be used to provide access to a shared logical file. For an example of this method, refer to the discussion of providing global objects in Section 18, "Using Libraries."

Section 20
Communication across Multihost Networks

The previous sections of this guide introduced several interprocess communication (IPC) features, including task attributes, events, global objects, parameters, libraries, and port files. You can use all these techniques to provide communication between processes that run on the same host system. However, some restrictions apply to the use of these IPC methods across multihost networks. This section explains which IPC methods can be used to route information between processes running on separate hosts. Before reading this section, you should be familiar with the concepts introduced in Section 12, "Tasking across Multihost Networks."

Port files provide an ideal method of transferring information between processes that run on different hosts. Subfile matching follows the same rules used for local port files. The YOURHOST file attribute is used to specify the host where the remote process is running. For a general overview of port files, and examples of port files, refer to Section 19, "Using Shared Files."

Of the task attributes used for interprocess communication, the TASKVALUE attribute and the SW1 through SW8 attributes are available for use across multihost networks. A parent process and a remote offspring can communicate with each other by reading and assigning these task attributes. However, the LOCKED, TARGET, and TASKSTRING task attributes are not supported across multihost networks.

The following methods of interprocess communication cannot be used across multihost networks:

- Global objects

  Because a remote process must be an external process, it cannot directly access any global objects declared in its parent.

- Call-by-reference and call-by-name parameters

  The only type of parameter allowed, a one-dimensional real array, must be passed by value.

- Libraries

  The user processes for a library must reside on the same host system as the library.

User-declared events cannot be shared between processes running on different hosts because global objects, call-by-reference parameters, and libraries are the only ways of sharing user-declared events between processes.

However, it is possible to use implicitly declared events. Processes that communicate by way of port files can wait on the event-valued port file attributes CHANGEEVENT,

INPUTEVENT, and OUTPUTEVENT. If necessary, you can create port files for the sole purpose of regulating the timing of related processes.

For example, suppose two processes communicate using a port subfile. The first process is waiting on the INPUTEVENT. The second process can cause the INPUTEVENT and thus reactivate the first process by writing a message into the subfile. When the waiting process is reactivated, it can reset the INPUTEVENT by reading all available messages in the input queue.

WFL jobs can use forms of the WAIT statement that create implicit events. For example, the following statements initiate an asynchronous task and cause the job to wait until the task's TASKVALUE attribute reaches a certain value:

```
PROCESS RUN OBJECT/ALGOL/TASK [T];
   HOSTNAME = TOLEDO;
WAIT T(TASKVALUE) = 3;
```

The EXCEPTIONEVENT task attribute has some use in remote interprocess communication because the exception event of a local parent is implicitly caused when its remote task terminates. However, a remote task cannot use a CAUSE statement to cause its parent's exception event because the EXCEPTIONEVENT task attribute is not supported by Host Services. The other task attributes that access events, ACCEPTEVENT and LOCKED, are also not supported by Host Services.

Glossary

A

abnormal termination
The type of termination that results when a process encounters a run-time error, or is discontinued by an operator command or a statement in another process.

access
To perform an action on an object. Possible actions depend on the type of object; for example, interrogating or assigning a value to a variable, reading from or writing to a file, or invoking a procedure.

accidental entry
See thunk.

activation record
A structure that is added to the process stack when a process enters a block. The activation record includes storage for objects declared in that block, a historical link, and an environmental link, as well as other items used by the operating system.

active
Pertaining to the state of a process that is executing normally, and is neither scheduled nor suspended.

actual parameter
An object or value that is specified in a procedure invocation statement and passed to a formal parameter.

actual segment descriptor (ASD)
A pointer to the location of a data or code item in memory or on a disk.

addressing environment
The set of objects that can be accessed by statements in a particular block.

ADM
See automatic display mode.

ALGOL
Algorithmic language. A structured, high-level programming language that provides the basis for the stack architecture of the Unisys A Series systems. ALGOL was the first block-structured language developed in the 1960s and served as a basis for such languages as Pascal and Ada. It is still used extensively on A Series systems, primarily for systems programming.

ancestor
The parent of a particular task, or the parent of any ancestor of the task.

APLB

A Programming Language B. A second-generation extended version of A Programming Language (APL).

ASCII

American Standard Code for Information Interchange. A standard 7-bit or 8-bit information code used to represent alphanumeric characters, control characters, and graphic characters on a computer system.

ASD

See actual segment descriptor.

ASD memory

The memory architecture used on A Series systems. In this memory architecture, memory is treated as a single continuous region that is indexed by the ASD table. Memory management is very flexible and is handled automatically by the operating system.

asynchronous process

A process that executes in parallel with its initiator.

automatic display mode (ADM)

A display mode that can be initiated at an operator display terminal (ODT) through the use of the ADM (Automatic Display Mode) system command. In this mode, various types of information about the system are displayed at regular intervals.

B

BASIC

Beginner's All-purpose Symbolic Instruction Code. A programming language that was developed as a tool for teaching computer programming. BASIC is similar to FORTRAN in many ways, but BASIC is easier to use because the instructions are structured more like English.

BDMSALGOL

A Unisys language based on Extended ALGOL that contains extensions for accessing Data Management System II (DMSII) databases.

beginning of job (BOJ)

The start of processing of a job.

beginning of task (BOT)

The start of processing of a task.

Binder

A program that enables separately compiled subprograms to be joined with a host object code file to produce a single object code file.

block

A program, or a part of a program, that is treated by the processor as a discrete unit. Examples are a procedure in ALGOL, a procedure or function in Pascal, a subroutine or function in FORTRAN, or a complete COBOL program.

BNA

The network architecture used on A Series, B 1000, and V Series systems as well as CP9500 and CP 2000 communications processors to connect multiple, independent, compatible computer systems into a network for distributed processing and resource sharing.

BOJ

See beginning of job.

BOT

See beginning of task.

C

call-by-name

Pertaining to one method of passing a parameter to a procedure. The system substitutes the actual parameter wherever the formal parameter is mentioned in the procedure body. Any assignments to the actual parameter immediately change the value of the formal parameter, and vice versa. *Synonym for* by name.

call-by-reference

Pertaining to one method of passing a parameter to a procedure. The system evaluates the location of the actual parameter and replaces the formal parameter with a reference to that location. Any change made to the formal parameter affects the actual parameter, and vice versa. *Synonym for* by reference.

call-by-value

Pertaining to one method of passing a parameter to a procedure. A copy of the value of the actual parameter is assigned to the formal parameter, which is thereafter handled as a variable that is local to the procedure body. Any change made to the value of a call-by-value formal parameter has no effect outside the procedure body. *Synonym for* by value.

CANDE

See Command and Edit.

COBOL

Common Business-Oriented Language. A widely used, procedure-oriented language intended for use in solving problems in business data processing. The main characteristics of COBOL are the easy readability of programs and a considerable degree of machine independence. COBOL is the most widely used procedure-oriented language.

code segment dictionary

A memory structure that is associated with a process and that indexes the memory addresses of the various segments of program code used by that process. The same code segment dictionary can be shared by more than one process, provided that each process

is an instance of the same procedure. A code segment dictionary is also referred to as a D1 stack.

Command and Edit (CANDE)
A time-sharing message control system (MCS) that enables a user to create and edit files, and to develop, test, and execute programs, interactively.

Communications Management System (COMS)
A general message control system (MCS) that controls online environments on A Series systems. COMS can support the processing of multiprogram transactions, single-station remote files, and multistation remote files.

compiler
A computer program that translates instructions written in a source language, such as COBOL or ALGOL, into machine-executable object code.

COMS
See Communications Management System.

constant
An object whose value is assigned during program compilation and cannot be changed during program execution.

control
(1) The path that execution takes among the various statements of a program. The general tendency is for control to progress, one statement at a time, from the start to the end of the program. However, some statements cause the flow of control to take an alternate path, skipping multiple lines forward or backward before resuming execution. (2) The path that execution takes among the processes in a process family.

coroutine
One of a group of processes that exist simultaneously, but take turns executing, so that only one of the processes is executing at any given time. The coroutine that is currently executing is called the active coroutine, and the others are called continuable coroutines.

cousin
A process that has an ancestor in common with some other process, but does not have the same parent as the other process.

critical block
For a dependent process, the block of the highest lexical level that includes the declaration of any critical objects used by the dependent process. The process that is executing the critical block is called the parent of the dependent process. If the parent exits the critical block while the dependent process is in use, the parent is discontinued and the dependent process is also discontinued.

critical object
A type of object that is used by a process, but was originally declared by another process. Critical objects include the task variable for the process, the procedure declaration for the process, and any objects passed as actual parameters to the process by name or by reference.

D

Data Communications ALGOL (DCALGOL)
A Unisys language based on ALGOL that contains extensions for writing message control system (MCS) programs and other specialized system programs.

data link processor (DLP)
A processor that serves as the system interface to a specific peripheral device, controller, or communications network.

Data Management ALGOL (DMALGOL)
A Unisys language based on ALGOL that contains extensions for writing Data Management System II (DMSII) software and other specialized system programs.

Data Management System II (DMSII)
A specialized system software package used to describe a database and maintain the relationships among the data elements in the database.

data specification
A section of a Work Flow Language (WFL) source program containing data that can be read by tasks of the WFL job. A data specification is also referred to as a data deck.

DCALGOL
See Data Communications ALGOL.

declaration
A programming language construct used to identify an object, such as a type or variable, to the compiler. A declaration can be used to associate a data type with the object so that the object can be used in a program.

declared external procedure
A dummy procedure declaration used in ALGOL or COBOL74 to enable a program to initiate a separate program.

dependent process
A process that depends on the continued existence of another process called the parent process. *See also* task.

descendant
An offspring of a particular process, or an offspring of a descendant of that process.

dialogue
See window dialogue.

direct addressing environment
The set of objects that can be accessed by statements in a particular procedure, but that are not passed as parameters to that procedure.

directory
(1) A table of contents listing the files contained on a device. The device is usually a disk or a tape. (2) A list of file names organized into a hierarchy according to similarities in their names. File names are grouped in a directory if their first name constants

(and associated usercodes) are identical. These groups are divided into subdirectories consisting of those file names whose first two name constants are identical, and so on.

discontinue
To cause a process to terminate abnormally. A process can be discontinued by operator commands, by statements in related processes, or by the system software.

discriminant
In Pascal, an item that appears in an array declaration and that specifies the highest or lowest numbered index for a particular dimension. If the discriminant is an integer, it is called a constant discriminant. If the discriminant is a variable, it is called a dynamic discriminant.

disk resource control (DRC) system
An optional feature of the disk subsystem that provides the ability to control disk space on a per user basis. The DRC system does not support interchange (IC) packs or installation-allocated disk (IAD) usage. DRC is not a security system, but normal security checking occurs.

DLP
See data link processor.

DMALGOL
See Data Management ALGOL.

DMSII
See Data Management System II.

DRC
See disk resource control (DRC) system.

D1 stack
See code segment dictionary.

E

EBCDIC
Extended Binary Coded Decimal Interchange Code. An 8-bit code representing 256 graphic and control characters that are the native character set of most mainframe systems.

element
A component of an array.

end of job (EOJ)
The termination of processing of a job.

end of task (EOT)
The termination of processing of a task.

entry
A type of procedure invocation that creates a new activation record in an existing process stack. The activation record exists until the procedure is exited.

EOJ
See end of job.

EOT
See end of task, end of transmission.

EPILOG procedure
A procedure that is automatically executed just before control exits the block in which the EPILOG procedure is declared. The EPILOG procedure is executed even if the block exit is caused by an error or a DS (Discontinue) system command.

exception task
A process that has a special relationship with another process, such that the following are true: the exception task's EXCEPTIONEVENT task attribute is caused whenever the status of the related process changes; and the related process can use the EXCEPTIONTASK task attribute to access the task attributes of the exception task.

execution
The act of processing statements in a program.

exit
To end the processing of an entered block. Exiting the block eliminates the activation record.

export object
The declaration of a library object in a library.

expression
A combination of operands and operators that results in the generation of one or more values.

extended addressing environment
The set of objects that can be accessed by statements in a particular procedure, including any objects that were passed as parameters to that procedure.

external procedure
A procedure whose procedure body is contained in an object code file different from the statement that invokes the procedure. External procedures are of three kinds: declared external procedures, passed external procedures, and library procedures.

external process
A process created by initiating an external procedure.

F

family
(1) One or more disks logically grouped and treated as a single entity by the system. Each family has a name, and all disks in the family must have been entered into the family with the RC (Reconfigure Disk) system command. (2) *See also* process family.

fatal
Referring to something capable of causing a process to be discontinued. An error that causes a process to be discontinued is called a fatal error.

FETCH specification
A statement in a Work Flow Language (WFL) job that provides a message an operator can display with a PF (Print Fetch) system command. Resetting the NOFETCH system option delays initiation of jobs with FETCH specifications until the operator enters an OK command for each job.

file attribute
An element that describes a characteristic of a file and provides information the system needs to handle the file. Examples of file attributes are the file title, record size, number of areas, and date of creation. For disk files, permanent file attribute values are stored in the disk file header.

formal parameter
An object that is declared in a procedure heading and that receives its value from an actual parameter when the procedure is invoked.

formal parameter specification
A portion of a procedure heading that names and describes a formal parameter.

FORTRAN
Formula Translation. A high-level, structured programming language intended primarily for scientific use.

FORTRAN77
A version of the FORTRAN language that is compatible with the ANSI X3.9-1978 standard.

function
(1) A subroutine that returns a value. (2) *See also* typed procedure.

G

global file assignment
A construct that can be included in a Work Flow Language (WFL) job to cause an offspring to use a file declared in the WFL job. This mechanism amounts to a hidden by-reference parameter, because the same logical file is used by the job and its offspring.

global object
An object that is declared outside a particular block, but that can be accessed by statements in that block.

guard file
A disk file created by the GUARDFILE utility program that describes the access rights of various users and programs to a program, data file, or database.

H

halt/load
A system-initialization procedure that temporarily halts the system and loads the master control program (MCP) from a disk to main memory.

HYDLP
See HYPERchannel data link processor.

HYPERchannel data link processor (HYDLP)
A specialized data link processor (DLP) that enables communication between systems through HYPERchannel adapters. HYPERchannel is a message-level I/O channel-to-channel communications interface between A Series systems. It can also provide an interface to other systems for which a HYPERchannel adapter exists.

I

I/O
Input/output. An operation in which the system reads data from or writes data to a file on a peripheral device such as a disk drive.

I/O processor (IOP)
A specialized processor for moving data between system memory and the I/O subsystem.

import object
The declaration of a library object in a user program.

in-use process
A process that has been submitted for initiation and has not yet terminated. The state of an in-use process can be scheduled, active, or suspended.

independent process
A process that does not depend on the continued existence of a parent process. An independent process is the head of any process family it is part of. *See also* job.

independent runner (IR)
A master control program (MCP) procedure that is initiated as an independent process. The procedure is executed in its own process stack rather than in the stack of a user process. An IR can be either visible or invisible. If the IR is visible, its status can be interrogated. If the IR is invisible, it does not appear in mix displays.

index
A value used to specify a particular element of an array variable.

indexed sequential-access method (ISAM)
A method that provides efficient, flexible random access to records identified by keys stored in an index.

InfoGuard
The Unisys security-enhancement software for A Series systems. InfoGuard provides such features as password management, selective logging and auditing, tape volume security, and simplified system-security configuration.

inheritance
The automatic transfer of particular task attribute values from a process to a descendant process. More broadly, inheritance also refers to the automatic transfer of values from job queue attributes or session attributes to the equivalent task attributes of a descendant process.

initiation
A type of procedure invocation that causes the creation of a new process, with its own process stack and process information block (PIB). Additionally, a new code segment dictionary is created if a code segment dictionary for that procedure is not already available.

initiator
The process that initiates a particular process. The initiator can be a different process from the parent process.

instance
A process that is an execution of a particular procedure and that has its own process stack. Multiple instances of a procedure can exist at the same time; a new instance is created each time the procedure is initiated.

interactive process
A process that reads input from a terminal or operator display terminal (ODT), and whose actions are largely determined by the input received. A data entry process, such as the Editor, is an example of an interactive process.

internal procedure
A procedure whose procedure body is contained in the same object code file as the statement that invokes the procedure.

internal process
The execution of an internal procedure that has been initiated.

intersystem control (ISC)
A direct hardware connection that enables data transfer between independent systems. The components that make up an ISC connection are a hub and its attached host control (HC) units. The HC unit type used to connect an I/O channel to a hub depends on the type of machine, specifically the I/O subsystem protocol.

invocation
The act that transfers control to the start of a specified procedure, initializes any parameters, and begins the execution of the statements of the procedure. Invocations are of two kinds: entrances and initiations.

IOP

See I/O processor.

ISAM

See indexed sequential-access method.

ISC

See intersystem control.

J

job

An independent process. The job of a particular task is the independent process that is the eldest ancestor of that task.

job description file

A system disk file that stores information about Work Flow Language (WFL) jobs, job queues, and various system settings. The job description file is also known as the JOBDESC file.

job file

A disk file that is associated with a job and contains the job log. The job file for a Work Flow Language (WFL) job also serves as the object code file for the job, and includes job restart information, data specifications, and a copy of the WFL source program.

job log

A log that is stored in a job file and contains log entries for a particular job and its descendant tasks. When the job terminates, the job log is processed to produce the job summary.

job queue

A structure in the system software that stores a list of jobs that have been compiled and are waiting to be initiated.

job summary

A file, produced after a job completes execution, that lists information such as the tasks initiated by the job, the beginning and ending times for each task, and the termination information for each task.

L

lex level

See lexical level.

lexical level (lex level)

A measure of the number of other blocks a block is nested within. The outer block of a program has a lex level of 2 or 3, depending on whether the program has a procedure heading. Each block has a lex level one higher than the block it is nested within.

library
A program that exports objects for use by user programs.

library directory
A memory structure associated with a library process stack that describes the objects exported by the library process.

library object
An object that is shared by a library and one or more user programs.

library process
An instance of the execution of a library. The sharing option of a library determines whether multiple user programs use the same instance of the library.

library program source
The program source file from which a library is compiled.

library template
A memory structure, associated with a user process stack, that describes objects imported from a library.

local host
The host computer system to which a user's station is physically attached.

local object
An object that is declared within a particular block.

local process
(1) A process running on the local host system. (2) A process that communicates with a remote process by way of a port subfile.

logical
Synonym for virtual.

logical file
A file variable declared in a program, which represents the file and its structure to the program. A logical file has no properties of its own until it is described by file attributes or associated with a physical file.

logical station number (LSN)
(1) In the Network Definition Language II (NDLII), a unique number assigned to each station in a network. Each station has an LSN assigned according to the order in which the stations are defined in NDLII. The first defined station is 1. (2) In the Interactive Datacomm Configurator (IDC), a unique number assigned to each station structure. When IDC creates the DATACOMINFO file from the network information file II (NIFII), it assigns an LSN to each structure sequentially, beginning with the number 2. The numbers allocated by IDC are the same as those used by the operating system to identify a station.

LSN
See logical station number.

M

MAKEUSER
A utility used to define, modify, or display information about the usercodes that are available on the system. The usercode information is stored in a file called the USERDATAFILE.

MARC
See Menu-Assisted Resource Control.

master control program (MCP)
The central program of the A Series operating system. The term applies to any master control program that Unisys may release for A Series systems.

MCP
See master control program.

MCS
See message control system.

Menu-Assisted Resource Control (MARC)
A menu-driven interface to A Series systems that also enables direct entry of commands.

message control system (MCS)
A program that controls the flow of messages between terminals, application programs, and the operating system. MCS functions can include message routing, access control, audit and recovery, system management, and message formatting.

microsecond
One-millionth of a second (.000001).

mix
The set of processes that currently exist on a particular computer. The mix can include active, scheduled, and suspended processes.

mix number
A 4-digit number that identifies a process while it is executing. This number is stored in the MIXNUMBER task attribute.

multiprocessing
A state in which two or more processors in the same system run under the control of a single operating system.

multiprogramming
The ability of a single computer system to execute many processes concurrently.

MYJOB
A predeclared task variable that a process can use to access the task attributes of its job.

MYSELF
A predeclared task variable that a process can use to access its own task attributes.

N

nesting
The practice of declaring a procedure within another procedure.

NEWP
A structured, high-level programming language used in developing some Unisys system software. Based on the ALGOL language, NEWP contains facilities necessary for the operating system to interact with A Series hardware.

normal termination
The termination of a process that has executed successfully, without any errors and without being terminated prematurely by an operator command or another process.

O

object
Any item declared in a program. Arrays, files, procedures, tasks, and variables are all examples of objects.

object code file
A file produced by a compiler when a program is compiled successfully. The file contains instructions in machine-executable object code.

ODT
See operator display terminal.

offspring
The dependent process whose critical block is owned by a particular parent process.

operator display terminal (ODT)
A terminal or other device that is connected to the system in such a way that it can communicate directly with the operating system. The ODT allows operations personnel to accomplish system operations functions through either of two operating modes: system command mode or data comm mode.

outer block
The portion of a program that has the lowest lexical level.

overlay
To load code or data into a memory area that was previously allocated to other code or data, and to write any data that previously occupied the area to a disk file if necessary.

P

parameter
An identifier associated in a special way with a procedure. A parameter is declared in the procedure heading and is automatically assigned a value when the procedure is invoked.

parameter passing
The act of passing an object or a value from an actual parameter to a formal parameter.

parent
A process that owns the critical block of a dependent process. If the parent exits the critical block before the dependent process terminates, the dependent process is discontinued.

partner process
The process that is specified by the PARTNER task attribute of another process. A process can transfer control to its partner process by executing a general continue statement.

Pascal
A high-level programming language developed by Niklaus Wirth, based on the block structuring nature of ALGOL 60 and the data structuring innovations of C.A.R. Hoare. Pascal is a general-purpose language.

passed external procedure
A procedure that is passed as a parameter from one program to another. Passed external procedures include both procedures that are explicitly passed and thunks that are implicitly passed.

passing mode
The mode by which an actual parameter is passed to a formal parameter. These modes are call-by-name, call-by-reference, or call-by-value.

performance
(1) A measurement of how efficiently a process uses resources such as processor time, I/O time, or elapsed time. (2) A measure of the amount of work a computer system is able to do in a given period of time.

physical file
A file as it is stored on a particular recording medium such as a disk or a tape.

PIB
See process information block.

PL/I
Programming Language I. A high-level, structured programming language designed primarily for scientific and commercial use.

port file
The following should replace all three definitions: A type of file for which file operations occur between a local user process and another process on the same host or on a remote host that is reachable through a network. A port file is made up of one or more subfiles, each of which supports one dialogue.

private process
A process whose task attributes cannot be accessed by other processes. Assigning the *private process* option to the OPTION task attribute causes a process to become a private process.

privilege
The ability to invoke actions that are not ordinarily allowed, such as accessing private files stored under other usercodes or invoking privileged functions such as SETSTATUS. The concept of privilege applies to usercodes, programs, and processes.

procedure
A block that can be invoked by statements elsewhere in the same program or, in some cases, by statements in another program. In most instances, a procedure has a procedure heading and a procedure body. Examples are a procedure in ALGOL, a procedure or function in Pascal, a subroutine or function in FORTRAN, or a complete COBOL program.

procedure body
The portion of a procedure that contains declarations and statements.

procedure entry
See entry.

procedure heading
The portion of a procedure that specifies the procedure name and the formal parameters, if any.

procedure initiation
See initiation.

process
The execution of a program or of a procedure that was initiated. The process has its own process stack and process information block (PIB). It also has a code segment dictionary, which can be shared with other processes that are executions of the same program or procedure.

process family
A group of processes consisting of a single job and any tasks that are descendants of that job.

process information block (PIB)
A memory structure that is associated with each process stack and code segment dictionary. The PIB contains control information that is visible only to the operating system. The PIB for a process stack also contains a reference to a task attribute block (TAB).

process stack
A memory structure that stores information about the current state of the execution of a procedure. The process stack includes activation records for all blocks that the process has entered and not yet exited.

process state
The current status of a process. The three process states are scheduled, active, or suspended.

processor
A hardware component that executes programs and procedures.

program
(1) A specification of the sequence of computational steps that solve a computational problem. The steps are written (coded) in a particular programming language. (2) An object code file.

pseudostation
A station created by the operating system that can be attached to, and controlled by, a message control system (MCS) like a *real* station. Unlike a real station, however, a pseudostation is not declared in the SOURCENDLII file or the DATACOMINFO file, has no line assigned, and does not need a corresponding physical terminal on the local host.

Q

queue
(1) A data structure used for storing objects; the objects are removed in the same order they are stored. (2) In Data Communications ALGOL (DCALGOL), a linked list of messages. (3) *See also* job queue, ready queue.

queue attribute
(1) Any attribute that can be assigned to a job queue using the MQ (Make or Modify Queue) system command. Queue attributes limit the use of system resources by jobs and tasks initiated from that queue. MIXLIMIT, PROCESSTIME, and PRIORITY are examples of queue attributes. (3) In Data Communications ALGOL (DCALGOL), any of a number of attributes that return information about a DCALGOL queue or affect usage of that queue.

R

ready queue (READYQ)
A list, maintained by the operating system, of the processes that are waiting for service from a processor.

READYQ
See ready queue.

remote file
A file with the KIND attribute specified as REMOTE. A remote file enables object programs to communicate interactively with a terminal.

remote host system
A system that can be accessed from the local host system by way of a BNA link.

remote process
A process initiated by a process that was running on another host system.

resuming
The act of changing a library process into a nonlibrary process. For example, a temporary library process resumes execution as a nonlibrary process when the last user process delinks from the library. *Contrast with* thawing.

RPG
Report Program Generator. A high-level, commercially oriented programming language used most frequently to produce reports based on information derived from data files.

RSVP message
A message the system displays for a suspended process that states the reason the process was suspended. RSVP messages ask for a reply such as *OK* or *DS*.

run-time error
An error occurring during the execution of a program, which causes the system software to terminate execution of that program abnormally.

S

scheduled process
A process whose initiation is delayed, either because the operator has entered an HS (Hold Schedule) system command or because the operating system estimates the process is likely to need more memory than is currently available.

schema
In Pascal, an array declaration that includes one or more dynamic array bounds. In other words, a schema is a type of incomplete array declaration. Using an array schema as a formal parameter makes it possible to pass arrays with different bounds and different numbers of elements to the same formal parameter.

scope
Those portions of a program or programs that can contain statements that access a particular object.

session
The interactions between a user and a message control system (MCS) during a particular period of time that is assigned an identifying session number. Logging on initiates a new session; logging off terminates a session. Each Menu-Assisted Resource Control (MARC) or Command and Edit (CANDE) dialogue at a terminal accesses a different session.

sibling
A task that has the same parent as another task.

simple array parameter
An array parameter that is declared with an explicitly specified lower bound. *Contrast with* unbounded array parameter.

SORT facility
An operating system procedure that sorts a file or a set of records. SORT can be activated through ALGOL, COBOL(68), COBOL74, PL/I, or the SORT compiler.

source file
A file in which a source program is stored.

source program
A program coded in a language that must be translated into machine language before execution. The translator program is usually a compiler.

stack
(1) A region of memory used to store data items in a particular order on a last-in, first-out basis. (2) A nonpreferred synonym for process stack.

station
The outer end of a communication line. A station can correspond to a single terminal connected on a single line, or several stations can be connected on a line.

support library
A library that is associated with a function name. User programs can access a support library by way of its function name instead of its object code file title. The operator uses the SL (Support Library) system command to link function names with libraries.

suspended process
A process that has temporarily stopped executing and cannot continue until appropriate operator or programmatic action is taken. A process can be suspended deliberately by an operator command or a statement in a program. In addition, the operating system can suspend a process automatically, for example, if the process has requested a file that is missing.

symbolic
A source program.

synchronous process
A process whose initiator waits after initiating the process. When the process terminates, the initiator resumes execution.

system command
Any of a set of commands used to communicate with the operating system. System commands can be entered at an operator display terminal (ODT), in a Menu-Assisted Resource Control (MARC) session, or by way of the DCKEYIN function in a privileged Data Communications ALGOL (DCALGOL) program.

system library
A library that is part of the system software and is accorded special privileges by the operating system. Two examples of system libraries are GENERALSUPPORT and PRINTSUPPORT.

system software
The master control program (MCP) and all other object code files necessary for system operation.

T

TAB
See task attribute block.

TADS

See Test and Debug System.

tanking

(1) The practice of temporarily storing output messages in a disk file because the destination station is unavailable. The operating system and the Communications Management System (COMS) both perform tanking. (2) The practice of temporarily storing messages from a Data Communications ALGOL (DCALGOL) queue in a disk file until the receiving process is ready to read the messages.

task

A dependent process.

task attribute

Any of a number of items that describe and control various aspects of process execution such as the usercode, priority, and the default family specification. Task attributes can be assigned interactively through task equations, or programmatically through statements that use task variables.

task attribute block (TAB)

A memory structure that stores the values of task attributes associated with a given task variable. Before the Mark 3.9 release, this information was part of the process information block (PIB).

task file

A printer backup file that is associated with each process, and that stores any program dumps generated by the process while the TOPRINTER program dump option is enabled. Processes can also write comments to the task file by way of the TASKFILE task attribute.

task variable

An object that is used to interrogate or modify the task attributes of a particular process.

tasking

The act of initiating, monitoring, or controlling processes. The processes can be either jobs or tasks. Operators and users can enter tasking commands from an operator display terminal (ODT), a Command and Edit (CANDE) session, or a Menu-Assisted Resource Control (MARC) session. Programs can initiate processes with such statements as CALL, PROCESS, or RUN. Programs can monitor and control processes by reading and assigning the values of various task attributes.

tasking program

A program that has been marked with tasking status by the MP (Mark Program) system command.

tasking status

A type of security status that permits a program to perform most of the actions that normally require message control system (MCS) privileges. A process receives tasking status while it is executing code from a tasking program.

termination
The act of permanently ceasing execution of a process. The process stack and process information block (PIB) are removed. The code segment dictionary can also be removed.

Test and Debug System (TADS)
A Unisys interactive tool for testing and debugging programs and libraries. TADS enables the programmer to monitor and control the execution of the software under testing and examine the data at any given point during program execution.

thawing
The act of changing a permanent or control library into a temporary library. *Contrast with* resuming.

thrashing
A state of degraded system performance caused by the overcrowding of main memory. The overcrowding of memory causes the system to spend an excessive amount of time performing overlays.

thunk
A compiler-generated procedure that calculates and returns the value of a constant or expression passed to a call-by-name formal parameter. The thunk is executed each time the formal parameter is used. A thunk is also referred to as an accidental entry.

transaction processor (TP)
A process initiated and controlled by the Communications Management System (COMS) to handle transactions through a particular window.

typed procedure
A procedure that is designed to return a value. Invoking such a procedure is similar to evaluating an expression. *See also* function.

U

unbounded array parameter
An array parameter that is declared with an unspecified lower bound. In ALGOL, for example, such a parameter is declared with an asterisk (*) in place of an explicit lower bound. *Contrast with* simple array parameter.

untyped procedure
A procedure that does not return a value. *Contrast with* typed procedure.

user process
(1) A process that is not an invisible independent runner, a message control system (MCS), or a system library. (2) A process that is linked to a library process and can import objects from that library process. *Synonym for* calling process, client process.

user program
(1) A program that is not part of the system software. (2) A program that uses objects imported from a library program.

usercode
An identification code used to establish user identity and control security, and to provide for segregation of files. Usercodes can be applied to every task, job, session, and file on the system. A valid usercode is identified by an entry in the USERDATAFILE.

usercode attribute
A characteristic that can be associated with a usercode in the USERDATAFILE. A set of standard usercode attributes, such as PU, MAXPW, IDENTITY, and PASSWORD, are supplied as part of the description of the USERDATAFILE structure. The system administrator or security administrator can define additional usercode attributes to meet the specific needs of an installation.

USERDATAFILE
A system database that defines valid usercodes and contains various data about each user (such as accesscodes, passwords, and chargecodes) and the population of users for a particular installation.

utility
A system software program that performs commonly used functions.

V

variable
An object in a program whose value can be changed during program execution.

virtual
Pertaining to an item whose existence is simulated by the system software. For example, a pseudostation is a virtual station, and a session is a virtual job.

W

WFL
See Work Flow Language.

WFL job
A Work Flow Language (WFL) program, or the execution of such a program.

window
In the Communications Management System (COMS) architecture, the concept that enables a number of program environments to be operated independently and simultaneously at one station. One of the program environments can be viewed while the others continue to operate.

window dialogue
In the Communications Management System (COMS), a particular access to a program environment through a COMS window at a station. The exact number of window dialogues allowed at one time for a given window depends on the constraints established through the COMS Utility. No more than eight window dialogues are allowed at one time for any window. Each dialogue has an identifying number within its window.

Work Flow Language (WFL)

A Unisys language used for constructing jobs that compile or run programs on A Series systems. WFL includes variables, expressions, and flow-of-control statements that offer the programmer a wide range of capabilities with regard to task control.

Bibliography

A Series ALGOL Programming Reference Manual, Volume 1: Basic Implementation (8600 0098) Unisys Corporation.

A Series APLB Programming Reference Manual (1203643). Unisys Corporation.

A Series BASIC Programming Reference Manual (1203650). Unisys Corporation.

A Series Binder Programming Reference Manual (8600 0304). Unisys Corporation.

A Series C Programming Reference Manual (3957 6061). Unisys Corporation.

A Series CANDE Operations Reference Manual (8600 1500). Unisys Corporation.

A Series COBOL ANSI-68 Programming Reference Manual (8600 0320). Unisys Corporation.

A Series COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation (8600 0296). Unisys Corporation.

A Series COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation (8600 1518). Unisys Corporation.

A Series Communications Management System (COMS) Programming Guide (8600 0650). Unisys Corporation.

A Series DCALGOL Programming Reference Manual (8600 0841). Unisys Corporation.

A Series Disk Subsystem Administration and Operations Guide (8600 0668). Unisys Corporation.

A Series Distributed Systems Service (DSS) Operations Guide (8600 0122). Unisys Corporation.

A Series File Attributes Programming Reference Manual (8600 0064). Unisys Corporation.

A Series FORTRAN Programming Reference Manual (1222691). Unisys Corporation.

A Series FORTRAN77 Programming Reference Manual (3957 6053). Unisys Corporation.

A Series I/O Subsystem Programming Guide (8600 0056). Unisys Corporation.

A Series Menu-Assisted Resource Control (MARC) Operations Guide (8600 0403). Unisys Corporation.

A Series MultiLingual System (MLS) Administration, Operations, and Programming Guide (8600 0288). Unisys Corporation.

A Series NEWP Programming Reference Manual (5044233). Unisys Corporation.

A Series Pascal Programming Reference Manual, Volume 1: Basic Implementation (8600 0080). Unisys Corporation.

A Series PL/I Reference Manual (1169620). Unisys Corporation.

A Series Print System (PrintS/ReprintS) Administration, Operations, and Programming Guide (8600 1039). Unisys Corporation.

A Series Report Program Generator (RPG) Programming Reference Manual, Volume 1: Basic Implementation (8600 0544). Unisys Corporation.

A Series Security Administration Guide (8600 0973). Unisys Corporation.

A Series Security Features Operations and Programming Guide (8600 0528). Unisys Corporation.

A Series System Administration Guide (8600 0437). Unisys Corporation.

A Series System Commands Operations Reference Manual (8600 0395). Unisys Corporation.

A Series System Configuration Guide (8600 0445). Unisys Corporation.

A Series System Operations Guide (8600 0387). Unisys Corporation.

A Series System Software Support Reference Manual (8600 0478). Unisys Corporation.

A Series System Software Utilities Operations Reference Manual (8600 0460). Unisys Corporation.

A Series Systems Functional Overview (8600 0353). Unisys Corporation.

A Series Task Attributes Programming Reference Manual (8600 0502). Unisys Corporation.

A Series Work Flow Language (WFL) Programming Reference Manual (8600 1047). Unisys Corporation.

A Series Work Flow Language (WFL) Programming Reference Manual (form 8600 1047). Unisys Corporation.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

UNISYS

Help Us To Help You

Publication Title

Form Number

Unisys Corporation is interested in your comments and suggestions reguarding this manual. We will use them to improve the quality of your Product Information. Please check type of suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

Name Telephone number ( )

Title Company

Address

City State Zip code

Cut along dotted line

Tape

Please Do Not Staple

Tape

Fold Here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 817 DETROIT, MI

POSTAGE WILL BE PAID BY ADDRESSEE

UNISYS CORPORATION
ATTN: PUBLICATIONS
25725 JERONIMO ROAD
MISSION VIEJO, CA 92691-9826

86000494-000