DataGeneral

Customer Documentation

DG/L™ Runtime Library (AOS and AOS/VS) User's Manual

DG/L™ Runtime Library (AOS and AOS/VS) User's Manual

093-000159-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000159

Printed in the United States of America
Revision 02, November 1985

NOTICE

DG/L™ Runtime Library
(AOS and AOS/VS)
User's Manual
093-000159

Revision History:

Original Release - October 1978

First Revision - November 1980

Second Revision - November 1985

Effective with:

DG/L™ Rev. 2.10)

CONTENT UNCHANGED

The content and change indicators in this revision are unchanged from 093-000159-01. This revision changes only printing and binding details.

Preface

The AOS DG/L™ Runtime Routine Library

DG/L™ (Data General Language) is a high-level, ALGOL-like language used for both systems and applications programming. This manual contains all the routines in the DG/L™ Runtime Library that you can run on Data General's Advanced Operating System (AOS).

Basically, a *runtime routine* is a procedure written in either assembly language or DG/L™ that you call by name in your program. In many ways, runtime routines are similar to built-in routines like READ and WRITE (see the *DG/L™ Reference Manual*). The code for the runtime routine is loaded from libraries supplied with the DG/L™ product.

In this manual, we document each routine's format, arguments, and purpose. We also describe many features of DG/L™ that use runtime calls; e.g., cache memory, error handling, input/output choices, multitask programming, and overlays. Additionally, we describe the structure of AOS memory while your DG/L™ program is running.

Before using this manual, you should be familiar with the *DG/L™ Reference Manual* (093-000229). You should also have handy the *Advanced Operating System (AOS) Programmer's Manual* (093-000120). A knowledge of assembly language programming is helpful, but not necessary, to use these runtime routines.

If you're running a 16-bit DG/L™ program on the 32-bit AOS/VS operating system, this manual also applies, and parts of the *AOS/VS Programmer's Manual* (093-000241) will also be useful. However, with slight differences, the DG/L™ compiler runs the same under both the AOS and the AOS/VS systems. We note the differences in the manual where they arise.

If you're writing DG/L™ programs to run on both AOS and RDOS (Real-time Disk Operating System), this manual is helpful, but the RDOS version of the manual, *DG/L™ Runtime Library User's Manual (RDOS),* should be your primary source of information.

What's Inside

The bulk of this manual gives descriptions of each runtime routine and is intended for reference use. The runtime routines are grouped in chapters according to functionality and are listed alphabetically within chapters.

Chapter 1	details the format we use in this manual to present routines, and explains how you interpret the information under a routine's description. This chapter also summarizes important information about how to declare runtime routines in your program and how to define arguments to routines correctly.
Chapter 2	discusses the runtime memory structure, how to write assembly language modules to use with DG/L™ programs, and how to call DG/L™ runtime routines from assembly language programs.
Chapter 3	is made up of seven groups of runtime routines. Each group covers a set of general-purpose functions, such as string manipulation, use of the system clock, addressing memory, and obtaining AOS Command Line Interpreter (CLI) information.
Chapter 4	covers error handling, and includes a description of the ways you can use error handling procedures and labels in your programs.
Chapter 5	contains the runtime routines you'll use to create and maintain files, manage directories, and use devices.
Chapter 6	includes all four types of DG/L™ input/output (I/O) calls: file, console, cache memory, and shared page.

Chapter 7	covers the calls that manipulate and monitor AOS processes and programs; including communications between processes, and the chaining and swapping of program modules to and from disk files.
Chapter 8	outlines DG/L™ multitask programming. It explains the memory environment during multitasking, how you create and synchronize tasks, and the options for delaying and terminating tasks.
Chapter 9	contains all the DG/L™ multitasking calls, organized into sections according to their functions.
Chapter 10	gives you the operating instructions for compiling, linking, and executing a DG/L™ program. This chapter also lists all the optional switches you can use on command lines to obtain error listing files, warning messages, etc.
Appendix A	lists alphabetically all DG/L™ runtime routines with a short description of each.
Appendix B	documents three different kinds of internal DG/L™ routines: mathematical, conversion, and formatting.
Appendix C	lists all DG/L™ error codes and messages, and discusses AOS exceptional condition codes.
Appendix D	discusses the three assembly source files, DGLSYM.SR, DGLMAC.SR, and DGLPARAM.SR, included with the product tape. It shows you how to modify parameters to tailor the runtime environment to your special needs.
Appendix E	is a complete example of a DG/L™ multitask program.
Appendix F	lists all the AOS DG/L™ runtime routines that are compatible with DGC's Real Time Disk Operating System (RDOS).

Related Manuals

DG/L™ Reference Manual	093-000229
AOS Programmer's Manual	093-000120
AOS Link User's Manual	093-000254
AOS Macroassembler Reference Manual	093-000192

only for the AOS/VS operating system:

AOS/VS Programmer's Manual	093-000241
AOS/VS Link User's Manual	093-000245

and only for the RDOS operating system:

DG/L™ Runtime Library User's Manual (RDOS)	093-000124

Conventions Followed in this Manual

- Unless explicitly indicated otherwise, *AOS* always refers to *both* the AOS and AOS/VS operating systems. Also, references to the *AOS Programmer's Manual* refer instead to the *AOS/VS Programmmer's Manual* if AOS/VS is the program environment.
- *Main memory* always refers to directly addressable user memory space, not disk or tape.
- We often use the terms *procedure* and *routine,* and, where more precise, *subroutine* interchangeably.
- We use the term *link* to mean the process of creating an executable program file (a file with a .PR extension). This term technically refers to Data General's LINK utility, but the general concept of "linking" is the same as "binding" or "loading."
- The term *a call* refers to the name of a runtime routine and its associated formatted arguments as they appear in your source code. *A routine* refers to the actual runtime code that performs the job for you. However, where the meaning is clear, we sometimes use these two terms interchangeably.
- *RDOS* always refers to Data General's Real-time Disk Operating System.

Use of Color

For easier referencing, this manual is divided by colored-edge pages into four broad sections:

- two introductory chapters
- the general runtime routines
- a discussion and description of multitasking runtime routines
- operating procedures and appendixes

Reader, Please Note:

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where	Means
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: $\begin{Bmatrix}\text{required}_1\\ \text{required}_2\end{Bmatrix}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[optional]	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	Means
)	Press the NEW LINE or RETURN key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, in examples we use

THIS TYPEFACE TO SHOW YOUR ENTRY)
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

) is the AOS CLI prompt.

Contacting Data General

If you:

- Have comments on this manual -- Please use the prepaid Remarks Form that appears after the Index.
- Require additional manuals -- Please contact your local Data General sales representative.
- Experience software problems -- Please notify your local Data General systems engineer.

End of Preface

Contents

Chapter 1 - Elements of a Runtime Call

Chapter 2 - The Runtime Environment and Writing Assembly Language Runtime Routines

Chapter 3 - All-Purpose Runtime Routines

Chapter 4 - Handling Errors

Chapter 5 - Managing Files, Directories, and Devices

Chapter 6 - Input/Output Routines

Chapter 7 - Process Communication, Manipulation, and Monitoring

Chapter 8 - DG/L Multitask Programming: An Overview

Chapter 9 - Multitasking: The Routines

Chapter 10 - Operating Instructions

Appendix A - Alphabetic List of DG/L Runtime Routines

Appendix B - Internal Routines

Appendix C - DG/L Runtime Error Codes and AOS Exceptional Condition Codes

Appendix D - DG/L Files, Tables, and User Options

Appendix E - A Multitask Program

Appendix F - DG/L Runtime Routines Implemented Under Both AOS and RDOS

Tables

Illustrations

Chapter 1
Elements of a Runtime Call

The DG/L™ runtime library, consisting of about 200 routines, offers you an extensive set of functions and interfaces with the AOS and AOS/VS operating systems. DG/L runtime routines make most system calls available to you as statements and functions in the high-level language DG/L. You should use the *DG/L Reference Manual* with this manual. Also, with some DG/L runtime routines, referring to the *AOS Programmer's Manual* is very helpful.

NOTE: Unless explicitly stated otherwise, any reference in this manual to AOS or the *AOS Programmer's Manual* refers instead to AOS/VS or the *AOS/VS Programmer's Manual* if you're running your 16-bit DG/L programs on that 32-bit operating system. (16-bit DG/L programs, with a few exceptions noted herein, run the same under both systems, and the two reference manuals are similar in design.)

Overview

By including DG/L runtime calls in DG/L programs, you can

- invoke sophisticated machine operations through single, efficient statements
- create a DG/L program that works more interactively than it otherwise could
- process errors within a program either before execution terminates or without terminating at all
- control input and output with disk, tape, printer, and terminal devices
- considerably extend the capacity of memory by using cache memory management, disk file I/O, multitask programming, and the swapping and chaining of programs,

This chapter gives the basic rules for using DG/L runtime routines, such as how to declare them, the format of a call, how to declare data types, what kinds of arguments you may pass, the capacity of the *error label* argument, and the importance, in some cases, of referring to the AOS system call that the DG/L runtime routine references. We'll show these general aspects of using DG/L runtime routines by discussing first their declaration, and then by elaborating on a the parts of a runtime routine's description as they appear in Chapters 2 through 9.

NOTE: As stated in the Preface, we sometimes use the terms "a call" and "a routine" interchangeably in this manual when referring to runtime routines. Strictly speaking, *a call* is the name of a routine in your program code, including the call's arguments. Whereas *a routine* is the actual code (DG/L or assembly language) which comprises the routine.

Declaring a Runtime Routine

In DG/L, there are two types of routines, built-in and runtime. Built-in routines, as well as DG/L arithmetic and general functions, are documented in the *DG/L Reference Manual.* Both built-in and runtime routines perform specific jobs for you in a DG/L program; their real difference is that you do not declare built-in routines (the compiler recognizes them), whereas you *must* declare runtime routines.

Before calling a runtime routine, you must declare it as an EXTERNAL PROCEDURE. We don't remind you to do this each time we describe a call in this manual. See the programs in Chapter 8 and Appendix E for examples of runtime routine declarations.

There are two types of DG/L runtime routines: *statements* and *functions.* You declare them with different formats. Functions explicitly return a result through their operation, while statements return results (if any) in passed variable arguments.

The declaration format for a *statement call* is

```
EXTERNAL PROCEDURE name;
```

The declaration format for a *function call* is

```
EXTERNAL data type PROCEDURE name;
```

The data type is STRING, INTEGER, BOOLEAN, REAL, or BIT, and must match the data type of the returned value.

Built-in Routines

Some of the routines in this manual are actually *built-in* routines; that is, the DG/L compiler recognizes them. (These routines are also documented in the *DG/L™ Reference Manual.)* Because they're recognized by the compiler, you shouldn't declare them. We've marked these routines with a (B) (for "built-in") right after the (boldface) name of the routine. If you accidentally declare a built-in routine, the compiler will expect a user-written module with that name. If one doesn't exist, you may get Link errors, since, for some built-in routines, the compiler generates special internal names.

What A Routine Does

The description of each runtime call in this manual begins with the name of the call and a sentence (in bold typeface) explaining what it does. In some cases this definition is complete, but many runtime routines include a "Notes" section that supplements this summary heading.

The Format of a Call

Following each heading description, we show the format you will use for the call in your source program. As stated earlier, DG/L runtime calls are either statements or functions. (A few of the calls in this manual are simple declarations, and are noted as such.)

Statement calls take this format

```
routine name [(arguments)];
```

All arguments must be put inside parentheses, and the routine's name must be uppercase. We use brackets, [ ], in a routine's format line to indicate optional arguments; don't enter the brackets in your code.

Function calls take the format of a DG/L assignment statement

```
variable := routine name [(arguments)]
```

Function calls can also be used as items in an arithmetic expression, or as arguments to a function or statement call:

```
OPEN (1,(GETCOUTPUT),ERRLAB);
```

Here, the function call GETCOUTPUT, which gets the output filename, functions as an argument in the built-in DG/L routine OPEN. (Another example of such nesting appears in this chapter under "The Error Label Argument.")

Notice that you enclose the embedded routine's name in an additional pair of parentheses. These parentheses ensure that the value GETOUTPUT returns will be the value returned, and not the address of the routine itself. If the embedded routine takes arguments, the parentheses around its argument list are sufficient to have its value, not its address, passed.

If you use a variable to receive the result of the function call, such as the s in the call

```
s := GETCOUTPUT [(error label)]
```

the data type of the variable(s) must be compatible with the returned value. In this case, you would declare s as a STRING. (See section of next chapter, "Internal Structure of Data," for a list of data types.) This manual uses the following conventions for indicating variables:

i	integer variable
s	string variable
bool	boolean variable

The format for a declaration is

```
routine name;
```

Arguments

In each runtime routine's description, after presenting the routine's format, we define the call's arguments. Each argument accepts certain data types, expressions, labels, etc., and passes or returns a value.

Generally, arguments accept a wide range of DG/L program elements, such as:

- Statement labels, to which the program can branch
- Constants and variables
- AOS parameter packets which the call passes to a system call
- File names or numbers
- Pointer expressions, arrays, etc.

As a general guideline, the following rules pertain to arguments:

- *Numerical values* can be variables, constants, or expressions, unless otherwise indicated. *Integer values* usually means single precision, with exceptions noted. In arguments which return values, such as in reporting errors, you must use a *variable,* rather than a constant or expression.

- *String values* can be constants or variables, unless otherwise indicated.

- *Arrays* can be any type of array, unless otherwise indicated. Of course, the declaration of the array must create the required number of words as a storage block.

Refer to the *DG/L™ Reference Manual* if you need to review types of DG/L program elements. For example, remember that pointer expressions act like variables.

A Note about AOS File Names

Some arguments you use will be names of files. AOS expects filenames you pass it to be strings terminated by a null, "<0>". DG/L string constants always have a null after the end of the string constant, but variables must have it explicitly set. Some examples

```
S := GETACL("CROWLEY");                    Good
S := "CROWLEY"; S := GETACL(S)             Bad
S := "CROWLEY<0>"; S := GETACL(S);         Good
```

A Note about Parentheses

In DG/L, parentheses around a constant, variable, or procedure cause it to be evaluated and the result used. Thus, in calling a runtime routine, parentheses around a variable argument cause the *value* of the variable to be passed instead of the variable itself. This way the call can't modify the variable. In the same way, you parenthesize constants passed to a routine to prevent the routine from modifying the constant.

When passing a function name as an argument to a routine, the name causes the address of the function to be passed. Parentheses around the name or around the argument list attached to it cause the function to be evaluated and the resulting *value* passed:

```
OPEN(1,MYNAME(3,I));
```

In this example, parentheses around the arguments 3,I suffice to ensure that the value of the call MYNAME is passed to OPEN.

The Error Label Argument

Many DG/L runtime routines have an optional argument, *error label.* This argument usually accompanies runtime calls to the operating system and must be a label that directs control to another part of the program. Error labels can prevent errors from terminating a program, provide additional processing before termination, or allow you to branch to somewhere else in the program.

For example, this call to ALLOCATE contains the error label optional argument

```
ALLOCATE (pointer,size [,error label] );
```

This call reserves a specified number of words in memory and returns their location in a pointer.

If you want to allocate all but 1000 words of the remaining memory in a program, you might use the following call

```
ALLOCATE (P,MEMORY - 1000,IERR);
```

(Notice how the runtime call MEMORY appears here as part of an expression.) If the expression returns a negative value (because of insufficient memory), an error condition results, and control branches to the statement label IERR.

At IERR you might write code to generate a message. You could also provide a series of statements to read the error code, correct the condition, and return to the subroutine. In the case of ALLOCATE, you might call FREE to allow re-allocation of a block of words your program no longer needs.

If you don't provide an error label (where the option exists), or other error interruption, and a fatal error occurs, your program terminates and control returns to the operating system.

See Chapter 4, "Handling Errors," for a complete outline of routines and options for handling runtime errors.

Examples in This Manual

After defining a call's arguments, we show you an example of the call as it might look in your source program. These examples assume that you've included code that interfaces with the call. For instance, we don't usually show the declaration portion of a program module (where you declare variables used later in runtime routines). And we don't show examples of code at an error label's branching destination, or pairs of runtime calls you might often use together. We try to cover all these important details in the introductory material of chapters and in explanatory chapters such as this one.

For most of the runtime routines, we give one example. Remember that this example will necessarily not cover all possible uses of a routine. Careful attention to the definitions of arguments and "Notes" under a call's description, as well as a good knowledge of DG/L and AOS, will extend your capacity to use any runtime routine to its maximum advantage.

Notes

After the "Example," for some routines you'll find additional information under "Notes." These notes provide information ranging from optional hints to essential conditions for using a routine. They supplement a call's description, point out other routines that you regularly use with that routine, or explain variations you can try.

Error Conditions

Under each routine's description we list possible error conditions and their meanings, including the mnemonic codes for AOS and DG/L errors.

For example, under the routine CLOSE, the following error codes appear

ERIBM Inconsistency in block allocation map.

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

These error conditions can result from the system call to ?CLOSE. Other routines may return DG/L runtime error codes, as well as system codes. All AOS errors have a mnemonic name beginning with ER and DG/L runtime errors begin with AI or AE.

To find the applicable error codes for AOS errors, look up each type; e.g., FILE SYSTEM, SYSTEM CALL, and CHANNEL-RELATED, in Appendix A of the *AOS Programmer's Manual.* That appendix gives the mnemonic name and the meaning of each AOS exceptional condition (error) code. For the numerical (octal) value of an AOS error code, refer to file PARU.LS in Appendix E of the *AOS Programmer's Manual.* For DG/L error codes and messages, see Appendix C of this manual.

You may want to use these error codes to define error-handling procedures in conjunction with routines described in Chapter 4, "Handling Errors."

References to AOS System Calls

Most of the routines in the DG/L runtime library make system calls to AOS. The system calls perform input and output, change programs and tasks, manage memory, and communicate with peripheral devices. In this manual, the description of a routine that calls the system includes a last section called "References." For example, under the call OPEN you'll find the following "Reference"

?OPEN (System call)

The OPEN routine uses the AOS system call ?OPEN.

To follow up these references, look up the system call in the index of the *AOS Programmer's Manual.* That manual gives you further information about the process, contents of accumulators, and side effects of the system or task call. In many cases, this information is not essential for using a DG/L runtime routine. However, in some cases the *AOS Programmer's Manual* provides necessary tables of bit codes, parameter packets, status words, or other useful information.

Examples of Calls to Runtime Routines

The runtime routine SETCURRENT resets the current length of a string variable. The format of the call is

```
SETCURRENT (string,length);
```

The routine sets the length of the variable named in string to the number of bytes specified in the integer length. In an actual program, the call might look like this

```
EXTERNAL PROCEDURE SETCURRENT;
STRING (25) CHARSTRING;
.
.
.
SETCURRENT (CHARSTRING, 20);
```

This call would set the length of the string CHARSTRING to 20 bytes until the next call to SETCURRENT or a string operation.

The runtime routine MEMORY returns the number of words of memory currently remaining for the user. Its format is

```
i:= MEMORY
```

i is an integer variable that receives the number of words. In a program, the declaration of the call looks like this:

```
EXTERNAL INTEGER PROCEDURE
MEMORY;
```

A call to MEMORY might look like this:

```
LEFT := MEMORY;
```

You might also use MEMORY as an expression in your program; for example, in a conditional statement

```
IF MEMORY < 1000 THEN WRITE (1,
"NOT ENOUGH MEMORY");
```

This DG/L statement calls MEMORY and compares the value it returns to 1000. If the value is less than 1000, the condition is met and the program produces the message. The actual value that MEMORY returns is not stored.

End of Chapter

Chapter 2
The Runtime Environment and Writing Assembly Language Runtime Routines

This chapter is made up of four sections: "The Runtime Environment," "Writing Assembly Language Runtime Routines," "Using DG/L Runtime Routines in an Assembly Language Program," and "Internal Structure of Data." This first section, "The Runtime Environment," discusses the allocation and management of memory during execution of a running program. We explain the structure of stack memory, how the stack records task information, and how the different kinds of non-stack memory operate dynamically under AOS single-task and multitask programs.

This section should help you use DG/L runtime routines effectively and manage memory efficiently. If you're writing your own assembly language routine to use in a DG/L program, you'll need a thorough understanding of the runtime environment (as well as information in the next section, "Writing Your Own Runtime Routine").

Many of the terms and concepts associated with the DG/L runtime environment necessarily relate to AOS concepts, so Chapter 3 of the *AOS Programmer's Manual,* "Memory Management," is also helpful to understanding the AOS DG/L runtime environment.

NOTE: If you're running 16-bit DG/L programs under the AOS/VS operating system, you should consult that system's reference manual for supplementary information about the runtime environment.

The Runtime Environment

The columns in Figure 2-1 represent a DG/L user's memory locations under AOS. They show the memory space as a vertical column with addresses increasing toward the top of the illustration.

Memory Allocation after Loading

The two columns show the differences between an executing single-task program and an executing multitask program.

When you first load a program into memory, it occupies several memory areas that remain stable throughout execution (see Figure 2-1). Here's a brief description of each of these areas:

- The *operating system* uses the first 20_8 addresses and areas outside the user address space (not shown).
- *Page zero,* in locations 20_8 to 377_8, contains information about the program. It includes pointers to other areas of memory, to some runtime routines, to the boundaries of the user stack, and to task state information. (See later section "Page Zero Pointers" for more information.)
- The *User Status Table* (UST) and *Task Control Blocks* (TCBs) start at location 400_8. The LINK command line (see Chapter 10) specifies the number of TCBs the program requires (the default is 1).
- The next memory area contains *OWN variables;* for example, DG/L can store OWN array pointers and OWN string specifiers here, above the TCBs and UST. These pointers reference OWN arrays and strings stored in the "heap" memory area (see next section).
- The next higher addresses of your memory space hold GLOBAL variables (see "Internal Structure of Data," in this chapter, for a definition of GLOBALs).
- The module DGLINIT, which initializes your DG/L program, occupies the locations after the USTs, TCBs, and OWN and GLOBAL variables. The location .LLOC marks its end, and the end of unshared memory in a linked program.
- The system allocates the user program (executable code), runtime routines, and user-written runtimes as shared memory at the top of memory addresses, starting at a 1K-word boundary.

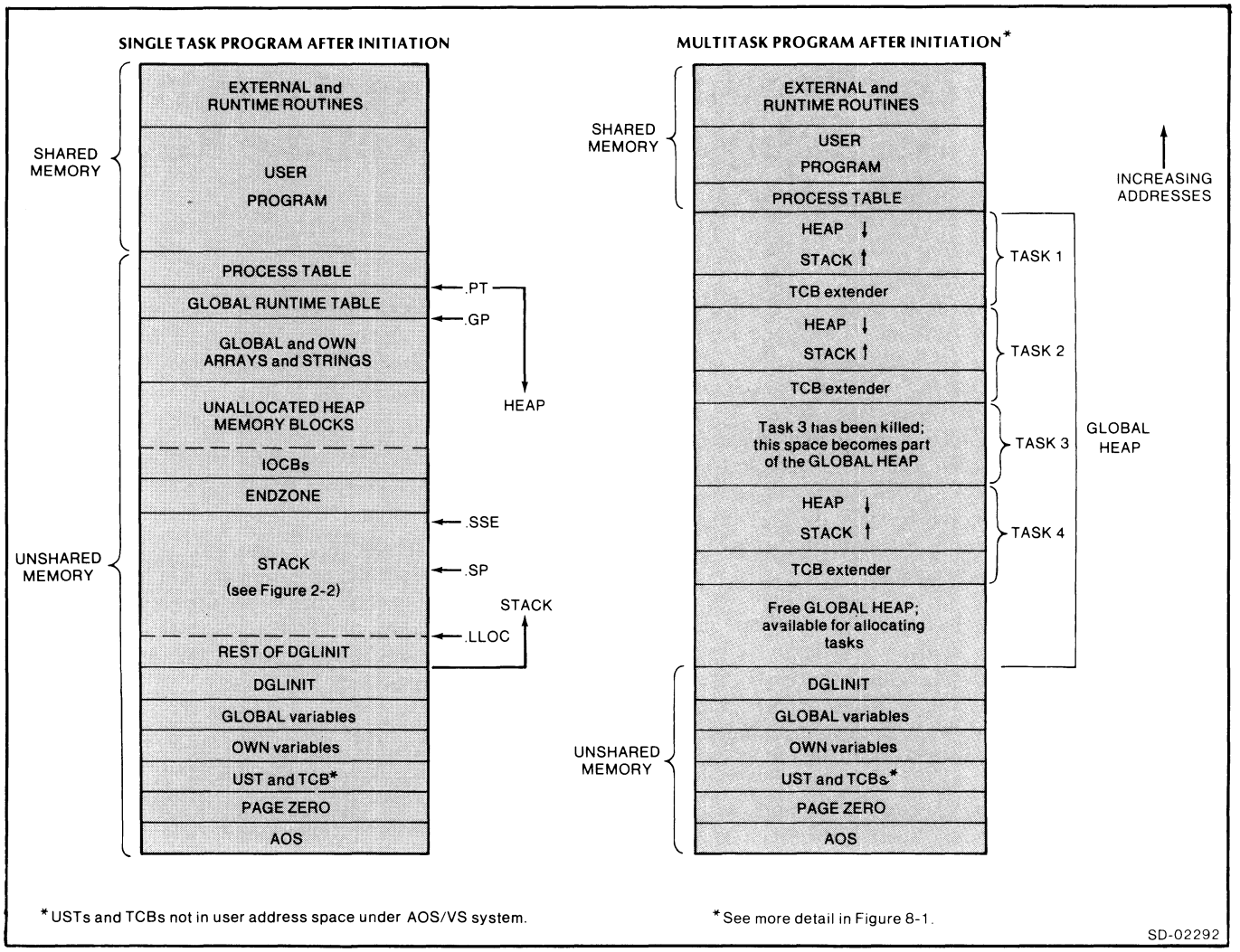

Figure 2-1. The Runtime Environment

Memory Allocation after Initiation

The program's execution begins with DGLINIT, which creates areas for dynamic allocation, releases some of its own code to the stack after it executes, and calls your program as a subroutine. If your program terminates normally, control returns to DGLINIT at the end of execution. DGLINIT partitions your memory space into data and table areas, as follows:

- A single *Process Table* contains information relevant to an entire DG/L program or process. The page zero pointer .PT points to word zero of the table, with displacements of its contents in ascending order from .PT. To see the contents of the Process Table type or print DGLSYM.SR, a file on your system.
- A *Runtime Global Table* contains information specific to eack task in a DG/L program, including input/output information. In the single-task environment, it occupies the locations immediately below the Process Table. In multitasking, you allocate it dynamically as tasks become active. A separate table exists for each task. The page zero pointer .GP points to the currently active task's information. Again, see DGLSYM.SR, a file defining DG/L symbols, for the contents of the Runtime Global Table.
- The *heap* begins at the address below the Runtime Global Table, and grows downward into lower addresses as your program requires data space. The heap stores allocated blocks of memory that are global to the entire program, and remain in memory from the time of declaration until the program completes execution. Allocated heap storage is useful for creating linked lists, trees, and other dynamic data structures. It allows you to define data areas accessible with pointers.

The heap also stores OWN and GLOBAL arrays and string variables.

- *Input/Output Control Blocks* (IOCBs) are allocated dynamically from the heap, as needed. The IOCBs contain information for formatting READ or WRITE commands. Unlike OWN and GLOBAL data, these blocks are released once the formatted WRITE is complete.

- To protect the downward-growing heap from a stack overflow, DG/L maintains a blank area called the *Endzone* between the stack and heap. A stack overflow error occurs when you attempt to make the stack pointer greater than the stack limit. DG/L reports the error and terminates the program.

- The *stack* begins above the initializer, DGLINIT, and grows upward. The page zero pointer .SP indicates its current highest address. (See "The Runtime Stack" in this chapter for a complete discussion of the stack.)

Changing the Size of Unshared Memory Area

As shown in Figure 2-1, a user's memory space under AOS divides into shared and unshared areas. (See the *AOS Programmer's Manual,* Chapter 3, for details on shared and unshared memory management.) The initializer DGLINIT, using the DG/L symbol .NMAX, defines the upper limit of the unshared area to be 32, the maximum number of unshared pages.

However, to reduce system use of main memory for programs not needing 32 pages of unshared memory, you may want to modify .NMAX to reduce its size. See Appendix D, under "User Options," or Chapter 6, under "Shared Input/Output," for directions on redefining .NMAX.

Page Zero Pointers

Page zero contains indexes to the currently loaded program and runtime stack. It holds pointers to the beginning of the program, to stack areas, to the Input/Output Control Blocks, and to the task frames where task information resides. The entire program can access the information and pointer values in page zero, which are

Pointer	Address Indicated
.SP	Pointer to current top of the user stack
.FP	Pointer to start of the current stack frame
.SSE	Pointer to the top of available stack space (per task)
.RP .RSV	Temporary page zero locations (per task)
.PT	Pointer to beginning of the Process Table
.GP	Pointer to the beginning of the Runtime Global Table (per task)
.ND1 .ND2	Temporary words used for emulating ECLIPSE® computer instructions on the NOVA® computer (per task)

Through page zero and the User Status Table (both are accessible from any part of memory) you can locate the currently active code, variables, task information and global information. Using these tables and pointers, you can address one part of the program from another, and chain tasks and subroutines in a logical order. You can stop execution of one subprogram or task temporarily and then reactivate it.

NOTE: Under AOS/VS the UST and TCBs aren't accessible.

How Control Transfers to a Runtime Routine

When a program calls a DG/L runtime routine or a user-written routine, the program's execution temporarily suspends. In order to later resume execution of the program, the call must preserve information about the program.

A call to a runtime routine starts a series of operations. The calling routine loads the stack pointer value into AC2 and the addresses of the call's arguments go to the top of the stack, in reverse order.

Control then passes to the routine. All DG/L-compiled procedures and runtime routines start with a SAVE instruction that allocates a five word frame and stores the current contents of the accumulators, the old frame pointer, and the return and carry. (See Figure 2-2.) The same command allocates stack space for the routine. Execution of the routine's code then begins.

When the routine completes execution, a RTN command restores the accumulators, releases the frame, and resumes execution of the calling program at the point from which the call was made.

If you're using routines in the DG/L runtime library, all these operations are invisible. However, if you want to insert an assembly language subroutine into a DG/L program, you need to follow these steps in detail. "Writing Assembly Language Runtime Routines" explains how to combine assembly language routines with a DG/L program.

The Runtime Stack

DG/L procedures allocate memory dynamically; runtime routines and programs can be re-entrant and recursive; and one or more tasks can use the same routine concurrently. The purpose of the *runtime stack*, then, is to temporarily store significant values between executions of a procedure or subroutine.

The stack is part of your unshared memory and operates on a last-in, first-out (LIFO) basis that preserves the nested block structure of a program. Each task has exactly one stack (a single-task program has one stack), and runtime routines use the same stack as DG/L programs.

The stack preserves three kinds of information

- Five contiguous words of information needed for the current subprogram and the state of its calling program (see Figure 2-2).
- Local variables (all but GLOBAL and OWN).
- Temporary values that must be stored while processing expressions.

The stack follows the logic of block structure for all kinds of data.

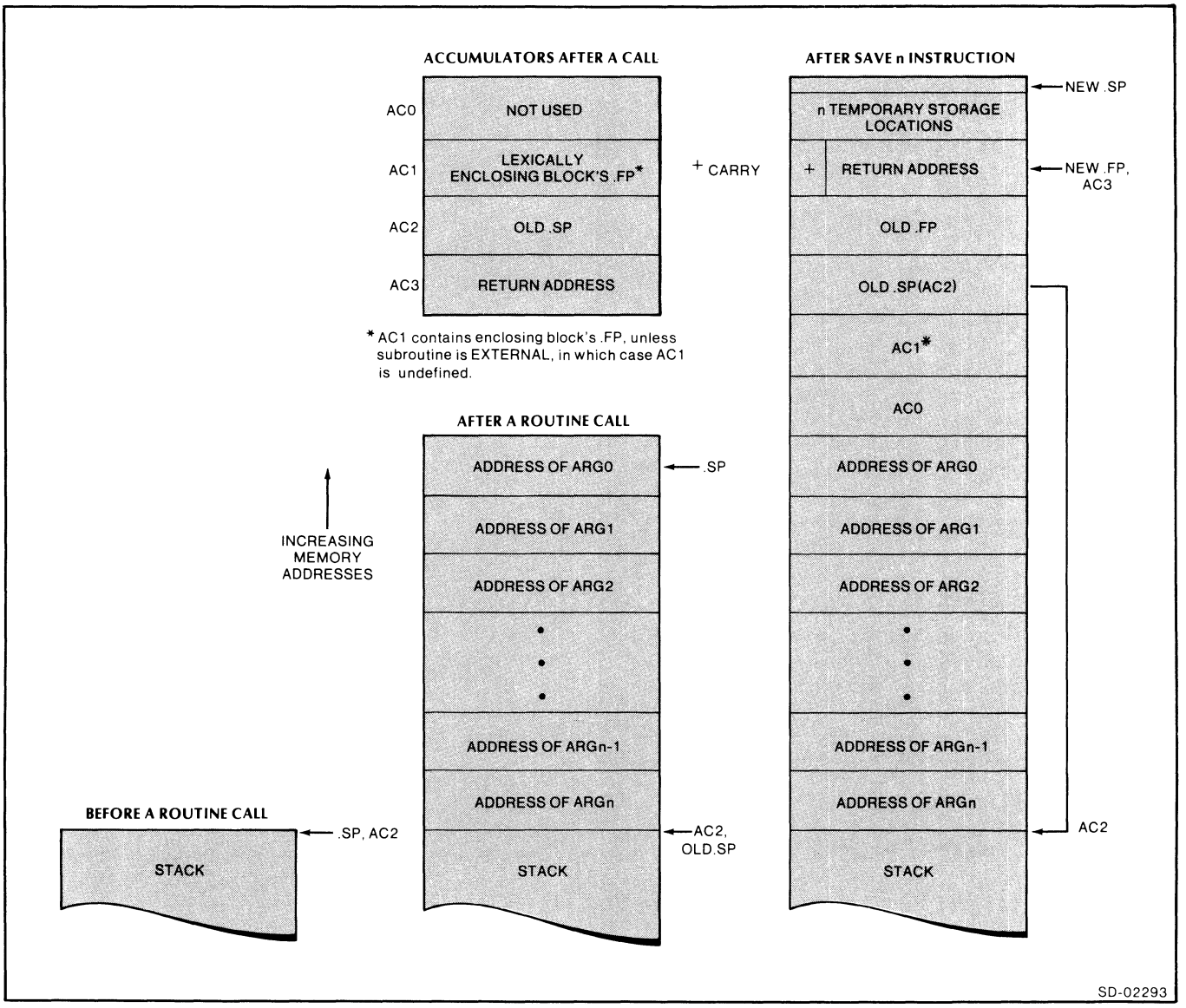

Figure 2-2. A Runtime Stack Frame, Stack Storage, and Accumulators at Various Stages of a Runtime Routine Call

Figure 2-2 represents the configuration of the runtime stack, and relevant accumulator references to it. In studying this figure along with the following text, remember that this illustration is necessarily a static view of a *dynamic process*. Stack space is continually being allocated, freed, and re-allocated, on a last-in, first-out basis.

Stack Frames

For each new call to a subroutine a new *stack frame* exists. The stack frame is a block of five contiguous locations, and then storage locations for variables and temporary values (see Figure 2-2).

When a program calls a subroutine, a new stack frame (the subroutine's) supersedes the caller's. When the subroutine's execution is completed, the code releases the subroutine's stack frame and the arguments, and returns control to the caller's address stored in the subroutine's frame. (See Figure 2-2.) If an internal procedure needs to reference variables declared in an enclosing procedure, it references the data using the (lexically) enclosing procedure's frame pointer, which is passed to it in AC1, so is located in FAC1 of its SAVE block. This enclosing procedure might not be the immediate caller of the internal procedure.

Variables

DG/L allocates local variables on the stack during program execution. It locates them by using the frame pointer (.FP) of the block that declared them.

Local variables are located in the stack frame, and in some cases (strings, arrays), the frame is increased in size when the variable is initialized, since the actual size may not be known until execution time. Variables for subroutine calls are pushed on the stack before the call and are automatically released (popped) after the subroutine finishes execution.

The stack maintains block structure by adding the data for the subroutine on top of the calling routine's data, and returning to the calling routine's data when the subroutine's data is released.

The Multitask Runtime Environment

Chapter 8, "Multitasking: An Overview," contains a detailed illustration and complete discussion of multitasking.

Figure 2-1 shows the multitask environment after initiation. Each task contains areas similar to a single-task program, except that in the multitask environment, DGLINIT first creates a *global heap* out of all non-code memory. Then, each task has its own local stack and heap area allocated from the global heap.

Like single-task heap data, the global heap grows downward from the top of unshared memory (see Figure 2-1). Each task has a Global Runtime Table, a Task Control Block (TCB), a TCB extender, a runtime stack, and a local heap, all allocated on a single block. When the task isn't executing, the TCB extender holds the values for its page zero pointers and the floating point unit.

In the multitask environment, DG/L sets the default size for each task during runtime. But you may allocate a different size to each task (see Chapter 8). When a task ends, its area of memory returns to the global heap of available memory blocks for re-allocation, but the blocks of other tasks remain in their original locations until they finish executing. When a new request to allocate space for a task fails to find a large enough free area, DG/L merges *adjacent* free blocks (see Figure 2-1).

Again, please refer to Chapter 8, "Multitasking: An Overview," for complete details on multitask programming.

Writing Assembly Language Runtime Routines

Before reading this section on writing assembly language routines to use in DG/L programs, you should read the previous section, "The Runtime Environment."

DG/L allows you to write assembly language routines, and to execute them as external procedures in a DG/L program at runtime. You might want to do so to run part of your program that must execute many times, and would thus benefit greatly from the speed that hand-optimized assembly language code provides.

Remember that you can also write DG/L language external procedures that also execute at runtime, as do assembly language routines and the routines in this manual.

To write an assembly language routine, you must know the instruction set for your DG computer, documented in the *AOS Macroassembler Reference Manual,* and the particulars of the DG/L runtime environment described earlier in this chapter.

Assembling and Linking an Assembly Language Routine

After creating an assembly language source file, assemble it using the Macroassembler MASM. An object file with a .OB extension will result.

Then create an executable program by using LINK, placing the name of the assembly language file in the LINK command line with your DG/L program (main program and external procedures). This creates a file with a .PR extension; for example,

```
X LINK/NSLS DGL_PRCG DGL_SUB_1
ASSEM_PROG [DGLIB]
```

Refer to Chapter 10 of this manual, "Operating Instructions," for details about linking, and refer to the *AOS Macroassembler (MASM) Reference Manual* for assembling instructions.

The Declaration and Format of Your Routine

In your program, you declare an assembly language runtime routine in the same format as a DG/L routine

```
EXTERNAL PROCEDURE MYFUNC;
```

or

```
EXTERNAL data type PROCEDURE MYFUNC;
```

Then call it with

```
MYFUNC [(arguments)];
```

or

```
identifier := MYFUNC [(arguments)]
```

Remember, in routines that return a value, you must declare the identifier to be the same data type as data type in the procedure's declaration; e.g., INTEGER, BOOLEAN, STRING, REAL.

Steps in the Operation of Your Assembly Language Routine

Assembly routines can be recursive and re-entrant. At any given time, a series of active and inactive stack frames may exist for separate tasks and routines. This chain of frames allows you to return to higher levels of the program in block order.

To interface with your DG/L program, the call to your assembly language routine must follow the format and rules that apply to DG/L routines. Like a call to a DG/L runtime library routine, the calling of your assembly language routine must include the following sequence of operations:

1. Pushes the addresses of arguments onto the stack, in reverse order.
2. Transfers control to the routine.
3. Allocates a stack frame for temporary storage.
4. Preserves information (.SP and RETURN address) about the calling program or routine.
5. Executes the routine.
6. Releases the frame and its storage space.
7. Returns to the calling program by following the pointers stored in the released stack frame.
8. Removes the argument list (addresses) from the top of the stack.

Useful Macros and Instructions

A *macro* is a named sequence of instructions that you can include in an assembly language routine by writing the name of the macro as if it were an instruction. In this section we'll explain the use of the macros TITLE, ENTRY, and R.T.N, and the instruction SAVE. By using these four words, you set up and return from an assembly language routine in the same way DG/L automatically handles runtime library routines.

When you call your own assembly language routine, DG/L automatically pushes the addresses of the arguments onto the stack and transfers control to the routine (steps 1 and 2 above). Your routine should include the macros TITLE, ENTRY, and R.T.N and the instruction SAVE in the following manner:

```
TITLE MYFUNC
ENTRY MYFUNC
SAVE  n
.
.
.
R.T.N
```

TITLE initializes an environment for the ENTRY macro; ENTRY creates an entry point; SAVE allocates a stack frame for temporary storage, and preserves information (.SP and RETURN address) about the calling program or routine (steps 3 and 4 above); and R.T.N releases the frame and its storage space and returns to the calling program (steps 6 and 7 above). We define these macros in the file DGLMACS.SR (see Appendix D and the current DGLMACS.SR file on your product tape).

Let's go through these steps in the operation of a runtime routine in more detail. Control transfers to your routine after the system loads the contents of the current .SP into AC2, and pushes the addresses of all arguments onto the stack *in reverse order* (see Figure 2-2).

If the called routine is a locally-declared procedure, the system loads AC1 with the frame pointer of the enclosing procedure's most recent activation. For EXTERNAL procedures, the contents of AC1 aren't defined (see "EXTERNAL Data" section).

A routine's arguments are numbered left to right, as they appear in the call, starting with 0. For example, in a call to SUBR,

```
SUBR (A,B,C,D);
```

A is ARG0; B, ARG1; C, ARG2, and D, ARG3. In a call to FUNC,

```
A := FUNC (B,C,D);
```

A is ARG0, B is ARG1, etc.

After pushing the arguments onto the stack, the system passes control to your routine, using EJSR, or ?RCALL if the routine is in an overlay (see Chapter 10 under "Overlays"). Figure 2-2 shows the stack and accumulator addresses before and after a subroutine gains control.

We already mentioned the purpose of the macro TITLE. Your routine must use the macro ENTRY to define its entry point. ENTRY creates a label with the routine's name on the first instruction. It also generates a .PENT for that routine's name that enables your DG/L program to reference the name. Your routine can have more than one entry point defined by the ENTRY macro.

Using the SAVE Instruction for Temporary Storage

The SAVE instruction (SAVE macro on the NOVA) creates a stack frame. This frame stores information about the calling task and provides storage for local variables and the routine's data manipulations.

This stack frame consists of the contents of the accumulators, the calling program's .FP, the carry, and the return address, plus the new temporary storage area. Figure 2-2 shows the stack after a SAVE instruction.

Once the old .SP is stored, SAVE updates .SP and .FP, and puts the new .FP into AC3. SAVE then allocates locations beyond the new .FP as a temporary data stack. It then updates .SP to point to the last location in the stack.

Using SAVE protects the area above .FP when the routine suspends execution. When execution reaches R.T.N, it releases the frame and temporary stack.

The numerical argument n in the SAVE command (see above section "Useful Macros and Instructions"). allocates the total number of words you want for temporary storage. The called routine's stack begins one word past the word pointed to by .FP and extends upward in memory for n words (see Figure 2-2). To determine the number of words you need to store each data type in your routine, see "Internal Structure of Data" at the end of this chapter.

Using Symbolic Names

You can assign symbolic names to data addresses that are displaced from the frame pointer, as in the following examples:

```
SINT = 1            ;1 PAST .FP (INTEGER)

SINT2 = 2           ;2 PAST .FP (INTEGER(2))

SPTR = SINT2+2      ;4 PAST .FP (POINTER)
```

Once you've assigned symbolic addresses, you can use them to store and reference data items on the stack. For example, to retrieve a single-word integer and save a copy of it locally, you could use

```
LDA 2, @ARG1, 3     ;GET FROM ARG LIST

STA 2, SINT, 3      ;STORE AS TEMP ON STACK
```

To dynamically store values that your routine creates, you can also use the instructions (macros on NOVA computers) PSH, POP, and MSP.

Temporary Storage: Making a Routine Re-entrant

Since each task has a separate stack, if you use these macros and the SAVE instruction, and use the stack frame for all local storage, your assembly language routine, like the routines in the DG/L runtime library, will allow for recursion and re-entrance. When you place the SAVE command at the beginning of the routine, the routine dynamically allocates a new storage area on the stack for each separate activation of itself. Also, space for variables doesn't occupy memory when the routine isn't currently active.

For more information about macros, and for general information about writing assembly language code, see the *MASM Programmer's Reference Manual.* When you need definitions and examples of instructions, see the *ECLIPSE-Computers Programmer's Reference Manual.*

Reference to Arguments

Once your routine has done a SAVE, there's an easy way to check how many arguments were passed in a DG/L call to your assembly language routine. The calling routine loaded AC2 with the stack pointer before pushing the argument addresses (if any). The SAVE retained this value in AC2 and also saved it on the stack. Subtracting this number plus five (for the SAVE block) from the current frame pointer (in AC3 after the SAVE) yields the count of argument addresses pushed.

The SAVE instruction also loads the current (new) frame pointer (.FP) value into AC3. The RTN instruction (or R.T.N macro), when used after a SAVE, returns the previous frame pointer to AC3. As the examples here show, it's good practice to keep the .FP in AC3, allowing easy indexed access to both current stack frame locations (positive offsets) and to arguments to the call (negative offsets).

Using Symbols to Reference Arguments

Within your routine, you can reference the arguments by using the symbols ARG0, ARG1, ARG2, etc. These symbols represent the arguments as negative displacements from .FP. A file, DGLSYM.SR, in Appendix D and on your product tape, defines the offset values of these symbols.

Your assembly language code can thus refer to an argument in the following form:

```
LDA 2, ARG3, 3
```

This instruction loads the *address* of the *fourth* argument into AC2 (first argument is ARG0). .FP, which is often kept in AC3, is the point of reference. The instruction

```
LDA 2, @ARG3, 3
```

loads the first 16-bit word at the address specified by the fourth argument into AC2.

Terminating the Routine

The R.T.N macro terminates the routine and returns control to the calling program. (On ECLIPSE computers, this macro simply produces a RTN instruction.) It returns control to the program as follows:

- reloads the accumulators and carry from the current frame
- reinstates the .SP and .FP to their previous values
- returns to the calling program at the address given in the return address in the frame

Before transferring control to the routine, the compiler generated code to load the current .SP into AC2 and to push the arguments onto the stack. The R.T.N macro simply frees the five-word frame and any temporary storage words that follow it.

The compiler-generated code then frees the space occupied by the pushed arguments by resetting .SP to its previous value. The code (on an ECLIPSE computer) is as follows:

```
LDA    2,.SP                ;SAVE current value
<push the arguments>
EJSR   name
STA    2,.SP                ;restore value
```

Handling Errors in Your Runtime Routine

Your assembly language routines, like DG/L routines, can use the DG/L error handler (.RTER). See Chapter 4, "Handling Errors," for a complete discussion of default error handling. The error handler can process fatal and non-fatal errors within the routine, and either continue the routine's execution, or transfer control to a statement label in the calling program or to an error intercept location.

If one of the arguments in your call is an error label, you should use .RTER to pass control to the error label.

You may also use the error handler to invoke a pre-set user-written routine to process the error before returning to the calling program, instead of simply returning an error code.

For non-fatal errors, you can use .RTER to produce an error message and continue the subroutine's execution.

The Error Code

The runtime routine .RTER can accept operating system, DG/L, or user-defined error codes. When an error occurs, your code must place the error code in the error accumulator, ERAC. (See the file DGLSYM.SR for a definition of ERAC). And your code must also load the error label code into AC1.

A fourth type of error is "error text," in which FRAC contains the word address of a text string to be output.

The Error Label

If your routine's call provides an error label as an argument, you must load a label indicator into AC1 when an error occurs. The code you enter into AC1 must be one of the following:

Code	Meaning
1B0	No error label is provided
-2	ARG0 is the error label (if present)
-1	ARG1 is the error label (if present)
0	If label is present, there is an even number of arguments, and the label is the last one
1	If label is present, there is an odd number of arguments, and the label is the last one
n	If label is present, it is argument n (counting from 0)

These codes apply also to routines where the error label is an optional argument. Codes 0 and 1 are unambiguous only where the error label is the only optional argument.

When a system call creates an error condition, the system puts the error code into ERAC. Chapter 9 of the *AOS Programmer's Manual* defines the error accumulator for AOS. (Other operating systems may use different accumulators.)

Examples of Calls to .RTER

If you defined a routine MYFUNC(PRT1,I) with no error label, the routine might use the sequence in Figure 2-3 when an error occurs. The series of instructions in that figure prints an error message provided by the subroutine in ERAC, then transfers control to .RTER, which will terminate the program. (In the example, we use RT.ERR, a DG/L macro which calls .RTER. All DG/L macros are defined in file DGLMACS.SR., in Appendix D and on your product tape.)

If you defined MYFUNC with an optional error label (ERR1), the call might look like this

```
MYFUNC (PRT1,I,ERR1);
```

If MYFUNC makes a system call, the exception return should branch to the label, as shown in Figure 2-3. This sequence passes control to .RTER and loads the value 2 as the error label indicator into AC1. The error handler returns control to the program at ERR1.

Like an error label you define using DG/L routines, your assembly error label might be followed by code to read the error's value, process, and terminate under ERRINTERCEPT, process without terminating with ERRTRAP, or branch elsewhere in the program. (See Chapter 4, "Handling Errors," for details on these DG/L routines.)

```
/*MYFUNC without an error label*/

            .TXTM    1              ; MAKES SURE "MESSAGE" IS
                                    ; STORED LEFT-TO-RIGHT
 ERET:      JSR      ERET1          ; TRICK TO GET ADDRESS OF MESSAGE
                                    ; INTO AC3
            .TXT     "MESSAGE"
 ERET1:      MOV      3,FRAC        ; PUT POINTER TO "MESSAGE" IN
                                    ; FREE AC
            XLDA     ERAC,= ECTUT   ; "USER-DEFINED TEXT" IN ERAC,
                                    ; PLUS FATAL ERROR
            SUBZR    1,1            ; WE HAVE NO ERROR LABEL
            RT.ERR                  ; CALL THE ERROR HANDLER .RTER
                                    ; NO R.T.N NEEDED BECAUSE ERRORS
                                    ; ARE DEFINED AS ABSOLUTELY FATAL

/*MYFUNC with an error label*/

 ERET:      SUBZL 1,1              ; AC1 <-1 MEANS ODD # OF ARGS
            RT.ERR                 ; ERROR RETURN
            R.T.N                  ; IN CASE NON-FATAL
```

Figure 2-3. MYFUNC With and Without An Error Label

Example of an Assembly Runtime Call

To conclude this section on how to write assembly language routines to interface with DG/L programs, we show you a typical DG/L runtime routine that is written in assembly language, CLIMESSAGE, which gets CLI information from the system. Notice the use of the macros discussed earlier (TITLE, ENTRY, and R.T.N), as well as the SAVE instruction. Also, notice the calling of the DG/L error handler, .RTER, with the macro RT.ERR.

Using DG/L Runtime Routines in an Assembly Language Program

The complement of using assembly language routines in DG/L programs is using the DG/L runtime routines in this manual in assembly language programming.

DG/L runtime routines can be a simple, efficient way of coding complex functions and system calls in an assembly language program. The call to a DG/L runtime routine from an assembly language program follows the sequence described in "Steps in the Operation of Your Assembly Language Routine" earlier in this chapter. However, the code in your assembly language program must perform the stack maintenance operations that DG/L performs in a DG/L program.

Assembling and Linking the DG/L Runtime Routine

Assemble and link (bind) the DG/L routine as you would any DG/L program.

Calling DG/L Runtimes in Assembly Language Programs

When you call a DG/L runtime routine, the routine itself provides the SAVE and R.T.N commands. Your assembly code must load the accumulators, push arguments, and perform the call in the following steps

- First, it loads the current value of the stack pointer into AC2
- Second, in reverse order, it pushes the addresses of all the arguments onto the stack
- Third, it calls the DG/L subroutine in this form:

```
?RCALL name
```

Passing Arguments

Your program must push all the arguments onto the stack in reverse order

```
ELEF   0,2,3        ;get address of ARGn
ELEF   1,5,3        ;get address of
                    ;ARGn-1
PSH    0,1          ;then push them onto
                    ;the stack
```

and repeat this process as many times as necessary. If you pass one of the incoming arguments to your routine, you can use the same pointer passed:

```
LDA 0,ARG1,3
PSH 0,0
```

```
        TITLE   CLIMESSAGE

ENTRY   CLIMESSAGE
        SAVE    0
        LDA     2,ARG0,3        ;GET PACKET ADDRESS
        ?GTMES                  ;DO THE SYSTEM CALL
        JMP     ERET            ;IF ERROR IN CALL, GO TO ERROR CODE
        STA     0,@ARG1,3       ;ELSE RETURN THE AC0 ARGUMENT,
        STA     1,@ARG2,3                   ;THE AC1 ARGUMENT,
        R.T.N                               ;AND RETURN TO CALLER

ERET:   SUBZR   1,1             ;NOTE: ?GTMES PUTS ERROR CODE IN ERAC
0                               ;(AC0 UNDER AOS)
        RT.ERR                  ;CALL THE DG/L ERROR HANDLER .RTER

        E.ND
```

Figure 2-4. Example of an Assembly Language Runtime Routine

Examples

The examples in Figure 2-5 demonstrate calls in assembly code to BLKWRITE and MEMORY. The last instruction in both calls restores the stack pointer to its initial value, releasing the stack frame and temporary storage created for the subroutine.

The call to BLKWRITE shows several arguments being passed, while the call to MEMORY shows a routine with a single argument, a returned value.

NOTE: If you write an assembly language routine that performs a ?RCALL to a DG/L routine, you must allow two extra words in the stack frame for the system to store resource information. Your local data must start at offset 3 from the frame pointer, not offset 1. See the *AOS Programmer's Reference Manual* for further details.

Internal Structure of Data

To use assembly language routines in a DG/L program, you must know how DG/L stores and references data. For any assembly language module (routine) you create, use the conventions outlined in this section.

- Address *INTEGER, REAL, BOOLEAN,* and *POINTER* variables by the first word of a block of n words:

Type	n Words
INTEGER	1
INTEGER(2)	2
REAL	2
REAL(4)	4
BOOLEAN	1
POINTER	1

- Define STRING VARIABLES (bit and character strings) by a three-word entity, called the *string descriptor,* where

Word 0	is a byte pointer to string (character strings) or a word pointer to string (bit string).
Word 1	is the maximum length of data (fixed).
Word 2	is the current length of data (changes can be modified by string operations).

```
                                        ; A CALL TO BLKWRITE

START:  ZLDA        2,.SP               ; LOAD .SP INTO AC2
        ELEF        0,6,3               ; LOAD ADDRESS OF LAST ARG
        ELEF        1,ARRAY,0           ; LOAD ADDRESS OF THIRD ARG
        PSH         0,1                 ; PUSH ADDRESSES OF LAST
                                        ; AND THIRD ARGS ONTO STACK

        ELEF        0,4,3               ; LOAD ADDRESS OF ARG1
        ELEF        1,BUFFER,0          ; LOAD ADDRESS OF ARG0
        PSH         0,1                 ; PUSH ADDRESSES OF FIRST ARGS

        ?RCALL      BLKWRITE            ; CALL THE SUBROUTINE

END:    ZSTA        2,.SP               ; RESTORE THE STACK POINTER


                                        ; A CALL TO MEMORY

      START: ZLDA   2, .SP
             ELEF   1,4,3               ;ADDRESS FOR
                                        ;RETURNED VALUE
             PSH    1,1

             EJSR MEMORY

END:         ZSTA   2, .SP
```

Figure 2-5. Assembly Language Calls to BLKWRITE and MEMORY

- Define SUBSTRINGS by three-word descriptors, where

Word 0	is a word pointer to the parent string specifier.
Word 1	is a negative byte (or bit) offset into the parent string.
Word 2	is the current length of data.

- Address any ARRAY as a pointer to the first address of a block of memory. The block comprises the number of words required for the data in the array. A specifier of 2n + 1 words (n being the number of dimensions) is located immediately before the block.

  The contents of the specifier are as follows, with the words numbered as negative displacements from the address:

Offset	Contains
-((2n)+1)	Lower bound of dimension n
-(2n)	Upper bound of dimension n
.	.
.	.
.	.
-5	Lower bound of dimension 2
-4	Upper bound of dimension 2
-3	Lower bound of dimension 1
-2	Upper bound of dimension 1
-1	Number of dimensions
0	The actual array

  String and bit arrays consist of elements that are three-word string specifiers, each pointing to and giving the size of a string located elsewhere (usually on the heap).

EXTERNAL Data

If you have two separately-compiled DG/L programs that refer to the same EXTERNAL data, you must create global storage for the data. You do this in one of two ways:

- define variables in an assembly file with the .ENT pseudo-op instruction and declare them EXTERNAL (.EXTN or .EXTD in assembly) in each program that uses them.

- declare variables as GLOBAL in one program and as EXTERNAL in any other programs which use them.

If you use an assembly source to define EXTERNAL data, you'll be able to set the variable to any initial value you want.

A GLOBAL declaration is usually the easier way to define global data. To use an assembly language module to do so, you must create a source assembly file defining the variable, assemble it using MASM, and load it with the rest of your program using LINK.

Examples of definitions of variables appear in Figure 2-6. This listing shows the assembler code for EXTERNAL variables with explanatory comments. The code must include entry labels and allocate a block of the appropriate size. To use nonzero initial values, you replace the .BLK items with actual data values.

```
        .ENT  INT1
INT1:   .BLK 1                  ; SINGLE PRECISION INTEGER

        .ENT  INT2
INT2:   .BLK  2                 ; DOUBLE PRECISION INTEGER

        .ENT  RL1
RL1:    .BLK  2                 ; SINGLE PRECISION REAL

        .ENT  RL2
RL2:    .BLK  4                 ; DOUBLE PRECISION REAL

        .ENT  BOOL
BOOL:   .BLK  1                 ; BOOLEAN

        .ENT  PTR
PTR:    .DAT                    ; POINTER TO .DAT
.DAT:   .BLK  2000              ; .DAT CAN BE ANYTHING

        .ENT  PTR2
PTR2:   .BLK  1                 ; UNINITIALIZED POINTER

        .ENT STR1               ; STRING
STR1:   .STR*2                  ; BYTE POINTER TO 3 WORDS
        10                      ; MAXIMUM SIZE
        0                       ; CURRENT SIZE
.STR:   .BLK 5                  ; RESERVE SPACE FOR STRING

        .ENT 12ARR              ; ARRAY

12ARR:  .1ARR                   ; INTEGER(2) ARRAY
        5                       ; LBOUND
        10                      ; HBOUND
        1                       ; # OF DIMENSIONS
.1ARR:  .BLK 12                 ; 12 WORDS OF ARRAY DATA

        .ENT BIT1               ; BIT
BIT1:   .BIT                    ; WORD POINTER
        64                      ; MAXIMUM SIZE
        1                       ; CURRENT SIZE
.BIT    .BLK 4                  ; RESERVE WORDS
```

Figure 2-6. Using Assembly Code for External Variables

End of Chapter

Chapter 3
All-Purpose Runtime Routines

The runtime routines (calls) in this chapter provide you with a wide variety of functions and statements useful for both single-task and multitask programming.

In order of appearance, the sections in this chapter are:

- Performing Mathematical Operations with Integers and Words
- Bit and String Manipulation
- Obtaining Information About Arrays
- Command Line Handling
- Using the System Clock
- Managing Memory
- Miscellaneous Routines

Performing Mathematical Operations with Integers and Words

REM (B)

Divides two integers and obtains an integer result and remainder.

Format

REM (dividend, divisor, quotient *[,remainder]*);

Arguments

dividend	is an integer expression that specifies the value to be divided.
divisor	is an integer expression.
quotient	is an integer variable that receives the result of division.
remainder	is an integer variable that receives the overflow.

Examples

```
REM(INT1,TEST,RESULT,OVER);
        .
        .
        .
TEMP := TEMP + OVER;

REM((RANDOM),(6),RX,DIE1);
/* THROW THE DICE */
REM((RANDOM),(6),RX,DIE2);
DIE1 := DIE1 + 1;
DIE2 := DIE2 + 1;
```

Notes

The value in *remainder* will always be between 0 and divisor minus 1. If divisor is zero, results are meaningless.

Error Conditions

No error condition can occur.

ROTATE (B)

Rotates a word a specified number of bit positions.

Format

```
i := ROTATE (datum, count);
```

Arguments

i	is an integer variable that receives the value of the rotated word.
datum	is an identifier that contains the input word.
count	is a signed integer specifying the number of bits you want to rotate. A positive value indicates a rotation to the right; a negative value indicates a rotation to the left.

Examples

```
IROT := ROTATE (W, 8);

I := 1;
UNTIL (I := ROTATE (I,1)) = 1 DO
/*DO IT 16 TIMES*/
END;
```

Notes

Bits rotated off either end of the word are brought back on the other end.

Error Conditions

No error condition can occur.

SHIFT (B)

Shifts a word a specified number of bit positions.

Format

```
i := SHIFT (datum, count);
```

Arguments

i	is an integer value that receives the value of the shifted word.
datum	is a one-word variable, such as an integer or pointer
count	is a signed integer specifying the number of bits you want to shift. A positive value indicates a shift to the right; a negative value indicates a shift to the left.

Examples

```
INEW := SHIFT (IOLD,-5);

IF (SHIFT (J,-8) = 0) GO TO TEST;
        /* CHECK FOR A NULL LOW BYTE */

IX := SHIFT (CHAR1, -8) + CHAR2;
        /*BYTE PACK TWO 8-BIT VALUES*/
```

Notes

Bits shifted off either end of the word are lost. Bits shifted in are all zero.

Error Conditions

No error condition can occur.

UMUL (B)

Multiplies two unsigned integers (16 bits), adds another, and obtains the result and the overflow.

Format

```
UMUL (multiplicand, multiplier, addend, overflow,
      product);
```

Arguments

multiplicand	is an integer expression that specifies the value to be multiplied.
multiplier	is an integer expression that specifies the value to use in multiplying.
addend	is an integer expression that specifies the value to add.
overflow	is an integer variable that receives the overflow (greater than 16 bits) of the product.
product	is an integer variable that receives the result of the multiplication.

Example

```
UMUL (IX,16,0,OFLO,RES);
      /*PICK UP HIGH ORDER 4 BITS FROM IX,
      STORE IN OFLO*/
```

Notes

UMUL performs multiplication in this manner:

```
(multiplicand x multiplier + addend)
      = (product,overflow)
```

Error Conditions

No error condition can occur.

Bit and String Manipulation

CBIT (B)

Clears a bit in a bit string.

Format

```
CBIT (bit string, index);
```

Arguments

bit string	is a bit string in which you want to clear a bit to zero.
index	is an integer expression that specifies the position of the bit you want to clear. Bits are numbered from left to right, starting with 1.

Example

```
CBIT (BITSTRX,8);
```

Notes

A call to CBIT that gives an index outside the declared maximum length of the string will not be performed, but will not return an error. If you call CBIT giving an index outside the *current* length, the string will be extended to that length, and the intervening bits set to zero.

Error Conditions

No error condition can occur.

INDEX (B)

Searches for a pattern of characters (or bits) in a string (or bit string) and obtains the position of the first character (or bit).

Format

```
i := INDEX (string, pattern)
```

Arguments

i	is an integer variable that returns the position of the first character (or bit) of the pattern. Characters (and bits) are numbered from left to right, starting with 1.
string	is a character (or bit) string expression that specifies the string to search in.
pattern	is a string expression that specifies the characters to search for.

Example

```
STR := "ABCDE"; I := INDEX (STR,"DE");
/*WIL   L RETURN 4 IN I*/
i := INDEX (BITSTR, "10110");
```

Notes

If INDEX does not find the pattern, i returns the value 0.

Error Conditions

No error condition can occur.

LENGTH (B)

Obtains the current length of a string variable.

Format

```
i := LENGTH (string)
```

Arguments

i	is an integer variable that receives the current length.
string	is a character or bit string expression.

Example

```
SLENGTH := LENGTH(STR);
```

Notes

To return the *declared* length of a string, use the SIZE call.

Error Conditions

No error condition can occur.

SBIT (B)

Sets a bit in a bit string to 1.

Format

SBIT (bit string, index);

Arguments

bit string	is the bit string in which you want to set a bit.
index	is an integer expression that specifies the position of the bit you want to set. Bits are numbered from left to right, starting with 1.

Example

```
SBIT (BITSTR,3);
```

Notes

A call to SBIT that gives an index outside the maximum length of the bit string is not performed, but does not return an error. If you call SBIT giving an index greater than the *current* length, the string will be extended to that length and the intervening bits set to zero.

Error Conditions

No error condition can occur.

SETCURRENT (B)

Sets the current length of a string or bit string variable.

Format

SETCURRENT (string, length);

Arguments

string	is a character or bit string variable you want to set.
length	is an integer expression that specifies the number of characters you want to set string to.

Example

```
LINEREAD (S,ADDRESS(S),LEN);
        .
        .
        .
SETCURRENT (S,LEN);
```

Error Conditions

Giving length a negative value or a value greater than the declared maximum length of string, will result in a fatal-to-process error, AISET.

SIZE (B)

Determines the number of elements in an array or the declared length of a string or bit string.

Format

```
i := SIZE (name)
```

Arguments

i	is an integer variable that receives the number of elements in the array or the declared length of the string or bit string.
name	is the name of the array, string, or bit string variable.

Example

```
ISIZ := SIZE(STR);
```

Error Conditions

No error condition can occur.

Reference

.ASIZE (Internal call)

SUBSTR (B)

Delimits a substring of a character string, bit string, substring, integer or integer array.

Format

s := SUBSTR (expr, starting index *[,terminating index]*)

or

SUBSTR(s, starting index *[,terminating index]*) := expr

Arguments

s	is a string or bit variable that receives the substring value.
expr	an expression or variable, particularly a string or integer variable.
starting index	is an integer expression that specifies the position of the first character (byte) you want to obtain or respecify (1 is the leftmost).
terminating index	an integer expression that specifies the position of the last character (or byte) you want to return. If you do not specify a *terminating index,* it is assumed to be the same as the starting index and s will receive one character.

Examples

1.
```
S := "H" !! SUBSTR("YELLOW",2,5)!!
" THERE"
        /*WILL RETURN "HELLO THERE"*/
```

2.
```
STRING S;
INTEGER I;
S := SUBSTR(I,2);
/*WILL RETURN THE LOWER BYTE OF INTEGER
I*/
```

3.
```
STRING (10) ARRAY INSTR [6];
                .
                .
                .
S := SUBSTR(INSTR[4],3,5);
/*WILL RETURN THE 3RD TO 5TH CHARACTERS
OF THE FIFTH
STRING IN THE STRING ARRAY INSTR*/
```

4. SUBSTR("ABCDE",3,10) IS "CDE", AND
   SUBSTR("ABCDE",6,9) IS A NULL STRING.

5. STRING (10) S;
   S := "ABC";
   SUBSTR(S,7,10) := "7890";
   /*WILL RESULT IN S TAKING THE VALUE
   "ABC□□□7890"*/

Notes

When **expr** is a string, and **starting index** is less than 1, or **terminating index** is greater than the length of the string, SUBSTR returns only the bytes that actually lie in the string. (See Example 4.)

For data types other than strings (such as integers), SUBSTR does not do bounds checking.

If you use SUBSTR to put characters in positions beyond the string's current length, but within its maximum length, the string will first be extended with spaces up to, but not including, the specified character positions, and then any specified character replacement(s) will be performed. If the new character positions are partially or completely out of range (less than 1 or greater than the maximum length) only the character positions that are in range will be replaced. (See Example 4).

CAUTION: The assignment statement SUBSTR(S,I+1,J+1) := SUBSTR(S,I,J), used to move each character in the string to the right one place, will *not* work. The assignment moves one character at a time, and thus moves the first character over one position, and then moves it over again, etc. Instead, use a temporary string.

As shown in the above formats, you can assign from or to a SUBSTR. In this way SUBSTR is analogous to an array reference, where **expr** corresponds to the array name, and the indexes correspond to the array index. However, for SUBSTR, **expr** isn't limited to variables.

You can probably best appreciate the operation of SUBSTR by thinking of it as a type of pointer expression. The actual movement of data is performed by the assignment statement.

The DG/L compiler recognizes different data types for SUBSTR and calls different subroutines for each: .USUBSTR if it's a bit string; otherwise, it calls .ISUBSTR.

Error Conditions

The following error condition can occur:

AISOV	String overflow (if /T global compiler switch is used; see Chapter 10, "Operating Instructions").

TBIT (B)

Tests a bit in a bit string.

Format

```
bool := TBIT (bit string, index)
```

Arguments

bool	is a boolean variable that receives the result: True if 1, False if 0.
bit string	is the bit string in which you want to test the bit.
index	is an integer expression that specifies the position of the bit you want to test. Bits are numbered left to right, starting with 1.

Example

```
J := TBIT (BITX,K);
        .
        .
        .
IF (TBIT (J,3)) GO TO ERR;
/*PASS CONTROL TO STMNT LBL ERR
IF BIT 3 OF J IS SET*/
```

Notes

CAUTION: TBIT does not check for out of range bits. If the offset you give is less than zero or greater than the current bit-string length, TBIT extrapolates in main memory to check a bit there. If the address is not in memory this can cause a hardware trap.

Error Conditions

No error condition can occur.

Obtaining Information about Arrays

DIM

Determines the width of an array dimension.

Format

```
i := DIM (array name, dimension);
```

Arguments

i	is an integer variable that receives the width (number of elements) of the dimension.
array name	is a string expression that specifies the array.
dimension	is an integer expression that specifies which of the array's dimensions you want measured. 1 is the dimension that appears leftmost in the ARRAY declaration.

Example

```
IWIDTH := DIM(IARAY10, 1);
```

Error Conditions

The following error code may be returned:

AIBND	Illegal bound specification

HBOUND (B)

Obtains the upper bound value of an array dimension.

Format

```
i := HBOUND (array name, dimension);
```

Arguments

i	is an integer variable that receives the upper bound value.
array name	is a string expression that specifies the array.
dimension	is an integer expression that specifies which of the array's dimensions you want measured. The value 1 is the dimension that appears leftmost in the ARRAY declaration.

Example

```
IBOUND := HBOUND(IARAY10,2);
```

Error Conditions

The following error code may be returned:

AIBND	Illegal bound specification.

LBOUND (B)

Obtains the lower bound value of an array dimension.

Format

```
i := LBOUND (array name, dimension);
```

Arguments

i	is an integer variable that receives the lower bound value.
array name	is an identifier that specifies the array.
dimension	is an integer expression that specifies which of the array's dimensions you want to measure. The dimension that appears leftmost in the ARRAY declaration is numbered 1.

Example

```
INTEGER ARRAY DUCA [1:10,1:5];
ILOWER := LBOUND(DUCA,2);
   /*WILL RETURN 1*/;
```

Error Conditions

The following error code may be returned:

AIBND	Illegal bound specification.

SIZE (B)

Determines the number of elements in an array or the declared length of a string.

Format

```
i := SIZE (name);
```

Arguments

i	is an integer variable that receives the number of elements in the array or the declared length of the string.
name	is the name of the array or string variable.

Example

```
ISIZ := SIZE(IRAY);
```

Error Conditions

No error condition can occur.

Reference

.ASIZE (Internal call)

Command Line Handling

CLIMESSAGE

Gets CLI command line information

Format

```
CLIMESSAGE (packet, AC0, AC1 [,error label] );
```

Arguments

packet	is a four-word entity that specifies the type of CLI command line processing that will be done. See ?GTMES in the *AOS Programmer's Manual* for more information.
AC0 and AC1	are integer variables whose use depends on the definitions in the AOS packet.
error label	a statement label to which control transfers if an error occurs.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
IPC codes.
MISCELLANEOUS codes.

References

?GTMES (System call)

See the *AOS Programmer's Manual* for request and parameter tables.

Example

```
I := 0;
DO BEGIN
        CLIPACK [$GREQ] := $GARG;       /* ARGUMENT READING */
        CLIPACK [$GNUM] := I;                /* FOR ARG NUMBER I */
        CLIPACK [$GRES] := 2*ADDRESS (ARG[I]);  /* READ INTO ARG TABLE*/
        CLIMESSAGE (CLIPACK,AC0,AC1,CLIEOF);  /* READ THE ARG*/
        SETCURRENT (ARG[I],AC0);             /* SET THE CORRECT LEN */
        CLIPACK [$GREQ] := $GTSWS;             /* NOW GET THE SWITCHES */
        CLIMESSAGE (CLIPACK, SW [I,1], SW [I,2]); /* GET THEM */
        I:=I+1
        END;
CLIEOF:                                 /* NO MORE ARGS */
```

COMARG (B)

Reads command line operands and switches.

Format

COMARG (file number, string *[[,switches]error label]*);

Arguments

file number	is an integer expression. It is a dummy argument that allows source-level compatibility with RDOS programs; AOS ignores this number.
string	is a string variable of up to 133 characters that receives the argument of the command file.
switches	is a 26-element boolean array that receives the value TRUE for each corresponding alphabetic switch that is set.
error label	is a statement label to which control transfers if an error occurs.

Example

Assume you typed

```
MYPROG/C KRUPP/B FREEMAN
```

to start execution of the program MYPROG. The following statements show how you could apply calls to COMARG:

```
COMARG (0,ITEXT,ISWITCH,IERR);
        .
        .
        .
COMARG (0,ITEXT,ISWITCH,IERR);
        .
        .
        .
COMARG (0,ITEXT,ISWITCH,IERR);
```

In the first call to COMARG, ITEXT receives the string MYPROG and ISWITCH receives the value of switch C. In the second call, ITEXT receives the string KRUPP and ISWITCH receives the value of switch B. In the third call, ITEXT receives the string FREEMAN and ISWITCH receives the value 0. Note that the three calls to COMARG have identical formats.

Notes

If you give only one optional argument, the system assumes that it's an *error label*.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
IPC codes.
MISCELLANEOUS codes.

References

See Appendix B of the *Command Line Interpreter User's Manual* for information on CLI messages.

?GTMES (System call)

RCOMARG

Rewinds the file of command arguments to make the next COMARG number zero.

Format

RCOMARG (file number *[,error label]*);

Arguments

file number	is an integer expression. It is a dummy argument that allows source-level compatibility with RDOS programs; AOS ignores this number.
error label	is a statement label to which control transfers if an error occurs.

Example

```
RCOMARG (0, IERR);
```

Error Conditions

None under AOS.

Using the System Clock

DELAY

Suspends a task for a specified number of clock pulses.

Format

DELAY (delay count *[,time units [,error label]]*);

Arguments

delay count	is an integer expression that gives the number of real-time clock pulses to suspend the task.
time units	is an integer expression that indicates the unit of measurement with one of the following values: 0 Basic System Units 1 Milliseconds 2 Seconds 3 Minutes Giving no argument is the same as giving zero.
error label	is a statement label to which control transfers if an error occurs.

Examples

```
DELAY (IPULS,0);

DELAY (10,2); / * DELAY FOR 10 SECONDS * /
```

Error Conditions

None currently defined.

Reference

?DELAY (System call)

GETFREQUENCY
Gets the real-time clock frequency.

Format

```
i := GETFREQUENCY ( [(error label] );
```

Arguments

i	is an integer variable that receives one of the following values:

Code	Meaning
0	Frequency is 60 HZ (line frequency).
1	Frequency is 10 HZ.
2	Frequency is 100 HZ.
3	Frequency is 1000 HZ.
4	Frequency is 50 HZ (line frequency).

error label	is a statement label to which control transfers if an error occurs.

Example

```
I := GETFREQUENCY;
```

Error Conditions

No error condition can occur.

Reference

?GHRZ (System call)

GTIME (B)
Obtains the current time.

Format

```
GTIME (year, month, day, hour, minute, second);
```

Arguments

The arguments are integer variables that receive the following values:

year	Calendar year minus 1900
month	1-12
day	1-31
hour	0-23
minute	0-59
second	0-59

Example

```
GTIME (YR,MO,DA,HR,MIN,SEC);
  :
WRITE (1, MO,"/",DA,"/",YR);
```

Error Conditions

No error condition can occur.

References

?GDAY (System call)
?GTOD (System call)

STIME (B)

Sets the time.

Format

STIME (year, month, day, hour, minute, second
[,error label]);

Arguments

The first six arguments are integer expressions that specify the following values:

year	Calendar year minus 1900
month	1-12
day	1-31
hour	0-23
minute	0-59
second	0-59
error label	a statement label to which control transfers if an error occurs.

Example

```
INTEGER ARRAY TIME [1:6];
GTIME (TIME[1],TIME[2],TIME[3],TIME[4],
       TIME[5],TIME[6]);
TIME[4] := 10;
TIME[5] := 0;
TIME[6] := 0;
STIME (TIME[1],TIME[2],TIME[3],TIME[4],
       TIME[5],TIME[6]);
/*SET THE CLOCK TO 10:00 AM*/
```

Error Conditions

The following error conditions can occur:

ERTIME	Illegal time of day.
ERPRV	Caller lacks privilege to set time.

References

?SDAY (System call)
?STOD (System Call)

Managing Memory

ALLOCATE (B)

Reserves a number of words of memory, obtains their location, and initializes the words to zero.

Format

ALLOCATE (pointer, size *[,error label]*);

Arguments

pointer	is a pointer variable that receives the beginning address of the reserved area of storage.
size	is a positive integer expression that specifies the number of words of storage for the area.
error label	is a statement label to which control is transferred if an error occurs.

Example

```
ALLOCATE (IAREA, 200, IERR);
```

Notes

Allocation follows an *exact-fit* method; if there is no block of the specified size in heap memory (see Figure 2-1), one is created from a larger block by splitting, or, if no larger block exists, one is allocated from the top of the heap, thus lowering the heap's limit.

ALLOCATE reserves size+1 words, and returns a pointer to the second word. The word at offset -1 from the pointer contains the length of the allocated area. This word should never be modified by the user.

Error Conditions

The following error codes may be returned:

ERMEM	Insufficient memory.
AISIZ	Invalid size (< = 0).

FREE (B)

Frees an allocated block of words and links it into the free chain.

Format

FREE (pointer *[,error label]*);

Arguments

pointer	is a pointer expression returned by a previous call to ALLOCATE.
error label	is a statement label to which control transfers if an error occurs.

Example

```
FREE (IPNT,IERR);
```

Notes

If the block was at the heap's limit, it's space is returned to stack space (raising the limit). Otherwise, it goes on a "free chain" linked list, possibly being merged with contiguous blocks on the list.

Error Conditions

The following error codes may be returned:

AICOR Attempt to free an unallocated or already freed block.

MEMORY (B)

Obtains the number of remaining words of storage available to the user.

Format

```
i := MEMORY
```

Arguments

i	is an integer variable that receives the amount of memory remaining.

Examples

```
ISIZE := MEMORY;

ALLOCATE (D,MEMORY-1000);
```

Notes

The amount of storage available doesn't include any blocks on the free chain, but it does include blocks returned to the stack space by FREE.

Error Conditions

No error conditions can occur.

ADDRESS (B)

Obtains the address of any datum.

Format

p := ADDRESS (reference)

Arguments

p	is a pointer variable that receives the word address of the argument.
reference	is any variable or constant.

Example

```
IADDR := ADDRESS ("STRING");
```

Error Conditions

A nonfatal error results if reference is a substring which does not lie on a word boundary.

The following error code may be returned:

AEOBA Unaligned string address.

ARGCOUNT

Obtains the number of arguments that the current procedure received when it was called.

Format

i := ARGCOUNT

Arguments

i	is an integer variable which receives the number of arguments that were passed to the procedure.

Example

```
PROCEDURE A (B,C,D,E);
        .
        .
        .
IF ARGCOUNT <= 2 THEN
        PROC1 (B,C)
        ELSE
        PROC2 (D,C);
```

Notes

ARGCOUNT counts the return argument as well as the passed argument(s): I := PROCEDURE (A, B) is a three-argument call. This is consistent with the fact that I := PROCEDURE (A, B) is the same as PROCEDURE (I, A, B).

ARGCOUNT is the number of arguments to the *current* call, not the number of declared formal parameters.

Error Conditions

No error condition can occur.

BYTE (ASCII) (B)

Obtains the integer value of a byte in a data element, or sets the value of a byte in any data element.

Format

```
i := BYTE (data element, byte)
```

or

```
BYTE (data element, byte) := expr
```

Arguments

i	is an integer variable that receives the integer value of the byte.
data element	is a string, a variable, or an expression in which the specified byte exists.
byte	is an integer expression that specifies the position of the byte in **data element** . The first byte of a string or data element is 1.
expr	is an integer value to be put into the byte specified

Examples

```
LBYTE := BYTE (J,1);
RBYTE := BYTE (J,2);
        /*RETURN THE LEFT AND RIGHT BYTES OF
        J*/

BYTE (J,2) := BYTE (J,2) OR 40R8;
        /*IF THE RIGHT BYTE OF J IS AN
        ALPHABETIC CHARACTER, THIS WILL MAKE
        SURE IT IS LOWER CASE BY SETTING BIT
        #2 TO 1*/
```

Notes

Notice that, like SUBSTR, BYTE works on both sides of the assignment statement, either to return an eight-bit quantity (right-justified in a word), or to set a new value in a byte in a variable or memory.

BYTE doesn't convert its data element arguments; it selects the appropriate byte of whatever data is passed to it.

BYTE of a BIT or character string selects from the string data, not the descriptor.

BYTE does not check for out of range.

BYTE starts counting at 1, not 0 (the addressed word has bytes numbered 1 and 2).

Error Conditions

No error condition can occur.

CLASSIFY (B)

Returns an integer value indicating the location of a given value within a table of ranges.

Format

```
i := CLASSIFY (test value, pointer)
```

Arguments

i	is an integer variable that receives the integer value from the table.
test value	is an integer expression that evaluates to a signed integer.
pointer	is a pointer expression that points to a user-written table of ranges.

The user-written table consists of three-word entries:

Word 0 lower limit of range.
Word 1 upper limit of range.
Word 2 value to be returned if test value is in this range.

The last entry should cover all possible values:

Word 0 100000R8P1
Word 1 +77777R8
Word 2 value if entry not found in ranges.

Example

```
J := CLASSIFY (ITST, IPNT);
GO TO TEST_TYPE [CLASSIFY(ITST,IPNT)];
```

Notes

If you do not define the last entry as shown above, CLASSIFY might continue searching through memory until a range is found.

Error Conditions

No error condition can occur.

GETLIST

Obtains the name of the current list file.

Format

```
s := GETLIST
```

Arguments

s	is a string variable that receives the list filename.

Example

```
OPEN (0,(GETLIST));
```

Notes

Under AOS, the listing filename is @LIST; under RDOS, it is $LPT. This routine makes a program transportable between the two systems on the DG/L source level.

Error Conditions

No error condition can occur.

NOTE: In the example above, however, the OPEN will take an error condition if the generic filename @ LIST is used. (Using an error label on the OPEN call, however, creates a file if it doesn't exist.)

RANDOM

Obtains a random number from a pseudo-random sequence of integers in the range of -32768 to 32767.

Format

i := RANDOM

Arguments

i	is an integer variable that receives the random number.

Error Conditions

No error condition can occur.

Example

```
INBR := RANDOM;
```

Notes

RANDOM must be declared as an EXTERNAL *INTEGER* PROCEDURE.

RANDOM uses a method which generates a linear-congruential sequence:

$X_{n+1} := (X_n * A + C) \text{ MOD } 2 \uparrow 16$

where

A = 5 + 2↑11
C = 33031R8
X = ith random number

RANDOM will always provide the same default seed value, unless you use the SEED routine. SEED can also provide a non-constant default value. The number returned by RANDOM is a 16-bit integer.

SEED

Provides an initial number to base random number generation on.

Format

SEED *[(number)];*

Arguments

number	is an integer value you wish to initialize random number from.

Example

```
SEED (1234);
```

Notes

If you do not specify *number* in the call to SEED, the random number generator will perform the following calculation to obtain an initial number:

```
initial number := ROTATE (default number, 8);
```

where default number is the number of seconds since midnight added to the number of days since January 1.

If you call RANDOM directly, without calling SEED first, a fixed initial number of 52352R8 is used. This assures repeatability of data.

The number given to SEED seed must be a 16-bit integer.

Error Conditions

No error condition can occur.

References

?GTOD (System call)
?GDAY (System call)

SUPERUSER

Changes Superuser mode.

Format

SUPERUSER (mode *[,error label]*);

Arguments

mode	is an integer expression or variable set to one of the following: -1 Turn on Superuser mode. 1 Turn off Superuser mode. 0 Return Superuser mode. If you set mode to zero, upon return from the call, mode will contain one of the following: -1 Superuser mode is on. 1 Superuser mode is off.
error label	is a statement label to which control transfers if an error occurs.

Example

```
SUPERUSER (I := 0); /*TEST SUPERUSER MODE*/
```

Notes

SUPERUSER permits the calling process to access any file, bypassing the system's control mechanism.

A process must have the Superuser privilege, ?PVSU, to enter or leave Superuser mode.

Error Conditions

The following error codes may be returned:

ERPRV Attempt to issue call without having ?PVSU.

Reference

?SUSR (System call)

SYSTEM

Provides a generalized interface with the system, allowing you to use AOS system calls.

Format

SYSTEM (system word, AC0, AC1, AC2 *[,error label];*)

Arguments

system word	is an integer expression whose value is one of the system call numeric codes listed in SYSID.SR.
AC0, AC1, AC2	are integer variables that specify the values of the accumulators passed to the system. On return from the call to SYSTEM, these arguments receive the returned values of the accumulators.
error label	is a statement label to which control transfers if an error occurs.

Example

```
LITERAL $GDAY;
        .
        .
        .
SYSTEM ($GDAY,MONTH,DAY,YEAR);
```

Notes

Even if the system call doesn't require all accumulators, you must provide four arguments. The accumulator arguments must not be constants, because they return values from the system call.

Error Conditions

Errors depend on the system call you are executing.

Reference

SYSID.SR *AOS Programmer's Manual.*

XCT1

Executes a one-word instruction on the assembly level.

Format

XCT1 (instruction, AC0, AC1, AC2, AC3, carry
[skip-label]);

Arguments

instruction	is a one-word, assembly language instruction to the machine.
AC0,AC1, AC2 and AC3	are integer variables or parenthesized constants whose values are to be set into the four accumulators prior to executing the instructions.
carry	is the desired input value of the carry, an integer set to 0 or 1.
skip-label	is a label to which control transfers if the instruction results in a skip.

Example

```
XCT1 (BLM,(0),ADDRESS(A1),ADDRESS(A2),(28),(0));

/* MOVE 28R10 WORDS FROM THE BEGINNING OF
A1 TO THE BEGINNING OF A2 */
```

Notes

The values for the accumulators and carry must not be constants, because the contents of the accumulators and carry after the instruction is executed are stored back into the argument locations after the instruction is executed. These values may be expressions or parenthesized constants, in which case they'll exist as temporaries on the stack and not create any permanent effects if changed. They may also be variables, whose values may be modified by the call.

Error Conditions

Errors depend on the instruction you are executing; an MSP, for example, might result in a stack overflow.

XCT2

Executes a two-word instruction on the assembly level.

Format

XCT2 (instruction, AC0, AC1, AC2, AC3, carry
[,skip-label]);

Arguments

instruction	is a two-word assembly language instruction to the machine.
AC0,AC1, AC2 and AC3	are integer variables or parenthesized constants whose values are to be set into the four accumulators prior to executing the instructions.
carry	is the desired input value of the carry, an integer set to 0 or 1.
skip-label	is a label to which control transfers if the instruction results in a skip.

Example

```
LITERAL $FSST(103350R8P1);
FSST [1] := $FSST;
/* STORE FLOATING POINT STATUS */
FSST [2] := ADDRESS (FPUSTAT);
/* DISPLACEMENT = 2 WORD STATUS ARRAY */
XCT2 (FSST,(0),(0),(0),(0),(0));
/* STORE THE STATUS */
```

Notes

The values for the accumulators and carry must not be constants, because the contents of the accumulators and carry after the instruction is executed are stored back into the argument locations after the instruction is executed. These values may be expressions or parenthesized constants, in which case they exist as temporaries on the stack and will not create any permanent effects if changed. They may also be variables, whose values may be modified by the call.

Error Conditions

Errors depend on the instruction you are executing.

End of Chapter

Chapter 4
Handling Errors

Overview

Runtime errors can occur for many different reasons; e.g., you might pass incorrect data types, or set up a system packet incorrectly; discrepancies between a call and the current state of memory or files may occur; or conflicts with hardware and software limitations may cause a runtime error. In all cases, the way you handle an error will determine the specific result of the error condition.

A runtime error can be one of the following four types:

- System error codes (returned by the operating system)
- DG/L runtime error codes
- User-signalled error codes
- User-signalled error messages (without a numeric code)

A runtime error also falls into one of four *severity levels:*

- Non-fatal (an error message appears at the terminal or line printer and your program continues)
- Fatal to task
- Fatal to process
- Absolutely fatal (not interceptible in any way)

DG/L handles all error types and all severity levels, although it can't trap absolutely fatal errors (see "ERRINTERCEPT and ERRTRAP"). For non-fatal errors, the sequence of events described in the "non-fatal" category above occurs, or you can trap the error. For fatal errors, you can either define your own error-handling routines and messages, or let the DG/L default error handler take control. The default error handler reports information about the error and terminates the task or program.

Appendix C in this manual lists all DG/L error mnemonics, numeric codes, and messages. Appendixes A and E in the *AOS Programmer's Manual* give the numeric and mnemonic values for all AOS error conditions.

In this introduction, we'll first discuss default error handling, and then the various ways you can handle errors: the *error label* argument available with many runtime calls, procedures for trapping and intercepting errors within program blocks, writing your own error messages, and using the error codes returned by the system. In reviewing this discussion, remember that these calls may be nested and called, one from the other, in many different ways. At the end of this introduction, we show an example of interrelated error handling calls used in a procedure that opens a disk file. For more details about these DG/L error handling calls, refer to individual calls later in this chapter.

Default Error Handling

Default error handling occurs when you don't pass an optional *error label* argument to a runtime call or user-written routine, and you haven't specified an error handling procedure or trap address, and an error occurs in that call or routine. Control passes from your program to the DG/L default error handling routine, .RTER. .RTER stops execution of the task or process that caused the error, generates an *error report,* and transfers control to the next higher level of execution, if the error is fatal to the process.

.RTER's error report contains the following information: severity level, error type, the location of the procedure that detected the error, and the location of the procedure call or operation that caused the error. .RTER puts out this report to the default output device, typically to your terminal or line printer. The report looks like this

FATAL TO PROCESS ERROR 20000
< error message if any >
REPORTED FROM 74057
ROUTINE CALLED FROM 73011

where "fatal to process" can also be "fatal to task," "absolutely fatal," or "non-fatal." By matching the "CALLED FROM" address with a procedure's name in your Link map, you can locate the offending procedure.

If you do provide code for execution when an error occurs (see later sections), .RTER still stores the error report information, but doesn't print it. If you want the system to print the error message, use ERPRINT in your error handling code. ERRFATAL also prints the system's error message, and returns control to the system, avoiding any further error handling.

Several other calls; e.g., READERROR, and FPUERROR, retrieve information stored by .RTER. You may want to use this information in your own error handling routines (see "Using Error Codes" and "Error Handling Example").

SHORTMESSAGE and NOMESSAGE

You can control the length of the error report by declaring either SHORTMESSAGE or NOMESSAGE within the declaration part of your program. Because they're EXTERNALs, these declarations remain in effect for the full execution of your program. You can't declare both calls in a single process.

SHORTMESSAGE gives you the numeric value of the error, the location of the procedure that detected the error, and the location of the procedure or operation that caused the error. It doesn't include the textual error message.

NOMESSAGE produces no message when an error occurs. It actually frees memory space otherwise needed to store error messages.

AOS System Error Report

On a fatal error processed by .RTER, when control returns to the system, the system also prints an error message, which looks like this

ERROR
UNKNOWN ERROR CODE 20000
(or error message if code is
known to system)
ERROR: FROM PROGRAM
X,PROGRAMNAME,ARGUMENT
(command line that executed program)

User Error Handling

You can alter default error handling with labels, DG/L runtime calls, or your own procedures. You can process errors before termination and even continue your program's execution after a fatal error.

The Error Label

Many of the DG/L runtime routines in this manual have an optional argument, which is always last, called *error label.* It is a statement label to which the program jumps if an error occurs while your program is performing the call with the label. You must decide what action to take at the code which the label branches to. For example, you could use error handling routines in this chapter to read the information about the error stored in .RTER and, depending on what kind of error occurred, print a message and terminate the program, or call another program module, or merely continue execution.

ERRINTERCEPT and ERRTRAP

Like the error label argument, these two runtime routines allow you to decide where control will go if an error occurs. Unlike the error label argument, both calls handle errors *anywhere* in the block(s) in which you call them. However, they can't process absolutely fatal errors.

ERRINTERCEPT specifies a user-written *procedure* which executes *before* the default-handler takes control. You pass the name of your error-handling procedure as the first argument to ERRINTERCEPT. Your procedure might include code to generate a detailed error report, or it could write buffers to preserve current data. (See the example of ERRINTERCEPT under its description.) After your error intercepting procedure finishes, control passes to the default handler, which terminates the program if the error was fatal (unless you cleared the error in your error-handling procedure).

With ERRTRAP control never passes to the default handler. Like the error label argument, ERRTRAP specifies a *label* to which control passes if an error occurs. The code you write at the label might correct the error condition and try again, or just continue.

Any number of ERRINTERCEPTs and ERRTRAPs may be in effect concurrently at any point in a program. The numerical "mode" you declare as an argument to these calls (see calls for details) defines the call as handling different, overlapping, or identical classes of errors.

NOTE: Under multitasking (see Chapter 8), any error handling procedures or labels apply only within tasks which call those routines.

Rules of Scope

Scoping rules apply to these two calls. In this respect, error handling procedures are like DG/L declarations. For example, you might call procedure B from A, and C from B. If B does an ERRTRAP or ERRINTERCEPT to prepare for possible errors, and then an error occurs *after* B returns to A, the error won't be trapped or intercepted. But an error that occurred in procedure C, which was called from B, would be trapped or intercepted.

If you declare a new error handler for a given type of error in the same block as an earlier declaration for that error type, the new handler overrides the earlier one.

In Figure 4-1 notice that, like all DG/L runtime calls except those to built-in routines (see Chapter 1), you must first declare ERRTRAP as an EXTERNAL PROCEDURE. Each block's call to ERRTRAP then specifies different labels for reading and handling errors. Thus you can limit the procedure's application to the specific types and severities of errors you want to trap. In Figure 4-1 the octal value arguments indicate the types of errors and handling. The actual code for the error handling routines begins at the error label passed as an argument to ERRTRAP, LAB2 or LAB1.

If an error were to occur in the second inner block (after the declaration of INTEGER Y), the entire block would terminate when control transferred to LAB1. If you wanted to preserve the values of that block's variables, you would need to have an error routine in that inner block, as is the case with LAB2.

```
      .
   BEGIN
      INTEGER I;
      EXTERNAL PROCEDURE ERRTRAP;
       .
       .                                  /*NO ERROR LABEL EXISTS*/
       .
      ERRTRAP (LAB1,74000R8);
       .
       .                                  /*LAB1 IS THE ERROR TRAP LABEL*/
       .
       BEGIN
         STRING (10) S1
         ERRTRAP (LAB2,74030R8);
          .
          .                               /*LAB2 IS THE ERROR TRAP LABEL*/
          .                               /*FOR ERROR CODE 30R8 ONLY*/
       LAB2:
          .
          .
         END;
       BEGIN
         INTEGER Y;
          .
          .                               /*LAB1 IS THE ERROR TRAP LABEL*/
          .
    END;
    .
    .
LAB1:
.
.                                         /*LAB1 IS THE ERROR TRAP LABEL*/
.
END;
```

Figure 4-1. An Example of Block Structure of Error-Handling Routines

Using Error Codes

As we mentioned earlier, when a fatal error condition terminates a program, the system stores an octal error code and the code's message. You may want to use this error code as a value in an error handling routine. If your program successfully intercepts an error, the system won't print the report. To retrieve the code and message, use READERROR. You can then use the error code's value in a conditional statement to define error handling actions (see Figure 4-2).

You could also pass the error code's value to ERRUSER, which calls .RTER as a DG/L runtime routine, specifying a type of error for .RTER to handle. By passing ERRRUSER your own error code, you could intentionally cause your program to call the error handler .RTER.

Writing Your Own Error Message

To provide your own error messages, you can either include a WRITE statement in an error handling routine in your program, or use the DG/L runtime call to ERROR.

Example of Error Handling

Figure 4-2 shows how several error handling routines work together in a procedure for opening files. (I/O operations are a common source of runtime errors.)

The procedure uses the DG/L runtime routine SYSTEM to open a file on the assembly language level. While the call to SYSTEM reduces overhead from the usual OPEN call, it doesn't *create* a file the way OPEN would (OPEN creates a file when you don't pass it an error label). Instead, SYSTEM goes to the error label, if a file doesn't exist.

The code at RETRY tries to open the file, and transfers control to MYERR if it fails. MYERR provides two branches. First it reads the error code and compares it to the value ER$FDE. Notice that the procedure previously defines ER$FDE to have the same value (25_8) as the AOS mnemonic, ERFDE, for *FILE DOES NOT EXIST.* The system only stores only the numeric value, not ER$FDE. However, using LITERALS like ER$FDE makes your code readable and easy to update.

If the error value does indeed indicate that a file doesn't exist, the code creates a file and returns to RETRY. Otherwise, it calls ERRUSER, which passes control to the DG/L error handler.

For information about DG/L *compiler* error messages, refer to Appendix B in the *DG/L™ Reference Manual.* For information on DG/L error handling *at link time,* see Chapter 10 of this manual, "Operating Instructions."

```
PROCEDURE QOPEN (PACKET, LAB);
INTEGER ARRAY PACKET (0:11);
LABEL LAB;
BEGIN
 INTEGER I;
 EXTERNAL PROCEDURE SYSTEM, CREATE;
 EXTERNAL INTEGER PROCEDURE READERROR;
 LITERAL ER$FDE (25R8);

RETRY:   SYSTEM ($OPEN,PACKET,MYERR);
GO TO ENDIT;

MYERR:  IF (I := READERROR) = ER$FDE THEN BEGIN
  CREATE (FILENAME);
  GO TO RETRY;
  END;
  ERRUSER (1, ERR);

ENDIT:  END;
```

Figure 4-2. An Example of Error Handling in File I/O

BREAKFILE

Creates a message for the DG/L error handler to write and a break file.

Format

```
BREAKFILE (message);
```

Argument

message — is a string ending in a null that the error handler will print.

Example

```
BREAKFILE ("FILE IS EMPTY, TERMINATING");
```

Notes

The default action on an error termination is not to create a break file (see Chapter 2 in the *AOS Programmer's Manual* for a discussion of break files).

See ERROR for rules on appending a null (< NUL >) to strings. And see BREAKSWITCH for an alternate method of obtaining a break file.

Error Conditions

No error condition can occur.

BREAKSWITCH

Is a declaration that creates a break file if a process terminates on an error condition.

Format

```
EXTERNAL INTEGER BREAKSWITCH;
```

Arguments

Does not apply.

Example

```
BREAKSWITCH := 1;
```

Notes

Declare EXTERNAL INTEGER BREAKSWITCH. BREAKSWITCH is initially zero. Any time you set it to a non-zero value, a DG/L runtime termination will cause a break file. You can switch BREAKSWITCH on and off in your program.

Also see BREAKFILE.

ERETURN

Terminates a program and indicates an error.

Format

ERETURN (error *[,error label]*);

Arguments

error	is an integer variable that contains an error code.
error label	is a statement label in the program to which control transfers if the call to ERETURN cannot be executed.

Notes

Upon execution of a call to ERETURN, the system makes an unconditional return to the next higher level program, the father process. If the father process is not the CLI or EXEC, the message may be in any format agreed on by the father and son. If the next higher level program is the CLI, one of the following occurs:

If the error status code is an AOS error code, the user terminal will display the message *WARNING*, *ERROR*, or *ABORT*, and the appropriate message.

If you specify one of your own error status codes that the CLI doesn't recognize, then UNKNOWN ERROR CODE n is displayed, where n is your error status code in octal.

This call is also documented in Chapter 7, under "Swapping and Chaining Programs."

Example

```
DELETE ("FILE20",IERR);

IERR:   ERETURN ((READERROR));
```

Error Conditions

No error condition can occur.

References

?RETURN (System call)

See Appendix C for tables of all AOS exceptional condition codes and DG/L error codes.

ERPRINT

Prints out the last error message (if not yet printed).

Format

ERPRINT;

Arguments

None.

Example

```
TRY_AGAIN: IF READERROR <> 25
        THEN ERPRINT
        ELSE BEGIN
        .
        .
        .
```

Notes

As shown in the example above, you typically use ERPRINT with error labels and other error handling branchings, such as ERRINTERCEPT, which intercept errors before an error message is printed.

Notice that ERPRINT is a simple declaration.

Error Conditions

No error condition can occur.

ERRFATAL

Prints an error message without using the stack and returns to the system.

Format

ERRFATAL *[(error number)];*

Arguments

error number is an integer variable that contains the code of the error:

Bit Code	Octal Equivalent	Meaning
	Error code: Severity	
0B1	000000R8P1	Fatal to process
1B1	040000R8P1	Fatal to task
2B1	100000R8P1	Nonfatal
3B1	140000R8P1	Absolutely fatal
	Error code: Type	
0B3	00000R8	System error
1B3	10000R8	User error code User error text *
3B3	30000R8	Runtime error

Bits 7-15 (the last three octal places) contain the AOS or DG/L error codes.

* To pass a "user error text", use ERROR call.

Notes

If the error was not already printed, ERRFATAL prints a full message.

If *error number* is not present, then ERRFATAL prints the last error message, if it hasn't already.

Example

```
ERRFATAL (ER$EOF);
```

Error Conditions

The following errors may be returned:

FILE SYSTEM codes.

References

?RETURN (System call)

See Appendix C for tables of all AOS exceptional condition codes and DG/L error codes.

ERRINTERCEPT

Establishes an error-handling procedure for intercepting (some) errors.

Format

ERRINTERCEPT *[(procedure, datum, mode [,error label])] ;*

Arguments

procedure is the name of the handling procedure. You must declare the procedure either globally or locally to the call to ERRINTERCEPT.

datum is an integer or interger variable (but not an expression) whose value passes to the procedure.

mode is an integer whose bit values define the manner of error handling - printing, severity, type, and code to be handled by the procedure. Bits are numbered from 0 at the left to 15 at the right.

Code	Octal Equivalent	Meaning
	Printing:	
0B0	000000R8	Suppress printing of error message
1B0	100000R8	Print error message before handling
	Severity:	
1B1	040000R8	Handle errors fatal to process
1B2	020000R8	Handle errors fatal to task
1B3	010000R8	Handle nonfatal errors
	Type:	
1B4	004000R8	Handle system errors
1B5	002000R8	Handle user error codes
1B6	001000R8	Handle user error text
1B7	000400R8	Handle DG/L runtime errors
	Code:	

Bits 8-15 either specify an error code for the procedure to handle, or give 377R8 for handling any code.

error label is a statement label passed to the procedure. Control will pass to the statement label after the procedure executes.

Notes

You must specify the severity and type, and give either an error code to be processed or 377R8.

ERRINTERCEPT is block-structured, so you may redefine intercepts at lower block levels, or replace intercepts on the same block level.

Declare the procedure in this format:

```
PROCEDURE procedure (datum [,error label] );
        procedure body
```

where datum, and *error label* will be passed by ERRINTERCEPT.

ERRINTERCEPT without arguments removes any active ERRINTERCEPT.

When an error occurs, .RTER selects the most recently established ERRINTERCEPT procedure (or ERRTRAP label) whose parameters allow it to handle the given error. .RTER then calls the routine. If control returns to .RTER (after the end of the procedure), .RTER then passes control to the user program at the point of error, if it was a non-fatal error or if the error was cleared by the intercept procedure. If the error was fatal, .RTER terminates the program. You can avoid returning at all to .RTER by having your intercept procedure jump to the optional error label.

Examples

```
ERRINTERCEPT (MYERRS,0,72377);
        /*INTERCEPT USER-DEFINED ERRORS
        WITH "MYERRS"*/

ERRINTERCEPT (SYSERRS,0,74777);
        /*INTERCEPT SYSTEM ERRORS WITH
        "SYSERRS"*/
```

Examples (continued)

```
PROCEDURE PETE;

BEGIN
  PROCEDURE BAD_KEY( KEY );
    INTEGER KEY;
    WRITE( 2, "Key ", KEY, " is an unknown key, try another.<NL>");

   /*end of internal procedure BAD_KEY*/

  EXTERNAL PROCEDURE ERRINTERCEPT, ERRUSER;
  EXTERNAL BOOLEAN PROCEDURE UNKNOWN;
  LITERAL $NON_FATAL (10000R8);
  LITERAL $USER_ERROR(2000R8);
  LITERAL $BAD_KEY   (7);

  OPEN( 1,(GETCINPUT));
  OPEN( 2,(GETCOUTPUT));
  ERRINTERCEPT( BAD_KEY, KEY, $NON_FATAL + $USER_ERROR + $BAD_KEY);

TRY_AGAIN;
  READ( 1, KEY);

  IF UNKNOWN( KEY ) THEN ERRUSER( 100000R8 +  $BAD_KEY, TRY_AGAIN);
     .
     .
     .

END;
```

Error Conditions

No error condition can occur.

ERRKILL

Prints an error message and kills the calling task.

Format

ERRKILL *[(error number);*]

Arguments

error number — is an integer variable that contains the code of the error:

Bit Code	Octal Equivalent	Meaning
	Error code: Severity	
0B1	000000R8P1	Fatal to process
1B1	040000R8P1	Fatal to task
2B1	100000R8P1	Nonfatal
3B1	140000R8P1	Absolutely fatal
	Error code: Type	
0B3	00000R8	System error
1B3	10000R8	User error code User error text *
3B3	30000R8	Runtime error

Bits 7-15 (the last three octal places) contain the AOS or DG/L error codes.

* To pass a "user error text", use ERROR call.

Example

```
ERRKILL (ER$XMT);
```

Notes

If no stack is available the following message is printed:

< error number > ERROR KILLED < task number >

where *< task number >* is 0 in a single-task environment. Otherwise, a full error message is printed.

Error Conditions

No error condition can occur.

References

?KILL (System call)
?REC (System call)
?XMT (System call)

See Appendix C for tables of all AOS exceptional condition codes and DG/L error codes.

ERRMESSAGE

Returns a message line (associated with an error code) from the AOS message file, or from a message file in AOS format.

Format

s := ERRMESSAGE (error code *[[,file number] ,error label]*);

Arguments

s	is a string variable that receives the textual message.
error code	is an integer that specifies the error whose message will be returned.
file number	is an integer that specifies the channel for an open message file you want read. If no number is specified, the system file ERMES will be read.
error label	is a statement label to which control transfers if an error occurs.

Example

```
ERROR (ERRMESSAGE (CODE := 145R8, ERRFILE));
```

Notes

Notice that if you pass only one optional argument, the system assumes it's an *error label*.

See the *AOS Programmer's Manual* for a definition of non-AOS message files.

Error Conditions

The following error codes may be returned:

ERTXT	String too short for message, or no message received.

FILE SYSTEM codes.

Reference

?ERMSG (System call)

ERROR (B)

Creates a message for the DG/L error handler to write.

Format

```
ERROR (message);
```

Argument

message	is a string ending in null (< NUL >) that the error handler will print.

Examples

```
ERROR ("UNABLE TO PROCEED, TERMINATING");

ERROR ("CANT START TASK" !!NUM!! "<NUL>");
```

Notes

The error signalled to .RTER will be type "user error text".

All DG/L string constants (see first example) end in null (< NUL >). However, if you create a message at runtime (see second example), you must concatenate a null to the end of the string.

Error Conditions

No error condition can occur.

ERRTRAP

Establishes a label as a location for trapping errors.

Format

ERRTRAP *[(error label, mode)] ;*

Arguments

error label is a statement label to which control transfers when an error occurs.

mode is an integer whose bit values indicate the type of error handling - printing, severity, type, and code. Bits are numbered from 0 at the left to 15 at the right.

Bit Code	Octal Equivalent	Meaning
	Printing:	
0B0	000000R8	Suppress printing of error message
1B0	100000R8	Print error message before handling
	Severity:	
1B1	040000R8	Handle errors fatal to process
1B2	020000R8	Handle errors fatal to task
1B3	010000R8	Handle nonfatal to errors
	Type:	
1B4	004000R8	Handle system errors
1B5	002000R8	Handle user error codes
1B6	001000R8	Handle user error text
1B7	000400R8	Handle DG/L runtime errors
	Code:	

Bits 8-15 either specify an error code for the system to handle, or give 377R8 for handling any code.

Example

```
ERRTRAP (EOF,74030R8)
/*TRAPS ALL SEVERITY LEVELS (4+2+1)
OF SYSTEM ERRORS (4) WITH
"END OF FILE" (30) AOS CODE*/;
```

Notes

When an error occurs, .RTER reviews all traps (labels) and intercepts (procedures) currently established (most recent first), until it finds a trap or intercept whose mode includes the current error type, severity, and code. If it finds an ERRTRAP label, .RTER does a non-local GOTO to this label, thus unwinding the stack of any procedures called in the meantime.

You must specify the severity and type, and give either an error code you want processed or 377R8.

ERRTRAP is block structured, so you may redefine traps at lower block levels, or replace traps on the same block level. Multiple traps and intercepts can be active at once, with overlapping, distinct, or identical types of codes. The search rule is latest-first.

ERRTRAP without arguments removes any active ERRTRAP established in the current block. It won't carry over to surrounding blocks.

If the statement label the call branches to is at a higher level block than the ERRTRAP, then all ERRTRAPs at the same and intervening levels will be released, along with the procedure's stack frames. If the branchings are on the same level, the traps are retained.

Error Conditions

No error condition can occur.

ERRUSER

Calls the DG/L error handler with an error code.

Format

ERRUSER (error code *[,error label]*);

Arguments

error code — is an expression that represents a user, DG/L, or system error number.

Bit Code	Octal Equivalent	Meaning
	Error code: Severity	
0B1	000000R8P1	Fatal to process
1B1	040000R8P1	Fatal to task
2B1	100000R8P1	Nonfatal
3B1	140000R8P1	Absolutely fatal
	Error code: Type	
0B3	00000R8	System error
1B3	10000R8	User error code
2B3	20000R8	User error text (no code)
3B3	30000R8	Runtime error

Bits 7-15 (the last three octal places) contain the AOS or DG/L error code.

error label — is a statement label to which control transfers if an error occurs.

Example

```
ERRUSER (ENUM,IERR)
```

Notes

This routine allows you to signal an error just as a DG/L runtime routine might, thus allowing convenient transfer of control to code set up using ERRINTERCEPT or ERRTRAP.

Error Conditions

No error condition can occur.

Reference

See Appendix C for tables of all AOS exceptional condition codes and DG/L error codes.

FPUERROR

Returns boolean and string values for FPU errors.

Formats

```
bool := FPUMANTISSA

bool := FPUUNDERFLOW

bool := FPUOVERFLOW

bool := FPUDIVIDE

i := FPUERROR
```

Arguments

bool — is a boolean variable that receives the result of each type of floating-point error. It returns true if an error has occurred.

i — is an integer variable that returns the code of the error:

AEMAN	Mantissa overflow
AEUND	Underflow
AEOVF	Overflow
AEZDV	Zero divide

See Appendix C and ERRFATAL for numeric codes.

Example

```
BOOLEAN BOOL;
        .
        .
        .
        IF (BOOL := FPUUNDERFLOW) THEN
```

Notes

You must separately declare each of the calls your program uses (FPUMANTISSA, etc.) as an EXTERNAL BOOLEAN routine.

Error Conditions

No error condition can occur.

FPUTRAP

Is a declaration for enabling the trapping of floating point errors.

Format

EXTERNAL INTEGER FPUTRAP;

Arguments

None.

Notes

FPUTRAP and NOFPUTRAP let you set, but not change at runtime, whether floating point errors are trapped. The default is NOFPUTRAP. If declaration of both occurs with the same LINK, FPUTRAP is used.

NOFPUTRAP

Is a declaration for ignoring floating point errors.

Format

EXTERNAL INTEGER NOFPUTRAP;

Arguments

None.

Notes

FPUTRAP and NOFPUTRAP let you set, but not change at runtime, whether floating point errors are trapped. The default is NOFPUTRAP.

Calls to ERRINTERCEPT and ERRTRAP normally catch FPU errors. Declare NOFPUTRAP if you want these errors ignored.

Error Conditions

No error condition can occur.

NOMESSAGE

Is a declaration for removing error messages and the error reporter from storage.

Format

EXTERNAL INTEGER NOMESSAGE;

Arguments

None.

Notes

If you want to use error messages for debugging, and then suppress them during execution, you can link an assembly module of this form into you program:

```
.EXTN NOMESSAGE
```

This saves re-compiling.

See also SHORTMESSAGE. You can't declare NOMESSAGE and SHORTMESSAGE in one program.

Error Conditions

No error condition can occur.

READERROR

Checks and reads information about errors.

Format

i := READERROR *[(severity [,text address [,error nrel [,caller nrel [,clear flag]]]])]*

Arguments

i — is an integer variable that receives the latest outstanding error code. If no error is outstanding, the value -1 is returned.

severity — is an integer variable that receives a severity code:

0	Fatal to process
1	Fatal to task
2	Nonfatal error

text address — is an integer variable that receives the address in memory of the error message or returns a code:

-1	Error message was already printed
0	Error message was not printed; error not reported through a call to ERROR.

error nrel — is an integer variable that returns the address in the program of the routine that caused the error.

caller nrel — is an integer variable that returns the address in the progam of the routine that called the error handler.

clear flag — is an integer expression that specifies whether to clear the error. The value 0 or no argument sets the error code to -1. Any other value prevents clearing.

READERROR (continued)

Example

```
ENUM := READERROR (SEV, ETEXT, LOC, ROUTINE,
0);
```

Error Conditions

No error condition can occur.

Notes

Clear flag is the only value that you input when calling READERROR; all the other arguments return values.

Reference

See Appendix C for tables of all AOS exceptional condition codes and DG/L error codes.

SHORTMESSAGE

Is a declaration for printing short error messages instead of full error messages.

Format

```
EXTERNAL INTEGER SHORTMESSAGE;
```

Arguments

None.

Notes

The short message has this form:

task id ERROR error code FROM location

where:

task id is an integer indicating the task where the error occurred.

error code is a 6-digit octal number that gives the error code as follows:

Bit Code	Octal Equivalent	Meaning
	Error code: Severity	
0B1	000000R8P1	Fatal to process
1B1	040000R8P1	Fatal to task
2B1	100000R8P1	Nonfatal
3B1	140000R8P1	Absolutely fatal
	Error code: Type	
0B3	00000R8	System error
1B3	10000R8	User error code
2B3	20000R8	User error text (no code)
3B3	30000R8	Runtime error

Bits 7-15 (the last three octal places) contain the AOS or DG/L error code.

location is the NREL location of the call that caused the error.

You can also use an assembly language module of the form:

```
.EXTN SHORTMESSAGE
```

to link with your program after debugging, avoiding re-compiling.

See also NOMESSAGE. You cannot declare both calls in one program.

Error Conditions

No error condition can occur.

Reference

See Appendix C for tables of all AOS exceptional condition codes and DG/L error codes.

End of Chapter

Chapter 5
Managing Files, Directories, and Devices

Managing Files

Choosing a DG/L Call for File Creation

This section, "Managing Files," includes five DG/L runtime routines that create files: ACREATE, CREATE, CRAND, CCONT, and NCONT. In this introduction, we will explain what kind of file each of these DG/L routines creates to help you choose the right call for your needs. If you're completely familiar with AOS file concepts, such as "file element size" and "record format type," you may want to skip the following section, "AOS File Concepts."

AOS File Concepts

Chapter 5 of the *AOS Programmmer's Manual* discusses file concepts in detail. Most of the basic operations the system performs to create files are user-transparent, but knowing some of these AOS processes may help you to choose what kind of file structure you want.

- The operating system builds disk files out of one or more 256-word (512-byte) disk blocks.
- The basic unit of file storage is a *file element,* which is one or more contiguous blocks; i.e., blocks with sequential physical addresses.
- When your data requirements grow beyond the *file element size* specified by you or by one of these DG/L runtimes, the system automatically creates new file elements of the same size wherever disk space is available. The system then uses index files to locate these "randomly" situated file elements.
- Finally, you read/write data to/from your files either in 256-word blocks or in one of the following four forms, which are usually smaller units than blocks:

  A *dynamic record,* whose length you specify (in 8-bit bytes) when reading or writing.

  A *fixed-length record* that has a constant length.

  A *data-sensitive record* that is a series of characters terminated by a character defined to be a delimiter.

  A *variable-length record* that has a fixed-length header describing the number of bytes of data that it contains.

Which DG/L Runtime Meets Your Needs?

- ACREATE simply creates an AOS file and lets you pass a *parameter packet* as an argument. You specify in the packet the kind of AOS file you want to create. Use this call for maximum flexibility.
- CREATE sets up a standard AOS file, that is, a "random" file with a *dynamic* record format. A "random" file gives you the freedom to locate a specific block and datum without requiring the system find a large disk area of contiguous storage.
- CRAND is particularly useful in writing RDOS-compatible programs. It also creates a "random" file, but with a *fixed-length* record format.
- CCONT is also mainly for RDOS-compatible programming. It creates a contiguous file with a *fixed-length* record format. (Under AOS, NCONT is the same call as CCONT.)

  A "contiguous" file allows random access, but is limited to a fixed number of contiguous blocks. With a contiguous file, you can access each of its blocks with a single disk read or write, because each block occupies a fixed location on disk. Random files, on the other hand, ordinarily require several accesses to locate a block because they use a tree structure of disk pointers. Contiguous files thus allow quick access to blocks and data.

You may choose what size to fix the file's record format at OPEN time or at READ/WRITE time for both CRAND and CCONT.

See "File I/O" in the next chapter for more information about the interrelations between file creation, file opening/closing, and file reading/writing. And refer to the individual descriptions of routines in this section for more details about a particular file creation routine.

ACREATE

Creates an AOS file.

Format

ACREATE (pathname, packet *[,error label]*);

Arguments

pathname	is a string expression that gives the name of the file (directory entry).
packet	is an array that defines the type of entry you want to create. See the tables of packet contents in the *AOS Programmer's Manual.*
error label	is a statement label to which control transfers if an error occurs.

Example

```
ACREATE ("ANGELA_DUCA",CR_PAK);
```

Notes

See also CCONT, CRAND, CREATE and NCONT.

ACREATE creates an AOS-defined file. See the *AOS Programmer's Manual* for definitions of necessary parameter packets.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?CREATE (System call)

ATTRIBUTE

Obtains the file status parameters and device characteristics of an opened file.

Format

ATTRIBUTE (file number, attributes *[,device characteristics [,error label]]*);

Arguments

file number	is an integer expression whose value is associated with the open file.
attributes	is an integer array (23 words long) that receives the file status information.
device characteristics	is a three-word integer array that receives the file's device characteristics.
error label	is a statement label to which control transfers if an error occurs.

Example

```
INTEGER ARRAY ATTS[22], DEVCH[2];
        .
        .
        .
ATTRIBUTE (1,ATTS,DEVCH,IERR);
```

Notes

See the *AOS Programmer's Manual* for tables of attribute and bit-to-characteristic correspondences.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
INITIALIZATION and RELEASE codes.

References

?FSTAT (System call)

?GCHR (System call)

CCONT (NCONT)

Creates a contiguous file of a fixed length and initializes all words to zero.

Format

CCONT (filename, block count *[,error label]*);

Arguments

filename	is a string that specifies the name of the file you want to create.
block count	is an integer expression that specifies the number of 256-word blocks you want to allocate.
error label	is a statement label to which control transfers if an error occurs.

Example

```
CCONT ("JOEL.BT",2000,IERR);
```

Notes

You specify the record size of a CCONT file at read/write time.

NCONT is the same call as CCONT under AOS; but under RDOS, NCONT doesn't initialize words to zero.

Also see ACREATE, CRAND, and CREATE.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?CREATE (System call)

CHSTATUS

Obtains the current file status information for an opened file.

Format

CHSTATUS (file number, status *[,error label]);*

Arguments

file number	is an integer expression that specifies the opened file.
status	is an array that receives the 23 words of file status information. See the *AOS Programmer's Manual* for tables of status information.
error label	is a statement label to which control transfers if an error occurs.

Example

```
INTEGER ARRAY ISTAT [0:22];
        .
        .
        .
CHSTATUS (3,ISTAT,IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?FSTAT (System call)

CRAND

Creates a random file with fixed record size.

Format

CRAND (filename *[,error label]*);

Arguments

filename	is a string expression that specifies the name of the file you want to create.
error label	is a statement label to which control transfers if an error occurs.

Example

```
CRAND ("FILNM.DC",IERR);
```

Notes

You set the record size of a CRAND file at read/write time.

The element size of a CRAND file is one.

See also ACREATE, CCONT, and CREATE.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?CREATE (System call)

CREATE

Creates a random file with a dynamic record format.

Format

CREATE (filename *[,error label]*);

Arguments

filename	is a string expression that specifies the name of the file you want to create.
error label	is a statement label to which control transfers if an error occurs.

Example

```
CREATE ("BIRGITTA",IERR);
```

Notes

The element size of a CREATE file is one, the default AOS value.

See also ACREATE, CCONT, CRAND, and NCONT.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?CREATE (System call)

DELETE (B)

Deletes a disk file.

Format

DELETE (filename *[,error label]*);

Arguments

filename	is a string expression that specifies the name of the disk file you want to delete.
error label	is a statement label to which control transfers if an error occurs.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Example

```
DELETE ("ANGIOLILLO",IERR);
```

Notes

If you do not provide an error label, errors will be ignored.

Reference

?DELETE (System call)

FILESIZE (B)

Computes the current size in bytes of a file.

Format

FILESIZE (file number, byte count *[,error label];*)

Arguments

file number	is an integer expression that specifies the file.
byte count	is a two-word integer array or a double precision integer variable that receives the number of bytes.
error label	is a statement label to which control transfers if an error occurs.

Example

```
INTEGER (2) BCNT;
      .
      .
      .
FILESIZE (13,BCNT,IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?FSTAT (System call)

GCHANNEL

Obtains the number of an available channel.

Format

i := GCHANNEL *[(error label)];*

Arguments

i	is an integer variable that receives the channel number; i.e., a file number that is not in use.
error label	is a statement label to which control transfers if an error occurs.

Example

```
OPEN (LISTFILE := GCHANNEL (CANTOPEN),
      ("DATA1"), CANTOPEN);
      .
      .
      .
WRITE (LISTFILE, "<NL>");
```

Notes

GCHANNEL does not reserve a channel. Therefore, in a multitask environment, it is possible that an OPEN following a GCHANNEL might result in a channel-in-use error.

Error Conditions

The following error codes may be returned:

ERNMC	No more channels available.

GETACL

Obtains an entry's access control list (ACL).

Format

s := GETACL (pathname *[,error label]*);

Arguments

s	is a 256-byte string variable that will receive the access control information in the following form: username < null > access type *[username < null > access type]* . . . < null >
pathname	is a string expression that specifies the entry whose ACL will be returned.
error label	is a statement label to which control transfers if an error occurs.

Example

```
SETACL ("PROG1.PR", GETACL("PROG1.PR")!!
USERNAME(-1,0)!!"<NUL><3><NUL>");
      /*SEE ROUTINE USERNAME FOR ITS
      ARGUMENTS*/
```

Notes

USERNAME is a text string.

Access type is a single byte that may have up to five bits set as follows:

Bit	Access Type
20_8	owner access (O)
10_8	write access (W)
4_8	append access (A)
2_8	read access (R)
1_8	execute access (E)

The ACL string for the ACL "DEA,OWARE,+,RE" would be

"DEA < 0 > < 37 > + < 0 > < 3 > < 0 >".

See the *AOS Programmer's Manual* for a complete discussion of ACL concepts.

To use GETACL, the caller must have read access to the entry's parent directory, or owner access to the file itself.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?GACL (System call)

GLINK

Gets the contents of a link entry.

Format

s := GLINK (pathname *[,error label]*)

Arguments

s	is a string variable that will receive the complete link entry.
pathname	is a string expression that names the link entry you want returned.
error label	is a statement label to which control transfers if an error occurs.

Example

```
OPEN (0,GLINK("DATA.FILE.IN"));
```

Notes

To use GLINK, the caller must have read access to the entry's parent directory.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?GLINK (System call)

LINK

Creates a link entry in the current directory to a file in the same or another directory.

Format

LINK (link filename, resolution filename *[,error label]*);

Arguments

link filename	is a string expression that specifies the name of the link entry you want to create.
resolution filename	is a string expression that specifies the name of the file being linked to.
error label	is a statement label to which control transfers if an error occurs.

Example

```
LINK ("CASH.PR","CDR10.PR",IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?CREATE (System call)

RENAME (B)

Changes the name of a file.

Format

RENAME (old name, new name *[,error label]*);

Arguments

old name	is a string expression that specifies the name of the disk file you want to rename.
new name	is a string expression that specifies the new name of the disk file.
error label	is a statement label to which control transfers if an error occurs.

Example

```
RENAME ("MATH.DC","CHAP4.DC",IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?RENAME (System call)

SETACL

Creates an entry's access control list (ACL).

Format

SETACL (pathname, new ACL *[,error label]*);

Arguments

pathname — is a string that specifies the entry whose Access Control List (ACL) you want to modify.

new ACL — is a string variable, up to 256 bytes long, that specifies the access control information in the following form:

username < null > access type
[username < null > access type]
. . .
< null >

Passing an integer value of -1 instead of a string deletes the old ACL.

error label — is a statement label to which control transfers if an error occurs.

Example

```
SETACL ("PROG1.PR", GETACL("PROG1.PR")!!
USERNAME(-1,0)!!"<NUL><3><NUL>");
```

Notes

Access type is a single byte that may have up to five bits set as follows:

Bit	Access Type
20_8	% owner access (O)
10_8	% write access (W)
4_8	% append access (A)
2_8	% read access (R)
1_8	% execute access (E)

To use SETACL, the caller must have write access to the entry's parent directory, or owner access to the file itself.

See the *AOS Programmer's Manual* for a complete discussion of ACL.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?SACL (System call)

STATUS

Obtains the current directory status of a specified file.

Format

STATUS (filename, status *[,error label]*);

Arguments

filename	is a string expression that specifies the name of the file whose information you want to obtain.
status	is an array that receives the file status information. See the *AOS Programmer's Manual* for its contents.
error label	is a statement label to which control transfers if an error occurs.

Example

```
INTEGER ARRAY FILUFD [0:22];
        .
        .
        .
STATUS ("IOLINK",FILUFD,IERR);
```

Notes

For an open file, STATUS returns information current when the file was opened. While CHSTATUS returns information updated on the call.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?FSTAT (System call)

UNLINK

Deletes a link entry.

Format

UNLINK (link name *[,error label]*);

Arguments

link name	is a string expression that specifies the link entry you want to delete.
error label	is a statement label to which control transfers if an error occurs.

Example

```
UNLINK ("SYS.PR",IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?DELETE (System call)

ASSIGN

Assigns a device for the calling process to use exclusively.

Format

ASSIGN (device *[,error label]*);

Arguments

device	is a string expression that gives the name of the device. See the *AOS Programmer's Manual* for a list of device names.
error label	is a statement label to which control transfers if an error occurs.

Example

```
ASSIGN ("@MTA0", NOTAPE);
```

Notes

An assignment remains in effect until the process terminates or calls DEASSIGN.

You cannot assign a device which is currently being spooled.

Error Conditions

The following error codes may be returned:

ERDAI Device is already assigned.

FILE SYSTEM codes.

References

?ASSIGN (System call)

CDIR

Creates a directory.

Format

CDIR (directory entry name *[,error label]*);

Arguments

directory entry name	is a string expression that specifies the name of the directory entry you want to create.
error label	is a statement label to which control transfers if an error occurs.

Example

```
CDIR ("SUBDR",IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?CREATE (System call)

CPART

Creates a control-point directory (CPD).

Format

CPART (directory name, size *[,error label]*);

Arguments

directory name	is a string expression that specifies the name of the control-point directory.
size	is an integer that specifies the maximum size for the control point directory in bytes.
error label	is a statement label to which control transfers if an error occurs.

Example

```
CPART ("PARTNM",2000,IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?CREATE (System call)

DEASSIGN

Deassigns the device previously ASSIGNed to the calling process.

Format

DEASSIGN (device *[,error label]*);

Arguments

device	is a string expression naming the device previously assigned by a call to ASSIGN. See the *AOS Programmer's Manual* for a list of device names.
error label	is a statement label to which control transfers if an error occurs.

Example

```
DEASSIGN ("@MTA0");
```

Error Conditions

The following error codes may be returned:

ERDAI	Device is currently opened (cannot yet be deassigned).
ERARU	Device is already deassigned.
ERARC	Attempt to release the process terminal.

FILE SYSTEM codes.

Reference

?DEASSIGN (System call)

DIR

Changes the current working directory.

Format

DIR (directory name *[,error label]*);

Arguments

directory name	is a string expression that specifies the name of the new working directory.
error label	is a statement label to which control transfers if an error occurs.

Example

```
DIR (":UDD:SOURCES",IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?DIR (System call)

GETDEV

Obtains the characteristics of a device.

Format

GETDEV (device, characteristics *[,error label]*);

Arguments

device	is a string expression that names the device. See the *AOS Programmer's Manual* for a list of device names.
characteristics	is a three-word entity that receives the bit code of the characteristics.
error label	is a statement label to which control transfers if an error occurs.

Example

```
GETDEV ("@LPT", CHARACS, GDEV$ERR);
```

Notes

See the *AOS Programmer's Manual* for a table of the bit-to-characteristic correspondences.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
INITIALIZATION and RELEASE codes.

Reference

?GCHR (System call)

GETDIR

Obtains the name of the current directory.

Format

s := GETDIR *[(error label)]*

Arguments

s	is a string variable that receives the pathname of the current directory.
error label	is a statement label to which control transfers if an error occurs.

Example

```
STR := GETDIR (IERR);
```

Error Conditions

No error condition can occur.

Reference

?GNAME (System call)

GETSEARCH

Writes the current search list into a string variable.

Format

GETSEARCH (list *[,error label]);*

Arguments

list	is a string variable that will receive the searchlist.
error label	is a statement label to which control transfers if an error occurs.

Example

```
STRING (512) SLIST;

GETSEARCH (SLIST);
```

Notes

The search list image appears in this format:

pathname < 0 > ... pathname < 0 > < 0 >

where < 0 > is a null.

Because the system immediately resolves each pathname into a complete pathname, relative to the current working directory, the whole search list may ultimately have more than 511 bytes. A call to GETSEARCH will not return more than 511 bytes.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?GLIST (System call)

INIT

Initializes a logical disk unit or a device.

Format

INIT (directory name *[,error label]*);

Arguments

directory name	is a string expression that specifies the name of the logical disk unit or device you want to initialize.
error label	is a statement label to which control transfers if an error occurs.

Example

```
INIT ("@MTA0",IERR);
```

Notes

An initialized logical disk unit (LDU) or device remains initialized until you release it by a call to RELEASE.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
INITIALIZATION and RELEASE codes.

References

?INIT (System call)

See the *AOS System Manager's Guide* for a list of valid disk names.

PATHNAME

Obtains a complete pathname for a file name.

Format

s := PATHNAME (input pathname *[,error label]*);

Arguments

s	is a string variable that receives the complete pathname.
input pathname	is a file name whose complete pathname you want.
error label	is a statement label to which control transfers if an error occurs.

Example

```
SWAP (PATHNAME ("PROG1.PR"));
```

Error Conditions

The following error codes may be returned:

ERMPR Designated string is too small .

FILE SYSTEM codes.

Reference

?GNAME (System call)

RELEASE

Releases a logical disk unit or device from current use.

Format

RELEASE (directory name *[,error label]*);

Arguments

directory name	is a string expression that names the logical disk unit or device you want to release.
error label	is a statement label to which control tranfers if an error occurs.

Example

```
RELEASE ("DPD3",IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
INITIALIZATION and RELEASE codes.

Reference

?RELEASE (System call)

SETDEV

Sets the device characteristics.

Format

SETDEV (device, characteristics *[,error label]);*

Arguments

device	is a string expression that names the device. See the *AOS Programmer's Manual* for a list of device names.
characteristics	is a three-word entity that specifies the bit code of the characteristics.
error label	is a statement label to which control transfers if an error occurs.

Example

```
GETDEV ("@LPT", DEVCHAR);
DEVCHAR[0] := DEVCHAR[0] OR 20R8;
/*SET UPPER CASE ONLY */
SETDEV ("@LPT", DEVCHAR);
```

Notes

See the *AOS Programmer's Manual* for a table of the bit-to-characteristic correspondences.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
INITIALIZATION and RELEASE codes.

Reference

?SCHR (System call)

SETSEARCH
Sets the search list.

Format

SETSEARCH (search list *[,error label]*);

Arguments

search list	is a string that defines the search list. It must be no longer than 511 bytes and in this format [name < 0 > < 0 > ... name < 0 > < 0 >]. < 0 > is a null or NEW LINE and each **name** names directories the system is to search through to locate a file.
error label	is a statement label to which control transfers if an error occurs.

Notes

The search list may have from zero to eight pathnames, and must be no more than 511 characters long. The system immediately resolves each name into a complete pathname, relative to the current working directory. Thus, the search list may ultimately have more than 511 characters. A call to GETSEARCH will not return the entire search list if it's longer than 511 characters (bytes).

Example

```
STRING (512) SLIST;
GETSEARCH (SLIST);
SLIST := ":UDD:USER2<NUL>"!! SLIST;
SETSEARCH (SLIST);
```

Error Conditions

The following error codes may be returned:

ERSRE Search list contains too many pathnames.

FILE SYSTEM codes.

Reference

?SLIST (System call)

End of Chapter

Chapter 6
Input/Output Routines

This chapter is divided into four sections, "File I/O," "Terminal I/O," "Shared Memory I/O," and "Cache Memory I/O." File I/O provides you with all the "flavors" of input/output you can perform to and from AOS disk and tape files and your program. Similarly, terminal I/O covers the input/output routines you need to communicate with a terminal.

Shared memory I/O and cache memory I/O are two special kinds of file I/O. Shared memory I/O allows more than one process to access the same data files, using the AOS shared page feature (see *AOS Programmer's Manual*).

Cache memory I/O, a DG/L feature, lets your program access large volumes of data through system-monitored buffers in your unshared memory area.

File Input/Output (I/O)

We offer you several different sets of runtime routines (calls) for transferring data to and from disk and tape files. These calls open a file, read/write, and close the file.

All methods of DG/L file I/O, except for block I/O, operate *sequentially*. In other words, you must read/write from the beginning of a file and continue sequentially. However, using two runtime calls, FILEPOSITION and POSITION, you can specify a starting position other than the beginning of the file, and then read/write from/to the file.

The format of your records and the amount of system overhead your program can afford affect your choice of a data transfer method. A brief description of each available method follows:

DATAREAD and DATAWRITE use buffers for transferring blocks of data from files into main memory. The use of buffers decreases the number of system input/output (I/O) calls required. But you must use DATAOPEN and DATACLOSE between reading and writing to buffers to ensure proper storage of data on disk.

BYTEREAD and BYTEWRITE perform binary I/O. Consequently, the system doesn't have to interpret character delimiters, and can transfer any number of bytes you specify.

LINEREAD and LINEWRITE perform data-sensitive I/O. They transfer bytes (usually character strings) up to a terminating character; e.g., NEW LINE, carriage return, form feed, or NUL. A maximum of 133 characters, can be transferred. This type of I/O is useful in terminal interactions.

READSTRING and WRITESTRING use a string's descriptor to automatically set the maximum length of read/write and the length of the read/written string. (READSTRING, however, does a data-sensitive read, while WRITESTRING does a simpler, binary, dynamic write -- see the *AOS Programmer's Manual* for details about kinds of I/O.)

SCREENREAD lets you use AOS screen edit commands on a string before reading it into memory (see the description of SCREENREAD for available options).

BLKREAD and BLKWRITE read and write blocks of 256 words at a time. You can use them with any files to read blocks into main memory from any location in the file.

GRDB and GWRB let you specify a block's AOS I/O parameters.

QREAD and QWRITE let you define non-standard I/O by specifying an AOS parameter packet.

Compare the descriptions of these calls carefully to determine which you should use for what kind of data manipulation, and also refer to the *AOS Programmer's Manual* for more details about file input/output.

Table 6-1 summarizes four characteristics of DG/L I/O calls.

For other kinds of DG/L input/output, see the other sections in this chapter: terminal, shared page, and cache memory I/O.

Table 6-1. Types of File I/O

Call	Record Size	Sequential	Buffered	Related Calls
BLKREAD BLKWRITE	256-word	No	No	OPEN, APPEND, EOPEN, ROPEN, CLOSE
GRDB GWRB	256-word	No	No	GOPEN, GCLOSE
LINEREAD LINEWRITE	data-sensitive	Yes	No	OPEN, APPEND, EOPEN ROPEN, CLOSE
READSTRING WRITESTRING	data-sensitive dynamic (bytes)	Yes Yes	No No	OPEN, APPEND, EOPEN, ROPEN, CLOSE
SCREENREAD	data-sensitive	Yes	No	OPEN, CLOSE
BYTEREAD BYTEWRITE	dynamic (bytes)	Yes	No	OPEN, APPEND, EOPEN, ROPEN, CLOSE
DATAREAD DATAWRITE	dynamic (words)	Yes	Yes	DATAOPEN, DATACLOSE
QREAD QWRITE	user-defined	NA	NA	NA

APPEND (B)

Opens a file for appending.

Format

APPEND (file number, filename *[,error label]*);

Arguments

file number	is an integer expression whose value will be associated with the open file.
filename	is a string expression that specifies the name of the file.
error label	is a statement label to which control transfers if an error occurs.

Example

```
APPEND (10,"MYFIL",IERR);
```

Notes

APPEND is similar to OPEN, but APPEND opens a file specifically for appending; i.e., the initial fileposition is the end of the file, rather than the beginning.

In writing to a line printer, APPEND will not generate an initial form feed, but OPEN will.

If you don't pass APPEND an error label, and the file doesn't exist, then APPEND creates the file. Otherwise, the error return is taken.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?OPEN (System call)

BLKREAD

Reads a series of blocks from a file.

Format

BLKREAD (file number, starting block, pointer, count *[,error label];*)

Arguments

file number	is an integer expression that specifies the number of the file you want to read.
starting block	is an integer expression that specifies the disk file block where you want to start reading. (The first block is 0.)
pointer	is a pointer expression that points to the first address in memory which you want to receive the blocks.
count	is an integer *variable* that specifies the total number of 256-word blocks you want read. If there is an error or an end-of-file condition, count will return the number of blocks actually read.
error label	is a statement label to which control transfers if an error occurs.

Example

```
POINTER FILPT;
OPEN (7,"DATA.OL");
BLKREAD (7,5,FILPT,SEGM,IERR);
COMMENT READ A SEGMENT OF BLOCKS
        INTO MEMORY AT FILPT;
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?RDB (System call)

BLKWRITE

Writes a series of blocks to a file.

Format

BLKWRITE (file number, starting block, pointer, count *[,error label]*);

Arguments

file number	is an integer expression that specifies the number of the file you want to read.
starting block	is an integer expression that specifies the first disk file block you want written. (The first block is 0.)
pointer	is a pointer expression that points to the first address in memory from which you want to write.
count	is an integer *variable* that specifies the total number of blocks you want written. If there is an error or an end-of-file condition, count returns the number of blocks actually written.
error label	is a statement label to which control transfers if an error occurs.

Example

```
POINTER FILPT;
OPEN (9,"FILE9");
BLKWRITE (9,4,FILPT,NUMB,IERR);
COMMENT WRITES NUMBER OF BLOCKS SPECIFIED
        IN NUMB FROM MEMORY AT FILPT
        INTO FILE 9 AT BLOCK 4;
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?WRB (System call)

BYTEREAD (B)

Reads a specified number of bytes from a file into memory.

Format

BYTEREAD (file number, pointer, count *[,error label]*);

Arguments

file number	is an integer expression that specifies the number of the file you want to read.
pointer	is a pointer expression that points to the first address in memory to receive the data.
count	is an integer expression that specifies the total number of bytes you want to read.
error label	is a statement label to which control transfers if an error occurs.

Notes

On an end-of-file error condition, count will be set to the total number of bytes read. As a result, count should not be a constant. It should be an expression, a variable, or a parenthesized constant.

BYTEREAD is a dynamic, binary read. See the *AOS Programmer's Manual* under ?READ for more details.

Example

```
POINTER IPNT;
OPEN (14,"INPUTDATA");
BYTEREAD (14,IPNT,ICNT,IERR);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes
CHANNEL-RELATED codes.

Reference

?READ (System call)

BYTEWRITE (B)

Writes a specified number of bytes to a file.

Format

BYTEWRITE (file number, pointer, count *[,error label]*);

Arguments

file number	is an integer expression that specifies the number of the file you want to write to.
pointer	is a pointer expression that points to the first address of the data in memory.
count	is an integer expression that specifies the number of bytes you want written.
error label	is a statement label to which control transfers if an error occurs.

Example

```
POINTER MESAREA;
OPEN (7,"OUTDATA");
BYTEWRITE (7,MESAREA,2,IERR);
        /*WRITE 2 BYTES*/
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?WRITE (System call)

CHANNEL

Gets an AOS channel equivalent to a DG/L channel.

Format

i := CHANNEL (DG/L channel)

Arguments

i	is an integer variable that returns the equivalent AOS channel number from the Process Table.
DG/L channel	is the DG/L channel number.

Example

```
READ_PACKET [$ICH] := CHANNEL(0);
```

Error Conditions

No error condition can occur.

CLOSE (B)
Closes a file.

Format

CLOSE (file number *[,error label]);*

Arguments

file number	is an integer expression that specifies the channel number of the file you want to close.
error label	is a statement label to which control transfers if an error occurs.

Example

```
CLOSE (7,IERR);
```

Error Conditions

The following error codes may be returned:

ERIBM Inconsistency in block allocation map.

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?CLOSE (System call)

DATACLOSE
Closes a file opened for word buffering.

Format

DATACLOSE (file number *[,error label]*);

Arguments

file number	is an integer expression that specifies the open file. The number was assigned in the DATAOPEN call.
error label	is a statement label to which control transfers if an error occurs.

Example

```
DATACLOSE (5);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Notes

See DATAOPEN, DATAREAD, and DATAWRITE.

Reference

?GCLOSE (System call)

DATAOPEN

Opens a file at its beginning for word buffering.

Format

DATAOPEN (file number, filename *[,error label]*);

Arguments

file number	is an integer expression whose value will be associated with the filename.
filename	is a string expression that specifies the file you want to open. If the file does not exist, it is created.
error label	is a statement label to which control transfers if an error occurs.

Example

```
DATAOPEN (6, "NOCAN.DO");
```

Notes

If *error label* isn't passed, and the file doesn't exist, then the file is created. If *error label* is passed and the file doesn't exist, the error label is taken.

See DATACLOSE, DATAREAD, and DATAWRITE.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?GOPEN (System call)

DATAREAD

Does a buffered read from a file into a specified area of memory.

Format

DATAREAD (file number, address, *[count ,error label]*);

Arguments

file number	is an integer expression that specifies the channel number of the file you want to read from.
address	is a pointer expression that points to the first address in memory to receive the data.
count	is an integer *variable* that specifies the number of words you want to read. If an error or an end-of-file is encountered, count is set to the number of words actually read. If count is not passed, one word is read.
error label	is a statement label to which control transfers if an error occurs.

Example

```
DATAREAD (3,ADDRESS(DATAARRAY),5,EOF3);
```

Notes

You must close and reopen a file with DATACLOSE and DATAOPEN between DATAREAD and DATAWRITE.

DATAREAD reads 256-word blocks.

Error Conditions

The following error codes may be returned:

AISIZ Illegal size specified ($<=0$).

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?RDB (System call)

DATAWRITE

Does a buffered write of a specified number of words from memory to a file.

Format

DATAWRITE (file number, address *[,count [,error label]]*);

Arguments

file number	is an integer expression that specifies the number of the file you want to write to.
address	is a pointer expression that points to the first location in memory where you want to write.
count	is an integer expression that specifies the number of words you want written. If count is not passed, one word will be written.
error label	is a statement label to which control transfers if an error occurs.

Notes

You must close and reopen a file with DATACLOSE and DATAOPEN between DATAREAD and DATAWRITE.

DATAWRITE writes 256-word blocks.

Example

```
DATAWRITE (4,ADDRESS(AREA1),2);
```

Error Conditions

The following error codes may be returned:

AISIZ Illegal size specified (=0).

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?WRB (System call)

EOPEN

Exclusively opens a file.

Format

EOPEN (file number, filename *[,error label]*);

Arguments

file number	is an integer expression whose value will be associated with the opened file.
filename	is a string expression that specifies the file you want to open.
error label	is a statement label to which control transfers if an error occurs.

Example

```
EOPEN (6, "NOCAN.DO");
```

Notes

The file is opened with the fileposition at its start.

If the named file does not exist, EOPEN creates a random file (file element-size of one and indexed).

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?OPEN (System call)

FILEPOSITION (B)

Computes the file pointer position.

Format

FILEPOSITION (file number, position *[,error label]*);

Arguments

file number	is an integer expression that specifies the file.
position	is an integer array, double-precision integer variable, or other two-word entity that receives the 32-bit file position value (in bytes). The first byte of the file is byte 0.
error label	is a statement label to which control transfers if an error occurs.

Example

```
INTEGER ARRAY POSTN [1:2];
FILEPOSITION (2,POSTN);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?GPOS (System call)

GCLOSE

Closes a file opened for AOS block input/output.

Format

GCLOSE (file number *[,error label]*);

Arguments

file number	is an integer expression whose value was associated with the file in a call to GOPEN.
error label	is a statement label to which control transfers if an error occurs.

Example

```
GCLOSE (5);
```

Notes

This call is used with GOPEN, GRDB, and GWRB to perform block input/output on magnetic tape, disks, or Multiprocessor Communications Adapters (MCA) links. You can access any block within a tape volume.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?GCLOSE (System call)

GOPEN

Opens a file for AOS block input/output.

Format

GOPEN (file number, pathname, return packet
[,error label]);

Arguments

file number	is an integer expression whose value will be associated with the opened file.
pathname	is a string that specifies the file you want to open.
return packet	is the name of a four-word entity that receives the information packed for the opened file. See the *AOS Programmer's Manual* for its contents.
error label	is a statement label to which control tranfers if an error occurs.

Example

```
GOPEN (5, "DATA.BLOCKS", BLOCK_10);
```

Notes

This call is used with GCLOSE, GRDB, and GWRB to perform block input/output on magnetic tape, disks, or MCA links. You can access any block within a tape volume. See the *AOS Programmer's Manual* for special considerations about tape and MCA input/output.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?GOPEN (System call)

GRDB

Reads blocks of data from tape, disk, or MCA links into memory.

Format

GRDB (file number, packet, count *[,error label]*);

Arguments

file number	is an integer expression whose value was associated with the file in a call to GOPEN.
packet	is the name of a seven-word entity that defines the block's input/output parameters. See the *AOS Programmer's Manual* under ?RDB for its contents.
count	is an integer variable that specifies the number of bytes you want read, and returns the number of bytes that were read.
error label	is a statement label to which control transfers if an error occurs.

Example

```
GRDB (5,BLOCK_10,I,EOF);
```

Notes

Specify disk and tape blocks by block number; MCA links by link number. You can access any block within a tape volume.

You use this call with GCLOSE, GOPEN, and GWRB to perform block input/output on magnetic tape, disks, or MCA links.

See the *AOS Programmer's Manual* for special considerations about tape and MCA input/output.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?RDB (System call)

GWRB

Writes blocks of data from memory to tape, disk, or MCA links.

Format

GWRB (file number, packet, count */,error label/*);

Arguments

file number	is an integer expression whose value was associated with the file in a call to GOPEN.
packet	is the name of a seven-word entity that defines the block's input/output parameters. See the *AOS Programmer's Manual* under ?WRB for its contents.
count	is an integer variable that specifies the number of bytes that you want written, and receives the number of bytes that were written.
error label	is a statement label to which control transfers if an error occurs.

Example

```
GWRB (5, BLOCK_10, I);
```

Notes

Specify disk and tape blocks by block number, and MCA links by link number. You can access any block within a tape volume.

You use this call with GCLOSE, GOPEN, and GRDB to perform block input/output on magnetic tape, disk, or MCA links.

See the *AOS Programmer's Manual* for special considerations about tape and MCA input/output.

Error Conditions

The following error code may be returned:

FILE SYSTEM codes.

Reference

?WRB (System call)

LINEREAD (B)

Does a data-sensitive read of a line from a file into memory.

Format

```
LINEREAD (file number, address, return-count /,error
      label/ );
```

or

```
LINEREAD (file number, address, return-count,
      limit-count, error label );
```

Arguments

file number	is an integer expression that specifies the number of the file you want to read.
address	is an expression used as a pointer to the first address in memory to receive the line. The memory space should be at least limit count bytes long.
return-count	is a dummy variable that receives the number of bytes read if the system detects an error or an end-of-file condition.
limit-count	is an integer expression that defines the maximum count you want to use in the read. If LINEREAD reads the maximum count without reaching a line delimiter, the call will terminate and control will pass to the error label.
error label	is a statement label to which control transfers if an error occurs.

Example

```
INTEGER IBYTE;
LINEREAD (T,ADDRESS(ARAYD),IBYTE,IERR);
IF IBYTE = 1 GO TO NOINPUT;
SETCURRENT (ARAYD, IBYTE);
COMMENT BRANCH IF TERMINATOR IS THE ONLY
      CHARACTER;
```

LINEREAD (B) (continued)

Notes

If limit count is not passed, 133 is used.

The error label argument is not optional in the limit-count calling sequence.

See the *DG/L™ Reference Manual* for more notes on the use of LINEREAD.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?READ (System call)

LINEWRITE (B)

Does a data-sensitive write of a line from memory to a file.

Format

LINEWRITE (file number, pointer, return-count */,error label/*);

or

LINEWRITE (file number, pointer, return-count, limit-count, error label);

Arguments

file number	is an integer expression that specifies the number of the file you want to write to.
pointer	is a pointer expression that points to the first address you want to write out.
return-count	is a dummy integer variable that receives the number of bytes actually written if LINEREAD reached an end-of-file or over limit count characters.
limit-count	is an integer expression that defines the maximum count you want to use in the read. If LINEREAD reads the maximum count without reaching a line delimiter, it will terminate the read and go to the error label.
error label	is a statement label to which control transfers if an error occurs.

Example

```
LINEWRITE(4,ADDRESS("HI THERE<NUL>"),9);
        /*WRITE "HI THERE" <NUL>*/
```

Notes

If limit count isn't passed, 133 is used.

No delimiter is added at the end of a write terminated by the limit-count.

The error label is not optional in the limit-count calling sequence.

See the *DG/L™ Reference Manual* for additional information on the use of LINEWRITE.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?WRITE (System call)

OPEN (B)

Opens a file for reading, writing, or appending.

Format

OPEN (file number, filename *[,error label]*);

Arguments

file number	is an integer expression whose value will be associated with the opened file.
filename	is a string expression that specifies the name of the file.
error label	is a statement label to which control transfers if an error occurs.

Example

```
OPEN (6,"MYFILE",IERR);
```

Notes

The file is opened at its start.

If the file doesn't exist, then, if an error label is passed, it will be jumped to; or, if no error label is passed, a random file will be created.

See the *DG/L™ Reference Manual* for additional notes on OPEN.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?OPEN (System call)

POSITION (B)

Changes the position of a file pointer.

Format

POSITION (file number, position *[,error label]*);

Arguments

file number	is an integer expression that specifies the file.
position	is a two-element array, a double precision integer expression, or some other two-word entity with a two-word integer value that points to the byte where the file will be positioned. (The first byte of the file is zero.)
error label	is a statement label to which control transfers if an error occurs.

Example

```
INTEGER ARRAY POSTN [1:14];
POSITION (2,POSTN [1],IERR);
```

Notes

To reposition the file at the beginning, rewind it with the call

```
POSITION (file number, 0P2);
```

See the *AOS Programmer's Manual* for a table of position values.

You may use position to restore a position saved by a call to FILEPOSITION.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?SPOS (System call)

QCLOSE

Closes an open file or device using AOS parameters.

Format

QCLOSE (packet *[,error label]);*

Arguments

packet	is an array that contains the input/output parameters set in the call to QOPEN.
error label	is a statement label to which control transfers if an error occurs.

Example

```
QCLOSE (PACKET1);
```

Error Conditions

The following error codes may be returned:

ERIBM Inconsistency in block allocation map.

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?CLOSE (System call)

QOPEN

Opens a file using AOS parameters.

Format

QOPEN (packet *[,error label]*);

Arguments

packet	is a 12-word array that defines the file type, format record, and access type. See the table of parameters in the *AOS Programmer's Manual* under ?OPEN.
error label	is a statement label to which control transfers if an error occurs.

Example

```
QOPEN (PACKET1);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Notes

QOPEN gives you the flexibility of an assembly-language ?OPEN call, since you provide all packet information.

Reference

?OPEN (System call)

QREAD

Reads from a file using AOS parameters.

Format

QREAD (packet *[,error label]*);

Arguments

packet	is a 12-word entity that defines the parameters for the read operation. See the *AOS Programmer's Manual* for a table of input/output parameters.
error label	is a statement label to which control transfers if an error occurs.

Example

```
QREAD (PACKET1, EOF);
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Notes

QREAD gives you the flexibility of an assembly-language ?READ call, since you provide all packet information.

Reference

?READ (System call)

QWRITE

Writes to a file using AOS parameters.

Format

QWRITE (packet *[,error label]);*

Arguments

packet	is a 12-word entity that defines the parameters for the write operation. See the *AOS Programmer's Manual* for a table of input/output parameters.
error label	is a statement label to which control transfers if an error occurs.

Example

```
QWRITE (PACKET1);
```

Notes

QWRITE gives you the flexibility of an assembly-language call to ?WRITE, since you provide all packet information.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?WRITE (System call)

READSTRING

Does a data-sensitive read into a string variable.

Format

READSTRING (file number, string *[,error label]*);

Arguments

file number	is an integer expression whose value is associated with a channel.
string	is a string which will receive the data.
error label	is a statement label to which control transfers if an error occurs.

Example

```
READSTRING (3, "We are always ready", ERR);
```

Notes

READSTRING is functionally similar to LINEREAD, and performs a data- sensitive read. It takes its maximum length to be read from the string's maximum length, and sets the length read into the string's current length. Thus, unlike LINEREAD, no subsequent SETCURRENT call is necessary.

The string argument can also be a substring.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?READ (System call)

ROPEN

Opens a file for read access only.

Format

ROPEN (file number, filename *[,error label]*);

Arguments

file number	is an integer expression whose value will be associated with the opened file.
filename	is a string expression that specifies the file you want to open.
error label	is a statement label to which control transfers if an error occurs.

Example

```
ROPEN (6, "NOCAN.DO");
```

Notes

If the file does not exist, it will not be created.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?OPEN (System call)

SCREENREAD

Does an AOS screen edit read.

Format

SCREENREAD (file number, in-string,*[,error label];*)

or

SCREENREAD (file number, in-string, old string,
cursor position *[,row, column [,error label]]*);

Arguments

file number	is an integer expression that specifies the opened file.
in-string	is a string variable that specifies the string to be read in.
old string	is a string variable or constant which you want write out to the screen for editing before reading in.
cursor position	is an integer expression that specifies the position of the cursor in old string.
row	is an integer expression specifying the row on the screen where old string will appear.
column	is an integer expression specifying the column on the screen where old string will appear.
error label	is a statement label to which control transfers if an error occurs.

Examples

```
SCREENREAD (3, STRVAR, ERR);
```

or

```
SCREENREAD (3, STRVAR, "WE ARE DEVO,"
        11, 10, 15, ERR);
```

SCREENREAD (continued)

Notes

If the optional arguments aren't used, a CTRL-A allows the user to retrieve the previous buffer contents; for example, the previous CLI command.

If the optional pair of arguments is passed, SCREENREAD displays **old string** and puts the cursor in the **cursor position** in the string. The string can then be modified on the terminal using edit commands, before it's read in. The cursor position assumes that the first character position is 0, not 1.

old string and **cursor position** must both be included or both left out. The same is true of **row** and **column**, which must appear together. Row 0, column 0 is the upper lefthand corner of the screen.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?READ (System call)

WRITESTRING

Writes a passed string onto a file.

Format

WRITESTRING (file number, string *[,error label]*);

Arguments

file number	is an integer expression whose value is associated with a file.
string	is a string to be written.
error label	is a statement label to which control transfers if an error occurs.

Example

```
WRITESTRING (3, STRVAR, ERR);
```

Notes

WRITESTRING uses the current length of the string to determine how many characters to write. Note that READSTRING and WRITESTRING aren't exactly complementary, since WRITESTRING does a simple binary write, not a data-sensitive one.

The **string** argument may also be a substring.

Error Conditions

The following error conditions may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Reference

?WRITE (System call)

.CONSOLE

Writes to the terminal without using the stack (assembly language call).

Format

JSR @.CONSOLE

Arguments

AC0	contains an integer that specifies the type of message you want to write:	
	0	Write an octal number plus two spaces.
	-1	Write a carriage return or line feed.
	other value	AC0 is the word address of a string.
AC1	contains the message number if AC0 is set to zero.	

Example

```
        JSR    STKOF              ;put text
        .TXT   "STACK OVERFLOW"   address in AC3

STKOF:  MOV    3,0                ;put text
        JSR    @.CONSOLE          address in AC0
                                  ;write out
                                  "STACK
                                  OVERFLOW"
```

Notes

This is an assembly language call only.

See the runtime call TWROPERATOR in Chapter 9 for two related procedures, EXTERNAL PROCEDURE LOCKCONSOLE and UNLOCKCONSOLE, which control the use of .CONSOLE in a multitasking environment.

Error Conditions

No error condition can occur.

GETCINPUT

Obtains the input filename.

Format

s := GETCINPUT *[(error label)]*

Arguments

s	is a string variable that receives the name of the current input file, @ INPUT.
error label	is a statement label to which control transfers if an error occurs (RDOS compatible).

Example

```
OPEN (0, (GETCINPUT));
```

Notes

This call provides source code compatibility with RDOS programs. Under RDOS, GETCINPUT returns the name of the input terminal; under AOS, it always returns @INPUT.

Error Conditions

No AOS error condition can occur.

GETCOUTPUT
Obtains the output filename.

Format

s := GETCOUTPUT *[(error label)]*

Arguments

s	is a string variable that receives the name of the current output file, @OUTPUT.
error label	is a statement label to which control transfers if an error occurs (RDOS compatible).

Example

```
OPEN (1, (GETCOUTPUT));
```

Notes

This call provides source code compatibility with RDOS programs. Under RDOS, GETCOUTPUT returns the name of the output terminal; under AOS, it always returns @OUTPUT.

Error Conditions

No AOS error condition can occur.

GKI
Performs polled keyboard input.

Format

GKI (channel number, input word, *[,error label]*);

Arguments

channel number	is an integer value associated with the number of the terminal channel.
input word	is an integer variable which receives the character input from the terminal.
error label	is a statement label to which control transfers if an error occurs.

Notes

GKI keeps a constant one-byte ?READ call on the channel given, so that the latest key hit is always refleted in input word.

You must clear input word yourself after each use (see Figure 6-1).

GKI is primarily meant for use in a single-task program. If you use GKI in a multitasking program, be aware that it works by starting up a *task* (which reads a byte from the channel). It uses task # 2, runs at priority 1, and causes the calling task (or procedure) to run at priority 2.

GKI leaves the character read in the low order byte of input word.

GKI is not reentrant.

If the input word is a function key (the upper row of keys on a DASHER terminal) bit 8 of input word is set, otherwise bit 8 is zero.

Error Conditions

The following error conditions may be returned:

FILE SYSTEM codes.
CHANNEL-RELATED codes.
TASK codes.

```
An example of GKI and KGKI's use for the main loop of a real-time
space game:

INTEGER INPUT_WORD;
OPEN(0,(GETCINPUT));
OPEN(1,(GETCOUTPUT));

WHILE PLAY_AGAIN DO
BEGIN
        GKI(0, INPUT_WORD);

        DO BEGIN
                GOTO ACTION[INPUT_WORD];
                GTO NEXT;       /*ignore bad input*/

        ACTION[31R8]:           /*left cursor arrow*/
                MOVE_LEFT;
                GOTO NEXT;

        ACTION[30R8]:           /*right cursor arrow*/
                MOVE_RIGHT;
                GOTO NEXT;

        ACTION[10R8]:           /* 'home' keyj */
                TOGGLE_HYPERSPACE;
                GOTO NEXT;

        ACTION[106R8]:          /* F */
        ACTION[141R8]:          /* f */
                FIRE_TORPEDO;
                GOTO NEXT;

        ACTION[101R8]:          /* 'A' */
        ACTION[141R8]:          /* 'a' */
                GOTO ABORT;

        NEXT:
                INPUT_WORD := 0;        /* we are done with this input*/
                UPDATE_SCREEN;
                IF GOT_KILLED THEN GOTO ABORT;
        END of game loop;

ABORT:

        KGKI;

        WRITE(1,"Do you want to play again?");


        STRINGREAD(0, ANSWER);
        PLAY_AGAIN := IF ANSWER = "yes" THEN TRUE ELSE FALSE;
END of main loop;
```

Figure 6-1. Example of GKI and KGKI

KGKI

Kills the task that GKI sets up.

Format

KGKI (*[error label]*);

Arguments

error label	is a statement label to which control transfers if an error occurs.

Example

```
KGKI;
```

Notes

If you want to perform regular I/O from the same keyboard you have been polling, using GKI, first call KGKI.

Error Conditions

The following error codes maybe returned:

TASK codes.

ODIS

Disables terminal interrupts.

Format

ODIS;

Arguments

None.

Example

```
ODIS;
```

Notes

The system automatically enables program interrupts when a program is initiated, whether from the CLI or by a call to CHAIN or ACHAIN. To re-enable interrupts after issuing this call, use OEBL.

Error Conditions

No error condition can occur.

Reference

?ODIS (System call)

OEBL

Enables terminal interrupts.

Format

OEBL;

Arguments

None.

Example

OEBL;

Notes

The system automatically enables console interrupts when a program is initiated, including by a call to CHAIN or ACHAIN. This call re-enables interrupts after they have been disabled by ODIS.

Error Conditions

No error condition can occur.

Reference

?OEBL (System call)

Shared Memory I/O

DG/L shared input/output uses the AOS shared page memory functions. AOS divides user memory space into shared and unshared areas. The unshared area includes the user stack, heap, and ICOBs, as well as GLOBAL and OWN variables (see Figure 2-1 in Chapter 2). The system allocates shared memory for program code and runtime routines; i.e., the user's executable address space. The shared memory area comprises one to thirty-one pages; each page is a block of 2048 bytes (1024 words). Shared memory reduces system overhead by allowing more than one process access to the same physical memory pages.

Using the runtime routines in this section, you can associate data files, or parts of them, with the shared memory area. Thus, commonly-used data become accessible to more than one process at a time. When a process opens a file that's already in main memory, the system merely remaps the calling process's access, instead of performing I/O to load a new copy of the file.

And besides sharing data, shared page I/O is a method of performing interprocess communication quickly.

Shared-Page Calls

To perform input/output in the shared memory area, you must open a file with the runtime call SHOPEN and read the file into memory with SHREAD. To open a file with SHOPEN, you must pass a parameter packet to the system; see the *AOS Programmer's Manual* for a definition of the packet.

When you're finished using a shared page, release it with a call to PAGERELEASE. After this call, your process no longer has access to the page. If no other process is using the page, it's written to disk; otherwise, it remains in main memory.

To close the file and flush it to disk, call SHCLOSE. If you opened the file for other than read-only access, the system automatically writes out the shared pages to disk on closing; no write operation is necessary.

GETSHARED, SHPARTITION, and PAGERELEASE allow you to redefine your shared memory area. You may, for example, wish to change memory space allocation for different parts of a program. You may also want to increase the shared memory area if a call to SHREAD fails to find available memory.

GETSHARED returns the current size and page locations of your process's shared memory area. SHPARTITION redefines the pages the shared area includes.

Redefining Your Shared and Unshared Memory Areas

Each time the system changes the process it's executing, it may swap out the *unshared* memory area of a process to disk. To reduce this system overhead and provide more *shared* memory area, you can reduce the size of a process's unshared area.

There are three ways of doing this:

- Reset the value of the symbol .NMAX in the file DGLSYM.SR. .NMAX defines the highest numbered page that the DG/L initializer can take for stack/heap space (unshared area). .NMAX has an initial decimal value of 32. This value is reduced at runtime to accommodate the shared-page area. (See Figure 2-1 in Chapter 2 for an illustration of main memory during execution of a DG/L program.)

  If a program, for example, needs only 1K words of stack and heap space, you might set .NMAX to 2. The two pages would provide memory for the stack, heap, 256 words of ZREL, and the resident part of the initializer. All pages from page three up to the shared code are then free for allocation as shared pages.

  You *must* redefine .NMAX if your program calls SHPARTITION to increase the shared memory area. (See call in this section.) If you don't modify .NMAX, the error ERMEM (insufficient memory) occurs. See Appendix D, "User Options," for information on redefining parameters such as .NMAX.

- Specify the shared-page parameter when you start the program from another program with the DG/L runtime call PROC.

- Specify the shared-page parameter when you start the program using the assembly-language system call ?PROC.

- Use the PROC CLI command to start the program and use the optional /MEM= switch.

See Chapter 3 of the *AOS Programmer's Manual,* "Memory Management," for a full discussion of shared-page concepts.

GETSHARED

Lists the current size of the process's shared partition.

Format

GETSHARED (start page, size);

Arguments

start page	is an integer variable that receives the page number of the first shared page (1-31).
size	is an integer variable that receives the number of pages in the shared partition.

Example

```
GETSHARED (PGSTART,PGCOUNT);
```

Notes

If there is no shared partition, both **start page** and **size** return a value of zero.

Error Conditions

No error condition can occur.

Reference

?GSHPT (System call)

PAGERELEASE

Releases a shared page that a disk file has been read into.

Format

PAGERELEASE (address *[error label]*);

Arguments

address	is any address within the page that you want to release.
error label	is a statement label to which control transfers if an error occurs.

Example

```
PAGERELEASE (P,BADREL);
```

Notes

This call does not close the file.

Only after all its shared page users have closed it can you reopen the file for another purpose.

Error Conditions

The following error codes may be returned:

ERNSA Page is not within the shared area.

FILE SYSTEM codes.

Reference

?RPAGE (System call)

SHCLOSE

Closes a file opened for shared-page reading and flushes the read pages to disk.

Format

SHCLOSE (file number *[, error label]*);

Arguments

file number	is an integer expression whose value was associated with the file in the call to SHOPEN.
error label	is a statement label to which control transfers if an error occurs.

Example

```
SHCLOSE (10);
```

Notes

If you have opened the file for read/write access, the system assumes it is modified and flushes the file to disk at closing.

See SHOPEN and SHREAD.

Error Conditions

The following error codes may be returned:

CHANNEL-RELATED codes.

Reference

?SCLOSE (System call)

SHOPEN

Opens a file for shared-page I/O.

Format

SHOPEN (file number, filename, open type *[,error label]*);

Arguments

file number	is an integer expression whose value will be associated with the opened file.
filename	is a string expression that specifies the file you want opened.
open type	is an integer expression that you set to 0 for read-only access; any other value gives read/write access.
error label	is a statement label to which control transfers if an error occurs.

Example

```
SHOPEN (I := 0, "DATA", 0);
```

Notes

If you open with read/write access, the system will assume the file is modified and flush it to disk upon closing. Use the SHCLOSE call to close a shared-page file. See also SHREAD.

Error Conditions

The following error codes may be returned:

ERILO Attempt to open a file whose file element is not a multiple of four.

FILE SYSTEM codes.

Reference

?SOPEN (System call)

SHPARTITION

Establishes a new shared partition; i.e., changes the size of the shared area.

Format

SHPARTITION (start page, pages *[,error label]*);

Arguments

start page	is an integer expression that gives the page number (1-31) where the new partition begins.
pages	is an integer expression that gives the length in pages of the partition.
error label	is a statement label to which control transfers if an error occurs.

Example

```
SHPARTITION (PGSTART - 1,PGCOUNT + 1);
```

Notes

If you want to use SHPARTITION to increase shared pages, it's necessary first to modify .NMAX, or else specify fewer unshared pages in a PROC or ?PROC parameter. This is necessary because, before your program begins running, the DG/L initializer assigns *all* of a user's memory space that hasn't been allocated for the shared partition to unshared memory.

If you want to use SHPARTITION to decrease shared pages in memory, you must first release shared pages using PAGERELEASE. However, you cannot release any shared pages currently in use.

Error Conditions

The following error codes may be returned:

ERSHP Pages specify an unshared area or are illogical.
ERMEM Insufficient memory available.

Reference

?SSHPT (System call)

SHREAD

Reads sequential disk blocks from an open file into shared pages.

Format

SHREAD (file number, packet *[,error label]*);

Arguments

file number	is an integer expression whose value was associated with the file in a call to SHOPEN.
packet	is a seven-word ?SPAGE parameter packet that passes to the system and specifies the file blocks to read and the shared pages to read the blocks into.
error label	is a statement label to which control transfers if an error occurs.

Example

```
SHREAD (0,SHPACKET,EOF);
```

Notes

See the *AOS Programmer's Manual* for a table of the ?SPAGE parameter packet.

See also SHOPEN and SHCLOSE.

Error Conditions

The following error codes may be returned:

MEMORY codes.
FILE SYSTEM codes.
CHANNEL-RELATED codes.

Reference

?SPAGE (System call)

Cache Memory Management

Cache memory management (CMM) is a flexible method in DG/L for making large bodies of data accessible to a program. It allows random access to a file, improving performance by using buffers rather than always going to disk for each call.

You can move the data from a CMM buffer into your own data areas and perform any DG/L operation your program needs. You can read/write anywhere in the file, and read/write in multiple files using the same set of buffers.

Using CMM, your program addresses a file up to 64K elements. These elements may be from 1 to 256 words long, and are all the same size. Their size is user-selected at file open time. You can transfer as many words at a time as you wish, or only one word at a time. You can also read/write in terms of *nodes*. Nodes are blocks of data whose first word gives the block size, thus allowing variable sizes of blocks that are managed semi-transparently.

CMM manages all CMM data buffers transparently, and faster than if you were to do disk I/O for each reference of data.

The *DG/L™ Reference Manual* discusses the principles of Cache Memory Management in detail.

A variety of calls are available for setting up CMM access and performing different types of data movement:

- Create one or more buffer pools - BUFFER.
- Open a file for Cache Memory Management - ACCESS.
- Read/write explicitly to memory buffers using assignment statements - HASHWRITE, HASHREAD.
- Read/write any number of words - WORDWRITE, WORDREAD.
- Read/write a complete node (from file zero) - NODEWRITE, NODEREAD, NODESIZE.
- Read/write a single word (from file zero) - FETCH, STASH.
- After processing, write out modified buffers back to the disk file - HASHWRITE, FLUSH.

To reference nodes in file zero and words within the nodes, you need to determine the size of a node and the offset of the data from the node's first word. MINRES and NODESIZE return the necessary values. Refer to the *DG/L™ Reference Manual* for further information about these calls.

ACCESS (B)

Opens a file for Cache Memory Management, associates it with a buffer pool, and defines its element size.

Format

ACCESS (file number, filename, pointer *[,element size]*);

Arguments

file number — is an integer expression whose value will be associated with the open file.

filename — is a string expression that specifies the file.

pointer — is a pointer expression that points to a buffer pool created in a call to BUFFER. Usually this is the pointer returned by BUFFER.

element size — is an integer that indicates the number of words in a file element. Since you can access at most 64K elements, you specify the maximum size of your CMM file by the element size in the following table:

Element Size	Desired File Size
1	64K words
2	64-128K words
3	128-192K words
.	.
.	.
.	.

If you don't specify an *element size*, the system assumes the size is 1.

Example

```
POINTER BUFF;
BUFFER(BUFF,2000,7);
ACCESS(1,FILEA,BUFF);
```

Notes

You can access several files using the same buffer pool pointer.

If the file does not exist, the routine creates a randomly organized file.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?OPEN (System call)

See the *DG/L™ Reference Manual.*

BUFFER (B)

Creates a buffer pool area in memory and returns its address.

Format

BUFFER (pointer, pool size *[,virtual buffers]*);

Arguments

pointer	is a pointer variable that receives the address of the buffer pool.
pool size	is an integer expression that specifies the total number of words you want to allocate.
virtual buffers	is an integer expression that specifies the number of virtual RDOS memory buffers you want to create. AOS ignores this argument; it provides source code compatibility for AOS/RDOS programs.

Examples

```
BUFFER(BUF,2100,7);

BUFFER(BUFF,2000,NUMBUF-4);
```

Error Conditions

The following error conditions may occur:

AIBFS	Insufficient buffers.
ERICM	Illegal system command.
ERICD	Illegal command for device.
ERMEM	Insufficient memory available.

Reference

See the *DG/L™ Reference Manual.*

BUFLOCK (B)

Locks into cache memory a 256-word block

Format

BUFLOCK (file number, logical address *[,error label]*);

Arguments

file number	is an integer expression whose value is associated with the file.
logical address	is an integer expression whose value is an address within the block that will be locked into memory.
error label	is a statement label to which control transfers if an error occurs.

Example

```
BUFLOCK (MYBLOCK, DATA);
```

Notes

The block containing logical address will not be written out to disk even if it becomes the least recently used one. If the block is currently not in memory, it will be read in.

If all blocks are locked into memory, and you try to access a block not currently in memory, the process or task will terminate with a fatal error.

Error Conditions

The following error condition may occur:

AILCK All buffers already locked.

BUFUNLOCK (B)

Unlocks a 256-word block in cache memory.

Format

BUFUNLOCK (file no, logical address *[,error label]*);

Arguments

file no	is an integer expression whose value is associated with the file.
logical address	is an integer expression whose value is an address in the block which will be unlocked.
error label	is a statement label to which control transfers if an error occurs.

Example

```
BUFUNLOCK (FILE2, LOC);
```

Notes

The block containing the element at logical address will be unlocked, so that it may be written to disk if it becomes the least recently used block.

Error Conditions

No error condition can occur.

FETCH (B)

Retrieves a word from a CMM file on file zero.

Format

i := FETCH (*[logical address]* , offset)

Arguments

i	is an integer variable that receives the word.
logical address	is an integer that specifies the node. If unspecified, the system assumes its value is zero (the start of the file).
offset	is an offset into the node of the word to fetch.

Example

```
NEXT := FETCH(N,1);
```

Error Conditions

The following error conditions may occur:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

Notes

If MINRES (see call in this section) was -3, a FETCH(500,0) call would get the fourth word of element 500, not the first. FETCH assumes the first node word has offset MINRES, the next MINRES +1, etc. If logical address is left out (fetching from the start of disk file, useful for global parameters) then offset counts from 0, not MINRES.

References

?READ (System call)

See the *DG/L™ Reference Manual.*

FLUSH (B)

Writes out the contents of all modified buffers in the pool to disk.

Format

```
FLUSH (pointer);
```

Arguments

pointer	is a variable that points to the first address of the buffer pool (see BUFFER).

Example

```
FLUSH(BUFF1);
```

Notes

You must use FLUSH to update CMM files; CLOSE will not modify disk files. You should still CLOSE the file afterwards.

Error Conditions

The following error conditions may occur:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?WRITE (System call)

See the *DG/L Reference Manual.*

HASHREAD (B)

Reads a specified block of the hash file into memory.

Format

```
HASHREAD (file number, hash value, block pointer,
      offset);
```

Arguments

file number	is an integer expression whose value is associated with the file.
hash value	is an integer expression that specifies the block you want read in the file.
block pointer	is a variable that receives the address of the block read in.
offset	is an integer expression that specifies the location of the word in the block.

Notes

Do not call HASHREAD or HASHWRITE under multitasking.

Example

```
HASHREAD(1,ELEMENT,BLKNUM,BLKOFF);
DATA := (BLKNUM+BLKOFF) - BI
```

Error Conditions

The following error conditions may occur:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?READ (System call)

See the *DG/L™ Reference Manual.*

HASHWRITE (B)

Marks the current buffer as modified.

Format

HASHWRITE (pointer);

Arguments

pointer	is a variable that points to the first memory address of the buffer pool.

Notes

You must identify a modified buffer for it to be written out to disk under FLUSH.

Example

```
HASHWRITE(BUFF1);
```

Error Conditions

The following error conditions may occur:

ERFIU File is in use.

Reference

See the *DG/L™ Reference Manual.*

MINRES

Is a declaration to specify, modify, or otherwise access the default lower bound of the cache memory file.

Format

```
EXTERNAL INTEGER MINRES;
        .
        .
        .
MINRES := value
```

Arguments

value	is the offset to use in FETCH and STASH to access the first word in a node. Subsequent words will be value + 1, etc.

Notes

FETCH and STASH use MINRES to determine the word offset from the beginning of the node, by assuming the first word in the node is offest MINRES. MINRES has a default value of -3.

Example

```
BEGIN   MINRES := 2;
        INTEGER ARRAY LOC_ARRAY [MINRES:
        MINRES +
        NODESIZE (EMPTR)];
        END;
```

Error Conditions

No error condition can occur.

Reference

See the *DG/L™ Reference Manual.*

NODEREAD (B)

Reads a specified node from file zero into memory.

Format

```
NODEREAD (logical address, array);
```

Arguments

logical address	is an integer expression that specifies the logical (element) address in the file of the first word of the node.
array	is an integer array that receives the words.

Example

```
INTEGER ARRAY IDATA
[MINRES: MINRES + NODESIZE (ELEMENT)];
NODEREAD (ELEMENT,IDATA);
```

Notes

A node always starts at an element boundary; thus, no offset is needed. The first word of the node contains the node size.

Error Conditions

The following error conditions may occur:

AINOD Invalid node specified.

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?READ (System call)

See the *DG/L™ Reference Manual.*

NODESIZE (B)

Obtains the size of a node on file zero.

Format

```
i := NODESIZE (logical address)
```

Arguments

i	is an integer variable that receives the number of words in the node.
logical address	is an integer expression that specifies the logical (element) address in the file of the node's first word.

Example

```
BEGIN
        INTEGER ARRAY LOC ARRAY [MINRES:
        MINRES +
        NODESIZE (EMPTR)];
        / * ALLOCATE AN ARRAY TO NODEREAD
        EMPTR * /
```

Error Conditions

The following error conditions may occur:

AINOD Invalid node specified.

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?READ (System call)

See the *DG/L™ Reference Manual.*

NODEWRITE (B)

Writes a node from memory to file zero.

Format

NODEWRITE (logical address, array);

Arguments

logical address	is an integer expression that specifies the location you want to write to in the file.
array	is an integer array that contains the data you want written.

Example

```
NODEWRITE (LOC,IARAY);
```

Notes

The first word of the array must contain the number of words that will be written (the nodesize).

Error Conditions

The following error conditions may occur:

AINOD Invalid node specified.

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?WRITE (System call)

See the *DG/L Reference Manual.*

STASH (B)

Stores a word into a node on file zero.

Format

STASH (word */,logical address/* ,offset);

Arguments

word	is a variable containing the word you want to store.
logical address	is an integer expression that specifies the first address of the node array. If not included, the offset is an absolute offset from the start of the file.
offset	is the offset into the node of the word to fetch.

Examples

```
STASH(DATUM,N,0);
N := N + 1;

STASH(FETCH(P,$STAT) OR $DATABIT,P,$STAT);
        / * SET $DATABIT AT OFFSET $STAT IN
        NODE P * /
```

Notes

If MINRES (see call in this section) was -3 (the default value), a STASH(66,500,0) call would store the value 66 in the fourth word of element 500, not the first. STASH assumes that the first node word has offset MINRES; the next, MINRES +1, etc. If logical address is left out (fetching from the start of disk file, useful for global parameters), then offset counts from 0, not MINRES.

Error Conditions

The following error conditions may occur:

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?WRITE (System call)

See the *DG/L Reference Manual.*

WORDREAD (B)

Reads words from a file using cache modules.

Format

```
WORDREAD (file number, logical address, pointer
      [,count [,offset]] );
```

Arguments

file number	is an integer expression whose value is associated with the file (see ACCESS).
logical address	is an integer expression that specifies the logical (element) address into the file.
pointer	is a variable that points to the first location in memory to receive the data.
count	is an integer expression that specifies the number of words you want read. If you omit *count*, the routine uses the value of the first word read as the count; this is equivalent to NODEREAD.
offset	is an integer expression that specifies the number of words beyond the start of element logical address where you want reading to begin. If you omit offset, it's treated as zero.

Example

```
WORDREAD (1,256,ADDRESS(ARU)2,6)
COMMENT READ 2 WORDS, STARTING 6 WORDS
                PAST START OF ELEMENT 256;
```

Error Conditions

The following error conditions may occur:

AINOD Invalid node specified.

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?READ (System call)

See the *DG/L™ Reference Manual.*

WORDWRITE (B)

Writes words to a file using cache modules.

Format

```
WORDWRITE (file number, logical address, pointer
      [,count [,offset]] );
```

Arguments

file number	is an integer expression whose value is associated with the file (see ACCESS).
logical address	is an integer expression that specifies the logical (element) address into the file.
pointer	is an expression that points to the first word in memory you want written to the file.
count	is an integer expression that specifies the number of words you want written. If you omit *count*, the routine uses the value of the first word to be written as the count; this is equivalent to NODEWRITE.
offset	is an integer expression that specifies the number of words beyond the start of logical address where you want to begin writing. If you omit *offset*, it's treated as zero.

Example

```
WORDWRITE(1,2,ADDRESS(ARU),2,3)
      /*WRITE 2 WORDS, STARTING 3 WORDS
      PAST START OF
      ELEMENT 2*/
```

Error Conditions

The following error conditions may occur:

AINOD Invalid node specified.

FILE SYSTEM codes.
SYSTEM CALL codes.
CHANNEL-RELATED codes.

References

?WRITE (System call)

See the *DG/L™ Reference Manual.*

End of Chapter

Chapter 7
Process Communication, Manipulation, and Monitoring

This chapter is divided into three sections: "Process Manipulation and Monitoring," "Interprocess Communication," and "Swapping and Chaining Processes." Most of the calls in this chapter reference AOS system calls to perform process functions.

AOS is a multiprogram system; it allows several separate users to compete simultaneously for memory and CPU time. Each competing program currently in the system is called a *process*. Different kinds of processes (batch programs, compilations, and interactive processes) can all run at a given time.

For a detailed discussion of multiprocessing and the structure of the multiprocess environment, see the *AOS Programmer's Manual,* under "Process Creation and Management." Don't confuse multiprocessing with *multitasking,* discussed in Chapters 8 and 9 of this manual. Each process has a separate address space, while tasks share the same memory space within a single process.

The DG/L runtime library provides the full range of AOS multiprocessing calls. Those in the first section, "Process Manipulation and Monitoring," allow you to initiate a process, and monitor and manage its execution. The calls in the next section, "Interprocess Communication," return information from and handle communication between processes. The third and last section in this chapter, "Swapping and Chaining," discusses how you move entirely new files between your process's address space and disk storage.

Process Manipulation and Monitoring

Initiating a Process

A call to PROC initiates a process. To initiate a process, you must provide PROC with a parameter packet: a 16-word entity that defines the process's name, size, program, I/O file, and system privileges. (You can also start up processes using the CLI commands PROC and XEQ.)

Defining A Process

Each process has a number and a name. The number is an integer the system assigns the process on initiation, called the process ID or PID. The name consists of the username followed by a colon and the process name given in the packet. If one user runs several processes, each process has the same username and a different process name.

Whenever a call in this manual requires a process name, you may supply either the full process name or the simple process name.

Type

You must assign a type and you may assign a priority to a process when you initiate it. These two parameters determine the process's eligibility for memory and CPU time. AOS defines three process types:

Memory-resident	processes have first claim to memory space. When a memory-resident process is initiated, it enters memory immediately and remains there until termination.
Pre-emptible	processes may be written out to disk if no memory is available for a memory-resident process.
Swappable	processes enter memory only when all memory-resident and pre-emptible processes have been accommodated. Swappable processes are written out to disk if a higher-status process supersedes them, or if they enter a blocked state and the system needs the memory space for another process.

Most AOS processes are swappable. To change a process's type during execution, use the runtime call CHTYPE.

Priority

When you assign a process a priority level, you define its place in the competition for CPU time. When any two processes of the same type are ready to use the CPU, the one with a higher priority receives control. (Process priorities work like task priorities in a multitask environment; see Chapter 8, "Multitasking: An Overview".)

Eligibility

The system may define a process as eligible, ineligible, or blocked from use of the CPU. *Eligible* processes are ready for the CPU and have no outstanding I/O calls. An *ineligible* process is also ready, but hasn't yet been allocated memory space by the system.

A *blocked* process is waiting for an external event to occur; if it is a swappable process, it's written out to disk.

You can unconditionally suspend any process's execution by calling BLOCKPR, provided the calling process has the privilege to suspend execution. UNBLOCKPR unblocks a blocked process. These calls allow you to coordinate the activity of related processes.

Monitoring

With DG/L runtime routines, you can generate reports containing a process's use of computer resources. RUNTIME returns *runtime statistics:* a report of the process's time in the system, CPU use, and I/O operations.

You can also create a *histogram* that continuously monitors how often the process has control of the CPU. Begin histogram monitoring with IHIST; and end a histogram with KHIST.

BLOCKPR

Blocks a subordinate process.

Format

BLOCKPR (process, flag *[,error label]*);

Arguments

process	is either a string expression giving the process name or an integer expression giving the PID of the process you want blocked.
flag	indicates the contents of **process**; set it to -1 for a process name, 0 for a PID.
error label	is a statement label to which control transfers if an error occurs.

Example

```
BLOCKPR (10,0);
```

Notes

You must call UNBLOCKPR to remove the block.

Error Conditions

The following error codes may be returned:

ERPRH Attempt to access a process not in the process tree.

FILE SYSTEM codes.

Reference

?BLKPR (System call)

CHPRIORITY

Changes the AOS task priority of a process.

Format

CHPRIORITY (process, flag, new priority *[,error label]*);

Arguments

process	is either a string expression specifying the pathname or an integer expression specifying the PID of the process.
flag	is an integer expression that indicates the contents of process; set it to -1 if you have given a pathname, and to any other value to indicate a PID.
new priority	is an integer that specifies the task's new priority (0-255).
error label	is a statement label to which control transfers if an error occurs.

Example

```
CHPRIORITY ((TIDENTITY),(0),1);
```

Notes

You must have the ?PVPR privilege in your user profile to use this call. See Chapter 4 of the *AOS Programmer's Manual* for more details on priority-changing privileges.

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?PRIPR (System call)

CHTYPE

Changes the type of a process.

Format

CHTYPE (process, flag, new type *[,error label]*);

Arguments

process	is either a string expression that gives the name of the process or an integer expression that gives its PID.
flag	is an integer expression that indicates the contents of process; set it to -1 for a process name, 0 for PID.
new type	is the new process type:

Mnemonic	Code	Meaning
(the default type)	0	Swappable
?PRFP	1	Pre-emptible
?PFRS	2	Resident

error label	is a statement label to which control transfers if an error occurs.

Example

```
CHTYPE (5, $PFRS, NOCANDO);
```

Error Conditions

The following error codes may be returned:

ERPRH Attempt to reference a process not in the process tree.

FILE SYSTEM codes.

Reference

?CHTYPE (System call)

DEBUG

Enters the system debugger from your program.

Format

DEBUG *[(error label)]* ;

Arguments

error label	is a statement label to which control transfers if an error occurs.

Example

```
DEBUG;
```

Notes

You can return control to the statement following this call by issuing a "proceed" (P) command in the debugger.

Error Conditions

The following error codes may be returned:

ERNNF	Debugger shared library file does not exist (DEBUG.SL).

FILE SYSTEM codes.

Reference

?DEBUG (System call)

GBIAS

Obtains the system's bias factor.

Format

```
i :=GBIAS
```

Arguments

i	is an integer variable that returns the current system bias factor. If there is no current bias factor, i returns a value of zero. (See the *AOS Programmer's Manual* for more details on bias factors.)

Example

```
BIAS := GBIAS;
```

Error Conditions

No error condition can occur.

Reference

?GBIAS (System call)

IHIST
Initiates a histogram.

Format

IHIST (process, flag, packet *[,error label]*);

Arguments

process	is either a string expression giving the process name, an integer expression giving the PID of the specified process, or -1 for monitoring the calling process.
flag	is an integer expression that indicates the contents of process; set it to -1 for a process name, and to any other value for a PID or self monitoring.
packet	is a four-word entity that contains a histogram parameter packet. See the *AOS Programmer's Manual* for its contents.
error label	is a statement label to which control transfers if an error occurs.

Example

```
IHIST (-1,0,HPACK);
```

Error Conditions

The following error codes may be returned:

ERHIS	An attempt to define more than one histogram, or an illogical parameter.
ERPPV	Calling process is not a resident process.
ERMPR	Histogram array is outside caller's address space.

FILE SYSTEM codes.

Reference

?IHIST (System call)

KHIST
Terminates histogram monitoring.

Format

KHIST *[(error label)];*

Arguments

error label	is a statement label to which control transfers if an error occurs.

Example

```
KHIST;
```

Error Conditions

The following error codes may be returned:

ERHIS Caller has no active histogram.

Reference

?KHIST (System call)

PROC

Creates a process and returns a process identifier.

Format

PROC (PID, packet *[,error label]*);

Arguments

PID	is an integer variable that returns the number of the created process.
packet	is the name of an array or pointer to the first word of an area containing the 16-word ?PROC parameter packet. See the *AOS Programmer's Manual* for tables and descriptions of its contents.
error label	is a statement label to which control transfers if an error occurs.

Example

```
PROC (I, PROC_PAK,IERR);
```

Error Conditions

The following error codes may be returned:

ERPRV	Caller not privileged to make this call.
ERPRP	Illegal priority specified.
ERBMX	Illegal maximum size (caller's process).
ERPTY	Illegal process type.
ERPDF	Error in definition of User Status Table.
ERPRN	Attempt to create more than maximum number of processes.

Reference

?PROC (System call)

RUNTIME

Returns runtime statistics for a specified process.

Format

RUNTIME (process, flag, packet *[,error label]*);

Arguments

process	is either a string expression giving the process name or an integer expression giving the PID of the process whose statistics you want.
flag	is an integer expression that indicates the contents of process; set it to -1 for a process name, and any other value for a PID.
packet	is an eight-word entity that receives the runtime statistics. See the *AOS Programmer's Manual* under ?RUNTM for its contents.
error label	is a statement label to which control transfers if an error occurs.

Notes

RUNTIME can only return statistics for the calling process or one of its son processes.

The runtime statistics returned include: real- time elapsed since the process's creation (in seconds), CPU time used (in milliseconds), number of blocks read or written, and page usage time (in page-milliseconds).

To monitor CPU usage, use IHIST.

Example

```
RUNTIME ("PROC1.PR",0,RPACK);
```

Error Conditions

The following error codes may be returned:

ERPRH Attempt to reference a process not subordinate in the process tree.

FILE SYSTEM codes.

Reference

?RUNTM (System call)

SBIAS

Sets the system's bias factor.

Format

SBIAS (factor *[,error label]*);

Arguments

factor	is an integer expression whose value becomes the new bias factor.
error label	is a statement label to which control transfers if an error occurs.

Notes

Only the operator process may issue this call.

Example

```
SBIAS (NEWBIAS);
```

Error Conditions

The following error codes may be returned:

ERPRV Caller is not operator process.

Reference

?SBIAS (System call)

TERM

Terminates a process, with an optional message.

Format

TERM (process, flag *[[,header] ,error label]*);

Arguments

process	is either a string expression that names the process you want to terminate, an integer expression that gives the process's PID, or -1 to indicate the calling process will be terminated.
flag	is an integer expression that indicates the contents of process. Set it to -1 for a process name; otherwise set it to 0.
header	is a string variable containing a message you want to send to the next higher level process in an IPC header (on self-termination). If *header* is unspecified, the system sends a standard message.
error label	is a statement label to which control transfers if an error occurs.

Example

```
TERM (-1, 0, "GOODBYE FROM" !!
        USERNAME (-1,0));
```

Notes

A *header* may be passed only on self-termination.

The *header* cannot be an expression, only a variable.

TERM will not pass back a message properly if the message argument is a substring instead of a string.

TERM is commonly used to return a message to the CLI. If you execute a program with the command XEQ/S MYPROG, and the program terminates with TERM and passes a string to the CLI, then the CLI STRING will contain your message. This message can then be accessed (by using the pseudo-macro !STRING) to create a macro that runs your program and uses the output string to decide what to do next.

TERM (continued)

A null (zero length) string is also acceptable.

This call is also documented under "Chaining and Swapping Processes" later in this chapter.

Error Conditions

The following error codes may be returned:

ERPRH Attempt to reference a process not in the process tree.

FILE SYSTEM codes.

Reference

?TERM (System call)

UNBLOCKPR

Unblocks a previously blocked process.

Format

UNBLOCKPR (process, flag *[,error label]*);

Arguments

process	is either a string expression giving the process name or an integer expression giving the PID of the blocked process.
flag	is an integer expression that indicates the contents of process; set it to -1 for a process name, and set it to 0 for a PID.
error label	is a statement label to which control transfers if an error occurs.

Example

```
UNBLOCKPR ("LOWERPROC1.PR", -1);
```

Notes

This call unblocks a process blocked by BLOCKPR. If the process has no task ready to run at the time of the call, it returns to the blocked state.

Error Conditions

The following error codes may be returned:

ERPRH Attempt to reference a process not in the process tree.

FILE SYSTEM codes.

Reference

?UBLPR (System call)

USERNAME

Returns the username of a specified process.

Format

```
s := USERNAME (process, flag [,error label] );
```

Arguments

s	is a string variable that returns the username.
process	is either a string expression giving the process name, an integer expression giving the PID of the process whose username you want returned, or -1 if you want the name of the calling process.
flag	is an integer expression that indicates the contents of process; set it to -1 if it contains a name, and to 0 if it contains a PID.
error label	is a statement label to which control transfers if an error occurs.

Example

```
FOR I := 0 STEP 1 UNTIL 255 DO BEGIN
        S:= USERNAME (I,0,NOPID);
        FORMAT(O,"<NL>",S);
NOPID: END;
```

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?GUNM (System Call)

Interprocess Communication

AOS multiprocessing allows you to pass information from one process to another through Interprocess Communication (IPC). With IPC, you can define messages of varying length and pass them from one process to another. You use IPC messages to coordinate process activities, place and remove process blocks, and pass data and packets among processes.

The routines in this chapter make the full IPC package available through DG/L. These calls provide the necessary information for addressing a process and transferring the messages you define. Your process needs the ?PVIP privilege (see Chapter 4 of the *AOS Programmer's Manual*) to use these calls.

Transferring Messages

The calls PORTSEND and PORTRECEIVE transfer messages from one process to another. The system blocks a process waiting for a message at PORTRECEIVE until the message is sent. To use these calls, you must define a message header, identify the receiving process, and use a communications port number.

You can also send messages to a process's terminal with a call to ASEND. To use this call, you must define the receiver by process name, PID, or terminal name.

Headers

To use IPC, you must define an IPC header for each process. The header contains the location and size of the message you want to send. See Chapter 4 of the *AOS Programmer's Manual* for a list of its contents.

Ports

Processes use memory locations called *ports* to communicate. You can define up to 127 ports, numbered locally, for each process. When you make IPC system calls, the system assigns a global portnumber to each of the process's ports. The global portnumber is a 32-bit number unique for each port in the AOS multiprocess system.

To send a message to another process, you must use the global portnumber of the receiving process. GETGLOBAL translates a local portnumber into its global equivalent.

Obtaining Information

This chapter includes a number of (function) calls that return information about the calling process or other processes. Because global portnumbers and PIDs are set by the system, they must be treated as variables in IPC. You will need these values to locate receiving and sending processes.

You can also use calls that return a process name when you give them a number, such as the PID or portnumber of a process that sent a message. DADID returns the PID of a father process; PORTOWNER returns the PID of the process that sent a message; PIDENTITY returns the PID of the calling process, and PROCNAME takes a PID as input and returns the corresponding process name. All these calls make a program portable by allowing you to use functions to perform IPC.

For any runtime call that requires a process name as input, you may give either the full process name (with username) or the simple process name (pathname).

For a complete outline of IPC, see Chapter 4 of the *AOS Programmer's Manual.*

ASEND

Sends a message to a terminal or process.

Format

ASEND (receiver, message, flag *[,error label]*);

Arguments

receiver	may be a string expression specifying the name of the receiving terminal or process, or an integer expression giving the PID of the process.
message	is a string containing the message you want to send. Message can't be a substring.
flag	is an integer whose value identifies receiver: 0 contains a PID. 1 contains a process name. 2 contains a terminal name.
error label	is a statement label to which control transfers if an error occurs.

Example

```
ASEND ("@CON0","HELP...", 1);
```

Error Conditions

The following error codes may be returned:

ERNAC	Named device is not a console name.
ERMRD	Device has reception of messages disabled.
ERPRH	Attempt to reference a process not in the process tree.

FILE SYSTEM codes.

Reference

?SEND (System call)

DADID

Obtains the PID of the calling process's father process or of another father process.

Format

i := DADID (*[process [,error label]]*)

Arguments

i	is an integer variable that receives the father process's PID.
process	is an integer expression that gives the PID of the process whose father process you want to identify, or -1 to get caller's father's PID.
error label	is a statement label to which control transfers if an error occurs.

Example

```
S := "TERMINATING NORMALLY";
      .
      .
      .
ASEND ((DADID), S, 0);
```

Notes

If you pass no arguments, DADID returns the caller's father's PID.

Error Conditions

The following error codes may be returned:

PROCESS codes.
FILE SYSTEM codes.

Reference

?DADID (System call)

ENQUEUE

Queues a file entry into another process's queue.

Format

ENQUEUE (specifications, queue name *[,error label]*);

Arguments

specifications	is a string expression that names the entry you want queued, followed by any switches necessary to modify the way it will be output. The string must have this form: pathname/switch ... <NUL> It must terminate in a null. If there is a switch argument, such as a space, that would normally terminate the switch string, you can set the high-order bit in that character, preventing the system from interpreting the space as a terminator.
queue name	is a string expression specifying the queue that you want the file to enter.
error label	is a statement label to which control transfers if an error occurs.

Example

```
ENQUEUE ("PRINT_FILE<NUL>", "@LPT");
```

Error Conditions

The following error code may be returned:

FILE SYSTEM codes.

Reference

?ENQUE (System call)

EXEC

Requests EXEC to perform a function for the calling process.

Format

EXEC (packet *[,error label]*);

Arguments

packet	is the identifier of the calling process's ?EXEC parameter packet. See the *AOS Programmer's Manual* for a description of its contents.
error label	is a statement label to which control transfers if an error occurs.

Example

```
EXEC (TAPE_MOUNT_PACKET);
```

Notes

See the *AOS Operator's Guide* (9324400) for a description of EXEC.

Error Conditions

The following error codes may be returned:

ERESO	Caller is not a son process of EXEC.
ERRBO	Operator refused the request.
ERWMT	Attempt to dismount a reel or disk that was not mounted.
ERXNA	EXEC not available in this system.
ERXUP	Unknown function requested.

FILE SYSTEM codes.
IPC SYSTEM codes.

Reference

?EXEC (System call)

GETCPN

Gets a terminal portnumber.

Format

GETCPN (portnumber *[,portowner, flag, [,error label]]*);

Arguments

portnumber	is a double-precision integer that receives the terminal's portnumber.
portowner	is either an integer expression that gives the PID of a specified terminal or a string expression that gives the process name.
flag	is an integer expression that identifies the contents of *portowner;* set it to 0 for a PID, and to -1 for a process name.
error label	is a statement to which control transfers if an error occurs.

Example

```
GETCPN (CPN,(DADID), 0);
```

Notes

To return the calling process's terminal portnumber, omit the arguments *portowner* and *flag*.

Error Conditions

The following error codes may be returned:

FILE SYSTEM codes.

Reference

?GCPN (System call)

GETGLOBAL

Translates a local portnumber into its 32-bit global equivalent.

Format

GETGLOBAL (local, global *[,error label]*);

Arguments

local	is an integer expression that specifies the local portnumber.
global	is a double-precision integer that receives the global portnumber.
error label	is a statement label to which control transfers if an error occurs.

Example

```
GETGLOBAL (5,GPN);
```

Error Conditions

The following error codes may be returned:

ERIVP Invalid port number.

Reference

?TPORT (System call)

NAMEGROUND (UNIQUE)

Returns an identifier with the PID affixed.

Format

s := NAMEGROUND (name)

Arguments

s	is a string variable that returns the following: ?<PID><name>.TMP where PID is a 3-character representation of the current process' PID number.
name	is any string.

Example

```
OPEN (0,NAMEGROUND ("FILEA"));
```

Notes

This routine is useful for creating unique, temporary filenames to prevent cross-process interference.

The system maintains PIDs across program chains, as well as within the execution of a single .PR file. NAMEGROUND will yield the same unique names throughout the execution of a chained series of programs.

Under RDOS, NAMEGROUND returns an indication of the program ground. See the RDOS version of this manual for details.

UNIQUE is an alternate entry point for NAMEGROUND under AOS only.

Error Conditions

No error condition can occur.

Reference

?RNAME (System call)

PIDENTITY

Obtains the PID of the calling process.

Format

i := PIDENTITY

Arguments

i	is an integer variable that receives the PID.

Example

```
PROC (ID, PROCPAK,(PIDENTITY);
        / * START PROCESS, PASSING FATHER
        PROCESS'S PID * /
```

Error Conditions

No error condition can occur.

Reference

?PNAME (System call)

PORTOWNER

Obtains the PID of the process that owns a specified port.

Format

i := PORTOWNER (portnumber *[,error label]*);

Arguments

i	is an integer variable that returns the PID of the process the port is assigned to.
portnumber	is a double-precision integer that gives the port's global portnumber.
error label	is a statement label to which control transfers if an error occurs.

Example

```
ASEND (PORTOWNER(GLBLPORT), MESSAGE, 0);
```

Error Conditions

The following error codes may be returned:

ERIPV Portnumber not currently assigned.

Reference

?GPORT (System call)

PORTRECEIVE

Receives an Interprocess Communication.

Format

PORTRECEIVE (header *[,error label]*);

Arguments

header	is the calling process's IPC header (a seven-word array that receives IPC information). See the *AOS Programmer's Manual* for a list of the header's contents.
error label	is a statement label to which control transfers if an error occurs.

Example

```
PORTRECEIVE (HEADER);
```

Error Conditions

The following error codes may be returned:

IPC EXCEPTIONAL CONDITION codes.

Reference

?IREC (System call)

PORTSEND

Sends an Interprocess Communication to a designated port.

Format

PORTSEND (header *[,error label]*);

Arguments

header	is the sending process's IPC header (a seven-word array that contains IPC information). See the *AOS Programmer's Manual* for a list of the header's contents.
error label	is a statement label to which control transfers if an error occurs.

Example

```
PORTSEND (MYHEADER);
```

Error Conditions

The following error codes may be returned:

ERPRV Caller is not privileged for this action.

IPC EXCEPTIONAL CONDITION codes.

Reference

?ISEND (System call)

PROCNAME
Obtains the name of a process or its PID.

Format

PROCNAME (name, PID *[,error label]*);

Arguments

name — is a string variable (or an area, passed as a based string expression) that will return the name of the designated process, or pass the input string constant. (See "Notes" below.)

PID — is an integer variable or expression (not constant) that tells what process's request is made. (See "Notes" below.)

error label — is a statement label to which control transfers if an error occurs.

Example

```
PROCNAME (STR,(DADID));
```

Notes

Depending on whether you want to find out a process's name or a PID, you have several options for the PID argument:

PID	Action
0	PID returns the PID number corresponding to the process's name put in name.
-1	PID returns the caller's process PID and name returns the caller's process name.
> 0	name returns the process name corresponding to the PID given by PID.

PID can be a variable (if you want a result value, it should be), expression, or parenthesized constant. It should not be a simple constant, since PROCNAME always attempts to store a value in PID.

Error Conditions

The following error codes may be returned:

ERPRH Attempt to reference a process not in the process tree.

FILE SYSTEM codes.

Reference

?PNAME (System call)

Swapping and Chaining Processes

Swapping and *chaining* are two ways you can run programs of virtually unlimited length on either ECLIPSE or NOVA computers, by segmenting programs into separate units. In a program swap or chain, an executing process calls in from disk file a completely new program (.PR) file. Each new segment of code replaces the user's *entire* memory space. ("Overlays," described in Chapter 10, replace only a portion of the user's address space.)

Before we describe swapping and chaining, a note about terminology may be useful: code executing under an operating system, with its necessary system requirements for I/O, memory space, etc, is called a *process*. A *program* refers to either an entire systems program, an applications program, or a module of code. A *program file* refers to a file of code on disk.

Swapping

When you call SWAP, AOS suspends the calling process that is executing, and brings into memory a program file residing on disk. Under AOS, the system actually blocks the calling process, creates a son process, and begins executing the swapped-in son process.

Under normal termination, a swapped-in process returns control to the calling process. You may also use calls SYSRETURN or TERM to return control to the calling process (see individual call descriptions for details).

If the swapped-in program file generates a fatal error, the system will return control to the calling process. You can also write your own error procedure, as discussed in Chapter 4, "Handling Errors."

A swapped-in process can perform another swap. In this case, the third process's error return or normal return goes to the second process, not to the original process. If you want to call in more than one program file, but you don't want to create a hierarchy among the processes, use CHAIN or ACHAIN, instead of SWAP.

TERM is a useful way to pass a message to the father process during swapping.

Chaining

A CHAIN call also creates a new process, but instead of being a son of the calling process, a chained process replaces the calling process, taking over the same PID, characteristics, etc. On termination or an error condition, control returns to the next higher level, typically to the CLI.

ACHAIN is similar to CHAIN, but allows you to pass an IPC message to the new program segment. CHAIN is RDOS-compatible.

Operating Instructions

Compile and link program files you plan to swap or chain just as you would any other DG/L program. Each program file is an independent, executable file with a .PR extension. Remember that any process that uses that file must have *execute* access to it. (See Chapter 10, "Operating Instructions.")

ACHAIN

Terminates the current process (or program file) and executes a new process (or program file).

Format

ACHAIN (program *[,debug flag [,message [,error label]]]*);

Arguments

program — is the name of the file you want to execute.

debug flag — is an integer whose bit values specify the conditions of control when the new program enters memory, (bits are numbered from left to right (0-15).

Octal Value	Bit Code	
0	0B14	Do not flush outstanding IPC messages.
2	1B14	Flush outstanding IPC messages.
0	0B15	Pass control to the program at its entry point.
1	1B15	Pass control to the debugger.

message — is the name of a string variable that contains an IPC message header.

error label — is a statement label to which control transfers if an error occurs.

Example

```
ACHAIN ("NEXTPROG.PR",1);
```

Error Conditions

The following error codes may be returned:

ERNSW — Out of swap file space.
ERMEM — Insufficient memory available.

FILE SYSTEM codes.

Reference

?CHAIN (System call)

CHAIN (B)

Terminates the current process (or program file) segment and calls another disk file for execution.

Format

CHAIN (filename, *[pointer [,error label]]*);

Arguments

filename	is a string expression that names the file you want to execute.
pointer	is a pointer to an IPC header you're passing to the new process. (See the *AOS Programmer's Manual* for what to put in the IPC packet.)
error label	is a statement label to which control transfers if an error occurs.

Example

```
CHAIN ("FIL.PR",TDAT,IERR);
```

Error Conditions

The following error codes may be returned:

ERNSW	Out of swap file space on disk.
ERMEM	Insufficient memory available.

FILE SYSTEM codes.

Reference

?CHAIN (System call)

ERETURN

Terminates a process and indicates an error.

Format

ERETURN (error *[,error label]*);

Arguments

error	is an integer variable whose value will be passed to the next higher level process (typically an AOS ERMES error code).
error label	is a statement label in the program to which control transfers if the ERETURN cannot be executed.

Notes

Upon execution of a call to ERETURN, unconditional return is made to the next higher level program. If the father process is not EXEC or CLI, the message may be in any format agreed on by father and son. If the next higher level program is the CLI, one of the following occurs:

- If the error code is an AOS error code, the user console will display WARNING, ERROR, or ABORT, and the appropriate message.
- If you specify one of your own error status codes (see ERROR), i.e., if the CLI does not recognize the code, it displays UNKNOWN ERROR CODE n where n is your error status code in octal, followed by your message.

This call is also documented in Chapter 4, "Handling Errors."

Example

```
DELETE ("FILE_20",IERR);

IERR: ERETURN ((READERROR));
```

Error Conditions

No error condition can occur.

References

?RETURN (System call)

See Appendix C for a complete list of system error codes.

SWAP

Transfers control to a program called in from a disk file.

Format

SWAP (filename *[,pointer [,error label]]*);

Arguments

filename	is a string expression that names the program file you want to execute.
pointer	is a pointer to an IPC header to be passed to the new process. (See the *AOS Programmer's Manual* for information on what to put in the IPC packet.)
error label	is a statement label to which control transfers if an error occurs.

Example

```
SWAP ("FILE.PR",TDAT,IERR);
```

Error Conditions

The following error codes may be returned:

ERPRV	Caller not privileged to make this call.
ERPRP	Illegal priority
ERBMX	Attempt to create a process with an illegal maximum size.
ERPTY	Illegal process type.
ERPDF	Error in process's User Status Table definition.
ERPRN	Attempt to create more than the maximum number of processes.

FILE SYSTEM codes.

Reference

?PROC (System call)

SYSRETURN

Terminates a program with no error.

Format

SYSRETURN *[(error label)];*

Arguments

error label	is a statement label to which control is transferred if an error occurs.

Example

```
SYSRETURN (IERR);
```

Error Conditions

The following error codes may be returned:

ERPRH Attempt to reference a process not in the process tree.

FILE SYSTEM codes.

References

?RETURN	(System call)
?LEFD	(System call)

TERM

Terminates a process, with an optional message.

Format

TERM (process, flag *[[,header] ,error label]*);

Arguments

process	is either a string expression that names the process you want to terminate, an integer expression that gives the process's PID, or -1 to indicate the calling process will be terminated.
flag	is an integer expression that indicates the contents of process. Set it to -1 for a process name; otherwise set it to 0.
header	is a string variable containing a message you want to send to the next higher level process in an IPC header. If *header* is unspecified, the system sends a standard message.
error label	is a statement label to which control transfers if an error occurs.

Example

```
TERM (-1, 0, "GOODBYE FROM" !!
        USERNAME (-1,0));
```

Notes

The *header* can't be an expression, only a variable.

A *header* may be passed only on self-termination.

TERM will not pass back a message properly if the message argument is a substring instead of a string.

This call is also documented under "Process Manipulation and Monitoring" earlier in this chapter.

Error Conditions

The following error codes may be returned:

ERPRH Attempt to reference a process not in the process tree.

FILE SYSTEM codes.

Reference

?TERM (System call)

End of Chapter

Chapter 8
DG/L Multitask Programming: An Overview

What is Multitasking?

Multitask programming is the most challenging use of your DG computer's resources. Multitasking allows numerous concurrent executions, or tasks, within the same source code. These tasks may follow different paths of execution, each one maintaining its separate program counter, and storing variables and accumulator contents in a separate section of memory.

Any procedure may run as a separate, concurrently executing task; moreover, you can activate any single procedure a number of times as separate tasks.

Multitask runtime routines allow you to arrange these "asynchronous" executions of tasks in a priority scheme, to specify when and under what conditions a task will run, and to supervise and even change the tasks' alternating control of the Central Processing Unit (CPU).

Additionally, DG/L multitasking has available all the memory-extending and program-extending features outlined earlier in this manual; for example, you can use overlay programs, disk file I/O, and cache memory management with multitasking.

This chapter gives you an overview of the DG/L runtime routines that control the multitask DG/L environment. The next chapter lists all the DG/L multitask runtime routines, grouped according to functionality with detailed descriptions of their arguments.

For a detailed presentation of multitasking concepts, refer to Chapter 7 of the *AOS Programmer's Manual,* "Creating and Managing a Multitask Program." The *AOS Programmer's Manual* describes the basic multitask concepts of initiating and queueing tasks, changing task states, and communicating with tasks.

How Multitasking Works

The separate tasks execute pseudo- concurrently, each task taking control of the CPU by turns. Tasks receive control of the CPU according to two criteria: their priority level and their state of readiness. The multitask environment defines tasks as ready or suspended.

For example, when a task is executing a system call for input or output, a scheduler (whose operation is invisible to the user) defines it as suspended and the task doesn't get to execute. Meanwhile, the I/O operation proceeds while other tasks run. Once its I/O is complete, the task returns to the queue of ready tasks.

The tasks may alternate in a "round robin" cycle, but you may also give some tasks a higher priority than others. If several tasks are ready to execute at once, the one with the highest priority receives control.

To design an efficient multitask program, you must coordinate task activity. When the execution of some tasks depends on other tasks' operations, your algorithm must guarantee a proper sequence of events to prevent conflicts and task deadlocks. You may need to queue tasks, assign priority levels, and define conditions that ensure the correct order of execution and use computer time efficiently.

When To Use Multitasking

Multitasking can be useful for doing two or more things at once. Unlike separate processes, tasks share the same address space, and thus can communicate without system calls and use less memory resources. (See the preceeding chapter for process-related DG/L runtime routines.) Here are several kinds of problems you might want to apply multitasking to:

Programs that do many system I/O calls. (See Figure 8-3 and Appendix E.) An I/O task can fill buffers, while another task processes buffer information, and a third task outputs the results. By using multitasking calls to overlap I/O and processing time, you save computing time.

Multi-terminal applications. A single procedure can handle terminal I/O and formatting. This procedure can run as a separate task for each terminal. Global resources and activities can then be handled by one or more other tasks. In this way, the system handles timing, and terminals get response only when they need it, without the overhead of polling techniques.

Top-down structured programming. Often the best way to separate functions is to create several concurrent tasks. This is especially true in real-time and interactive jobs.

Smaller machine configurations. Normally, multitasking takes less hardware and systems software to operate than multiprocessing. (Multitasking is available for RDOS programs as well.)

The Multitask Runtime Environment

Figure 8-1 shows your address space as it looks in a multitask program. (Remember that the figure is only a static image of a dynamic allocation process.) The compiler creates a single area of shared program code and a single Process Table, which contains information about the program as a whole. (See file DGLSYM.SR for the contents of the Process Table, and refer to Figure 2-1 in Chapter 2, which compares the single-task runtime environment to this multitask structure.)

As shown in Figure 8-1, within the initially unallocated part of your main memory area (the *global heap*), space exists for separate tasks. Each task is, in effect, a user of the program code, and its access to the code through the CPU alternates with that of other tasks. Tasks may follow separate paths through the same program, just as different data might be used in different executions of the same program.

Because the tasks execute concurrently, however, the AOS task scheduler must have separate state information about each task in a Task Control Block (TCB). You use LINK to create TCBs at Link time (see "Linking" in this chapter).

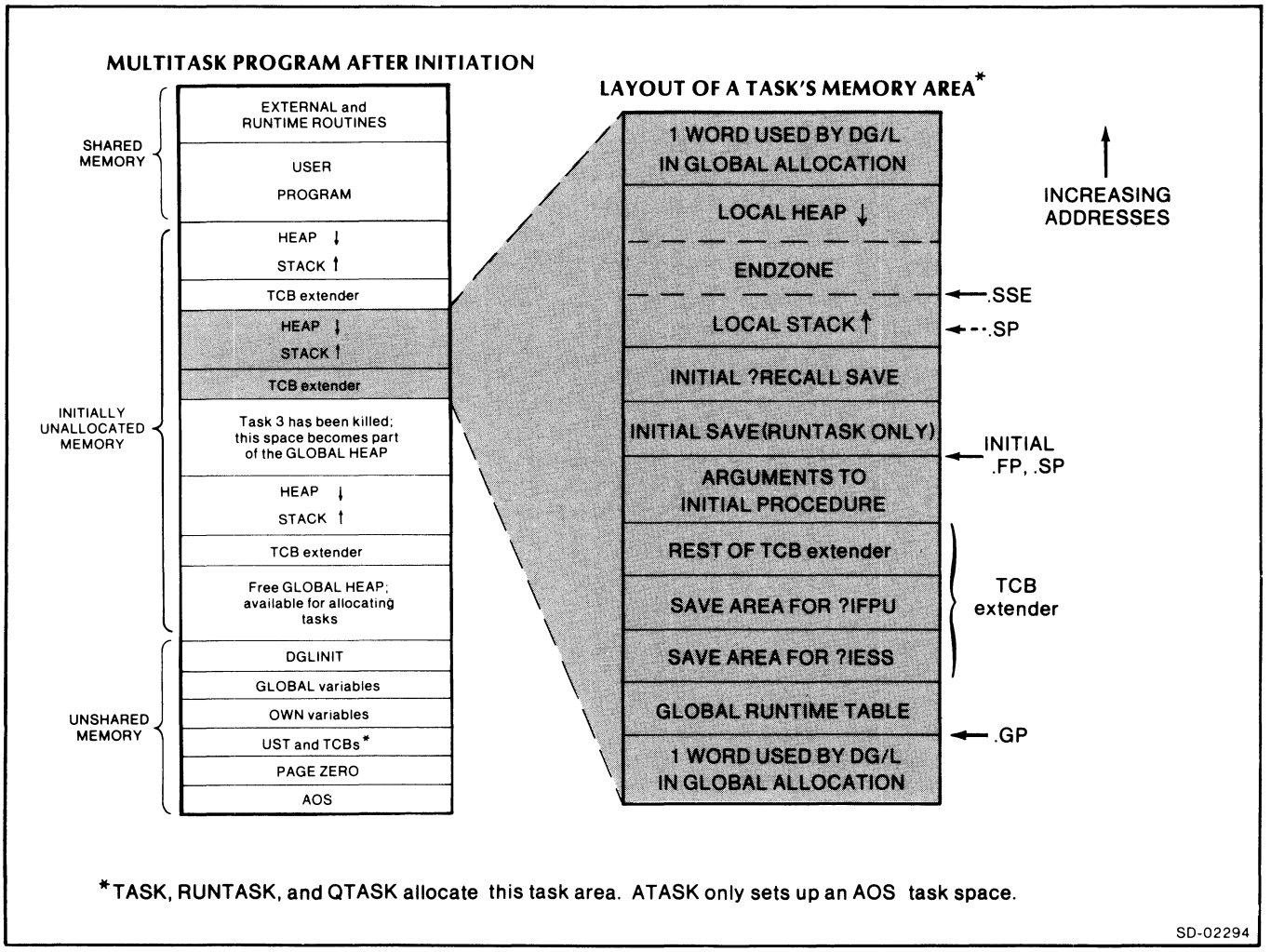

Figure 8-1. The Multitask Runtime Environment

TCB Extender

Because tasks alternate in execution, each needs a TCB extender (Figure 8-1) to preserve the contents of task-specific page zero locations, the floating point unit's contents and status, and other DG/L information. DG/L allocates a TCB extender when you initialize the task. When the task regains control of the CPU, the information in the TCB extender makes it possible to resume execution of a suspended task with the values the task had when it left off.

?IESS Area

Each task also contains an ?IESS area, pointing to ZREL locations, where task information is stored when a task is suspended. The ?IESS area's contents are

- .GP points to the Global Runtime Table that is unique to each task.
- .RP is a temporary area for storing the return pointer; you can also use .RP as a temporary storage area in assembly language modules.
- .RSP is the Return Stack Pointer, used with the interception of errors (see ERRTRAP and ERRINTERCEPT in Chapter 4).
- .ND1 .ND2 are two words of extra ?IESS storage used with NOVA® computers to emulate ECLIPSE® computer instructions.

NOTE: If your task isn't a DG/L task (see "AOS Task Calls" in this chapter), the ?IESS call isn't done. You can make the call yourself (see Chapter 7 in the *AOS Programmer's Manual*) and reserve 3 words for the area, or 5 words if you're writing code to run on RDOS/NOVA computers.

Creating a Task

The main program is the initial task to run, but you can create other tasks at runtime. To do so, you must first code any part of your program that will run as a separate task as an external procedure:

```
EXTERNAL PROCEDURE TASK_NAME;
```

or as a GLOBAL PROCEDURE within a CLUSTER.

Like any other DG/L procedure, the task's code begins with the declaration

```
PROCEDURE TASK_NAME [(argument)];
```

The procedure can of course call other procedures (including internal ones) to any nesting level.

You then create a task in your program by passing TASK_NAME to the DG/L TASK routine (or ATASK). You may also pass a one-word datum to the task as an argument. Declare any variables that individual tasks use *in common* as EXTERNAL variables in the subordinate tasks, and as GLOBAL variables in the main program. (See Chapter 2 of this manual, under "Internal Structure of Data," for details on EXTERNAL and GLOBAL variables).

Note: The call RUNTASK is different from TASK: it allows you to pass several arguments to a task. These arguments pass by reference, not by value, which means that if the initiated task modifies them, they remain modified. Also, the initiating task (which calls RUNTASK) must guarantee that the argument values are available for the task to use.

Compiling

Each task's code is in an external procedure (or CLUSTER); it must be separately compiled. (The same external procedure could be tasked up multiple times simultaneously). Here's an example of a compile command line for creating several tasks:

```
X DGL MAINPROGfile
X DGL procedure1
X DGL procedure2
```

Linking

When you link your program, include the name of each task in the command line, as in the following example:

```
LINK/NSLS/TASKS=3 MAINPROGfile&
procedure1 procedure2 [DGLIB]
```

In this example, three separate procedures run as tasks. The Link switch /TASKS=N directs the compiler to create a number of TCBs (one by default). When a program runs, there can be up to thirty-two tasks (in various combinations of activations and nestings) running at one time.

See Chapter 10 of this manual, "Operating Instructions," for a full discussion of compiling and linking a DG/L program.

Kinds of DG/L Multitasking Routines

The DG/L runtime library offers you the full range of AOS multitask calls. In addition, DG/L provides a call that allows you to suspend the multitask environment and let a task execute without interruption as a single-task program.

The basic calls you need to execute a multitask program initiate a task, allocate stack space, and run the task. As the task runs, you may want to coordinate its execution with that of others through intertask communication calls.

In addition to monitoring tasks, you may need to program the conditions of task execution into your code. DG/L routines for changing task states can ready, suspend, abort, or kill a task.

Initiating a Task

You have four options to initiate a task. With ATASK, you can be very flexible in defining a task and use an AOS Task Definition Packet. The calls TASK and RUNTASK start up a DG/L procedure as a task with a stack/heap and other requirements needed by DG/L programs. RUNTASK allows you to pass more than one argument to the task. Finally, by using the queuing call QTASK, you can specify a relative delay before the task begins to run, multiple periodic restarts, etc. (See the next chapter for complete descriptions of these four initiating calls.)

With all of these calls, you can specify the task's size, or else allocate a default size. (See section below, "Task Sizes.") With QTASK, you can specify when a task will begin to run, and the intervals of successive activations if it is to be created multiple times.

When you initiate a task, you give it a specific identification number, priority level, and stack size. The runtime environment allocates space for the task, instructs the system to associate a TCB with it, and enters the task into the list of running tasks.

Task Identifiers

Several DG/L task routines allow you to indicate the specific task you want affected by the call, e.g., TIDABORT, TIDKILL, TIDREADY. To use these calls, you must know the task's identification number, which you can get with TIDENTITY.

The task identification number is a 16-bit value ranging from 1 to 255. Also, multiple tasks of ID zero can be created. However, this sacrifices the ability to identify one particular task from code executing in another one, which might be important in some applications; e.g., where one task will kill other tasks.

Priority Levels

When you assign a task priority level, you direct the task scheduler to give the CPU to that task before any other ready tasks with a *lower* priority level. Tasks of lower priorities never receive control until all those at a higher level are waiting. Thus you may want to change a task's priority at certain stages in the program to ensure that it gets CPU time.

You may assign a priority level to a task when you initiate it, or else let it take the default value. The default priority level of the MAIN task (the first procedure in your program) is 255. Then each succeeding task takes the same default priority level of the procedure that initialized it. Zero is the *highest* priority, 255 the lowest. The priority level remains in effect until you change it with a call to PRIORITY. A task can change its own priority level or the priority level of another task. You obtain a priority level with the call IDPRIORITY. (IDPRIORITY can't be used in the 16-bit AOS/VS environment.)

Task Sizes

When you initiate a task, DG/L allocates it a block of your program's unshared memory space. This block is used for the task's stack/local heap, TCB extender, and global runtime area (see Figure 8-1). But you may find that the default size uses memory inefficiently, especially if your various tasks have quite different storage requirements.

The calls that initiate a task let you specify the block size. You can also define the block size for some tasks and allocate the default size for others in the same program.

The default size for the MAIN task is the size of available main memory divided by the number of tasks specified at link time. To change this value, change the contents of .MSTK in DGLPARAM.SR from zero to the desired block size. And then reassemble DGLPARAM.SR and include it in your LINK command line. Appendix D describes how you modify values in the file DGLPARAM.SR.

The default size for a task created by one of the DG/L calls TASK, RUNTASK, or QTASK, if not specified in the call itself, is 400_8 words. To modify this default value, change .DSTACK in DGLPARAM.SR to contain the desired value.

Each task's block is allocated from the global heap (see Figure 8-1). It's important to define task sizes carefully; once a task completes execution, its space becomes free for use by another task requesting the same or smaller block size. And other tasks that are still executing remain in the same memory locations. As a result, there may be small blocks of unused memory between tasks as old tasks finish and new tasks begin execution (see Figure 8-1).

When a heap allocation fails to find a large enough block of memory, the call merges *contiguous* free blocks and tries to find a block large enough. If this fails, an error condition occurs.

The calls that initiate tasks also give you the option of waiting for a block. Also, with QTASK, if you don't indicate the option of waiting for a default block, you may include an assembly language error-handling routine in your program. (You give the error-handling procedure's name as an argument to QTASK.)

Block Contents

To define the right size block, you should know certain facts about the words on the block which DG/L reserves for its own use (see Table 8-1). If you designate a block for your task, then the system always uses a Runtime Global Table, TCB extender, Endzone, Frame header, and two words for local non-stack ("heap") management. (See Chapter 2, under "The Runtime Environment" for definitions and locations of these stack areas.) The block allocated for any task, therefore, includes about 60 words not available for your actual use as stack/heap.

Table 8-1. Allocated Areas in a Task

Area	Size (decimal)
Runtime Global Table	7 words
TCB extender	24 words (7 words under AOS/VS)
Endzone	24 words
Input/Output Control block	106 words/IOCB*
Task Stack Space (per procedure nesting):	
Frame header	5 words
Arguments	Number of arguments received
Temporary Storage	Number of words given in SAVE n in code listing, plus space for arrays and string vaqriables

* used only during READ/WRITEs and FORMAT/OUTPUTs -- taken from local heap

Global Heap Management

Any of your *global* memory space not allocated for tasks remains as part of the global heap (see Figure 8-1). You manipulate blocks of global heap memory in a multitasking environment by using calls GALLOCATE, GFREE, and GMEMORY. But you still use single-task memory management calls, ALLOCATE, FREE, and MEMORY, to manipulate space from a task's local heap. (See Chapter 2 for more details on heap memory space.)

Task Control and Communications

As we mentioned under "Priority Levels," you can alter the "round robin" execution of tasks by specifying different priority levels when you initiate them.

Two kinds of runtime calls also control tasks and even let them communicate with each other, "Changing Task States" and "Intertask Communication."

Suspending, Readying, and Killing

Calls to SUSPEND, ASUSPEND, and TIDSUSPEND block a task's execution until another task executes a call to AREADY or TIDREADY. These calls let you to block a task's execution and allow another task to take precedence.

Calls to KILL, TIDKILL, and AKILL remove tasks and free their block spaces. These calls are useful when an error in one task affects other tasks. See "Changing Task States" in the next chapter for complete descriptions of these calls.

Waiting and Signalling

You can also define conditions that block a task temporarily at a given point in the code until another task sends a message. Refer to "Intertask Communications" in the next chapter for a discussion of these routines, and look at the example of intertask communication at the end of this chapter.

Termination

Because a multitask program executes asynchronously, it doesn't automatically terminate when the main program ends. The program (process) continues execution until the last task finishes. When a task terminates (including the main program), DG/L automatically releases its stack space.

DG/L and Non-DG/L Tasks

A "DG/L" task is created when a procedure passed to TASK (ATASK, etc.) is a DG/L procedure. However, it's also possible to run assembly language routines as tasks. The DG/L runtime environment handles both DG/L and non-DG/L tasks. However, non-DG/L tasks are subject to the following restriction:

> They can't use the floating point unit or the DG/L state variables .GP, .RP, and .RSP, or the NOVA .ND1 and .ND2.

You can code assembly language as a DG/L task by following one of several rules:

- Begin the assembly language task (procedure) with a SAVE instruction *or* give it a non-zero stack size. (On Nova computers, STA 3,.ND1 is recognized as a SAVE.) Either action tells the system to save and restore the floating point unit's state and any other state variables stored in the TCB extender.
- Or initiate a task with RUNTASK, which always creates a stack.
- Or use QTASK or TASK, and, in arguments passed with either, designate a non-zero stack size.

See Chapter 2, under "Writing Assembly Language Runtime Routines," for more information about writing assembly language routines to interface with DG/L programs.

Examples of Multitasking

The example in Figure 8-2 shows the skeletal structure of a multitask program and illustrates some of the combinations of initializing calls that you can use.

The main program (at the top) will be the initial task to execute.

```
BEGIN
 EXTERNAL PROCEDURE TASK, RUNTASK, QTASK, MULT1, MULT2, MULT3;
        .
        .
        .
      TASK (MULT1, argument);
        .
        .
      RUNTASK (MULT2, arguments);
        .
        .
      QTASK (MULT3, argument);
     END;

  /* IN SEPARATE FILES: */
 PROCEDURE MULT1 (arguments);
    BEGIN
      MULT3;
      TASK (MULT2,argument);
      TASK (MULT3,argument);
      END;

 PROCEDURE MULT2 (argument);
      .
      .
      .
    END;

 PROCEDURE MULT3 (argument);
    BEGIN
      .
      .
     END;
```

Figure 8-2. An Example of Multitask Program Structure

Notice that you must declare each procedure an EXTERNAL PROCEDURE that runs as a task, since each task is a separate routine, external to the others. MULT, MULT1, MULT2, and MULT3 all have separate entry points, and are in separate source files and compilations, unless they're put in a CLUSTER as GLOBAL procedures.

The procedure MULT1 shows that any task can initiate another procedure as a task, as it does in its calls to TASK for MULT2 and MULT3.

MULT1 calls MULT3 once as a subroutine, and later as a task. In the first call to MULT3 (as a subroutine), MULT3 variables will be stored in MULT1's stack space, with a stack frame for the subroutine.

When MULT1 later initiates MULT3 as a task, the multitask initializer allocates a separate stack space to MULT3, and the new task alternates execution with MULT1. When MULT3 runs as a task, it does not return values to the initiating task, MULT1.

A Multitask Example Using Intertask Communication

The example in Figure 8-3 shows how the task communication calls TRANSMIT and RECEIVE work. The program reads lines from separate terminals and writes them to a single disk file.

```
BEGIN
        GLOBAL INTEGER OUTCHAN, FILECLEAR;
        GLOBAL INTEGER (2) COUNT;
        INTEGER I, J;
        EXTERNAL PROCEDURE TRANSMIT, TASK, READIT;

        COUNT := 0;
        TRANSMIT (FILECLEAR, 1, OK);

        OK:     OPEN (OUTCHAN := 0, "OUTFILE");
                FOR I := 1 STEP 1 UNTIL 10 DO
                        TASK (READIT,I,I,5,400,NOSTACK);
                J := 11;
                GO TO CHECK;

        NOSTACK:J := I;

        CHECK:  READIT (J);
                DO BEGIN
                        DELAY (5,2); /* DELAY FOR 5 SECONDS */

                        FOR I := 1 STEP 1 UNTIL J DO
                                IF IDSTATUS (I) <> -1 THEN
                                        GO TO CONT;
                        GO TO NOMORE;

        CONT:   END;

        NO_MORE:        OPEN (J+I, @CON0);
                        WRITE (J+I, "<NL> NUMBER OF WRITES IS ",COUNT,"<NL>");
END;

        PROCEDURE READIT (ME);
                VALUE ME; INTEGER ME;
```

Figure 8-3. Multitask Example with Intertask Communication (continues)

```
BEGIN
        INTEGER DATUM; STRING S;
        EXTERNAL INTEGER FILECLEAR, OUTCHAN;
        EXTERNAL INTEGER (2) COUNT;
        EXTERNAL PROCEDURE RECEIVE, TRANSMIT;

        OPEN (ME, "@CON" !! ME);

        DO BEGIN
                READ (ME,S,EOF);
                DATUM := RECEIVE (FILECLEAR);
                WRITE (OUTCHAN,S);
                COUNT := COUNT + 1;
                TRANSMIT (FILECLEAR,DATUM);
                END;
EOF:    END;
```

Figure 8-3. Multitask Example with Intertask Communication (concluded)

RECEIVE and TRANSMIT

In this example, the two calls RECEIVE and TRANSMIT prevent two tasks from writing to the same file simultaneously.

The first block of the program declares a GLOBAL integer, FILECLEAR, as a message word. In the call to TRANSMIT, the message word, 1, allows a task to proceed past the RECEIVE call. When one task receives the message, the message word becomes 0 until the receiving task TRANSMITs another non-zero message word. Any task that attempts to RECEIVE the 0 message is suspended until the message word is set to a different value by another task's execution of TRANSMIT.

The call to RECEIVE is part of the procedure READIT. Any task can read the string S from a terminal, but only one task at a time can receive a 1 at FILECLEAR.

After the first task passes TRANSMIT, the next task with the highest priority can receive the message, clear the message word, and proceed through the WRITE portion of the procedure, which writes the string into the file "OUTFILE".

Creating the Tasks

The first call to TRANSMIT initializes the message word. Then using the *error label* argument, it transfers control to the label, OK, which starts the tasking program whether or not there was an error on TRANSMIT.

At OK, the program opens a file, referred to as OUTCHAN, to receive the string data that READIT returns.

The procedure uses a FOR UNTIL DO statement to start ten tasks that operate in a "round robin" cycle. Each task has an identifier set at an incremented value, I. Each task passes that value to the routine READIT, and each task takes a stack of 400_{80} words.

Operation of the Procedure

Each activation of the procedure READIT as a task has its own identifier and reads a string from a different terminal. The identifier I passes to the procedure, into the variable ME. For each task, the procedure uses the value in ME to tell the task which terminal to read, and gives the terminal a unique file number. The call to OPEN opens the terminal whose number corresponds to ME.

COUNT, which records the number of WRITEs, is declared as a GLOBAL variable, so that its value isn't part of any one task (or procedure).

When the task executes the READ, it is pended until the READ operation is complete. Then when it reaches RECEIVE, the task waits, regardless of its readiness, if another task is writing.

Operation of the Program

The call to TASK gives an error handling routine as an argument. If any of the tasks initiated at OK cannot get stack space, an error condition will result. The statement label NOSTACK, used as an error label, then receives control. NOSTACK sets J to a value one greater than the highest initiated task. If stack space is available for all ten tasks, J is set to 11.

The code at CHECK serves two purposes. First, it allows the main task to act as a task reading from a terminal.

When the main task receives an end-of-file condition from the terminal, the subroutine terminates, and the main task continues with its second purpose, described next.

CHECK determines whether any procedures are currently running or not. Using the call to IDSTATUS, CHECK sees whether any task is active. If IDSTATUS returns the code -1, no task with the specified ID is running.

If no tasks are running, then the program will terminate. Control passes from CHECK to the label NOMORE, and the program writes out to the terminal the number of writes all of the tasks performed. If tasks are running, execution continues at the top of the loop, delaying the main task five seconds.

Terminating

When all the tasks have reached the END at EOF , and the main task has reached the END below NO_MORE, the process will terminate.

NOTE: Appendix E of this manual contains an extended example of multitask programming.

End of Chapter

Chapter 9
Multitasking: The Routines

This chapter concludes the manual's listing of DG/L runtime routines. In it, we give all the routines (calls) you'll use to enable, structure, monitor, manipulate, and disable multitask programs.

The chapter divides into eight sections. Each section groups a set of runtime routines which perform related multitask functions. The eight sections are

- Initiating Tasks
- Obtaining Task-Related Information
- Changing Task States
- Intertask Communication
- Queuing Tasks for Deferred or Periodic Execution
- Using Memory Management
- Task/Operator Communications
- Disabling and Enabling the Multitask Environment

Initiating Tasks in a Multitask Environment

ATASK

Start a task, providing all AOS packet information explicitedly.

Format

ATASK (packet *[,error label]*);

Arguments

packet	is an identifier that names an eleven- or fifteen-word Task Definition Packet. See the *AOS Programmer's Manual* for lists of its contents.
error label	is a statement label to which control transfers if an error occurs.

Example

```
ATASK (TASK1);
```

Notes

This call lets you specify all options yourself, thus giving more functionality.

You must establish a task manager if you want to queue tasks. See TASKMANAGER under "Queuing Tasks."

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?TASK (System call)

RUNTASK

Executes a routine as a task, with arguments and error handling.

Format

RUNTASK (procedure *[,arguments]* ,identifier, priority, block size, key, error label);

Arguments

procedure — is the procedure (or entry point in assembly language) you want to execute as a task.

arguments — are any variables whose values pass to the procedure. They pass by name and cannot be expressions.

identifier — is an integer expression that gives the identification number you want to assign to the task.

priority — is an integer expression that gives the priority you want to assign to the task.

block size — is an integer expression that specifies the block size you want to allocate to the passed task when it executes. Zero or no argument indicates a default size block of 400_8 words. A negative value indicates waiting for a default-size block.

key — is an integer variable. Before RUNTASK starts the task, it sets key to zero. When the task terminates, a SIGNAL will be done by RUNTASK code to locate key. The value passed on the SIGNAL will be 1 for normal termination, or an < error code + 3 > if the task terminated abnormally.

As an input argument to RUNTASK, key can take one of three parenthesized values:

Code	Meaning
-1	Intercept all errors, with errors killing the task. (Does an ERRINTERCEPT, which new task code can overwrite.)
0	Before starting the new task, do a WAITFOR on location key. This suspends the task calling RUNTASK.
other value	Start the task. Errors will be handled by .RTER default handling until/unless the new task does its own ERRINTERCEPT or ERRTRAP calls. (See "Handling Errors," Chapter 4 for details on these error-handling calls.)

error label — is a statement label to which control transfers if an error occurs. Or, if a -1 is passed, it indicates no error label is present.

Example

```
RUNTASK (SUBR1,ARG1,ARG2,10,2,800,(1),ERR);
```

Notes

RUNTASK is the only task initiation call that allows you to pass arguments.

Note that the key value must be parenthesized, so that it will be treated as an expression, not a constant.

The procedure being passed to RUNTASK must not be parenthesized or have an argument list, in order that DG/L knows to pass the procedure entry to RUNTASK. Its argument list should be coded as one or more extra arguments to RUNTASK.

Error Conditions

The following error codes may be returned:

TASK codes.

AISTK Stack overflow.

Reference

?TASK (System call)

TASK

Initiates a task.

Format

TASK (procedure *[,datum [,identifier [,priority [,block size [,error label]]]]]*);

Arguments

procedure — is the procedure (or entry point in assembly language) you want to initiate as a task.

datum — is a one-word variable, constant, or expression, whose value you want to pass to the procedure as an argument. This is a call-by-value argument; i.e., if the task modifies the argument, it won't change the value of the datum in the initiating task.

identifier — is an integer expression that gives the task identification number.

priority — is an integer expression that gives the task's priority number.

block size — is an integer expression that gives the block size you want to allocate to the program when it executes as a task. A positive value gives the size of the block. Zero or no argument indicates the default size of 400_8 words. A negative value indicates waiting for a default-size block.

error label — is a statement label to which control transfers if an error occurs.

Example

```
TASK (SUBR9,DATUM,15,4,1000,IERR);
```

Notes

The procedure being passed to TASK must not have parentheses around it or following it (i.e., an argument list). This ensures that DG/L passes the procedure itself, rather than a returned value from it, to TASK.

Error Conditions

The following error codes may be returned:

TASK codes.

AISTK Stack overflow.

Reference

?TASK (System call)

Obtaining Task-Related Information in a Multitask Environment

GETPRIORITY

Obtains the calling task's priority.

Format

```
i := GETPRIORITY
```

Arguments

i	is an integer variable that receives the task's priority number.

Example

```
I := GETPRIORITY;
```

Error Conditions

No error condition can occur.

IDPRIORITY

Obtains a specified task's priority.

Format

```
i := IDPRIORITY (identifier [,error label] )
```

Arguments

i	is an integer variable that receives the task's priority number.
identifier	is an integer expression that specifies the task's identification number.
error label	is a statement label to which control transfers if an error occurs.

Examples

```
PTASK := IDPRIORITY (5,IERR);

MYPRI := IDPRIORITY((GETIDENTIFIER));
        /* SIMULATE GETPRIORITY */
```

Notes

This call is not possible under AOS/VS.

Error Conditions

The following error codes may be returned:

ERTID You specified an identifier of zero, or no such task ID was found.

IDSTATUS

Obtains a specified task's status.

Format

```
i := IDSTATUS (identifier)
```

Arguments

i is a one-word variable that receives the task's status as a group of single-bit flags:

Event Flag	Event
?TSPN ?TSIG	Task has issued a system call executing in system space.
?TSSG	Waiting for overlay or shared routine; or waiting at ?XMTW or ?REC.
?TSSP	Suspended by ?IDSUS, ?PRSUS or ?SUS.
?TSRC	Waiting for a ?TRCON message.
?TSIW	System action is occurring to prepare for task's execution.
?TSGS ?TSAB	Used by system.
?TSUF	Used by DG/L.

identifier is an integer expression that specifies the task's identification number.

Example

```
I := IDSTATUS (124);
```

Notes

If the task ID specified doesn't exist, i returns a value of -1 (all 1 bits).

Error Conditions

No error conditions can occur.

Reference

?IDSTATUS (System call)

TIDENTITY (GETIDENTIFIER)

Obtains the calling task's identification number.

Format

```
i := TIDENTITY
```

Arguments

i is an integer variable that receives the calling task's identification number.

Example

```
TIDSTATUS ((TIDENTITY),STAT,TCBAD);
```

Notes

GETINDENTIFIER is RDOS-compatible; under AOS you can use either name.

Error Conditions

No error condition can occur.

TIDSTATUS

Obtains the AOS status (and TCB address) of an identified task.

Format

TIDSTATUS (identifier, status, TCB *[,error label]*);

Arguments

identifier — is an ID number that specifies the task whose status and TCB pointer you want returned.

status — is a one-word variable that receives the task's status as a bit code:

Event Flag	Event
?TSPN	Task has issued a system
?TSIG	call executing in system space.
?TSSG	Waiting for overlay or shared routine; or waiting at ?XMTW or ?REC.
?TSSP	Suspended by ?IDSUS, ?PRSUS, or ?SUS.
?TSRC	Waiting for a ?TRCON message.
?TSIW	System action is occurring to prepare for task's execution.
?TSGS ?TSAB	Used by system.
?TSUF	Used by DG/L (not under AOS/VS)

TCB — is a pointer variable that receives the address of the task's Task Control Block (TCB).

error label — is a statement label to which control transfers if an error occurs.

Example

```
TIDSTATUS ((TIDENTITY), STAT, TCBAD);
```

Notes

This call isn't operable under AOS/VS; use IDSTATUS instead.

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?IDSTAT (System call)

Changing Task States in a Multitask Environment

AKILL

Kills all tasks of a given priority.

Format

AKILL (priority);

Arguments

priority	is an integer expression that gives a task priority number.

Example

```
AKILL (4);
```

Notes

Any task supplied with a kill-processing routine will have control pass to that routine.

Error Conditions

No error code is returned; if no task exists at the priority given, no action is taken.

Reference

?PRKILL (System call)

AREADY

Readies all tasks of a given priority.

Format

AREADY (priority);

Arguments

priority	is an integer expression that gives a task priority number.

Example

```
AREADY (PRI);
```

Notes

AREADY will ready tasks suspended by user calls (ASUSPEND, SUSPEND, TIDSUSPEND, ?SUSP, ?IDSUSP, ?PRSUSP), but not tasks suspended due to an outstanding system call. If a task is suspended in both ways at the same time, AREADY will clear the explicit suspension and the task can start when its system call completes.

Error Conditions

No error code is returned; if no task exists at the priority given, no action is taken.

Reference

?PRDY (System call)

ASUSPEND
Suspends all tasks of a given priority.

Format
ASUSPEND (priority);

Arguments

priority	is an integer expression that gives a task priority number.

Example
```
ASUSPEND (2);
```

Error Conditions
No error code is returned; if no task exists at the priority given, no action is taken.

Reference
?PRSUSP (Task call)

KILL
Kills the calling task.

Format
KILL;

Arguments
None.

Example
```
IERR: KILL;

END;
```

Notes
Control returns to the Task Scheduler, which allocates system resources to the highest priority task that is ready. See the *AOS Programmer's Manual* for a discussion of kill processing.

Error Conditions
No error condition can occur.

Reference
?KILL (System call)

PRIORITY

Changes the priority of the calling task.

Format

PRIORITY (priority);

Arguments

priority	is an integer expression that gives the new priority of the calling task.

Example

```
PRIORITY (5);
```

Notes

The calling task receives the lowest priority in its new priority class.

If you request a priority greater than 255, only the value of bits 8 through 15 is accepted.

Reference

?PRI (System call)

SUSPEND

Suspends the calling task.

Format

SUSPEND;

Arguments

None.

Error Conditions

No error condition can occur.

Reference

?SUSP (Task call)

TIDABORT

Kills a task specified by an identification number.

Format

TIDABORT (identifier, *[,error label]*);

Arguments

identifier	is an integer expression that gives the identification number of the task you want to kill.
error label	is a statement label to which control transfers if an error occurs.

Error Conditions

The following error codes may be returned:

TASK codes.

Example

```
TIDABORT (201,IERR);
```

Notes

The task where the call TIDABORT appears may kill any ready or suspended task. Outstanding system calls from the killed task are terminated. Other killing calls (TIDKILL, AKILL) don't cause the task to terminate until it returns from an outstanding system call.

TIDABORT does not release open channels used by the killed task, but does release any overlays.

Error Conditions

No error condition can occur.

Reference

?IDGOTO (System call)

TIDKILL

Kills a task specified by an identification number.

Format

TIDKILL (identifier *[,error label]*);

Arguments

identifier	is an integer expression that gives the identification number of the task you want to kill.
error label	is a statement label to which control transfers if an error occurs.

Example

```
TIDKILL (99,IERR);
```

Notes

TIDKILL is a faster method of killing a task than KILL, because TIDKILL first resets a task's priority level to zero (highest). It doesn't release channels, but does release overlays.

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?IDKILL (System call)

TIDPRIORITY

Changes the priority of a task specified by an identification number.

Format

TIDPRIORITY (identifier, priority *[,error label]*);

Arguments

identifier	is an integer expression that gives the identification number of the task that will get the new priority.
priority	is an integer expression that gives the new priority of the task.
error label	is a statement label to which control transfers if an error occurs.

Example

```
TIDPRIORITY (100,1,IERR);
```

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?IDPRI (System call)

TIDREADY

Readies a task specified by an identification number.

Format

TIDREADY (identifier *[,error label]*);

Arguments

identifier	is an integer expression that gives the identification number of the task to ready.
error label	is a statement label to which control transfers if an error occurs.

Example

```
TIDREADY (ITSK,IERR);
```

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?IRDY (Task call)

TIDSUSPEND

Suspends a task specified by an identification number.

Format

TIDSUSPEND (identifier *[,error label]*);

Arguments

identifier	is an integer expression that gives the identification number of the task to suspend.
error label	is a statement label to which control transfers if an error occurs.

Example

```
TIDSUSPEND (55,IERR);
```

Notes

See note under AREADY in this section.

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?IDSUS (System call)

Intertask Communication

You use the calls in this section primarily to synchronize task activity. Tasks can signal their state of execution by communicating with each other. If, for example, there is a critical path of code in your program that only one task at a time may execute, these communication routines can override the system's usual transfer of control.

TRANSMIT and RECEIVE

These routines allow tasks to pass one-word messages to each other. The message can signal a task to wait before executing a block of code.

DG/L defines 0 as the value to suspend. You can use any non-zero integer value as a message to continue execution.

```
TRANSMIT (SYNCH,1);
        .
        .
        .
TEST := RECEIVE (SYNCH);
        .
        .
        .
critical path
        .
        .
        .
TRANSMIT (SYNCH,TEST);
```

Here is a code fragment being executed by several tasks simultaneously. In this example, the program sets the value of SYNCH to 1. When a task enters the critical path, it receives the value 1 in its variable TEST.

The call to RECEIVE clears the message word to zero again. Any task reaching the call to RECEIVE will be suspended because SYNCH contains a 0.

When the first task reaches the end of the critical code, it will transmit the value 1 to SYNCH. Now the task of the highest priority will receive a 1 in SYNCH, and proceed to execute the critical code. Again, the message word becomes 0, and no other task may execute beyond RECEIVE.

You can also "broadcast" a message at TRANSMIT by sending a value of -1. If the message is broadcast, all tasks waiting at RECEIVE will receive the message and pass to the next statement.

The message word (SYNCH) should not be manipulated directly, but only via TRANSMIT and RECEIVE.

Appendix E, "A Multitask Program", contains a detailed example of TRANSMIT and RECEIVE in action.

WAITALL and WAITFOR; SIGNAL and CLEAR

WAITALL and WAITFOR, using the message routines SIGNAL and CLEAR, also suspend a task's execution, but in a more complex way. Instead of a simple zero/nonzero condition, a combination of bits can signal some tasks to continue execution while others are suspended. WAITALL and WAITFOR can also allow more than one task at a time to execute a body of code. Tasks that do a WAITALL call won't continue execution until *all* bits in the specified mask are set; those waiting at WAITFOR resume execution as soon as one bit is set.

A program might, for example, have several tasks receiving the same condition word. Tasks 3, 4, and 5 could be suspended at

```
WAITALL (WAITWORD, 37R8);
```

while tasks 6 and 7 are waiting at

```
WAITFOR (WAITWORD,37R8);
```

where 37R8 specifies a 7-bit mask in the condition word WAITWORD. Both calls then use the last five bits of the condition word as the signal. Those tasks waiting at WAITFOR may continue as soon as one bit is set to 1; those at WAITALL wait for all five. When another task, say task 2, executes the statement

```
SIGNAL (WAITWORD, 1);
```

elsewhere in the program, tasks 6 and 7 will be allowed to continue execution, while tasks 3, 4, and 5 remain suspended. If task 2 or another executing task later executes

```
SIGNAL (WAITWORD,36R8);
```

then all five bits will be set and tasks 3, 4, and 5 may continue.

At this point, any task reaching WAITALL with the mask 37R8 will continue execution without waiting. If you again wish to stop tasks at WAITALL or WAITFOR, you must provide a call to CLEAR to return the condition word to all zeros.

It is possible to clear only selected bits of the condition word through the optional condition mask. In this case, a call such as

```
CLEAR (WAITWORD,101R2);
```

would allow tasks past the call to WAITFOR, but not the call to WAITALL.

WAITALL and WAITFOR do not return information to the calling task indicating whether or not a message has been received. To check this, you must monitor the receiving task with TRANSMIT and RECEIVE calls (and a separate set of message words).

See Appendix E, "A Multitask Program," for an example of these routines in a program.

CLEAR

Clears the bits in a condition word.

Format

CLEAR (condition word *[,mask]*);

Arguments

condition word	is a one-word variable, which cannot be an expression or constant.
mask	is a 16-bit mask of bits you want cleared. If it's not present, -1 is assumed.

Example

```
CLEAR (SYNCH,100R2);
```

Notes

This routine is used with SIGNAL, WAITALL, and WAITFOR to synchronize tasks or pass single bits of information between tasks.

CLEAR will not cause a suspended task to be readied.

Error Conditions

No error condition can occur.

NWRECEIVE

Receives a one-word message from another task, and does not suspend the receiving task if message is a zero.

Format

message := NWRECEIVE (key name)

Arguments

message	is an integer variable that receives the message.
key name	is a variable that contains the message.

Example

```
NWMSG := NWRECEIVE (DO_IT)
```

Notes

Unlike RECEIVE, this call doesn't suspend the receiving task if the message is zero. You can use it to clear message before doing a TRANSMIT or a WTRANSMIT.

If a task with a higher priority does a WTRANSMIT to key name, that task gains control first.

Error Conditions

No error condition can occur.

Reference

?RECNW (System call)

RECEIVE

Receives a one-word message from another task.

Format

message := RECEIVE (key name)

Arguments

message	is an integer variable that receives the message.
key name	is a variable that contains the message.

Example

```
IMSG := RECEIVE (AREA9)
```

Notes

After the message is received, keyname is cleared. If the message is -1, all tasks at RECEIVE will receive the message. See TRANSMIT, WTRANSMIT.

Error Conditions

No error condition can occur.

Reference

?RECEIVE (DG/L Task call)

SIGNAL

Signals a set of conditions. If the conditions are already true, the call has no effect.

Format

SIGNAL (condition word *[,mask]*);

Arguments

condition word	is a one-word variable that cannot be an expression or constant. Usually it will be GLOBAL/EXTERNAL.
mask	is a 16-bit mask of bits you want set. If it's not present, -1 is assumed.

Example

```
SIGNAL (SYNCH,11R2);
```

Notes

You must use SIGNAL to activate a task waiting in WAITFOR or WAITALL. Calls are cumulative: each mask is OR'd with the previous contents of condition word to get the new value of condition word.

Error Conditions

No error condition can occur.

TRANSMIT

Transmits a one-word message to a task.

Format

TRANSMIT (key name, message *[,error label]);*

Arguments

key name	is a one-word variable indicating where the message is to be written. Usually it will be GLOBAL/EXTERNAL.
message	is the one-word, non-zero message you want to transmit to another task.
error label	is a statement label to which control transfers if an error occurs.

Example

```
TRANSMIT (ITSK10,IMSG,100);
```

Notes

A message word set to -1 will be broadcast to all tasks waiting at RECEIVE.

While TRANSMIT simply deposits a message, WTRANSMIT deposits a message and suspends the caller until a RECEIVE on the key name is executed. However, if any task is pended on a RECEIVE at this key name , WTRANSMIT won't suspend the caller.

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?XMT (System call)

WAITALL

Waits for all bits in a condition word signaled by SIGNAL to be ones.

Format

WAITALL (condition word *[,mask]);*

Arguments

condition word	is a one-word variable that cannot be an expression or constant.
mask	is a 16-bit mask you want tested. If not present, -1 is assumed.

Example

```
WAITALL (SYNCH,377R8);
```

Notes

This routine is used with CLEAR, SIGNAL, and WAITFOR to synchronize tasks or pass single bits of information between tasks.

Error Conditions

No error condition can occur.

WAITFOR

Waits for any bit in a condition word signaled by SIGNAL to be true.

Format

WAITFOR (condition word *[,mask]*);

Arguments

condition word	is a one-word variable, which cannot be an expression or constant.
mask	is a 16-bit mask of bits you want tested. If not present, -1 is assumed.

Example

```
WAITFOR (SYNCH,402R8);
```

Notes

This routine is used with CLEAR, SIGNAL, and WAITALL to synchronize tasks or pass single bits of information between tasks.

Error Conditions

No error condition can occur.

WTRANSMIT

Transmits a one-word message to a task and waits for it to be received.

Format

WTRANSMIT (key name, message *[,error label]*);

Arguments

key name	is a variable indicating where the message is to be written.
message	is the one-word, non-zero message you want to transmit to another task.
error label	is a statement label to which control transfers if an error occurs.

Example

```
WTRANSMIT (IADDR,IABC,1050);
```

Notes

A message word set to -1 will be broadcast to all tasks waiting at RECEIVE.

While TRANSMIT simply deposits a message, WTRANSMIT deposits a message and suspends the caller. However, if any task is pended on a RECEIVE at this key name, WTRANSMIT will not suspend the caller.

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?XMTW (System call)

Queuing Tasks for Deferred or Periodic Execution

DEQUEUE

Dequeues one or more tasks queued using ATASK.

Format

DEQUEUE (packet *[,error label]*);

Arguments

packet	is the name of the Task Definition Packet established for the task in ATASK.
error label	is a statement label to which control transfers if an error occurs.

Example

```
DEQUEUE(TASK1);
```

Notes

See ATASK for the AOS initiation call. Any tasks active when you call DEQUEUE will remain active. Only tasks queued but not yet started will be DEQUEUEd.

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?DQTSK (System call)

QKILL

Dequeues a task queued with QTASK.

Format

QKILL (identifier *[,error label]*);

Arguments

identifier	is an integer expression that gives the task's identification number.
error label	is a statement label to which control transfers if an error occurs.

Example

```
QKILL (INUM,IERR);
```

Notes

This call removes the queue table corresponding to the task ID from the active chain, but does not alter the information in the queue table. Only a call to QTASK will requeue the task. When there are multiple tasks queued with the same ID number, QKILL dequeues the most recently queued task.

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?DQTSK (System call)

QTASK

Queues a task for execution after a specified delay.

Format

QTASK (procedure, datum, identifier, priority,
stack size, start hour, start min *[,start sec*
[,rerun count ,rerun increment [,time
units [,error handler [,error label]]]]]);

Arguments

procedure is an external procedure you want to execute as a task.

datum is a variable whose one-word value passes to the procedure.

identifier is an integer expression that will be the task identifier.

priority is an integer expression that specifies the task's priority level, from 0 (highest) to 255.

stack size is an integer expression whose value indicates the size of the stack/heap task block you request:

Code	Meaning
> 0	Specified size
0	Default size (400_8 words)
-1	Wait for default size
< -1	Queued task waits for stack

The block includes the global area, TCB extender, and stack/heap area.

start hour, **start min**, and *start sec* are integer expressions that specify the delay before task creation. A value of -1 (unlimited) for start hour indicates immediate execution.

rerun count is an integer expression that specifies the number of times to create a task. A value of -1 indicates unlimited repetitions of a new task.

rerun increment is an integer expression that specifies the number of time units between executions of a task.

time units is an integer expression that specifies one of the following values:

0	Basic system Units
1	Milliseconds
2	Seconds
3	Minutes (less than one day)
4	Hours (less than one day)

error handler is the name of an assembly language error procedure you want to pass control to if an error occurs when no stack has been allocated. You may specify ERRKILL, KILL, or your own routine. If you specify no procedure, ERRFATAL will be used. Your error-handling routine must not presuppose the existence of a stack or begin with a SAVE; it will receive the datum in AC2.

error label is a statement label to which control transfers if an error occurs.

Example

```
QTASK (ITSK,10,1,IERR);
```

Error Conditions

The following error codes may be returned:

AINUM Illegal number conversion.

TASK codes.

Reference

?TASK (System call)

TASKMANAGER

Creates a task manager for queued tasks.

Format

TASKMANAGER (priority, identifier *[,error label]*);

Arguments

priority	is an integer expression that specifies the priority you want to assign to the task manager. Setting priority to zero gives the manager the calling program's priority.
identifier	is the integer task identification number you want to give to the manager.
error label	is a statement label to which control transfers if an error occurs.

Notes

You must create a TASKMANAGER task before queueing tasks.

Example

```
TASKMANAGER (0, 255);
```

Error Conditions

The following error codes may be returned:

TASK codes.

Reference

?QTSK (System call)

Using Memory Management in a Multitask Environment

NOTE: In a multitasking environment, ALLOCATE, FREE, and MEMORY (see Chapter 3) all refer to a task's *local* heap, rather than to the global heap space. The following routines manipulate global memory space in a multitask environment.

GALLOCATE

Allocates a number of words of memory from the global heap and returns a pointer in a multitask environment.

Format

GALLOCATE (pointer, size *[,error label]*);

Arguments

pointer	is a pointer variable that receives the first memory address of the block.
size	is an integer expression that specifies the number of words you want to allocate. A negative value indicates that the task will wait for memory if none is currently available.
error label	is a statement label to which control transfers if an error occurs.

Example

```
GALLOCATE (P1,100);
```

Notes

The call reserves the first block that is equal to or larger than size . See Appendix D for for instructions on how to change this "first fit" method of allocation to an "exact fit," by modifying a DGLPARAM file value.

Unlike ALLOCATE, GALLOCATE does not initialize words to zero.

The call actually reserves size + 2 words. Pointer - 1 and pointer + size both contain the negative length of the area; these words should not be modified by the user.

You may use ALLOCATE and FREE in a multitask environment to allocate blocks of space from a task's local heap.

See Figure 8-1 in Chapter 8 for an illustration of the multitask memory environment.

Error Conditions

The following error codes may be returned:

ERMEM	Insufficient memory.
AICOR	Invalid core error.

GFREE

Frees a block of memory allocated by GALLOCATE from the global heap in a multitask environment.

Format

GFREE (pointer *[,error label]*);

Arguments

pointer	is a pointer variable giving the first memory address of the block (the same value returned from GALLOCATE).
error label	is a statement label to which control transfers if an error occurs.

Example

```
GFREE (P1);
```

Error Conditions

The following error code may be returned:

AICOR Invalid core error.

GMEMORY

Obtains the size of the largest free global heap memory block.

Format

i := GMEMORY

Arguments

i	is an integer variable that receives the number of words in the largest block. It doesn't include the 2-word overhead returned by GALLOCATE; GMEMORY instead returns the words available to the user.

Example

```
GALLOCATE (DATA,(GMEMORY),IERR);
```

Error Conditions

No error condition can occur.

Task/Operator Communications in a Multitask Environment

TRCONSOLE

Reads a message from the terminal (AOS only).

Format

s := TRCONSOLE *[(error label)]*

Arguments

s	is a string variable that receives the message. It should be at least 132 characters long. The string does not include the terminator.
error label	is a statement label to which control transfers if an error occurs.

Example

```
SMESS := TRCONSOLE;
```

Notes

More than one task within a program can receive a terminal message; this call will fail if the calling task has no identification number.

Error Conditions

The system may output two diagnostic messages to indicate errors in messages intended for tasks:

TID NOT FOUND	No task with the specified ID was found to be waiting for a keyboard message.
INPUT ERROR	Non-numeric character in task ID.

The following error codes may be returned:

TASK codes.

Reference

?TRCON (System call)

TRDOPERATOR
Reads a message from the terminal.

Format

s := TRDOPERATOR *[(error label)]*

Arguments

s	is a string variable that receives the message. It should be at least 132 characters long. The terminator is *not* included in the string.
error label	is a statement label to which control transfers if an error occurs.

Example

```
IMESS := TRDOPERATOR (IERR);
```

Notes

This call is the same as TRCONSOLE under AOS, but is also RDOS-compatible.

More than one task within a program may issue an outstanding request for a task message.

Error Conditions

The system may output two diagnostic messages to indicate errors in messages intended for tasks:

TID NOT FOUND	No task with the specified ID was found to be waiting for a keyboard message.
INPUT ERROR	Non-numeric character in task ID.

The following error codes may be returned:

TASK codes.

Reference

?TRCON (Task call)

TWROPERATOR
Writes a message to the terminal.

Format

TWROPERATOR (message *[,flag [,error label]]*);

Arguments

message	is a string up to 128 characters long including a terminator.
flag	is 0 if you want the task's identification number printed with the message, and -1 or omitted if you want the ID number suppressed.
error label	is a statement label to which control transfers if an error occurs.

Error Conditions

The following error codes may be returned:

TASK codes.

Example

```
TWROPERATOR ("FLAG IS 10",1,IERR);
```

Notes

The message appears in this form:

task identifier, message, terminator

TWROPERATOR calls the internal DG/L routine .CONSOLE (documented in Chapter 6). You can disable other callers of .CONSOLE, to prevent more than one task from writing to the same terminal by calling argumentless procedures EXTERNAL PROCEDURE LOCKCONSOLE and UNLOCKCONSOLE in your task's code. You can also call these two procedures in assembly language. LOCKCONSOLE locks the terminal, and error messages won't appear on it. UNLOCKCONSOLE unlocks the terminal.

Reference

.CONSOLE (DG/L internal routine)

Disabling and Enabling the Multitask Environment

DRESCHEDULE

Disables scheduling in a multitask environment.

Format

DRESCHEDULE;

Arguments

None.

Error Conditions

No error condition can occur.

Notes

No other task can gain control after one task's code has made this call. This allows protection of critical resources from race conditions, etc.

The system automatically enables scheduling when a multitask program begins. You must use the call to ERESCHEDULE to re-enable scheduling after this call.

Reference

?DRSCH (System call)

ERESCHEDULE

Enables scheduling in a multitask environment.

Format

ERESCHEDULE;

Arguments

None.

Error Conditions

No error condition can occur.

Notes

The system automatically enables scheduling when a multitask program begins. Use the call to ERESCHEDULE to re-enable scheduling after the call to DRESCHEDULE.

Reference

?ERSCH (System call)

MULTITASK

Enables the multitask environment.

Format

MULTITASK;

Arguments

None.

Error Conditions

No error condition can occur.

Notes

When the multitask environment is enabled, the task relinquishes control of the CPU. It then competes for control with the other tasks in the system. Use this call after SINGLETASK to enable multitasking.

This call is RDOS-compatible.

SINGLETASK

Disables the multitask environment.

Format

SINGLETASK;

Arguments

None.

Error Conditions

No error condition can occur.

Notes

This call gives the calling task sole control of the CPU (this process's share of it). The calling task loses control if the CPU if it suspends itself for anything but a system call, or if it dies.

This call is RDOS-compatible.

End of Chapter

Chapter 10
Operating Instructions

In this final chapter, we provide you with the information necessary to compile, link, and run DG/L programs from the AOS Command Line Interpreter (CLI).

The sections of this chapter describe:

- the command line format for compiling a DG/L program, creating a .OB file.
- the command line format for linking an executable program file, creating a .PR file.
- the rules for naming, searching for, and storing program files.
- all the optional switches you can use when compiling, linking, and executing programs.
- overlays, a means of storing blocks of code on disk, to expand available memory space (you designate overlays at link time).

Compilation

From the Command Line Interpreter (CLI), you call the DG/L compiler with this command format:

XEQ DGL *[/glob-sw ...]* input-file
[binary-file/B]
[list-file/L] [error-file/E] [option-list/O]
[code-option/C]

Search Rules When Compiling a Source File

The compiler looks for your source program in the following order: if input-file ends in .DG, then only input-file is searched for. If input-file does not end in .DG, then the compiler searches for a file with the name input-file.DG.

The compiler looks for a file in the order specified by your search list, starting from the working directory. If it finds input-file.DG anywhere in your search list, then it uses that file. If input-file.DG isn't found, then the compiler searches for input-file, and it uses the first file with that name in your search list.

If these attempts fail, an error occurs. DG/L displays errors, or sends them to a file, depending on what switches you append to the compile command line (see following sections).

See the *DG/L™ Reference Manual* for compiler error messages.

Commonly-used Optional Switches

As shown above, all switches in the compiler command line are optional. You can put these switch arguments in any order in the command line, except for global switches, which must be appended to DGL.

The optional arguments following input-file in the command line format shown above are actually in one of two possible forms, this one called *local* switches. Each local switch is made up of a filename, a backslash, and the switch symbol. Under the two sections that follow, "Global Switches" and "Local Switches," you'll notice that all five local switches have equivalent *global* switch forms. Thus, for these five most-common optional compilation functions, you can use either global or local switches.

The compiler always creates a relocatable binary object file that has a default name input-file.OB. You can specify another name by using the name/B local switch, or the /O=name global switch.

Listing information, including compiler error messages, goes to the list file, which takes the default name input-file.LS. To get a list file with default name, use the global switch /L. You can name a list file yourself by using the local switch name/L, or the global switch /L=name.

Error messages always go to the list file, if you specify one. Specifying an error file with local switch name/E or global /ERR=name sends an additional copy of error information to that file. If you specify no *list file* or *error file,* error messages go to the terminal.

The option list (option-list/O) argument is a character string that controls conditional compilation. Again, it's a local switch, and its global switch equivalent is /OPT=option-list.

The code generation argument *(code-option/C),* or global switch /CODE=code-option, specifies the environment where the program will run. N/C means NOVA/RDOS; E/C means ECLIPSE/RDOS; A/C means AOS. Any other specification will result in an error. If you omit the /C switch, DG/L assumes you will run in the current environment.

Output Files

The compiler uses the following rules to decide where to place your program's two main output files, the object file (.OB) and the list file (.LS).

If you give only the filename for the source file, for example, X DGL/L BOB, then the output files go into the current directory, even if the source file used was not found in the current directory, but elsewhere in the search list.

When you don't specify a filename for the output files (if any), the locating of the output files (filename.LS and filename.OB) depends on the source file specification. For example, X DGL :TEST:BOB, and X DGL ↑↑TEST:BOB both put the output file(s) into the directory :TEST, because a complete pathname was given.

If you specify a file name (/L, /O, /B), then, if the filename includes a pathname, the output file goes into the directory specified by the pathname; e.g., /L=↑↑LISTINGS:TESTLIST.LS. If you give just the filename, the file goes into the current directory; e.g., /L=TESTLIST.LS.

Global Switches

Global switches are switches you use in a compile command line by appending them to DGL (see example above). The DG/L global switch options are

/A — Orders the compiler to go beyond the phase that catches syntax errors, even if it detects a syntax error. Normally any errors discovered in the syntax phase cause the compiler to skip the other phases. Using this switch may help catch more errors with fewer compilations.

CAUTION: Some syntax errors cause invalid states in compiler databases. Thus use of the /A switch can cause compiler aborts in phases after the syntax phase. If you get such an abort, try compiling without /A.

/B — Makes the /L output listing a brief listing (source text and storage map only).

/C — Executes only the syntax phase. Semantics aren't checked, and code isn't generated. This option takes less time than a full compilation, and is therefore useful in early debugging of a program.

/CODE=[N,E,A] — Specifies the code generation option. This is the same as the /C local switch.

/ERR=name — Directs the error listing to file name. This is the same as the /E local switch.

/F — Nonfloating point program: if your program contains no floating point operations, you can declare this to the compiler with the /F switch. Compilation in this mode will generate object code with no floating point instructions, making the code usable on a machine without floating point hardware. If floating point operations or real numbers are used in a program compiled under /F, the compiler will give you error messages. The DG/L initializer checks for the existence of the FPU (floating point unit) when this switch is omitted.

/G — Puts the names of internal procedures into the symbol table. CAUTION: procedure names aren't scoped here; i.e., you'll get multiple definition errors if you have two identical names (up to eight characters) in your program and use /G.

Using the /LOCAL link switch (see later section) on a file compiled with the global switch /G causes source line numbers to appear in the symbol table for those procedures you so designate. Line numbers appear in the symbol table as either *LINE123* for .MAIN, or *MYOWN123* for PROCEDURE MYOWN. These source line numbers match those in the generated code section of the listing file.

DG/L may optimize certain statements so that the same line number appears several times. The debugger will then only look up one instance of the symbol and not the others. These switches, therefore, can be helpful for debugging, but are not a language feature.

/H — (For NOVA code only) Indicates that the target machine has hardware integer multiply/divide instructions (otherwise, DG/L will use software multiply/divide).

/I — Omits listing of INCLUDE files' contents on the compilation listing.

/L=name — Produces a listing. (See introductory text on naming and placing list file.)

/L — Produces a listing to the default list file name.

/M — This switch declares to the compiler that it can use short LEF instructions in the generated code (LEF mode will be enabled). The compiler assumes /M when it generates code for AOS systems, but not for ECLIPSE RDOS. See /NOLEF switch for complementary function.

/N — This switch tells the compiler not to generate object code, but otherwise to proceed with all compilation phases.

/NOLEF — Tells the compiler that although code is generated for AOS, it is not safe to use short LEF instructions.

/O=name — Directs the object file (name.OB) to file name. This is the same as the /B local switch.

/OPT=alpha- numeric string — Does conditional compilation using the specified string. This is the same as the /O local switch.

/P — Directs the compiler to assume that no less than the correct number of arguments will always be passed to all external procedures. This switch eliminates the need for many runtime checks in the generated code. Use /P on the called procedures, not the calling procedures.

/Q — Allows the use of the character ? in DG/L names (procedures, variables, literals, etc.). This is helpful for using parameter files that match AOS symbols, system calls, etc.

/R — Forces all integer division within subexpressions to be done using floating point arithmetic. This increases accuracy by reducing rounding and truncation errors.

/S — Directs the compiler to generate code for full subscript checking. Normally, this should be used only while debugging, since an object program without subscript checking runs faster and requires less memory space.

/T — Generates code for string overflow checking. If a program compiled under /T attempts to store before the first character or after the last character of a string, a non-fatal error is generated.

/V — Adds information to the left-hand margin of a compiler listing, describing the status of each line. More than one symbol may occur on a line. Symbols are

- C — For conditionally compiled
- * — Part of a comment continued from previous line
- " — Part of a quoted string constant continued from previous line
- I — Line is in an INCLUDE file block line number showing nesting level of blocks at current line start

/W — Directs the compiler to produce warning messages. (This option may produce voluminous output!.) These messages fall into two basic classes. The first class warns you when the compiler makes a necessary modification to your program. For example, if you tried to declare a REAL (15) number, the compiler would process it as a REAL (4). The second class informs you of potentially dangerous constructs in your program, such as passing a constant parameter to a procedure by reference, risking destruction of the constant.

/Y Puts all constants into shared code instead of shared data segments of memory. This change allows constants to go into overlays instead of remaining root-bound (see "Overlays" section of this chapter).

CAUTION: If you use constants as arguments in procedure calls, using /Y in an overlay environment can cause an error if the constant is not in main memory while a sub-procedure attempts to access it. To avoid this, simply parenthesize passed constants which will be in overlays; e.g., MYSUB (I, RP->BI, ("STRING_CONSTANT"), (37)). This turns the constant into an expression, passed on the stack.

/X Generates a full cross-reference table, including constant references, in the output listing. Without this switch, only variables will be cross-referenced.

/Z Declares to the compiler that you have placed all EXTERNAL variables referenced in this compilation in page zero of memory. If this is the case, the compiler generates shorter object code; e.g., LDA instead of ELDA.

Local Switches

/B Specifies the file to receive binary object code.

/C Specifies the code generation option. Legal combinations are N/C, E/C, and A/C for NOVA RDOS, ECLIPSE RDOS, and AOS, respectively. If you omit this switch, DG/L will generate code for the current environment.

/E Specifies the file to receive error messages.

/L Specifies the file to receive listing output.

/O Identifies the option code string for conditional compilation. (See example under "Overlays.")

Examples of Compile Command Lines

```
X DGL TEST
```

Compiles file TEST.DG or TEST, generating the binary object file TEST.OB, if no errors occur. However, since no error or listing file is designated, error information appears only on the terminal output device.

```
X DGL/S/X APROG.DG XYZ.OB/B
PROG_LIST/L PROG_ERR/E
```

Compiles file APROG.DG, generating a binary object file named XYZ.OB and a listing to file PROG_LIST. Error messages go to PROG_ERR, as well as to the program listing. The compiler generates code that fully checks all subscripts (/S), and has full cross-referencing (/X).

```
X DGL/F/Z PHI I2/O LIST/L
```

Compiles file PHI.DG (or PHI, if PHI.DG doesn't exist), generating code without floating point operations (/F) and taking advantage of EXTERNAL integers' position on page zero (/Z). The string 'I2' is read as a list of options for conditional compilation. A listing, along with any error messages, will be sent to LIST.

```
X DGL TESTER N/C NTESTER.RB/B
X DGL TESTER E/C ETESTER.RB/B
X DGL TESTER A/C ATESTER.OB/B
```

or, using global switches,

```
X DGL/CODE=N/O=NTESTER.RB TESTER
X DGL/CODE=E/O=NTESTER.RB TESTER
X DGL/CODE=A/O=NTESTER.OB TESTER
```

Compiles a program TESTER.DG for NOVA RDOS, ECLIPSE RDOS, and AOS, respectively. And designates files to receive binary object code.

For other examples of compile (and link) command lines for a DG/L program, see the program in Appendix E.

The /S XEQ Switch

If you use the /S switch with a DG/L compile command line by appending it to the XEQ CLI command (XEQ/S DGL pro_name), DG/L will return to the CLI STRING feature a null string if compilation was successful, or an error message string. You can then use this string in a CLI macro to determine if compilation produced a valid .OB file. For example, a macro to compile a program and then LINK it if there are no compilation errors might look like this:

```
XEQ/S DGL_PROGRAM/L %1%
[!EQUAL [!STRING],]
WRITE COMPILATION SUCCESSFUL
X LINK/NSLS %1-% [DGLIB]
[!ELSE]
WRITE COMPILATION FAILED, GAVE ERROR
MESSAGE: [!STRING]
[!END]
```

For more information about XEQ switches and other CLI features, refer to the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS).*

Building An Executable Program File (Linking)

You use the Link utility to load the compiler's output (the .OB file), the DG/L runtime libraries, and any EXTERNAL procedures needed to support your main program.

The command line format looks like this when linking for AOS systems:

```
X LINK/NSLS main-prog external-procedures
[DGLIB]
```

and like this

```
X LINK/SYS=RDOS program [DGLIBE]
```

or this

```
X LINK/SYS=VS16 program [DGLIB16]
```

when linking to run programs under the RDOS or the AOS/VS systems.

The brackets around DGLIB, the DG/L library macro (see next section), are part of the actual command line, not optional.

DG/L Library Macros

DGLIB is a convenient macro that lists the five DG/L runtime libraries in their proper order: DGLATASK.LB DGLAMATH.LB DGLAOPSYS.LB DGLAENV.LB URT.LB DGLAINIT.LB (URT.LB is the AOS system library).

Actually, you use one of five macros when linking your program, depending on the environment where the program will run. The five DG/L library macros are

- DGLIBA for linking to run under AOS
- DGLIBE for linking to run on ECLIPSE under RDOS
- DGLIBN for linking to run on NOVA under RDOS
- DGLIB same as DGLIBA, since DGLIBA is the most common one on AOS
- DGLIBF for linking to run under AOS using FPF hardware instructions (use DGLIBF on M600, C350, and S250 computers under AOS)
- DGLIB16 for linking to run 16-bit DG/L programs under AOS/VS

Link Switches

/NSLS — is always in the DG/L Link command line. /NSLS ("no system library search") prevents the system from doing a system library search, since the system library (URT.LB) already appears in its correct order in the DGLIB macros. (See the [DGLIB] macro expanded in the above section.)

/LOCAL — is a local switch on the file name. Discussed under the global switch /G, it designates the file(s) compiled with /G for which you want source listing numbers in the symbol table.

/TASKS=N — allocates the correct number of TCBs (task control blocks) in a multitask program. (See Chapter 8, "Multitasking.")

/SYS=RDOS or /SYS=VS16 — produces a program file that can execute respectively under the RDOS or AOS/VS operating systems. The Link line looks like this

```
X LINK/NSLS/SYS=RDOS prog
[DGLIBN]
```

/CHANNELS=N — allows you to increase the maximum number of channels open at any one time beyond the default value of 16. However, using an N value that is *less* than 16 won't decrease the number of channels. To do this, you must modify DGLPARAM.SR (see Appendix D). The available channel numbers are 0 through N-1 (default 0 through 15).

For an example of a Link command line with switches, see the program example in Appendix E.

For a more detailed discussion of Link concepts, refer to the *AOS Link User's Manual* or the *AOS/VS Link User's Manual*. If you're using the Data General's Bind utility, instead of Link, please refer to the *DG/L™ Reference Manual* and BIND manuals.

Executing a Linked Program

The CLI command line for executing a DG/L program is

```
X PROGRAM_NAME
```

Getting the RDOS System Library in AOS Format for Generating RDOS code on AOS

As shown under "Link Switches," the Link command line for loading a program that will run under the Data General's RDOS operating system looks like this

```
X    DGL/NSLS/SYS=RDOS    aprogram
[DGLIBE] (or [DGLIBN])
```

where the DG/L library is [DGLIBE] or [DGLIBN] for Eclipse RDOS or Nova RDOS, respectively.

The two macros DGLIBE and DGLIBN include the RDOS system library SYS.LB (just as DGLIB and DGLIBA include the AOS library URT.LB). However, to work on AOS, the RDOS SYS.LB must be changed to .OB format. Currently, the way you can change the RDOS system libraries to .OB format is by following this sequence of steps:

1. Under RDOS, use LFE to list the titles of the library routines in the selected library.

   ```
   LFE T ASYS.LB
   ```

2. Still under RDOS, use LFE to extract each of the .RB library routines.

   ```
   LFE X ASYS.LB <titles in ASYS.LB>
   ```

3. Dump the .RB files to tape.

   ```
   INIT MT0 /*initializes the magnetic tape for dumping*/

   DUMP/V MT0:0 <titles in ASYS.LB>

   RELEASE MT0 /*releases and rewinds magnetic tape*/
   ```

4. Load them onto your AOS system.

   ```
   XEQ (or X) RDOS LOAD/V @MT0:0
   ```

5. Convert the .RB files to .OB format.

   ```
   XEQ CONVERT <titles in ASYS.LB>
   ```

6. Use the AOS LFE to remerge the routines into the system library in the original order

   ```
   LFE N ASYS.LB/O <titles in ASYS.LB
   in order given from line 1>
   ```

Overlays

You can handle large, modular programs efficiently by using overlays. Overlays let you store blocks of code on disk until your program executes them. Your program calls a procedure in an overlay the same way it calls any other procedure, by simply calling its name in the program. When your program calls a procedure in an overlay, the operating system automatically loads the overlay into main memory.

In this section, guidelines on how you use overlays follow a general discussion of overlays. Refer to the *AOS Link User's Manual* for a more general discussion of overlays.

The Link Command Line with Overlay Designators

You wait until Link time to decide which procedures to put in overlays. Designating overlays at Link time (a method of overlaying known as "load-on-call") makes creating overlays fast and simple, and eliminates the need to recompile programs if you decide to modify an initial overlay structure.

The following example and discussion of a Link command line that creates overlays should help you to decide the following:

- how many overlay areas and overlays to create
- which of your procedures to make overlays
- which procedures should go into the same overlays and overlay areas, and which should be in different overlays.

Here's an example of a Link command line you might use to create overlays. Figure 10-1 shows the same overlay structure which this command line creates.

```
X LINK/NSLS MAINPRO !*□A B C□!□D
E□*!□!*□F□!□G□*! [DGLIB]
```

NOTE: Boxes show required spaces between overlay symbols and the names of procedures.

MAINPRO is the main program, and the characters A to G represent individual procedures. The !* symbols delimit two overlay areas, and the ! symbols establish two separate overlays in each overlay area.

Overlayed procedures A, B, and C all go into main memory together, in one overlay, when the program calls any one of them (see Figure 10-1). Procedures D and E also make one overlay, an overlay associated with the same overlay area as A, B, and C. If the overlay containing A, B, and C is in main memory and your program (possibly A, B, or C) calls D or E, then the overlay file made of D and E, which is being stored on disk, simply overwrites the overlay containing A, B, and C.

The same process applies to procedures F and G, which are each in a separate overlay, both associated with another overlay area in main memory (see figure 10-1).

Any procedure in an overlay can call any another procedure, even in a different overlay in the same overlay area.

To summarize then, two *overlay designators,* !* and !, in the Link command line

- create *overlay areas* in main memory (one program can have up to 63 overlay areas assigned to it). An overlay area is as large as the largest overlay associated with it, rounded to block boundaries.
- associate overlays with each overlay area (up to 511 overlays per overlay area).
- include one or more modules (procedures or clusters) in an overlay.

How to Decide on an Overlay Structure

Since the size of an overlay area depends on the *largest* overlay which you assign to it, you should try to group procedures into overlays so that you don't waste memory space. Notice that A, B, and C together are about the same size as D and E combined (Figure 10-1).

Another consideration, when deciding on an overlay structure, is which procedures will be calling which other procedures, and how frequently. For example, you'd save on input/output overhead with the overlay

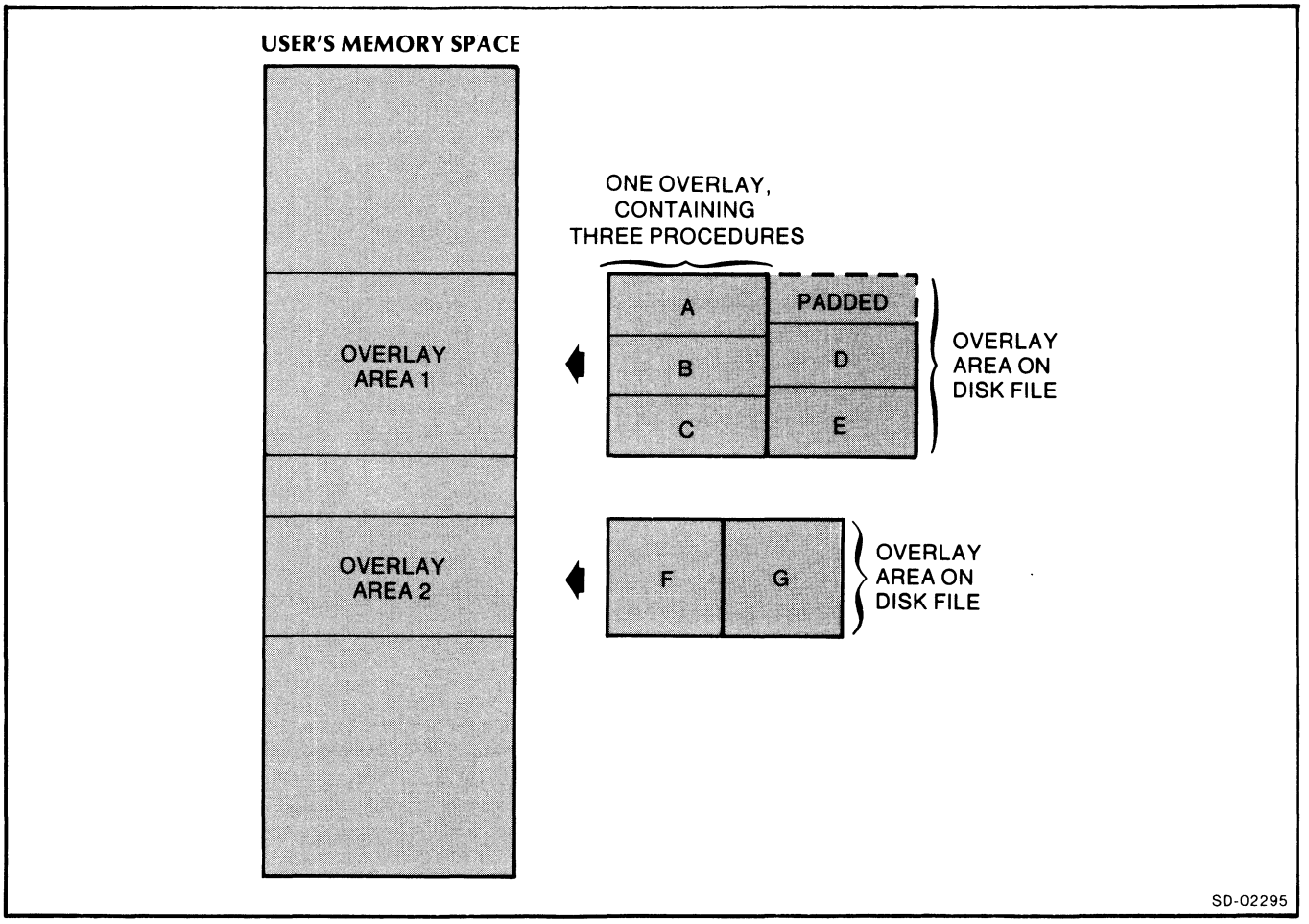

Figure 10-1. An Overlay Structure

structure in Figure 10-1 if A calls B and C frequently, and if F calls G. Or if one procedure in an overlay area calls a procedure in another overlay area (B calls G, for example), since then neither overlayed procedure will be swapped out of main memory.

Some Guidelines for Using Overlays

Don't put either your program's MAIN module or DG/L libraries into overlays.

Your program's constants normally go into a shared *data* area, instead of the shared code area. Shared data is linked into main memory, not into an overlay. If you have many overlays, their constants may use up too much of main memory. To avoid this extra overhead, use the /Y switch when you compile. /Y loads constants into an overlay with code. But if you do so, be sure not to make a subroutine call from the overlayed routine if the subroutine has an argument which is a constant. A *pointer* is passed to the constant, and if the overlayed routine isn't in main memory when the called procedure references it, the pointer won't find the value. However, parenthesizing constants will make such references possible by copying the constant onto the stack.

If you call DG/L routines from assembly language, be sure to call overlayed routines using ?RCALL, not EJSR. Also, be sure to use the hardware stack. You should have a stack frame of your own with the first two words reserved for use by ?RCALL.

At execution time, be sure the correct overlay file (.OL) is first in the search list when executing its companion program file (.PR). Normally, you would keep them in the same directory.

Multitask Programming with Overlays

Any procedures you designate as tasks, using calls RUNTASK, ATASK, TASK, or QTASK, must be in the root module rather than in an overlay. However, If you wish, you can create a small dummy DG/L procedure in the root that calls a routine in an overlay.

Multiple tasks can run in an overlay environment. But if your program suspends a given task that is executing in an overlay, then that overlay is no longer swappable, but is locked into memory while the task's program counter is in the overlay. This is an AOS feature, but it affects how you plan your multitask programs.

If your program kills a task whose program counter is in an overlay, then an internal DG/L routine automatically releases the overlay.

The Linking Overlay Mechanism

The DG/L compiler produces ?RCALL system call linkages for all external procedures, including overlays. Link then detects unnecessary ?RCALLs, such as root-calling-root, and converts them to EJSR calls. For more information on the load-on-call overlay mechanism, see the *AOS Programmer's Manual.*

Transporting Files From AOS to RDOS

You must use explicit calls in your program, such as OVLOD and OVREL, to load and release overlays under the RDOS system (see the *RDOS DG/L™ Runtime Library Manual* 93000124). Because these RDOS overlay routines are user-declared externals, the DG/L compiler doesn't recognize their names. Thus, you can compile and link RDOS programs with overlay declarations under AOS without receiving definition errors.

You can test, as well as compile and link, RDOS programs under AOS in one of the following ways:

- Conditionally code all RDOS overlay calls. For example, if you write

  ```
  /**R OVLOD (OV12); */
  ```

  you can use the switch /OPT=R when compiling for RDOS but not when compiling for AOS. This feature saves having to edit the source file.

- Leave the overlay calls in your object module for both AOS and RDOS. Then create your own DG/L external routines for OVLOD, OVREL, etc. Include these external routines in your Link command line before the DG/L library names. Your external routines can be dummy routines, such as `PROCEDURE OVLOD; END;` or debugging aids to print out a message; e.g., "loaded overlay 12 here".

You should carefully note functional differences between RDOS and AOS overlay environments which can affect programs, such as the location of overlayed code and data in main memory. Refer to RDOS and AOS system reference manuals for more information about the overlay environment.

End of Chapter

Appendix A
Alphabetic List of DG/L Runtime Routines

Routine	Function
ACCESS	Opens a file for cache memory management, associates it with a buffer pool, and defines its element size.
ACHAIN	Terminates the current program and executes a new program file.
ACREATE	Creates an AOS file.
ADDRESS	Obtains the address of any datum.
AKILL	Kills all tasks of a given priority.
ALLOCATE	Reserves a number of words of memory, obtains their location, and initializes the words to zero.
APPEND	Opens a file for appending writing to its end.
AREADY	Readies all tasks of a given priority.
ARGCOUNT	Obtains the number of arguments the current procedure received when it was called.
ASCII	(See BYTE.)
ASEND	Sends a message to a terminal or process.
ASSIGN	Assigns a device for the calling process to use exclusively.
ASUSPEND	Suspends all tasks of a given priority.
ATASK	Starts a task, providing all AOS packet information explicitly.
ATTRIBUTE	Obtains the file status parameters and device characteristics of an opened file.
BLKREAD	Reads a series of blocks from a file.
BLKWRITE	Writes a series of blocks to a file.
BLOCKPR	Blocks a subordinate process.
BREAKFILE	Creates a message for the DG/L error handler to write and a break file.
BREAKSWITCH	Is a declaration that creates a break file if a process terminates on an error condition.

Routine	Function
BUFFER	Creates a buffer pool area in memory and returns its address.
BUFLOCK	Locks into cache memory a 256-word block.
BUFUNLOCK	Unlocks a 256-word block in cache memory.
BYTE	Obtains the integer value of a byte in a datum, or sets the value of a byte in any datum.
BYTEREAD	Reads a specified number of bytes from a file into memory.
BYTEWRITE	Writes a specified number of bytes to a file.
CBIT	Clears a bit in a bit string.
CCONT	Creates a contiguous file of a specified length and initializes all words to zero.
CDIR	Creates a subdirectory.
CHAIN	Terminates the current process and loads another disk file for execution.
CHANNEL	Gets an AOS channel equivalent to a DG/L channel.
CHPRIORITY	Changes the AOS task priority of a process.
CHSTATUS	Obtains the current file status information for an opened file.
CHTYPE	Changes the type of a process.
CLASSIFY	Returns an integer value indicating the location of a given value within a table of ranges.
CLEAR	Clears the bits in a condition word.
CLIMESSAGE	Gets CLI command line information.
CLOSE	Closes a file.
COMARG	Reads command line operands and switches.
.CONSOLE	Writes to the terminal without using the stack.
CPART	Creates a fixed-size directory.
CRAND	Creates a random file with a fixed record size.
CREATE	Creates a random file with a dynamic record format.
DADID	Obtains the PID of the calling process's father process, or of another father process.
DATACLOSE	Closes a file opened for word buffering.
DATAOPEN	Opens a file at its beginning for word buffering.

Routine	Function
DATAREAD	Does a buffered read from a file into a specified area of memory.
DATAWRITE	Does a buffered write of a specified number of words from memory to a file.
DEASSIGN	Deassigns the device previously ASSIGNed to the calling process.
DEBUG	Enters the debugger from your program.
DELAY	Suspends a task for a specified number of clock pulses.
DELETE	Deletes a disk file.
DEQUEUE	Dequeues one or more tasks queued with ATASK.
DIM	Determines the width of an array dimension.
DIR	Changes the current working directory.
DRESCHEDULE	Disables scheduling in a multitask environment.
ENQUEUE	Enqueues a file entry into another process's queue.
EOPEN	Exclusively opens a file.
ERESCHEDULE	Enables scheduling in a multitask environment.
ERETURN	Terminates a program and indicates an error.
ERPRINT	Prints out the last error message if it is not yet printed.
ERRFATAL	Prints an error message without using the stack and returns to the system. If the error was stored, prints the full message.
ERRINTERCEPT	Establishes an error-handling procedure for intercepting (some) errors.
ERRKILL	Prints an error message and kills the calling task.
ERRMESSAGE	Returns a message line (associated with an error code) from the AOS message file, or from a message file in AOS format.
ERROR	Creates a message for the DG/L error handler to write.
ERRTRAP	Establishes a label as a location for trapping errors.
ERRUSER	Calls the error handler with an error code.
EXEC	Requests EXEC to perform a function for the calling process.
FETCH	Retrieves a word from CMM file on file zero.
FILEPOSITION	Computes the file pointer position.
FILESIZE	Computes the current size in bytes of a file.

Routine	Function
FLUSH	Writes out the contents of all modified buffers in the pool to disk.
FPUERROR	Returns boolean and string values for FPU errors.
FPUTRAP	Is a declaration for enabling the trapping of floating point errors.
FREE	Frees an allocated block of words and links it into the free chain.
GALLOCATE	Allocates a number of words of memory from the global heap, and returns a pointer in a multitask environment.
GBIAS	Obtains the system's bias factor.
GCHANNEL	Obtains the number of an available channel.
GCLOSE	Closes a file opened for AOS block input/output.
GETACL	Obtains an entry's Access Control List.
GETCINPUT	Obtains the input filename.
GETCOUTPUT	Obtains the output filename.
GETCPN	Gets a terminal portnumber.
GETDEV	Obtains the device characteristics.
GETDIR	Obtains the name of the current directory.
GETFREQUENCY	Obtains the real-time clock frequency.
GETGLOBAL	Translates a local portnumber into its 32-bit global equivalent.
GETIDENTIFIER	Obtains the calling task's identification number.
GETLIST	Obtains the name of the current list file.
GETPRIORITY	Obtains the calling task's priority.
GETSEARCH	Writes the current search list into a string variable.
GETSHARED	Lists the current size of the process's shared partition.
GFREE	Frees a block of memory allocated by GALLOCATE from the global heap in a multitask environment.
GKI	Performs polled keyboard input.
GLINK	Gets the contents of a link entry.
GMEMORY	Obtains the size of the largest free global memory block.
GOPEN	Opens a file for AOS block input/output.
GRDB	Reads blocks of data from tape, disk, or MCA links into memory.
GTIME	Obtains the current time.

Routine	Function
GWRB	Writes blocks of data from memory to tape, disk, or MCA links.
HASHREAD	Reads a specified block of the hash file into memory.
HASHWRITE	Marks the current buffer as modified.
HBOUND	Obtains the upper bound of an array dimension.
IDPRIORITY	Obtains a specified task's priority.
IDSTATUS	Obtains a specified task's status.
IHIST	Initiates histogram monitoring.
INDEX	Searches for a pattern of characters (or bits) in a string (or bit string) and obtains the position of the first character (or bit).
INIT	Initializes a logical disk unit or a device.
.ISUBSTR	See SUBSTR.
KGKI	Kills the task that GKI sets up.
KHIST	Terminates histogram monitoring.
KILL	Kills the calling task.
LBOUND	Obtains the lower bound of an array dimension.
LENGTH	Obtains the current length of a string variable.
LINEREAD	Does a data-sensitive read of a line from a file into memory.
LINEWRITE	Does a data-sensitive write of a line from memory to a file.
LINK	Creates a link entry in the current directory to a file in the same or another directory.
MEMORY	Obtains the number of remaining words of storage available to the user.
MINRES	Is a declaration to specify, modify, or otherwise access the default lower bound of the cache memory file.
MULTITASK	Enables the multitask environment.
NAMEGROUND	Returns an identifier with the PID affixed.
NCONT	Creates a contiguous file of a specified length without initializing all words to zero.
NODEREAD	Reads a specified node from file zero into memory.
NODESIZE	Obtains the size of a node on file zero.
NODEWRITE	Writes a node from memory to file zero.
NOFPUTRAP	Ignores floating point errors.

Routine	Function
NOMESSAGE	Gains extra memory space by removing error messages and the error reporter from storage.
NWRECEIVE	Receives a one-word message from another task, and does not suspend the receiving task if message is zero.
ODIS	Disables terminal interrupts.
OEBL	Enables terminal interrupts.
OKILL	Exits an overlay and kills the task.
OPEN	Opens a file for reading, writing, or appending.
PAGERELEASE	Releases a shared page that a disk file has been read into.
PATHNAME	Obtains a complete pathname for a file name.
PIDENTITY	Obtains the Process Identification Number (PID) of the calling process.
PORTOWNER	Obtains the PID of the process that owns a specified port.
PORTRECEIVE	Receives an Interprocess Communication.
PORTSEND	Sends an Interprocess Communication to a designated port.
POSITION	Changes the position of a file pointer.
PRIORITY	Changes the priority of a calling task.
PROC	Creates a process and returns a process identifier.
PROCNAME	Obtains the name of a process or its PID.
QCLOSE	Closes an open file or device using AOS parameters.
QKILL	Dequeues a task started with QTASK.
QOPEN	Opens a file using AOS parameters.
QREAD	Reads from a file using AOS parameters.
QTASK	Queues a task for execution after a specified delay.
QWRITE	Writes to a file using AOS parameters.
RANDOM	Obtains a random number from a pseudo-random sequence of integers in the range of -32768 to +32767.
RCOMARG	Rewinds the file of command arguments to make the next COMARG argument number zero.
READERROR	Checks and reads information about errors.
READSTRING	Does a data-sensitive read into a string variable.

Routine	Function
RECEIVE	Receives a one-word message from another task.
RELEASE	Releases a logical disk unit or device from current use.
REM	Divides two integers and obtains an integer result and remainder.
RENAME	Changes the name of a file.
ROPEN	Opens a file for read access only.
ROTATE	Rotates a word a specified number of bit positions.
RUNTASK	Executes a routine as a task, with arguments and error handling.
RUNTIME	Returns runtime information on a specified process.
SBIAS	Sets the system's bias factor.
SBIT	Sets a bit in a bit string to 1.
SCREENREAD	Does an AOS screenedit read.
SEED	Provides an initial number on which to base random number generation.
SETACL	Creates an entry's Access Control List.
SETCURRENT	Sets the length of a string or bit string variable.
SETDEV	Sets the device characteristics.
SETSEARCH	Sets the search list.
SHCLOSE	Closes a file opened for shared-page reading and flushes the read pages to disk.
SHIFT	Shifts a word a specified number of bit positions.
SHOPEN	Opens a file for shared-page input/output.
SHORTMESSAGE	Is a declaration for printing short error messages, instead of full error messages.
SHPARTITION	Establishes a new shared partition; i.e., changes the size of the shared area.
SHREAD	Reads sequential disk blocks from an open file into shared pages.
SIGNAL	Signals a set of conditions.
SINGLETASK	Disables the multitask environment.
SIZE	Determines the number of elements in an array or the declared length of a string or bit string.
STASH	Stores a word in a node on file zero.

Routine	Function
STATUS	Obtains the current directory status of a specified file.
STIME	Sets the time.
SUBSTR	Delimits a substring of a character string, bit string, substring, integer, or integer array.
SUPERUSER	Changes Superuser mode.
SUSPEND	Suspends the calling task.
SWAP	Transfers control to a program called in from a disk file.
SYSRETURN	Terminates a program with no error.
SYSTEM	Provides a generalized interface to the system, allowing you to use AOS system calls.
TASK	Initiates a task and allocates storage for it.
TASKMANAGER	Creates a task manager for queued tasks.
TBIT	Tests a bit in a bit string.
TERM	Terminates a process, with an optional message.
TIDABORT	Kills the task specified by an identification number.
TIDENTITY	AOS call to obtain task identifier.
TIDKILL	Kills the task specified by an identification number.
TIDPRIORITY	Changes the priority of a task specified by an identification number.
TIDREADY	Readies a task specified by an identification number.
TIDSTATUS	Obtains the AOS status of a task specified by an identifier.
TIDSUSPEND	Suspends the task specified by an identification number.
TRANSMIT	Transmits a one-word message to a task.
TRCONSOLE	Reads a message from the terminal (AOS) only.
TRDOPERATOR	Reads a task message from the terminal.
TWROPERATOR	Writes a message to the terminal.
UMUL	Multiplies two unsigned integers, adds another, and obtains the result and overflow.
UNBLOCKPR	Unblocks a previously blocked process.
UNIQUE	(Same as NAMEGROUND.)
UNLINK	Deletes a link entry.

Routine	Function
USERCLOCK	Defines a location to count clock pulses.
USERNAME	Returns the username of a specified process.
.USUBSTR	See SUBSTR.
WAITALL	Waits for all bits in a condition word signaled by SIGNAL to be true.
WAITFOR	Waits for any bit in a condition word signaled by SIGNAL to be true.
WRITESTRING	Writes a passed string onto a file.
WORDREAD	Reads words from a file using cache modules.
WORDWRITE	Writes words to a file using cache modules.
WTRANSMIT	Transmits a one-word message to a task and waits for it to be received.
XCT1	Executes a one-word instruction on the assembly level.
XCT2	Executes a two-word instruction on the assembly level.

End of Appendix

Appendix B
Internal Routines

The DG/L compiler produces internal routines for many of the constructs you use in DG/L programs. The three general categories of internal routines the compiler generates are conversion, formatting, and mathematical. Here we list all of the conversion and formatting routines, and show their calling sequences. Then we describe the format of the math routines, and how you can find them; we don't list all the internal math routines here. The operation of these internal routines is transparent.

Conversion Routines

CLRE Conversion Routines

Several of the DG/L internal conversion routines are CLRE (Common Language Runtime Environment) routines (see "Mathematical Routines" for information on the CLRE). These CLRE routines have names based on their data types:

	output	input
	1	1
CVT32?	2*	2*
	3	3
	4	4

where

1 means a single precision integer value

2 means a double precision integer value *

3 means a single precision real value

4 means a double precision real value

* double precision integers are not yet in CLRE form; see next section for those routines.

For example, the routine named CVT32?13 converts a single precision real value to a single precision integer value. The order of data types reads from *right to left,* like an assignment statement.

The four CLRE DG/L conversion routines are:

Name	Function	Input	Output
CVT32?13	Converts single precision real to single precision integer.	FPAC0	AC1
CVT32?31	Converts single precision integer to single precision real.	AC1	FPAC0
CVT32?14	Converts double precision real to single precision integer	FPAC0	AC1
CVT32?41	Converts single precision integer to double precison real	AC1	FPAC0

Non-CLRE Conversion Routines

The remaining DG/L internal conversion routines are not CLRE routines. The accumulators for input and output contain the *addresses* of the arguments, not the argument values themselves. AC0 should contain the input argument, and AC1 the output argument. Both arguments should be passed by reference, not value, and space must be allotted ahead of time.

Name	Function
.BLJ	Boolean to double precision integer.
.CONCAT	Concatenates two strings, yielding result string.
.DBJ	Double precision real to double precision integer.
FIX	Converts single precision real argument to integer by truncation.
.INJ	Integer to double precision integer.
.INTST	Integer to string.
.PTRST	Pointer to string.
.JST	Double precision integer to string.
.JBL	Double precision integer to boolean.
.JDB	Double precision integer to double precision real.
.JIN	Double precision integer to integer.
.JRL	Double precision integer to real
.RLJ	Real to double precision integer.
.RLST	Real to string.

Name	Function
.DBLST	Double precision real to string.
.STBOOL	String to boolean. (TRUE if string starts with T.)
.STINT	String to integer.
.STPTR	String to pointer.
.STJ	String to double precision integer (.STJ).
.STREAL	String to real.
.STDBL	String to double precision real.
.TMV	Move DG/L string to DG/L string.
.TMVN	Move string ending with a null to a DG/L string.

Formatting Routines

Name	Function	Calling Sequence
.FCHAR	Insert a character into the the format output buffer	AC0 right-adjusted character JSR @ .FCHAR
.FOPB	Format output of a boolean value	AC1 address of boolean value JSR @ FOPB
.FOPD	Format output of a double precision real value	AC1 address of value JSR @ .FOPD
.FOPI	Format output of an integer	AC1 address of integer JSR @ .FOPI
.FOPJ	Format output of a double precision integer	AC1 address of a value JSR @ .FOPJ
.FOPP	Format output of a pointer	AC1 address of pointer JSR @ .FOPP
.FOPR	Format output of a single precision real value	AC1 address of value JSR @ .FOPR
.FOPS	Format output of a character string	AC1 address of string specifier JSR @ .FOPS
.FOPU	Format output of a bit string	AC1 address of string specifier JSR @ .FOPU
.FWRB	Format writing of a boolean value	AC1 address of boolean JSR @ .FWRB
.FWRD	Format writing of a double precision real number	AC1 address of double precision real JSR @ .FWRD

Name	Function	Calling Sequence	
.FWRI	Format writing of a single precision integer or pointer	AC1 JSR	address of integer (pointer) @ .FWRI(.FWRP)
.FWRJ	Format writing of a double precision integer	AC1 JSR	address of double precision integer @ .FWRJ
.FWRR	Format writing of a single precision or double precision real number	AC1 JSR	address of single or double precision real @ .FWRR(.FWRD)
.FWRS	Format writing of a string datum	AC1 JSR	address of string specifier @ .FWRS
.FWRU	Format writing of a bit string datum	AC1 JSR	address of bit string specifier @ .FWRU
.IFLE	Initialize parameters for an unformatted read	AC0 AC1 JSR	address of error label specifier address of file number @ .IFLE
.IFMT	Initialize parameters for formatted output	AC1 AC0 IARG2 JSR	address of file number address of format string specifier address of error label specifier @ .IFMT
.IWRT	Initialize parameters for write output	AC1 AC0 JSR	address of file number address of error label @ .IWRT
.TFLE	Terminate the current unformatted read	JSR	@ .TFLE
.TFMT	Terminate the current formatted write	JSR	@ .TFMT
.TFOP	Terminate the current formatted output	JSR	@ .TFOP
.TWRT	Terminate the current write	JSR	@ .TWRT
.URDB	Do an unformatted read of a boolean	AC1 JSR	address of boolean datum @ .URDB
.URDD	Do an unformatted read of a double precision real number	AC1 JSR	address of double precision real @ .URDD
.URDI	Do an unformatted read of an integer	AC1 JSR	address of integer @ .URDI
.URDJ	Do an unformatted read of a double precision integer	AC1 JSR	address of value @ .URDJ
.URDP	Do an unformatted read of a pointer	AC1 JSR	address of pointer @ .URDP
.URDR	Do an unformatted read of a single precison real number	AC1 JSR	address of value @ .URDR

Name	Function	Calling Sequence
.URDS	Do an unformatted read of a string datum	AC1 address of string specifier JSR @.URDS
.URDU	Do an unformatted read of a bit string datum	AC1 address of specifier JSR @.URDU
.XCHAR	Insert a character into the format output buffer	AC0 right-adjusted character JSR @.FCHAR*

*NOTE: XCHAR and FCHAR are alternative routines. XCHAR differs from FCHAR in that it maintains the value LIST; it does not output when LIST is False; and the routine maintains an output character position pointer CRCNT. LIST and CRCNT should be declared as EXTERNAL BOOLEAN and EXTERNAL INTEGER, respectively.

Mathematical Routines

Many of the DG/L mathematical runtime routines are part of the Common Language Runtime Environment (CLRE). The CLRE is a runtime environment that is the same under several Data General utilities, including Fortran V, PL/I, INFOS, and the Array Processor.

These CLRE DG/L math routines have a naming format and calling sequence that is different from non-CLRE DG/L math routines.

(DG/L also offers built-in arithmetic functions callable from DG/L, documented in the *DG/L™ Reference Manual.*)

CLRE Math Routines

The CLRE math routines have a standard naming convention:

```
                    1
<root name of routine>?32  2*
                    3
                    4
```

where

1 means a single precision integer value

2 means a double precision integer value *

3 means a single precision real value

4 means a double precision real value

* DG/L CLRE routines have not yet been implemented for double precision integers

For example, the routine ABS32?4 computes the absolute value of a double precision real argument.

Calling Sequence for CLRE Math Routines

Before calling a math routine, you put arguments into accumulators and receive a result in the same accumulator generally. For each CLRE math routine there is one primary input area, either AC0 for integer values or FPAC0 (the floating point accumulator) for floating point values. If a second input argument exists, AC2 must contain the *address* of the second argument. CLRE functions with more than two arguments; e.g., MIN and MAX, use the assembly language EXTERNAL calling sequence. Many CLRE routines are callable from DG/L or assembly language.

Non-CLRE Math Routines

The calling sequence for non-CLRE DG/L math routines is different from CLRE math routines. Here is an example of such a non-CLRE calling sequence for a routine i := ROUTINE(j,k).

```
LDA 2,SP
LEF 0,j
PSH 0,0
LEF 0,k
LEF 1,i
JSR @ROUTINE
STA 2,SP
```

For a routine with one input argument, such as i := ROUTINE(j), the calling sequence might look like this:

```
LEF 0,j
LEF 1,i
JSR @ROUTINE
```

To find what internal math routines the DG/L compiler generates, look at the .LS file of a program (see Chapter 10, "Operating Instructions"). This file contains the assembly language code and runtime calls that the compiler generates to compile functions such as ARCCOS, LOG10, TAN, SQRT, SIGN, COS, ABS, etc. Within the listing of this assembly code will be the names of the internal math routines.

End of Appendix

Appendix C
DG/L Runtime Error Codes and AOS Exceptional Condition Codes

AOS Exceptional Condition Codes

Under a routine's description in this manual, you sometimes find a general category of errors, listed under "Error Conditions", e.g., TASK codes, FILE SYSTEM codes, CHANNEL-RELATED codes, etc. Appendix A of the *AOS Programmer's Manual* lists the mnemonics and meanings of all AOS exceptional condition codes, separated into these same categories.

When the system returns to your program the parametric (numeric) value of an AOS error condition, you can associate this value with a mnemonic name by referencing Appendix E of the *AOS Programmer's Manual,* which reproduces the AOS file PARU.SR, a listing of all AOS parameters.

The following is a list of the AOS exceptional condition code categories in the order they're most likely to occur:

Channel-Related
File System
Initialization and Release
IPC
Memory
Miscellaneous
Process
System Call
Task
User Device
Connection Management
Synchronous Line

The system also provides an error message file named ERMES that contains a textual description of each error code; you can read the description associated with an error code by issuing the DG/L call READERROR or the AOS CLI command MES.

DG/L Runtime Errors

The following table lists all DG/L runtime error codes, values, and messages. You can suppress the messages with the declarations EXTERNAL INTEGER SHORTMESSAGE or EXTERNAL INTEGER NOMESSAGE. The numeric codes are returned by READERROR and FPUERROR. Chapter 4 in this manual discusses error-handling routines in detail. For DG/L compiler error messages, refer to the *DG/L Reference Manual.*

NOTE: The latest version of DG/L runtime error codes always appears in the file DGLSYM, supplied on your product tape.

DG/L Runtime Errors

Mnemonic	Numeric Code	Message
AICOR	1+ECTRT+ECSAF	INVALID POINTER OR CORE
AISUB	2+ECTRT	SUBSCRIPT OUT OF BOUNDS
AINUM	3+ECTRT	ILLEGAL NUMBER CONVERSION
AIFMT	4+ECTRT	MISSING FORMAT OUTPUT FIELD
AISET	5+ECTRT	PROBABLE LINEREAD OVERFLOW
AEOBA	6+ECTRT	REQUEST FOR ODD-BYTE STRING ADDRESS
AINOD	7+ECTRT	INVALID NODE SIZE
AIBFS	10+ECTRT	INSUFFICIENT BUFFER POOL ALLOCATION

Mnemonic	Numeric Code	Message
AISIZ	11+ECTRT	ILLEGAL SIZE
AISTK	12+ECTRT	STACK OVERFLOW
AILCK	13+ECTRT	ALL CM BUFFERS ARE LOCKED
AIQTS	14+ECTRT	NO ROOM FOR QTASK STACK
AIARG	15+ECTRT	NOT ENOUGH ARGUMENTS TO RUNTIME
AIPTR	16+ECTRT	ILLEGAL POINTER TO ROUTINE
AESQT	20+ECTRT	ILLEGAL ARGUMENT TO SQRT
AEEXP	21+ECTRT	ILLEGAL ARGUMENT TO EXP
AELOG	22+ECTRT	ILLEGAL ARGUMENT TO LOG
AEPWR	23+ECTRT	ILLEGAL EXPONENTIATION
AEASC	24+ECTRT	ILLEGAL ARGUMENT TO ASIN/ACOS
AEATN	25+ECTRT	ILLEGAL ARGUMENT TO ATAN2
AEINT	26+ECTRT	ILLEGAL ARGUMENT TO INT
AEFPU	50+ECTRT	FPU Error
AEMAN	AEFPU+(1*ECSCB)	MANTISSA OVERFLOW
AEUND	AEFPU+(2*ECSCB)	FPU UNDERFLOW
AEOVF	AEFPU+(4*ECSCB)	FPU OVERFLOW
AEZDV	AEFPU+(10*ECSCB)	ZERO DIVIDE
AIBND	60+ECTRT	ILLEGAL BOUND SPECIFICATION

Notes

ECTRT = Runtime error (30000R8)
ECSAF = Absolutely fatal error (140000R8P1)
ECSCB = Error subcode (400R8)

End of Appendix

Appendix D
DG/L Files, Tables, and User Options

DG/L provides three files (with your product tape) that you can use to create external routines in assembly language:

DGLSYM.SR includes definitions of contents of the DG/L runtime tables (Process Table, Global Runtime Table), I/O control block structure, symbolic DG/L error codes, fixed stack areas, and the structure of cache memory buffers.

DGLMACS.SR contains macro definitions that eliminate repetitive typing of code; e.g., TITLE. These macros let you use different hardware and operating systems with source file compatibility.

DGLPARAM.SR supplies default assembly definitions of locations and symbols that you can modify to make changes in the actual runtime environment of your programs. (See the next section, "User Options.")

Using these symbols and macros, you can create assembly language files that are readable and easy to maintain. For hard copies of these files in their latest revisions, print them from tape.

User Options

You may want to change certain default values in a program's symbol table to meet your needs. Modifying DGLPARAM.SR is the most common way to do this. In this section, we first list programming options, and indicate where in DGLPARAM.SR you'll find the parameter to change. In some cases, we mention other methods of modifying a value; e.g., using a command line switch at link time, or writing an EXTERNAL declaration in a main DG/L program.

Following this list of options and how to effect them, we explain the four alternative *ways* to modify values in DGLPARAM.SR. In the actual file on the product tape, DGLPARAM.SR, you'll find further instructions about how to change DGLPARAM.SR. Here are programming options you may want to take advantage of:

- Modify .DSTACK in DGLPARAM.SR to specify a new default stack size in multitask programming.
- Change .NMAX in DGLPARAM.SR to reduce the size of the user stack that the initializer takes. You might decrease the value of .NMAX to create more shared pages (see "Shared I/O" in Chapter 6).
- Alter the Cache Memory Management (CMM) values for the minimum and maximum number of virtual buffers in DGLPARAM.SR, .VMIN and .VMAX (see "Cache Memory Management" in Chapter 6).
- Change the maximum number of channels (from the default value of 16). To increase the maximum number of channels, the easiest method is to use the link switch /CHANNELS=N (see Chapter 10, "Operating Instructions"). However, to decrease the number of channels (to less than 16) you must modify .CNTFL in DGLPARAM.SR.
- Change the GALLOCATE memory allocation from "first fit" (see call's description in Chapter 9) to "exact fit," by changing .XFIT in DGLPARAM.SR to a nonzero value.
- Alter the default value of temporary string and bit string storage areas (from 32 (40R8) characters or bits). You can change this value by either modifying STSZ. in DGLPARAM.SR or by writing into your program

```
EXTERNAL INTEGER BITSIZE,
STRINGSIZE;
BITSIZE := <length you want for bit
temporaries>;
STRINGSIZE := <length you want
for string temporaries>;
```

- Control the output of FORMAT by using an EXTERNAL BOOLEAN LIST declaration. By default, LIST is true and I/O is done. If you set LIST to false, all computations in an expression are done but I/O isn't. You may still use your own procedure or variable named LIST, but having one prevents the use of this feature.

There are four ways to change DGLPARAM.SR values, thereby changing your symbol table's values:

- Modify and reassemble DGLPARAM.SR. See DGLPARAM.SR for details.

- Create an assembly language module. The following example modifies .NMAX:

```
        .ENT    .NMAX
.NMAX:  n
        .END
```

  where n is the new value (in this case the maximum number of pages to be used for unshared stack space).

  Assemble the module as you would an external routine, using MASM and LINK, creating a file with a .SV or .PR extension.

- Use DEDIT, as in the following example

  .NMAX: <current value new value> BYE

  The system will display the current value; you key in the new one. This will change the .PR file.

- Use the debugger to change a program in main memory

  .NMAX: <current value + new value>

  The system will display the current value; you key in the new one. (Refer to the *AOS Debugger and Disk File Editor* manual for more information on using DEDIT and the debugger.

The Process Table

The Process Table, outlined in DGLSYM.SR, contains information globally pertinent to a DG/L program or process. The page zero pointer .PT points to word zero of the table. See DGLSYM.SR for the Process Table's contents.

The Global Runtime Table

The Global Runtime Table, also described in DGLSYM.SR, contains information specific to each task in a DG/L program. In the single-task environment, it occupies the locations immediately below the Process Table. In multitasking, you allocate a global runtime table dynamically for each task as the tasks become active (see illustration in Chapter 8, "Multitasking: An Overview"). The page zero pointer .GP points to word zero of the table, and is updated on task switches by AOS (or AOS/VS or RDOS). See DGLSYM.SR for the Global Runtime Table's contents.

End of Appendix

Appendix E
A Multitask Program

Figure E-1 shows a complete DG/L multitask program, including all eight files used in the operation of the program. For details about the runtime routines in this program, particularly the routines which communicate among tasks (RECEIVE, TRANSMIT, WAITFOR, SIGNAL, and CLEAR) please refer to Chapter 8 and "Intertask Communication" in Chapter 9.

The names and functions of the eight files which comprise this DG/L applications program are

CREFILE.DG	A single-task program which builds the database for the multitask program to process
EXT.DG	INCLUDE file supplying EXTERNAL definitions
LIT.DG	INCLUDE file defining various LITERALs
NAME.DG	INCLUDE file defining filename
DATABASE.DG	INCLUDE file describing relevant offsets in single-task program CREFILE.DG
DEMO.DG	The main program
TASKA.DG	Task A
TASKB.DG	Task B
TASKC.DG	Task C

The order in which these eight files appear in Figure E-1 follows the logic of their creation and use.

The single-task program, CREFILE.DG, builds the database that the multitask demonstration program, DEMO.DG, will process.

The four INCLUDE files (one defining EXTERNAL variables and three defining LITERAL declarations) are used by the single-task program, the main multitask program, and the three tasks to create easy-to-read, easy-to-modify symbolic names.

And finally we show the main program and its three separately-compiled tasks.

This multitask program shows you how to use various multitasking calls, how to create tasks, how to use variables where required, and how to handle errors. It should also provide you with a review of many DG/L language elements and structures; e.g., LITERALs, INCLUDE files, DO-WHILE loops, and boolean declarations.

```
/************************beginning of CREFILE.DG************************/

BEGIN

COMMENT

    This file builds the database that the multitask demonstration
program needs to use.   It is not multitasked.  The console
user is asked for the various fields.
                ;

INCLUDE NAME;
INCLUDE DATABASE;
```

Figure E-1. Complete DG/L Multitask Program (continues)

```
EXTERNAL STRING PROCEDURE GETCOUTPUT, GETCINPUT;
EXTERNAL        PROCEDURE BLKWRITE,   NCONT;

LITERAL
        CON_INP ( 0 ),          /* File for console input */
        CON_OUT ( 1 ),          /* File for console output */
        DATAFIL ( 2 );          /* Datafile */

POINTER      P;                 /* Pointer to current block */
INTEGER ( 1) SEQ;               /* Block sequence number */
INTEGER ( 1) MAX_NUM;           /* Maximum number of blocks */
INTEGER ( 1) ONE;
INTEGER ( 1) I;
REAL    ( 2) PAY;               /* Temp for reading in pay */
STRING  (20) NAME;              /* Temp for reading in name */


DELETE( FILE_NAME );            /* Delete previous data file */
OPEN( CON_INP, (GETCINPUT));    /* Open files */
OPEN( CON_OUT, (GETCOUTPUT));

WRITE( CON_OUT, "<FF>Multitask demo database builder.<NL><NL>");
WRITE( CON_OUT, "How many blocks do you want to create? ");
READ( CON_INP, MAX_NUM );

NCONT( FILE_NAME, MAX_NUM );    /* create file of specified length */
OPEN( DATAFIL, FILE_NAME );     /* and open it */

ALLOCATE( P, 256 );             /* Allocate a block to use */

FOR I := 1 STEP 1 UNTIL MAX_NUM DO BEGIN
    WRITE( CON_OUT, "<NL>Block number ",I,":<NL>");
    WRITE( CON_OUT, "<HT>Employee ID number? ");
    READ( CON_INP, (P + OFF_ID) -> BI );
    WRITE( CON_OUT, "<HT>Employee Pay code?  ");
    READ( CON_INP, (P + OFF_PAY_CODE) -> BI );
    WRITE( CON_OUT, "<HT>Employee Pay rate?  ");
    READ( CON_INP, PAY );
    (P + OFF_PAY_RATE) -> B12 := PAY * 100.0 + 0.5;
    WRITE( CON_OUT, "<HT>Employee name?      ");
    READ( CON_INP, NAME );
    IF LENGTH(NAME) < 20                /* pad name with blanks */
       THEN SUBSTR(NAME,20) := " ";
    (P + OFF_NAME) -> BSTR := NAME;
    ONE := 1;                           /* write out block */
    BLKWRITE( DATAFIL, I-1, P, ONE );
END;  % of FOR statement

WRITE( CON_OUT, "<NL><NL>Done!<NL>" );

END of program;

/*********************end of CREFILE.DG*********************************/
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
/*********INCLUDE file DATABASE.DG - describes relevant offsets ******/

/*
 * The database is a collection of 256-word blocks, 1 block per
 * employee.  In this demo, we are concerned only with the first
 * few words:
 *      ID              - employee ID   (integer (1))
 *      PAY_CODE        - Pay code      (integer (1))
 *      PAY_RATE        - pay * 100     (integer (2))
 *      NAME            - employee name (string (20))
 */

LITERAL
        OFF_ID          ( 0 ),          /* ID offset */
        OFF_PAY_CODE    ( 1 ),          /* PAY_CODE offset */
        OFF_PAY_RATE    ( 2 ),          /* PAY_RATE offset */
        OFF_NAME        ( 4 );          /* NAME offest */

BASED   INTEGER (1)     BI;             /* Define Based names */
BASED   INTEGER (2)     BI2;
BASED   STRING  (20)    BSTR;

/********************** End of DATABASE.DG ***************************/


/*************** INCLUDE file NAME.DG - defines the filename *********/

LITERAL FILE_NAME       ("DATAFILE.DF");

/***************************** End of NAME.DG ************************/


/********** INCLUDE file LIT.DG - defines various LITERALs ************/

LITERAL A_START (  1R8 ),               /* Signal to start task A */
        B_START (  2R8 ),               /* Signal to start task B */
        C_START (  4R8 ),               /* Signal to start task C */

        A_DONE  ( 10R8 ),               /* Signal task A is done */
        B_DONE  ( 20R8 ),               /* Signal task B is done */
        C_DONE  ( 40R8 ),               /* Signal task C is done */

/**A     E$EOF   ( 30R8 ),               /* End of File error msg, AOS  */ */
/**R     E$EOF   (  6R8 ),               /* End of File error msg, RDOS */ */

        NUMBER    ( 10 ),               /* number of # buffers in memory */
        N_MINUS_1 (  9 ),               /* NUMBER-1 */

        DATAFILE ( 0 ),                 /* DG/L file # of Datafile */
        OUTPFILE ( 1 );                 /* DG/L file # for output */
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
/***** INCLUDE File EXT.DG - supplies EXTERNAL definitions *************/

EXTERNAL INTEGER (1)     A_SEQ, B_SEQ, C_SEQ;     %% current record #
EXTERNAL POINTER         A_PTR, B_PTR, C_PTR;     %% current index #

EXTERNAL INTEGER (2)     TOTAL_PAY;
EXTERNAL INTEGER (2)     TOTAL_RAISE;
EXTERNAL INTEGER (2)     TOTAL_NEW_PAY;

EXTERNAL POINTER ARRAY   BUFFER [0:N_MINUS_1];    %% ptr to buffer
EXTERNAL BOOLEAN ARRAY   MODIFIED [0:N_MINUS_1]; %% block changed?

EXTERNAL INTEGER (1)     NUM_CHANGED;             %% # blks modified
EXTERNAL INTEGER (1)     NUM_READ;                %% # blks read

EXTERNAL INTEGER (1)     FLAGS;  %% flags for task status
EXTERNAL INTEGER (1)     MUTEX;  %% semaphore to lock critical region


EXTERNAL INTEGER PROCEDURE       RECEIVE,  READERROR;
EXTERNAL         PROCEDURE       TRANSMIT, SIGNAL,   WAITFOR, CLEAR;
EXTERNAL         PROCEDURE       BLKREAD,  BLKWRITE, ERRFATAL;

/******************** End of EXT.DG ********************************/


/********************Beginning of DEMO.DG - main program ***********/


BEGIN

COMMENT

    This program, and its associated tasks (TASKA, TASKB, and TASKC)
are meant to demonstrate various features of DG/L multitasking.  This
program updates an employee database file asynchronously.  Its purpose
is to give certain employees a specified percentage pay raise.  Although
this application is just an example, it demon-
strates some of the basic uses of multitasking.

Tasks and their purposes
------------------------
    main program         - Initialization, and final report.
    TASKA                - read in blocks from the disk
    TASKB                - process the blocks
    TASKC                - write the blocks to the disk

Rules all of the tasks in this example must follow:
---------------------------------------------------
n   The critical region must be locked/unlocked with a TRANSMIT/RECEIVE
    on the GLOBAL MUTEX.

n   Tasks can only modify their sequence #'s or current buffer pointers
    inside of the critical region.
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
   π   References to other tasks' sequence #'s or current buffer pointers
       must be inside of the critical region.

   π   If a task finds it cannot continue on to the next block, it uses
       WAITFOR to wait until it is woken up.

   π   Task's B & C must initially start out asleep.

   π   Reading must be sequential, unless the algorithims are changed.


   Notes:
   ------
       The datafile must be created with the DG/L calls NCONT or CCONT on
   RDOS/DOS, because otherwise the "End of File" indicator will not be
   raised when there are no more blocks to be read in.

       The buffers are used in a circular manner; i.e., BUFFER[1] is used
   after BUFFER[0], BUFFER[0] is used after BUFFER[N-1], etc.  Also, the
   following relations hold true for the task/block numbers (the
   sequence # of the block that the task is currently operating on):

           A_SEQ >= B_SEQ >= C_SEQ
           A_SEQ - C_SEQ <  NUMBER
           B_SEQ = A_SEQ              only when TASK B is sleeping
           C_SEQ = B_SEQ              only when TASK C is sleeping

Options:
--------
    The following compile time options are used in the code:
        A - use AOS "End of File" error code
        R - use RDOS/DOS "End of File" error code

Building the demonstration:
---------------------------
    This program was designed to show multitasking, and to
run on the various computers and operating systems DG/L runs on.
The ways to compile and link the program are listed below.
Note, for the AOS => AOS
case, we assume that you have the DG/L compiler in your searchlist.  For the
RDOS => RDOS/DOS case, we assume that you either have the DG/L compiler and
RLDR in your directory, or links to it (and that the directory
it is in has been INIT'ed).  Also, you must have the correct system library
for the operating system you intend to run on.

π   Compile on AOS, run on AOS:

    DELETE DEMO.<LS,E,LM> TASK(A,B,C).<LS,E>
    X DGL/B/X/V/L=DEMO.LS/E=DEMO.E/OPT=A DEMO
    X DGL/B/X/V/L=TASKA.LS/E=TASKA.E/OPT=A TASKA
    X DGL/B/X/V/L=TASKB.LS/E=TASKB.E/OPT=A TASKB
    X DGL/B/X/V/L=TASKC.LS/E=TASKC.E/OPT=A TASKC
    X LINK/NSLS/ALPHA/NUMERIC/MAP/MODSYM/TASKS=3/L=DEMO.LM &
                          DEMO TASK<A,B,C> [DGLIBA]

    PROCESS/DEF/BLOCK/IOC/CALLS=3 DEMO
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
n   Compile on RDOS/Eclipse, run on RDOS or DOS Eclipse:

    DELETE DEMO.<LS,E,LM> TASK(A,B,C).<LS,E>
    DGL/B/X/V DEMO<,.LS/L,.E/E> R/O
    DGL/B/X/V TASKA<,.LS/L,.E/E> R/O
    DGL/B/X/V TASKB<,.LS/L,.E/E> R/O
    DGL/B/X/V TASKC<,.LS/L,.E/E> R/O
    RLDR/N/D/A DEMO TASK<A,B,C> @DGLIBE@ DEMO.LM/L

    DEMO

n   Compile on RDOS/Eclipse, run on RDOS or DOS Nova:

    DELETE DEMO.<LS,E,LM> TASK(A,B,C).<LS,E>
    DGL/B/X/V DEMO<,.LS/L,.E/E> N/C R/O
    DGL/B/X/V TASKA<,.LS/L,.E/E> N/C R/O
    DGL/B/X/V TASKB<,.LS/L,.E/E> N/C R/O
    DGL/B/X/V TASKC<,.LS/L,.E/E> N/C R/O
    RLDR/N/D/A DEMO TASK<A,B,C> @DGLIBE@ DEMO.LM/L


Definition of State Variables
-----------------------------
    A_SEQ               - next block # TASKA is to read
    B_SEQ               - next block # TASKB is to process
    C_SEQ               - next block # TASKC is to write
    A_POS               - buffer TASKA is to use
    B_POS               - buffer TASKB is to use
    C_POS               - buffer TASKC is to use
    BUFFER[0:NUMBER-1]  - array of pointers to the buffers
    FLAGS               - word to be used with SIGNAL/WAITFOR for
                          the tasks to wait for flags from the other tasks.
                          The flags from the tasks are OR'ed into the
                          variable FLAGS with SIGNAL, which wakes up
                          a task if it was waiting on the specific
                          flag.  If no task was waiting nothing happens.
                          Each task is responsible for clearing its
                          wake-up flag so that it doesn't process the
                          next block until the previous TASK is done
                          with it.
    MUTEX               - semaphore used with TRANSMIT/RECEIVE
                          to lock the critical region; i.e., the part
                          of code that modifies the sequence # or
                          buffer pointers, and checks whether the
                          task can proceede to the next block.  Only
                          one task can be executing inside the critical
                          region at a time.  The value of 0 indicates
                          that any task that executes RECEIVE on
                          MUTEX will wait until the task currently
                          executing the critical region does a TRANSMIT.
    TOTAL_PAY           - Total (*100) of all the current PAY fields.
    TOTAL_RAISE         - Total (*100) of all the raises given.
    TOTAL_NEW_PAY       - Total (*100) of all the updated PAY fields.
    NUM_READ            - Number of blocks read in
    NUM_CHANGED         - Number of blocks modified
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
    MODIFIED[0:NUMBER-1]- Boolean array to indicate that the block was
                          written back out to the disk)

    DEMO
              ;

COMMENT

   This is the beginning of the main program.

   INCLUDE NAME;
   INCLUDE LIT;


   GLOBAL  INTEGER (1)      MUTEX;
   GLOBAL  INTEGER (1)      FLAGS;
   GLOBAL  POINTER          A_PTR, B_PTR, C_PTR;
   GLOBAL  INTEGER (1)      A_SEQ, B_SEQ, C_SEQ;
   GLOBAL  INTEGER (2)      TOTAL_PAY;
   GLOBAL  INTEGER (2)      TOTAL_RAISE;
   GLOBAL  INTEGER (2)      TOTAL_NEW_PAY;
   GLOBAL  INTEGER (1)      NUM_READ, NUM_CHANGED;

   GLOBAL  POINTER ARRAY    BUFFER[0:N_MINUS_1];
   GLOBAL  BOOLEAN ARRAY    MODIFIED[0:N_MINUS_1];

   EXTERNAL PROCEDURE               WAITFOR, TASK, TASKA, TASKB, TASKC;
   EXTERNAL PROCEDURE               GALLOCATE;
   EXTERNAL STRING PROCEDURE        GETCOUTPUT;


   LITERAL HDR                      /* header for OUTPUT */
   ("<FF>DG/L Multitask example program.<NL><NL>
      Block      Id Name                          Pay      Raise
       New Pay<NL><NL>"),

           FMNT                     /* Format for totals */
   ("<NL>                    --- Totals ---       #######.## #######.##
    #######.##<NL><NL>
   Number of records read     = ######<NL>
   Number of records modified = ######<NL>");



   INTEGER I;                       /* local variables */
   REAL    REAL_PAY, REAL_RAISE, REAL_NEW_PAY;


   FOR I := 0 STEP 1 UNTIL N_MINUS_1 DO    /* allocate buffers */
       GALLOCATE( BUFFER[I], 256 );

   A_PTR := B_PTR := C_PTR := BUFFER[0];   /* initial buffer */
   A_SEQ := 0;                             /* initial seq. # */
   B_SEQ := C_SEQ := -1;

   TOTAL_PAY := 0;                         /* zero totals */
   TOTAL_RAISE := 0;
   TOTAL_NEW_PAY := 0;
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
MUTEX := 1;                             /* unlock critical region */
FLAGS := A_START;                       /* TASKA can start executing*/

NUM_CHANGED := NUM_READ := 0;           /* no blks read or modified yet*/

OPEN( OUTPFILE, (GETCOUTPUT));          /* output file */
OPEN( DATAFILE, FILE_NAME );            /* data file */

OUTPUT( OUTPFILE, HDR );

TASK( TASKA, 0, 1, 1 );                 /* initiate tasks */
TASK( TASKB, 0, 2, 1 );
TASK( TASKC, 0, 3, 1 );

WAITFOR( FLAGS, B_DONE );               /* wait for processing task to
                                           finish */

REAL_PAY := TOTAL_PAY / 100.00;         /* convert to dollars and cents*/
REAL_RAISE := TOTAL_RAISE / 100.00;
REAL_NEW_PAY := TOTAL_NEW_PAY / 100.00;

OUTPUT( OUTPFILE, FMNT, REAL_PAY, REAL_RAISE, REAL_NEW_PAY,
                NUM_READ, NUM_CHANGED );

END of main program;


   /*******************beginning of TASKA.DG*****************************/

   PROCEDURE TASKA;
   BEGIN

   COMMENT

       This task reads in blocks of data for the other tasks to use and
   possibly modify.  Each block is 256 words (the size of a disk block).

           ;

   INCLUDE LIT;
   INCLUDE EXT;

   INTEGER I, ONE;
   BOOLEAN MORE_BUF;

   DO BEGIN
       WAITFOR( FLAGS, A_START );           /* wait until awaken */
       DO BEGIN                             /* assume an empty buffer */
           CLEAR( FLAGS, A_START );         /* clear the wakeup flag */

           ONE := 1;
           BLKREAD( DATAFILE, A_SEQ, A_PTR, ONE, EOF );
                                            /* jump to EOF on error */
           NUM_READ := NUM_READ+1;          /* increment # read */

           I := RECEIVE( MUTEX );           /* lock the critical region */
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
        A_SEQ := A_SEQ + 1;              /* prepare to read next blk */
        A_PTR := BUFFER[ MOD( A_SEQ, NUMBER ) ];
        MORE_BUF := (A_SEQ - C_SEQ) < NUMBER;
                                         /* is next blk free? */

        TRANSMIT( MUTEX, 1 );            /* unlock critical region */

        SIGNAL( FLAGS, B_START );        /* start B it if was asleep */
        END
    WHILE MORE_BUF;

END;                                     /* loop until EOF abnormal rtn*

EOF:                                     /* handle errors in reading */
    IF (I := READERROR) <> ESEOF
        THEN ERRFATAL( I );              /* If not EOF, terminate */

    SIGNAL( FLAGS, A_DONE + B_START );   /* Tell the world no more
                                            buffers in file */
END of taska;

  /******************* end of TASKA.DG ************************/


  /***************** beginning of TASKB.DG ********************/

PROCEDURE TASKB;
BEGIN

COMMENT

    This task waits until a block has been read in, and then it
processes the record, and signals TASKC to write out the block
asynchronously.  The processing that this task does is as follows:

    1. If ( 10 <= PAY_CODE <= 20 ) is true, give the employee a
       10% raise.
    2. If PAY_CODE > 20 then give a 12% raise.
    3. If the pay changed, write it out to the list file

                        ;

INCLUDE LIT;
INCLUDE EXT;
INCLUDE DATABASE;

LITERAL
        RATE_1  ( 10 ),                  /* Parameterize rates */
        RATE_2  ( 20 ),

        PERCENT_1 ( 0.10 ),              /* and percentage raise */
        PERCENT_2 ( 0.12 );

LITERAL FMNT    (                        /* OUTPUT format */
"   #####   ##### ####################  ######.##  ######.##
  ######.##<NL>");
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
BOOLEAN PROCEDURE MORE_BLKS;                    /* TRUE if more to do */
BEGIN
    INTEGER I;

    I := RECEIVE( MUTEX );                      /* lock critical region */
    IF (MORE_BLKS := (B_SEQ+1) < A_SEQ) THEN BEGIN
        B_SEQ := B_SEQ + 1;                     /* go to next record */
        B_PTR := BUFFER[ MOD( B_SEQ, NUMBER )];
    END;
    TRANSMIT( MUTEX, 1 );                       /* unlock critical region */
END of more_blks;



BOOLEAN     CHANGED;
INTEGER     CODE;
INTEGER (2) NEW_PAY, RAISE, PAY;
REAL        REAL_PAY, REAL_NEW_PAY, REAL_RAISE;



DO BEGIN
    WAITFOR( FLAGS, B_START );                  /* wait for a block */
    WHILE MORE_BLKS DO BEGIN                    /* process each block */
        CHANGED := FALSE;
        CLEAR( FLAGS, B_START );                /* clear wakeup flag */
        CODE := (B_PTR + OFF_PAY_CODE) -> BI;
        PAY := (B_PTR + OFF_PAY_RATE) -> BI2;
        RAISE := 0;

                                                /* compute possible raise */
        IF (RATE_1 <= CODE) AND (CODE <= RATE_2) THEN BEGIN
            RAISE := PAY * PERCENT_1 + 0.5;
            CHANGED := TRUE;
            END
        ELSE IF CODE > RATE_2 THEN BEGIN
            RAISE := PAY * PERCENT_2 + 0.5;
            CHANGED := TRUE;
            END;

        MODIFIED[ MOD(B_SEQ, NUMBER) ] := CHANGED;
        IF CHANGED THEN BEGIN
            NUM_CHANGED := NUM_CHANGED + 1;
            (B_PTR + OFF_PAY_RATE) -> BI2 := NEW_PAY := PAY + RAISE;
            TOTAL_NEW_PAY := TOTAL_NEW_PAY + RAISE;
            TOTAL_RAISE := TOTAL_RAISE + RAISE;
        END ELSE
            NEW_PAY := PAY;

        REAL_PAY := PAY / 100.00;               /* change pay to $ & cents */
        REAL_RAISE := RAISE / 100.00;
        REAL_NEW_PAY := NEW_PAY / 100.00;
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
        OUTPUT( OUTPFILE, FMNT,                    /* write out info */
                          B_SEQ + 1,               /* sequence # */
                          (B_PTR + OFF_ID) -> BI, /* employee ID */
                          (B_PTR + OFF_NAME) -> BSTR,     /* employee Name */
                          REAL_PAY,                /* current pay */
                          REAL_RAISE,              /* raise given */
                          REAL_NEW_PAY );          /* new pay */

        TOTAL_NEW_PAY := TOTAL_NEW_PAY + PAY;
        TOTAL_PAY := TOTAL_PAY + PAY;

        SIGNAL( FLAGS, C_START );          /* wake TASKC if sleeping */
    END;

END UNTIL (FLAGS AND A_DONE) <> 0;         /* keep going until EOF */

SIGNAL( FLAGS, B_DONE + C_START );         /* tell world I'm done */

END of taskb;

 /****************** end of TASKB.DG ********************************/


 /*****************beginning of TASKC.DG ***************************/

PROCEDURE TASKC;
BEGIN

COMMENT

    This task write blocks (256 words each) out to the disk.  While
it is waiting for the I/O action to occur, the other tasks can be
executing.  If it finds that it has nothing to write out at the
moment, it puts itself to sleep (initially it starts out sleeping).
TASKB then wakes it up, when it has processed the block.  After
writing out a block, it signals to TASKA that a buffer has been
freed.  At the end, it signals that it is finished.
                ;

INCLUDE LIT;
INCLUDE EXT;

BOOLEAN PROCEDURE MORE_BLKS;               /* TRUE if more blocks to write*/
BEGIN
    INTEGER I;

    I := RECEIVE( MUTEX );                 /* lock the critical region */
    IF (MORE_BLKS := C_SEQ < B_SEQ) THEN BEGIN
        C_SEQ := C_SEQ + 1;                /* prepare to write next blk */
        C_PTR := BUFFER[ MOD( C_SEQ, NUMBER )];
    END;
    TRANSMIT( MUTEX, 1 );                  /* unlock the critical region */
END of more_blks;
```

Figure E-1. Complete DG/L Multitask Program (continued)

```
DO BEGIN
    INTEGER ONE;

    WAITFOR( FLAGS, C_START );          /* go to sleep */
    CLEAR( FLAGS, C_START );            /* clear the flag */
    WHILE MORE_BLKS DO BEGIN
        CLEAR( FLAGS, C_START );        /* in case TASKB did a signal
                                           while looping */
        ONE := 1;
        IF MODIFIED[ MOD(C_SEQ, NUMBER) ] THEN
            BLKWRITE( DATAFILE, C_SEQ, C_PTR, ONE );

        SIGNAL( FLAGS, A_START );       /* start A if sleeping */
    END;

END UNTIL (FLAGS AND B_DONE) <> 0;

SIGNAL( FLAGS, C_DONE );                /* tell the world I'm done */
END of taskc;

/********************** end of TASKC ************************/
```

Figure E-1. Complete DG/L Multitask Program (concluded)

End of Appendix

Appendix F
DG/L Runtime Routines Implemented Under Both AOS and RDOS

This list names all routines in this manual that will run under DGC's Real-Time Disk Operating System (RDOS), as well as AOS. To write DG/L programs that can run under either system, use only these runtime calls. For additional calls available only under RDOS, see the RDOS edition of this manual (093-124).

Routine	Chapter
ACCESS	9
ADDRESS	3
AKILL	9
ALLOCATE	3
APPEND	9
AREADY	9
ASCII	3
ASUSPEND	9
ATTRIBUTE	5
BLKREAD	6
BLKWRITE	6
BUFFER	6
BUFLOCK	6
BUFUNLOCK	6
BYTE	3
BYTEREAD	6
BYTERITE	6
CBIT	3
CCONT	5
CDIR	5
CHAIN	7
CLASSIFY	3
CLEAR	9
CLOSE	6
COMARG	3
.CONSOLE	6
CPART	5
CRAND	5
CREATE	5
DATACLOSE	6
DATAOPEN	6
DATAREAD	6
DATAWRITE	6
DELAY	3
DELETE	5
DIM	3
DIR	5
EOPEN	6
ERETURN	4, 7
ERPRINT	4
ERRFATAL	4
ERRINTERCEPT	4
ERRKILL	4
ERROR	4
ERRTRAP	4
ERRUSER	4
FETCH	6
FILEPOSITION	6
FILESIZE	5
FLUSH	6
FPUERROR	4
FREE	3
GALLOCATE	9
GETCINPUT	6
GETCOUTPUT	6
GETDIR	5
GETFREQUENCY	9
GETIDENTIFIER	9
GETLIST	3
GETPRIORITY	9
GFREE	9
GMEMORY	9
GTIME	3
HASHREAD	6
HASHWRITE	6
HBOUND	3
IDPRIORITY	9
IDSTATUS	9
INIT	5
INDEX	3
KILL	9
LBOUND	3
LENGTH	3
LINEREAD	6
LINEWRITE	6
LINK	5
MEMORY	3
MINRES	6
MULTITASK	9

Routine	Chapter	Routine	Chapter
NAMEGROUND	7	SIZE	3
NCONT	5	STASH	6
NODEREAD	6	STATUS	5
NODESIZE	6	STIME	3
NODEWRITE	6	SUBSTR	3
NOFPUTRAP	4	SUSPEND	9
NOMESSAGE	4	SWAP	7
		SYSRETURN	7
ODIS	6	SYSTEM	3
OEBL	6		
OPEN	6	TASK	9
		TBIT	3
POSITION	6	TIDABORT	9
PRIORITY	9	TIDKILL	9
		TIDPRIORITY	9
QKILL	9	TIDREADY	9
QTASK	9	TIDSUSPEND	9
		TOVKILL	9
RANDOM	3	TOVLOAD	9
RCOMARG	3	TOVRELEASE	9
READERROR	4	TRANSMIT	9
READSTRING	6	TRDOPERATOR	9
RECEIVE	9	TWROPERATOR	9
RELEASE	5		
REM	3	UMUL	3
RENAME	5	UNLINK	5
ROPEN	6		
ROTATE	3	WAITALL	9
RUNTASK	9	WAITFOR	9
		WORDREAD	6
SBIT	3	WORDWRITE	6
SEED	3	WRITESTRING	6
SETCURRENT	3	WTRANSMIT	9
SHIFT	3		
SHORTMESSAGE	4	XCT1	3
SIGNAL	9	XCT2	3
SINGLETASK	9		

End of Appendix

INDEX

Within this index, "f" or "ff" after a page number means "and the following page" (or "pages"). In addition, primary page references for each topic are listed first. Routine names, commands, and acronyms are in uppercase letters (e.g. CRREATE); all other entries are lowercase.

D

E

F

G

H

I

Q

R

S

X

DataGeneral

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
MV/Series ECLIPSE®		
Commercial ECLIPSE		
Scientific ECLIPSE		
Array Processors		
CS Series		
NOVA®4 Family		
Other NOVAs		
microNOVA® Family		
MPT Family		
Other _____ (Specify) _____		

3. Software

- ☐ AOS
- ☐ AOS/VS
- ☐ AOS/RT32
- ☐ MP/OS
- ☐ MP/AOS
- ☐ RDOS
- ☐ DOS
- ☐ RTOS
- ☐ Other

Specify _____

4. Languages

- ☐ ALGOL
- ☐ DG/L
- ☐ COBOL
- ☐ Interactive COBOL
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ FORTRAN 77
- ☐ FORTRAN 5
- ☐ RPG II
- ☐ PL/1
- ☐ APL
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communication

- ☐ HASP
- ☐ HASP II
- ☐ RJE80
- ☐ RCX 70
- ☐ RSTCP
- ☐ 4025
- ☐ X.25
- ☐ SAM
- ☐ CAM
- ☐ XODIAC™
- ☐ DG/SNA
- ☐ 3270
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

CUT ALONG DOTTED LINE

DataGeneral

Data General Corporation, Westboro, Massachusetts 01580 (617) 366-8911

FOLD FOLD

TAPE TAPE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

CUT ALONG DOTTED LINE

Data General

TP_____

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

SHIP TO: (if different)
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #_____
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No._____ Expiration Date_____

☐ Purchase Order Number:_____

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other _____

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order _____ Phone _____ Extension _____

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

Title

DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

Data General

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services – M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

User Documentation Remarks Form

Your Name _____ Your Title _____

Company _____

Street _____

City _____ State _____ Zip _____

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title _____ Manual No. _____

Who are you? ☐EDP Manager ☐Analyst/Programmer ☐Other _____
☐Senior Systems Analyst ☐Operator _____

What programming language(s) do you use? _____

How do you use this manual? *(List in order: 1 = Primary Use)* _____

___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide _____

About the manual:	Yes	Somewhat	No
Is it easy to read?	☐	☐	☐
Is it easy to understand?	☐	☐	☐
Are the topics logically organized?	☐	☐	☐
Is the technical information accurate?	☐	☐	☐
Can you easily find what you want?	☐	☐	☐
Does it tell you everything you need to know	☐	☐	☐
Do the illustrations help you?	☐	☐	☐

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

User Documentation Remarks Form

Your Name _____ Your Title _____

Company _____

Street _____

City _____ State _____ Zip _____

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title _____ Manual No. _____

Who are you? ☐EDP Manager ☐Analyst/Programmer ☐Other _____
☐Senior Systems Analyst ☐Operator _____

What programming language(s) do you use? _____

How do you use this manual? *(List in order: 1 = Primary Use)* _____

___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide _____

About the manual:	Yes	Somewhat	No
Is it easy to read?	☐	☐	☐
Is it easy to understand?	☐	☐	☐
Are the topics logically organized?	☐	☐	☐
Is the technical information accurate?	☐	☐	☐
Can you easily find what you want?	☐	☐	☐
Does it tell you everything you need to know	☐	☐	☐
Do the illustrations help you?	☐	☐	☐

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

Data General Corporation, Westboro, MA 01580

093-000159-02