DG/L™ Runtime Library

User's Manual

(RDOS)

093-000124-00

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. 093-000124

Printed in the United States of America
Revision 00, October 1978

NOTICE

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

Revision History:

Original Release - October 1978

Preface

This manual describes the DG/L™ Runtime Library under the Real Time Disk Operating System (RDOS). It documents the format, purpose, and arguments of each call in the DG/L Runtime Libraries.

To use this manual you should have programming experience in DG/L or another ALGOL-like language. You should read the *DG/L Reference Manual* (093-000229) and be familiar with the *Real Time Disk Operating System (RDOS) Reference Manual* (093-000075). An understanding of assembly language will help you use DG/L runtime routines, but you do not need assembly language programming experience to use the runtime library.

All DG/L Runtime Routines in this manual (except RDSW) also run under the Real-Time Operating System (RTOS) with appropriate equipment. If you are using RTOS, you should refer also to the *Real Time Operating System Reference Manual* (093-000056).

If you are using DG/L Runtime's interface to the INFOS® system, you will need the following publications: the *INFOS System User's Manual* (093-000114); *INFOS System Utilities Manual* (093-000182); *Introduction to the INFOS System* (093-000113).

The bulk of this manual is primary reference, giving brief descriptions of each call. We organized the routines in functional groups; related routines appear together in chapters. The routines appear alphabetically within the chapters.

Chapter 1, the Introduction has two parts: the first part describes the format and use of runtime calls, and explains your options for handling errors; the second part details the runtime environment and describes writing an assembly language runtime routine to use in a DG/L program.

The rest of the chapters, which describe routines, fall into four broad categories:

Chapters 2-8 describe general routines which use DG/L and system resources to obtain information, manipulate bits and strings, and report and handle errors.

Chapters 9-14 contain file-related routines which create and manage files and directories, perform input and output, and interface with INFOS.

Chapters 15-18 contain program-extending routines for swaps, overlays, foreground/background and multiple-processor programs.

Chapter 19 introduces DG/L multitask programming, and Chapters 20-29 describe multitasking routines that obtain information, change task states, and provide task communication.

Appendixes A to C alphabetically list DG/L runtime, mathematical, and conversion routines and their functions.

Appendix D documents assembly language routines that perform formatted input and output.

Appendix E lists all DG/L error codes and messages.

Appendix F lists alphabetically all DG/L runtime routines that run under both the Real-time Disk Operating System and DGC's Advanced Operating System (AOS).

Appendix G contains a line printer listing of DG/L tables.

To correctly use this manual, you should read the Introduction and become familiar with the chapter groupings, both for an overview of DG/L runtime and for easy reference. As you become familiar with the runtime, you should periodically review chapters you use often and try to learn the different approaches you can take to a given type of problem.

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: $\left\{\begin{matrix}\text{required}_1\\ \text{required}_2\end{matrix}\right\}$ 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 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.

R is the RDOS/DOS CLI prompt.

End of Preface

Contents

Chapter 1 - Introduction

Chapter 2 - General User Routines

Chapter 3 - Performing Mathematical Operations with Integers and Words

Chapter 4 - Bit and String Manipulation

Chapter 5 - Obtaining Information about Arrays

Chapter 6 - Command Line Handling

Chapter 7 - Using the System Clock

Chapter 8 - Handling Errors

Chapter 9 - Managing Directories and Devices

Chapter 10 - Maintaining Files

Chapter 11 - File Input/Output

Chapter 12 - Console Input/Output

Chapter 13 - Cache Memory Management

Chapter 14 - INFOS System Routines

Chapter 15 - Calling to and Returning from Programs

Chapter 16 - Using Overlays in a Single-Task Environment

Chapter 17 - Foreground/Background Programming

Chapter 18 - User Device and Multiple Processor Routines

Chapter 19 - Multitask Programming in DG/L

Chapter 20 - Initiating Tasks in a Multitask Environment

Chapter 21 - Obtaining Task-Related Information in a Multitask Environment

Chapter 22 - Changing Task States in a Multitask Environment

Chapter 23 - Queueing Tasks for Deferred or Periodic Execution

Chapter 24 - Using Overlays in a Multitask Environment

Chapter 25 - Using Memory Management in a Multitask Environment

Chapter 26 - Intertask Communication

Chapter 27 - Task/Operator Communications in a Multitask Environment

Chapter 28 - User/System Clock Commands

Chapter 29 - Disabling and Enabling the Multitask Environment

Appendix A - Alphabetic List of DG/L Runtime Routines

Appendix B - Alphabetic List of Mathematical Runtime Routines

Appendix C - Conversion Routines

Appendix D - Formatting Routines

Appendix E - DG/L Runtime Errors

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

Appendix G - DG/L Tables and Files

Illustrations

Chapter 1
Introduction

The DG/L™ Runtime Library offers you an extensive set of functions and interfaces with the operating system. It makes most of the RDOS system and task calls available as statements and functions in a high-level language. Using DG/L runtime routines you will be able to program sophisticated machine operations in a simple, efficient format.

The runtime library allows you to make a DG/L program work interactively; to process errors within a program either before execution terminates or without terminating at all; to control input and output with disk, tape, printer, and console devices; and to considerably extend the effective capacity of memory by using such processes as program swapping, overlays, Cache Memory Management, disk file I/O, and multitask programming.

Runtime Calls

This manual gives the following information about each routine in the DG/L Runtime Libraries: name, purpose, call format, definitions of arguments, error conditions, an example, and any necessary rules, notes, and references.

Call Formats

DG/L runtime routines are either statements or functions. This manual presents *statement calls* in the format:

routine name *[(arguments)];*

The arguments must appear in parentheses. We indicate optional arguments with square brackets.

Function calls appear in the format of an assignment statement:

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

However, in a program you may call the routine in an expression, just as you may with any other function call. The variable, if you use one, must be compatible with the value that is returned. This manual uses an i to indicate integer variables, an s for strings, and bool for booleans.

Declaration

You must declare each runtime routine as an external procedure before calling it. The declaration forms are:

EXTERNAL PROCEDURE name;

EXTERNAL data type PROCEDURE name;

Use the second declaration for function routines; the data type will be STRING, INTEGER, or BOOLEAN (or another type if you write your own routine).

You must declare all runtime routines in your program. Built-in routines--those which are also documented in the DG/L Reference Manual--need not be declared. (If you do declare a built-in routine, the compiler will first search for a user-written module of that name.)

Arguments

In each routine description, after presenting the call format, we define the call's arguments. These definitions explain the type of data or expression required for each argument, and note whether the argument passes or returns a value.

Numerical values can be variables, constants, or expressions unless otherwise indicated. You cannot use *expressions* as arguments in calls such as error reporting, where the routine must store and reference a value in an argument. *Integer expressions* must be single-precision unless otherwise indicated.

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

Data arrays can be variables of any type (excluding string) that store the required number of words as a block, unless the routine requires an array as in DIM. A required four-word data array, for example, could be a double-precision real variable.

The Error Label Argument

Many DG/L runtime routines have an optional error label argument. This argument, usually accompanying calls to the operating system, must be a statement label that directs control to another part of the program. If you provide an error label, you can prevent errors from terminating a program, provide additional processing before termination, or branch elsewhere in the program.

The call to ALLOCATE, for example, contains the error label option:

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);
```

As in the earlier conditional statement example, MEMORY appears here in an expression. This time, however, if the expression returns a negative value, an error condition results; control will then branch to the statement label IERR.

At IERR you might, as in the earlier example, 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 do not provide an error label (where the option exists), and a fatal error occurs, your program will terminate and control will return to the operating system.

See the section of this chapter entitled "Error Handling" for a complete outline of routines and options for handling errors.

Error Conditions

In each routine's description, we list possible error conditions after defining the call's arguments. This list includes the mnemonics of any RDOS system error codes (in order of their RDOS numbers) or DG/L error codes the system might return with a description of the error code's meaning.

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

ERFNO	Illegal channel number.
ERFOP	Attempt to reference an unopened unit.
EDDTO	Ten second disk timeout occured.

These three error conditions can result from the system call to .CLOSE. Other routines may return DG/L error codes as well as system codes.

All RDOS errors have a mnemonic beginning with ER; DG/L errors begin with AI or AE.

To find the currently defined numerical values for RDOS error codes, see the Error Message Summary in Appendix A of the *RDOS Reference Manual.* For DG/L error codes, see Appendix F of this manual.

You can use these error codes to define an error-handling procedure under the runtime routine READERROR. See the "Error Handling" section of this chapter and the routines in Chapter 8 for a discussion of routines for reporting and handling errors.

Rules and Notes

In some routines' descriptions you will find additional information under the headings RULES and NOTES. RULES point out information essential for calling the routine, whereas NOTES contain additional information about the call's operations and effects.

Examples

The runtime routine SETCURRENT resets the current length of a string variable. Its statement format 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 could 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 a value, which is the number of words of memory currently remaining for the user. Its function format is:

```
i := MEMORY
```

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

```
EXTERNAL INTEGER PROCEDURE MEMORY;
```

A call to MEMORY might look like this:

```
LEFT := MEMORY;
```

You might also call MEMORY as a function in your program; for example, as an expression 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, then the condition is met and the program produces the message. The actual value that MEMORY returns is not stored.

System References

Most of the routines in the DG/L runtime library make calls to RDOS. The system calls perform input and output, change programs and tasks, manage memory, and communicate with peripheral devices. In this manual, descriptions of routines that call the system include a section of references. Under the call to OPEN, for example, you will find the following under the head, REFERENCE:

.OPEN (System call)

.GTATR (System call)

.CRAND (System call)

The OPEN routine makes system calls to .OPEN, .GTATR, and .CRAND.

Some multitask routines make RDOS task calls. RECEIVE, for example, calls RDOS .REC:

Reference

.REC (Task call)

To follow up both kinds of reference, look up the mnemonic of the call in the index of the *RDOS Reference Manual.* It will give you further information about the process, contents of accumulators, and side effects of the system or task call. In most cases, this information is not essential for using the runtime routine. In some cases, however, the *RDOS Reference Manual* provides necessary tables of bit codes and error messages.

Error Handling

A runtime error condition occurs when the system is unable to complete a call from your program. Error conditions commonly involve inconsistencies in runtime calls, discrepancies between a call and the current state of memory or files, and conflicts with hardware or software limitations. Runtime errors may be non-fatal, fatal to a task, fatal to the process, or absolutely fatal (untrappable).

The result of an error condition depends on how your program defines error handling. On a fatal error, control passes from the system to the DG/L error handler, .RTER. If you do not specify an error-handling procedure and there is no error label, .RTER stops execution of the task or program that caused the error, generates an error report, and transfers control to the next higher level of execution--possibly to the operating system.

You may alter this error-handling process with error-handling declarations and routines, which allow you to modify the error reporting procedure, to process errors before termination, and even to continue your program's execution after a fatal error without termination.

Messages

If you do not specify an error-handling routine, on a fatal error .RTER produces a full error report and terminates the calling program. The report contains the full error message, the location of the procedure that detected the error, and the absolute location of the procedure call or operation that caused the error.

You may declare instead to have either short messages or no message.

The declaration EXTERNAL INTEGER SHORTMESSAGE produces abbreviated error reports. SHORTMESSAGE gives the error number but not the message, the location of the procedure that detected the error, and the absolute location of the procedure or operation that caused the error.

The declaration EXTERNAL INTEGER NOMESSAGE provides for output of no message when an error occurs. NOMESSAGE has the advantage of freeing memory space; it removes from storage the error messages and the code necessary to print them.

Declare SHORTMESSAGE and NOMESSAGE among the data declarations in your program. Because they are EXTERNAL integers, they are in effect for the full execution of the program. You may not make both declarations in a single process; if you do so, you will create a load error.

To provide your own error messages, you may include a WRITE statement in an error-handling portion of your program. You may also use the DG/L runtime routine ERROR to specify an error message within a given part of your program.

For a table of RDOS error codes see the Error Summary in the *RDOS Reference Manual.* For a table of DG/L error codes, see Appendix F of this manual.

Error-Handling Routines

The two routines ERRTRAP and ERRINTERCEPT allow you to process errors within a DG/L program. Their mechanism is similar to that described above in "The Error Label Argument." These routines identify an error-handling routine as a statement label or procedure (elsewhere in the program) that receives control. The error-handling routine thus operates as a subroutine to .RTER, which receives control when the user-defined routine is complete.

ERRINTERCEPT

ERRINTERCEPT allows you to define pre-termination error handlers. Its argument is the procedure name of your error handling routine. Your pre-termination error processing might include generating a detailed error report, or writing out buffers to preserve current data. When your subroutine is complete, control returns to .RTER and normal error termination continues.

ERRTRAP

ERRTRAP makes it possible to avoid termination entirely when an error occurs. Its argument is a statement label at the beginning of your error-handling code. With ERRTRAP, control never returns to the DG/L error handler until a new error occurs. The routine you write could correct the error condition and try again, branch to another part of the program, or simply terminate the task or program with a report.

.RTER

When no error-handling routine is in effect, either through a declaration or as an argument to the call, the runtime error handler .RTER receives control. This handler will print the appropriate error message and, if the error is fatal, terminate the task or program.

Declaring an Error Handler

Error-handling routines work as declarations; they are local to the block where you declare them, global to blocks nested at lower levels, and overridden by local declarations in lower blocks. You may specify an interceptor or trap for each type of error you wish to process in a given block, and all will be in effect concurrently. However, if you declare a new handler for a given type of error in the same block as an earlier declaration for that same error type, the new handler replaces the earlier one.

In multitasking, each task is logically separate, and thus requires a separate declaration of error-handling procedures. If control passes to an error handler at a higher block level, the lower block will be released when an error occurs and its data will be lost. The example in Figure 1-1 illustrates the block structure of error handling.

```
##
  BEGIN
      INTEGER I;
      EXTERNAL PROCEDURE ERRTRAP;
       .
       .
       .
      ERRTRAP (LAB1,74000R8);
       .
       .                    /*LAB1 WILL BE THE ERROR TRAP*/
       .
       BEGIN
         STRING (10) S1
         ERRTRAP (LAB2,74006R8);
          .
          .                              /*LAB2 WILL BE THE ERROR TRAP*/
          .
       LAB2:
         .
         .
         END;
       BEGIN
         INTEGER Y;
         .
         .                              /*LAB1 WILL BE THE ERROR TRAP*/
         .
 END;
 .
 .
LAB1:
.
.                       /*LAB1 WILL BE THE ERROR TRAP*/
.
END.
##
```

Figure 1-1. Block Structure of Error Handling

The declaration EXTERNAL PROCEDURE ERRTRAP provides the DG/L code for trapping the error. The actual code is part of your program, and begins at the statement label referenced in the call to ERRTRAP.

Each block's error handler can specify a different procedure for reading and handling an error. You can also limit the procedure's application to handle only specific types and severities of error. In Figure 1-1, the octal value arguments indicate the type of error and handling. (See Chapter 8 for charts of values for error masks.)

If an error were to occur in the second inner block, the block would terminate when control transferred to LAB1. If you wanted to preserve the values of that block's variables, you would need an intercepting routine within the lower block.

ERRINTERCEPT and ERRTRAP may be concurrently in effect at any point in a program. The numerical mask you declare for each can define them as handling different, overlapping, or identical classes of errors. When both an intercept and a trap are in effect for a single error, the error will be intercepted and then trapped.

Using Error Codes

When an error condition terminates a program, the system prints out an octal RDOS error code and the code's associated message at the output console. When you create an error-handling routine, however, this procedure may be modified. If your program successfully intercepts an error, the system will not report it. Moreover, you may want to use the error code as a value within the error-handling routine.

A call to the INTEGER PROCEDURE READERROR accesses the error code in system storage and returns its value. Your error routine can use this value in conditional statements to define different error-handling routines.

If you want the system to print the error message for intercepted errors, use the call to ERPRINT in your error handling; this routine will print out the last error message if it has not already been printed. ERRFATAL also prints out the system's error message, provides for a return to the system, and avoids any further error handling.

The example in Figure 1-2 uses several error-handling routines in a procedure for opening files:

```
PROCEDURE OPEN (FILENO, FILENAME, ERR);
INTEGER FILENO;
STRING FILENAME;
LABEL LAB;
BEGIN
 INTEGER I;
 STRING S (LENGTH(FILENAME)+1);
 EXTERNAL PROCEDURE SYSTEM, CRAND;
 EXTERNAL INTEGER PROCEDURE READERRCR;
 LITERAL $OPEN (14077R8);
 LITERAL ER$DLE (12R8);

RETRY:   SYSTEM ($OPEN,ADDRESS(S:= FILENAME!!"<NUL>"),(0),
                  FILENO,MYERR);
                  GO TO ENDIT;

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

ENDIT:  END;
```

Figure 1-2. Using Error Handling Routines

The procedure in Figure 1-2 uses a direct interface to the system (SYSTEM) to open a file on the assembly level. While the call to SYSTEM reduces overhead from the usual OPEN call, it does not create the files that the OPEN routine provides. The routine uses the error label to supply that feature.

The code at RETRY opens the file, and transfers control to MYERR if the open fails. MYERR provides two branches. It reads the error code and compares it to the value ER$DLE. If the code is 12_8 (FILE DOES NOT EXIST), it creates a file and returns to RETRY. Otherwise, it calls ERRUSER, which passes control to the DG/L error handler, along with the value of the error code.

Note that the procedure actually defines the value of the mnemonic ER$DLE. The mnemonic itself is not stored by the system; only the numerical code is. ER$DLE, however, makes the code more readable, and makes it easier to update the program.

Chapter 8 documents these and related DG/L error-handling and error-reporting routines.

The Runtime Environment

This section defines the DG/L runtime environment--the allocation and management of memory for a running program. It explains how the runtime stack is structured, how it records task information, and how it uses memory globally under RDOS single-tasking and multitasking.

If you have a good knowledge of this section, you should be able to use DG/L runtime routines effectively and manage memory efficiently. If you are writing an assembly language routine of your own to use in a DG/L program, you will need a thorough understanding of the runtime environment.

The diagrams in Figure 1-3 represent a user's memory locations for unmapped RDOS. They show the memory space as a vertical column with addresses increasing toward the top of memory (bottom of the illustration).

The two columns illustrate the difference between a bound program and a running one. When a program is first loaded into memory it occupies blocks of memory that remain stable throughout execution:

- The *operating system* occupies the first 20_8 addresses and an area at the top of memory (unmapped RDOS only).
- *Page zero,* 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.
- The *User Status Table* (UST) and *Task Control Blocks* (TCBs) start at location 400_8. The command line RLDR specifies the number of TCBs the program requires. (The default is 1.)

- The next area contains *OWN variables and pointers.* Scalar values are actually stored here. OWN arrays and strings are stored in the heap, with array pointers and string specifiers in this area.

- The *user program and runtime routines* follow. The executable code comprises the main program, subprograms, DG/L runtime routines, and system routines. The module DGLINIT, which initializes the program, occupies the locations after the executable code. The locaton .LLOC marks its end.

- If you are using a debugger there will be a symbol table after .LLOC. This area moves as a block to a zone at the top of available memory, just below the operating system.

Mapped RDOS's allocation of memory is identical except that the operating system occupies another memory unit.

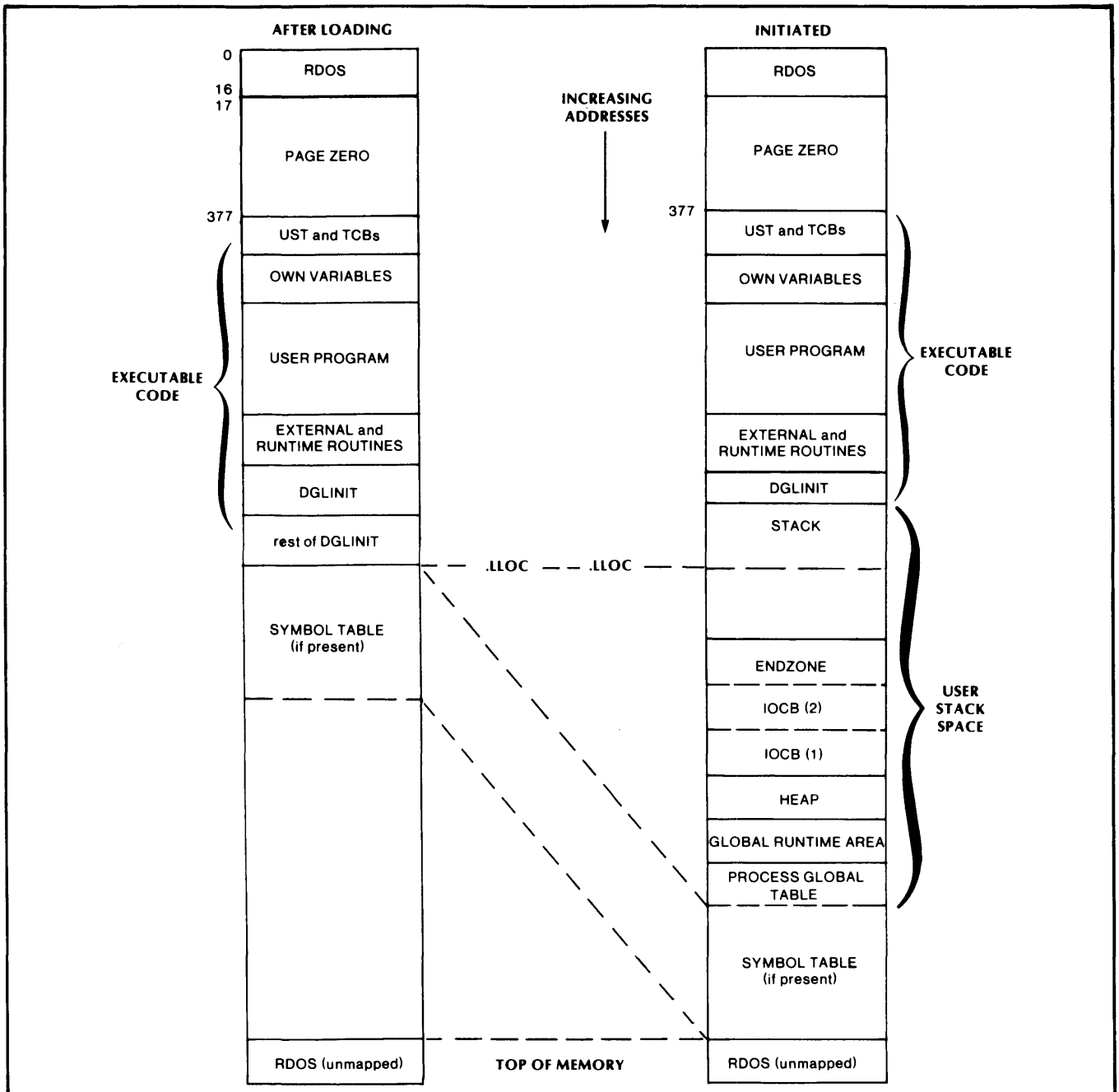


Figure 1-3. The Runtime Environment (Unmapped RDOS)

Initiating the Program

The program's execution begins with DGLINIT, which moves the symbol table (if present), creates areas for dynamic allocation, releases some of its own code to the stack after it has executed, and calls your program as a subroutine. If the program terminates normally, control returns to DGLINIT at the end of execution.

DGLINIT allocates a user stack space in memory, and partitions it into data and table areas.

- A single *Process Global Table* contains information global to all tasks and to the program as a whole. See Appendix G for a list of its contents. The page zero pointer .PT points to its beginning.
- The *user stack* occupies the area between the initializer and the lowest address of the Process Global Table. This area dynamically stores data -- the variables and temporaries for programs and subroutines, and links between programs and subroutines. The page zero pointer .SP indicates its current highest address.
- The *Global Runtime Area* maintains task control and input/output information. See Appendix G for a list of its contents. The page zero pointer .GP points to the currently active task's information.
- *The heap,* which stores OWN data variables, begins at the address below the Global Runtime Area, and grows downward into lower addresses as your program requires data space.

The size of the user stack is defined by the DG/L symbol .NMAX (in DGLSYM.SR). Applying the value in .NMAX, the initializer uses 77777_8 as the highest address of user space. If you want to define a smaller stack for a program, reset .NMAX to a lower value. See Appendix G for information on changing symbol and parameter default values.

The Multitask Environment

Figure 1-4 shows the multitask environment after initiation. The elements of the memory area are the same, but here each task has its own stack area within the heap. Like heap data, task stack areas are allocated downward from the top of memory.

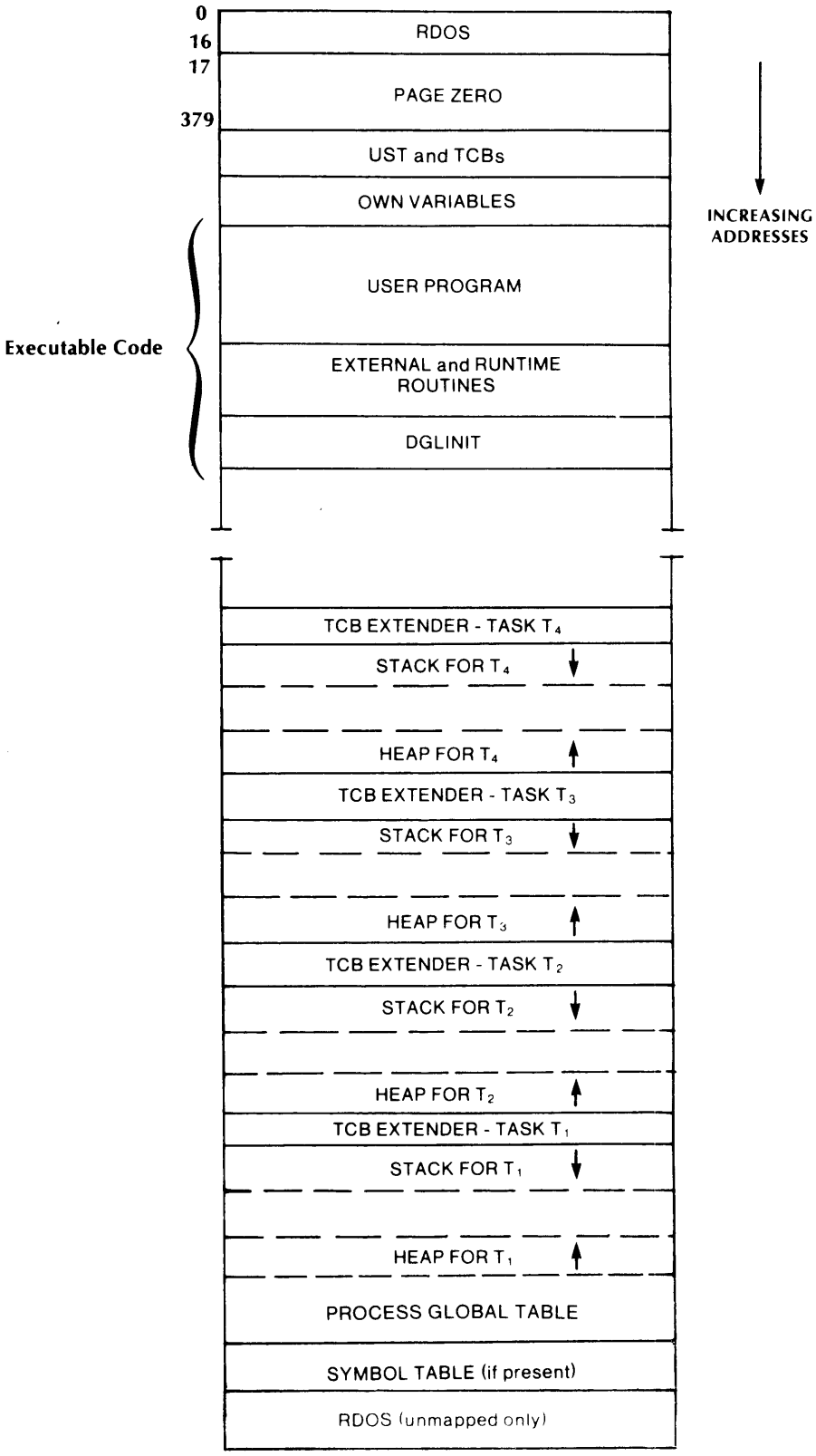
Each task has a Global Runtime Area, a Task Control Block (TCB), a temporary TCB extension and a runtime stack. When the task is not executing, the TCB extender stores the values for its page zero pointers and the floating point unit.

In the multitask environment, the task initializer fixes the size of each task stack space during runtime; you may allocate a different size of stack space to each task. When a task terminates, its stack area returns to the global free chain for re-allocation, but other stack spaces remain in the same locations while the tasks execute. When a request to allocate space fails to find a large enough free area, the allocation routine will merge free adjacent blocks.

Runtime Stack Discipline

With this system of runtime tables and pointers, you can refer to 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. DG/L allocates active variables on the stack in the program's sequence of execution, and locates them with reference to the frame pointer of the block that declared them. Through page zero and the User Status Table, which are accessible from any part of memory, you can locate the currently active code, variables, task information and global information.

The following section describes the logic in this process of reference and chaining. Your DG/L program operates with data, alternates among tasks and subroutines, and manages input/output operations through the user stack (i.e., the dynamically allocated addresses in memory).



SD-00994

Figure 1-4. The Multitasking Runtime Environment (Unmapped RDOS)

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 described in Table 1-1.

Table 1-1. Page Zero Pointer Values

Pointer	Address Indicated
.SP	Pointer to top of the user stack (per task).
.FP	Pointer to top of the current stack frame (per task).
.SSE	Pointer to the top of available stack space.
.RP .RSP	Temporary page zero locations (per task).
.PT	Pointer to beginning of the Process Global area (per process).
.GP	Pointer to the beginning of the Runtime Global area (per task).
.ND1 .ND2	Temporary words used for emulating ECLIPSE SAVE and other instructions on the NOVA (per task).

Because DG/L procedures allocate memory dynamically, runtime routines and programs can be re-entrant and recursive. One or more tasks can use the same routine concurrently, with their pertinent values stored between executions. (See Figures 1-5 and 1-6.)

The user stack maintains two kinds of information. First, it stores values for variables (excluding OWN data); second, it maintains frames with information about all currently active routines. The user stack follows the logic of block structure for both kinds of information.

New variables go to the top of the stack (at .SP), with addresses increasing; references to variables access data at addresses below that in .SP. Variables for subroutines enter the stack at its top, and are released when the subroutine finishes execution. The stack maintains block structure by adding the data for the subroutine on top of the calling routine's data, then returning down to the calling routine's data when the subroutine's data is released.

The stack stores information about the program's execution on the same principle. Each routine or subprogram maintains a stack frame. The frame, a block of five contiguous locations, holds information needed for the current subprogram and the state of the calling program.

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 system releases its stack frame and returns control to the address stored in the frame. If a subroutine needs data declared in an enclosing procedure, it acquires the data through the enclosing procedure's frame pointer.

Non-Stack Storage

DG/L maintains a *data heap* toward the top of memory, beginning under the Process Global Table. The heap does not reflect block structure; its data are global to the entire program, and remain in memory from the time of declaration until the program completes execution. All OWN arrays and strings go into the heap, with the system allocating new data below the lowest occupied address.

DG/L allocates *Input/Output Control Blocks* (IOCBs) just below the heap. The IOCB contains information needed for formatted READ or WRITE commands. Unlike OWN data, these blocks are released once the formatted WRITE is complete. IOCBs also reflect block structure; if a formatted WRITE statement calls a subroutine which executes a formatted WRITE statement, the subroutine call generates a second IOCB. This block supersedes the first IOCB until its WRITE operation is done. To protect the downward-growing heap from a stack overflow, DG/L maintains a blank area called the Endzone between the stack and heap.

Transferring Control to a Runtime Routine

When a program calls a runtime routine or user-written procedure, the program's execution temporarily suspends. In order to later resume the program's execution, the call must preserve information current at the time of the call.

A call to a runtime routine starts a series of operations. The compiler loads the contents of .SP into AC2 and stores the addresses of the call's arguments at the top of the stack, in reverse order.

Control then passes to the routine. All 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. The same command allocates stack space for the routine. Execution of the routine's code then begins.

When the routine completes execution, a R.T.N command restores the accumulators, releases the frame and resumes execution of the calling program at the point where the call was made.

If you are using routines in the DG/L runtime library, these operations will be invisible. However, if you want to insert an assembly language subroutine into a DG/L program, you will need to follow these steps in detail.

Writing a Runtime Routine

DG/L allows you to write your own runtime routines in assembly language and execute them as external procedures in a DG/L program.

To write a runtime routine you must know the assembly language for your DGC computer and the particulars of the DG/L runtime environment described in this chapter.

After creating a source file along the lines described below, assemble it using MAC. Your routine's filename will then have an .RB extension.

Create an executable program using RLDR, which creates a file with an .SV extension, as in the following example:

```
RLDR DGL_PROG ASSEM_PROG @DGLIB@
```

See Chapter 11 of the *DG/L Reference Manual* for a detailed explanation of compiling and binding procedures.

Your own runtime routine should follow the form of those in the DG/L runtime library. For a routine called "MYFUNC," for example, the call format would be one of these two:

MYFUNC *[(arguments)];*

or

identifier := MYFUNC *[(arguments)]*

where the identifier has the same data type as the value returned. In your program, declare and call your own runtime routine in the same format as you would a DG/L routine:

```
EXTERNAL PROCEDURE MYFUNC;
```

or

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

Calling Your Runtime Routine

The call to your runtime routine must follow the format and rules that apply to DG/L routines. Like a call to a runtime library routine, it begins a series of operations:

1. Pushing the addresses of arguments onto the stack.
2. Transferring control to the routine.
3. Allocating a stack frame for temporary storage.
4. Preserving information about the calling program or routine.
5. Executing the routine.
6. Releasing the frame and its storage space.
7. Returning to the calling program following the pointers stored in the released stack frame.

Runtime routines can be recursive, re-entrant, and nested. 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.

When you call your own assembled routine, DG/L will perform operations 1 and 2 listed above. However, your routine must include the macros or instructions ENTRY, SAVE, and R.T.N in its code. For example, in:

```
ENTRY MYFUNC

        SAVE n
        .
        .
        .
        R.T.N.
```

SAVE performs steps 3 and 4 above; and R.T.N, performs steps 6 and 7.

The DG/L compiler transfers control to your routine after:

1. Loading the contents of the current .SP to AC2.
2. Pushing the addresses of all the arguments onto the stack, in reverse order.

The arguments for a subroutine are numbered left to right (as they appear in the call), starting at 0. For example, in a call to SUBR:

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

A is ARG0, B is ARG1, C is ARG2, and D is ARG3.

In a call to FUNC,

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

the same numbering applies; A is ARG0, etc.

Loading and Pushing Arguments

If your call has any arguments, the compiler pushes them in reverse order onto the stack. The compiler loads the argument's address into an accumulator. Then the PSH instruction increments the stack pointer and stores the address in the stack pointer's new position.

After pushing the arguments, the compiler loads ARG0 and ARG1 into AC1 and AC0, and transfers control to the routine.

Figure 1-5 shows the stack and accumulators when control goes from the compiler to the routine.

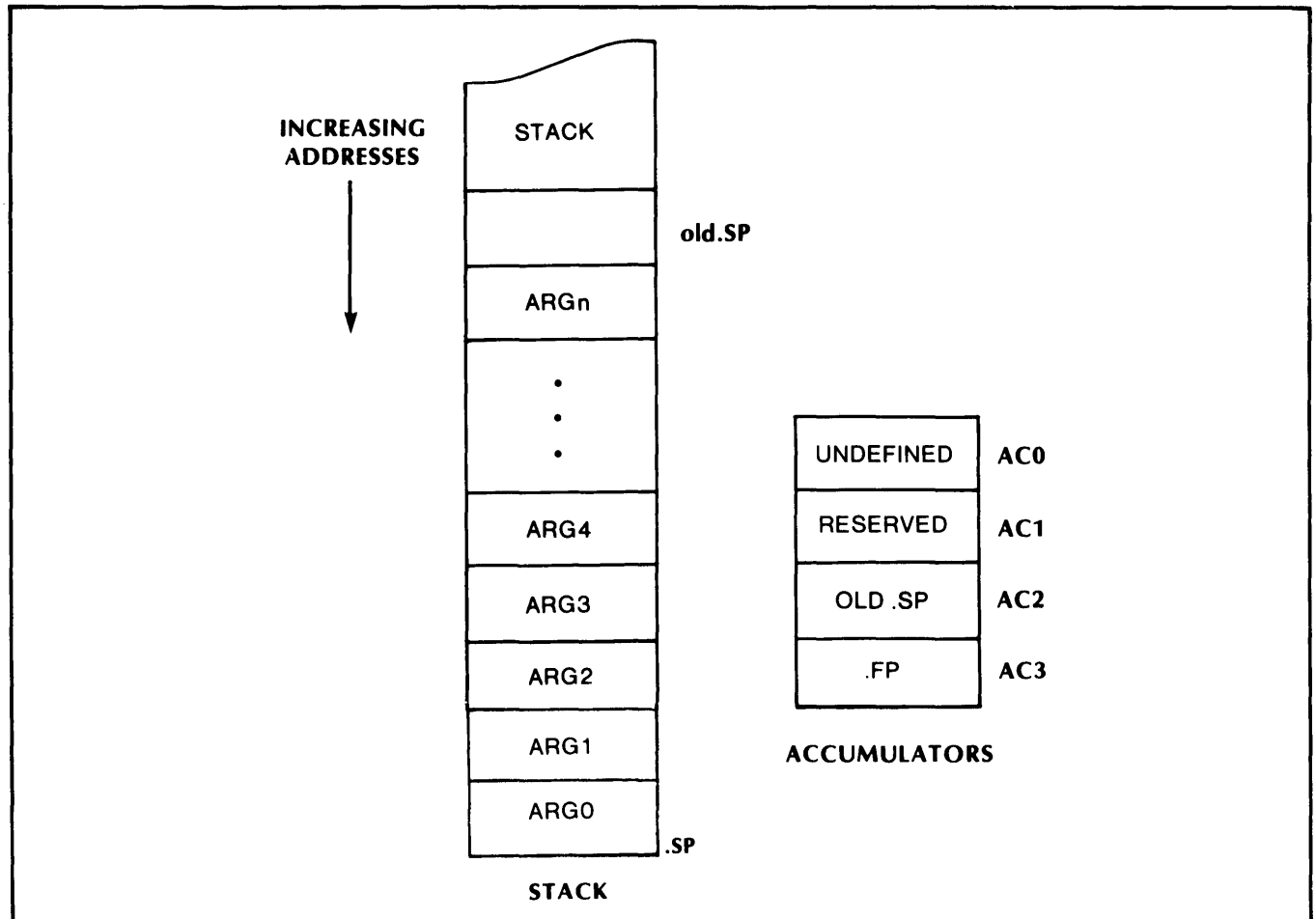


Figure 1-5. Stack Frame and Accumulators after Call to Routine

ENTRY

Your routine must use the macro ENTRY to define its entry points. ENTRY creates a label with the routine's name as the routine's first instruction. It also generates an .ENT for that name which enables a DG/L program to reference the name.

SAVE n

The SAVE instruction (SAVE macro on the NOVA) creates a stack frame which stores information about the calling task, provides storage for the routine's data manipulations, and updates the page zero pointers .SP (stack pointer) and .FP (frame pointer).

This stack frame consists of the contents of the accumulators, the calling program's .FP, the carry, and the return address.

Once the old .FP 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 updates .SP to point to the last location in the stack. The total space occupied will be 5 + n words, where n is the number of words requested for temporary storage. Figure 1-6 shows the stack after a SAVE instruction.

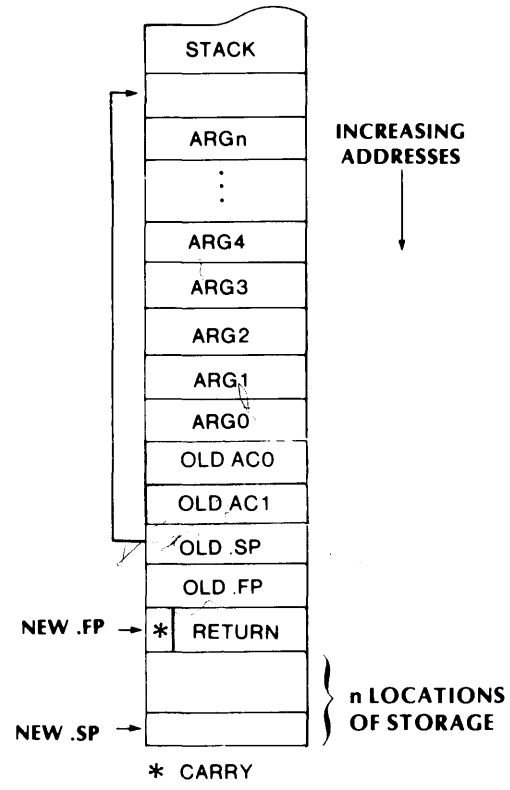


Figure 1-6. Stack Frame and Storage after SAVE n

Reference to Arguments

Once the arguments have been stored and the frame pointer updated, the routine can determine how many arguments it has received by computing the difference between AC2 (the old stack pointer) and AC3 (the new frame pointer). The difference equals the number of arguments plus five.

Included with the DG/L compiler and runtime libraries is a file DGLSYM.SR that defines the addresses of the arguments symbolically. You can address them left to right as ARG0, ARG1, ARG2, etc.; the file locates them as negative displacements from .FP.

Your assembly language code can thus refer to the arguments in this form:

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

This instruction will load the address of the fourth argument into AC2; .FP, which is in AC3, is the point of reference. The instruction

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

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

Temporary Storage Locations

The numerical argument n in the SAVE command is the total number of words required for temporary storage. The subroutine's stack begins one word past the word pointed to by .FP and extends upward in memory for n words.

To determine the number of words you need to store each data type in your routine, see the tables in "External Variables and the Internal Structure of Data" at the end of this chapter.

Temporary Storage: Making a Routine Re-entrant

Your own runtime routine should, like those in the DG/L runtime library, allow for recursion. If it will be called in a multitasking environment, it should also allow re-entrance.

Your routine needs to allocate temporary data areas for separate, concurrent executions of recursive and multitask operations. When you place the SAVE command at its beginning, the routine will dynamically allocate a new storage area for each separate activation. Also, space for variables will not occupy memory when the routine is not currently active.

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.

You can assign symbolic names to data addresses as displacements from the frame. For example:

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

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

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

Once you have created 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 it, you could use the following sequence:

```
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 (NOVA macros) PSH, POP, and MSP.

Error Handling in Your Runtime Routine

Your assembly language routine, like DG/L routines, can use the DG/L error handler, .RTER. The error handler can process fatal and non-fatal errors within the subroutine, and either continue the routine's execution or transfer control to a statement label in the calling program.

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

You may also use the error handler 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, or to correct the error condition and make the call again.

The Error Code

The runtime routine .RTER can accept operating system, DG/L, or user-defined error codes. When such an error occurs, its code goes into one accumulator, ERAC. (See the file DGLSYM.SR for definitions of ERAC.) Another accumulator, AC1, receives the error label code.

Your routine must load DG/L and system codes into the ERAC. For errors you define, you must put the word address of the error message into ERAC.

The Error Label

If your routine's call provides an error label as an argument, you must load a label indication 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 possible.
-2	ARG0 is the error label.
-1	ARG1 is the error label.
0	label is present, there is an even number of arguments.
1	If label is present, there is an odd number of arguments.
n	If label is present, it is argument n (counting from 0).

The last three codes apply 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 error code will go into the appropriate accumulator. The *RDOS Reference Manual* defines the error accumulator for system calls.

Examples

If you defined MYFUNC with no error label, the routine might use the sequence in Figure 1-7 for an orderly termination:

The series of instructions in that figure will print out an error message provided by the subroutine in ERAC, then transfer control to .RTER, which will terminate the program.

If you defined MYFUNC with an error label, the call might look like this:

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

If MYFUNC makes a system call, the exception return should branch to a label .ERET; see Figure 1-8.

This sequence passes control to .RTER, loading the error label indicator into AC1. The error handler will return control to the program at ERR1.

Like those you define using DG/L routines, the error label might read the error code, process and terminate under ERRINTERCEPT, process without termination with ERRTRAP, or branch elsewhere in the program.

Terminating the Routine

The R.T.N macro terminates execution of a subroutine and transfers control back to the calling program. It returns to the program in the following steps:

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

```
##
            .TXTM    1
            JSR      ERET
            .TXT     "MESSAGE"
 ERET:      MOV      3, 0            ; ERAC - ERROR MSG
            SUBZR    1,1             ; NO POSSIBLE ERROR LABEL
            RT.ERR   ECTUTIECSFP     ; ERROR RETURN
##
```

Figure 1-7. User-written Routine without Error Label

```
.ERET:  SUBZL  1,1          ; AC1 <-1 IMPLYING ODD # OF ARGS
        RT.ERR              ; ERROR RETURN
```

Figure 1-8. User-written Routine with Error Label

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

The compiler generates code to store the contents of AC2 into .SP, thus freeing all of the space occupied by the pushed arguments. Its code (on the ECLIPSE) is as follows:

```
LDA   2,.SP
push the arguments
EJSR  name
STA   2,.SP
```

Including Your Runtime Routine in a DG/L Program

To include your assembled routine in a DG/L program, enter it in the command line for RLDR. Chapter 11 of the *DG/L Reference Manual* describes binding an executable program.

Using DG/L Runtime in an Assembly Language Program

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 routine will follow the sequence described in "Calling Your Runtime Routine" earlier in this chapter. However, your assembly code must perform the stack maintenance operations that DG/L provides in a compiled DG/L program.

Assembling and Binding

Assemble and bind the subroutine as you would any DG/L program.

Calling DG/L Subroutines 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 and push arguments following these steps:

1. Load the current value of the stack pointer into AC2.

2. In reverse order, push the addresses of all the arguments onto the stack.

Call the DG/L subroutine in this form (on both NOVA and ECLIPSE):

```
NCALL name
```

Passing Arguments

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

```
ELEF    0,ARGn
ELEF    1,ARGn-1
PSH     0,1
```

Repeat this process as many times as necessary.

The example in Figure 1-9 demonstrates an ECLIPSE call to BLKWRITE. The last instruction in this figure restores the stack pointer to its initial value, releasing the stack frame and temporary storage created for the routine.

```
START:    ZLDA   2,.SP          ; LOAD .SP INTO AC2
          ELEF   0,ARG3         ; LOAD ADDRESS OF LAST ARG
          ELEF   1,ARG2         ; LOAD ADDRESS OF THIRD ARG
          PSH    0,1            ; PUSH ADDRESSES OF LAST
                              ; AND THIRD ARGS ONTO STACK

          ELEF   0,ARG1         ; LOAD ADDRESS OF ARG1
          ELEF   1,ARG0         ; LOAD ADDRESS OF ARG0

          EJSR   BLKWRITE       ; CALL THE SUBROUTINE

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

Figure 1-9. Call to BLKWRITE from an Assembly Language Program

```
##
  START:    LDA    2,.SP
            ELEF   1,MEM,3          ; ADDRESS FOR RETURNED VALUE
            PSH    1,1
            EJSR   MEMORY
  END:      STA    2,.SP
##
```

Figure 1-10. Call to MEMORY from an Assembly Language Program

The example in Figure 1-10, a call to MEMORY, shows a routine with a single argument (a returned value).

EXTERNAL Data and the Internal Structure of Data

If you need to use assembly language in a DG/L program, you must know how DG/L stores and references data. For any assembly language module you create, use the conventions outlined in this section.

EXTERNAL Data

If you have two, separately compiled DG/L programs that will refer to the same global data, you must create external storage for that data. You do this in two stages: by declaring variables as EXTERNAL in each program, and by defining them in an assembly file. To initialize an EXTERNAL variable with a value, you must define it with that value in the assembly file.

To create an executable file, you will need to create a source file defining the variable, assemble it using MAC, and load it with the rest of the program using RLDR. The loaded file's name will have an .SV extension. You must also declare this variable EXTERNAL in your program. The table in the following section gives the assembly-level definitions of each type of variable.

Internal Structure of Data

You will need the information in this section if you are using assembly code in connection with DG/L runtime. To create an EXTERNAL variable for a DG/L program or for storage in a runtime routine you have written, or to reference DG/L-defined data, you will need to allocate and address data following these conventions.

You 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

STRING VARIABLES: bit and character strings have a stack location defined as the first address of a three-word entity, called the string specifier, where:

Word 0	is the byte pointer to 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

SUBSTRINGS also have three-word specifiers:

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

You address *ARRAYS* of all types 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. An entity of 2n + 1 words (n being the number of dimensions) precedes the block.

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

Word	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

String arrays comprise arrays of three-word string specifiers, each pointing to and giving the size of the string.

Examples

The example in Figure 1-11 shows the assembler code for external variables with explanatory comments. The code must include entry labels and allocate a block of the appropriate size.

```
                .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
                MAXSIZE = 10        ; INITIALIZE MAXSIZE
                CURSIZE = 0         ; CURRENTLY EMPTY
STR1:           .STR*2              ; BYTE POINTER
                MAXSIZE             ; MAXIMUM SIZE
                CURSIZE             ; CURRENT SIZE
.STR            .BLK (MAXSIZE+1)/2      ; RESERVE WORDS
```

Figure 1-11. Assembler Code for External Variables

End of Chapter

Chapter 2
General User Routines

ADDRESS

Obtains the address of a referenced 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.

Error Conditions

A non-fatal 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.

Example

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

ALLOCATE

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 an integer expression in the range 1-32,767 inclusive 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.

Error Conditions

The following error codes may be returned:

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

Example

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

Notes

Allocation follows an exact-fit method; if there is no block of the specified size on the free chain, the block will be created from the top of the stack, lowering the stack limit. The call reserves size + 1 words. The word offset -1 from the pointer contains the length of the area.

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.

Error Conditions

No error condition can occur.

Example

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

BYTE (or ASCII)

Obtains the integer value of a byte in a datum.

Format

```
i := BYTE (datum,byte)
```

Arguments

i is an integer variable that receives the integer value of the byte.

datum 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 datum. The first byte of a string is 1.

Error Conditions

No error condition can occur.

Example

```
LBYTE := BYTE (J,1);
RBYTE := BYTE (J,2);
```

Notes

Giving byte a value outside the string will not produce an error, but the retrieval will not terminate.

CLASSIFY

Obtains an integer 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.

Rules

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

Word 0	lower limit of range.
Word 1	upper limit of range.
Word 2	value for range to be returned.

The last entry should be:

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

Error Conditions

No error condition can occur.

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 will continue searching through memory until a range is found.

FREE

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

Format

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

Arguments

pointer is a pointer expression that points to the beginning of a block of words previously reserved by a call to ALLOCATE.

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

Error Conditions

The following error code may be returned:

AICOR	Attempt to free an unallocated or already freed block.

Example

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

Notes

The freed block merges with any contiguous free block. It merges with the top of the stack if no free block is contiguous; otherwise the block is added to the free chain of freed blocks.

GETLIST

Obtains the name of the current list file.

Format

```
s := GETLIST
```

Arguments

s is a string variable that receives the list filename.

Error Conditions

No error condition can occur.

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.

MEMORY

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.

Error Conditions

No error condition can occur.

Examples

```
ISIZE := MEMORY;

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

Notes

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

RANDOM

Obtains a random number from a pseudo-random sequence of integers in the range of 0 to (2 ↑ 16)-1.

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 uses a method which generates a linear-congruential sequence:

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

Where

$A = 5 + 2 \uparrow 11$
$C = 33031R8$
X_i = the ith random number.

If you want to provide the initial number on which to base the random number generation, use the SEED routine instead. SEED can also provide a non-constant default value.

RDSW

Reads the CPU console data switches.

Format

i := RDSW *[(error label]*

Arguments

i is an integer variable that receives the configuration of the switches.

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

Error Conditions

Under RDOS, no error condition can occur.

RDSW does not exist in RTOS; the call will return the following error:

ERICM Illegal system command.

Example

```
ISWICH := RDSW (IERR);
```

Reference

.RDSW (System call)

SEED

Provides an initial number on which to base random number generation.

Format

SEED *[(number)]* ;

Arguments

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

Error Conditions

No error condition can occur.

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 do not issue a call to SEED but call RANDOM directly, a fixed initial number of 52352R8 is used. This method assures repeatability of data.

References

.GTOD (System call)
.GDAY (System call)

SYSTEM

Provides a generalized interface to the system, allowing you to use RDOS 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 OSID.SR.

AC0, AC1, AC2 are integer variables which specify the values of the accumulators when 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.

Error Conditions

Errors depend on the system call you are executing.

Example

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

Notes

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

Reference

OSID.SR -- *RDOS Reference 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 the integer variables or parenthesized constants whose values are required for the instruction.

carry is the desired 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.

Error Conditions

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

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 are stored back into the argument locations after the instruction is executed. These values may be expressions or parenthesized constants, which exist as temporaries on the stack and will not create any permanent effects if changed. They may also be variables, which do not exist as temporaries.

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 the integer variables or parenthesized constants whose values are required for the instruction.

carry is the desired 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.

Error Conditions

Errors depend on the instruction you are executing.

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 are stored back into the argument locations after the instruction is executed. These values may be expressions or parenthesized constants, which exist as temporaries on the stack and will not create any permanent effects if changed. They may also be variables, which do not exist as temporaries.

End of Chapter

Chapter 3
Performing Mathematical Operations with Integers and Words

REM

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.

Error Conditions

No error condition can occur.

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 - 1.

ROTATE

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 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.

Error Conditions

No error condition can occur.

Examples

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


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

SHIFT

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 variable that contains the word.

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.

Error Conditions

No error condition can occur.

Examples

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


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

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

UMUL

Multiplies two unsigned integers, 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 expession 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 of the product.

product is an integer variable that receives the result of the multiplication.

Error Conditions

No error condition can occur.

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)
```

End of Chapter

Chapter 4
Bit and String Manipulation

CBIT

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.

Error Conditions

No error condition can occur.

Example

```
SETCURRENT (BITX,0);
CBIT (BITX,SIZE(BITX));
    /*WILL CLEAR ALL BITS TO ZERO*/
```

Notes

A call to CBIT which gives an index outside the 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.

INDEX

Searches for a pattern of characters or bits in a 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 of the pattern. Bits are numbered from left to right, starting with 1.

string is a bit or character string expression that specifies the string to search in.

pattern is a string expression that specifies the characters to search for.

Error Conditions

No error condition can occur.

Example

```
IIND := INDEX (STR,"ING");
```

Notes

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

LENGTH

Obtains the current length of a string variable.

Format

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

Arguments

i is an integer variable that receives the current bound.

string is a character or bit string expression.

Error Conditions

No error condition can occur.

Example

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

Notes

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

SBIT

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.

Error Conditions

No error condition can occur.

Example

```
SBIT (I,3);
```

Notes

A call to SBIT which gives an index outside the maximum length of the bit string is not performed, but returns no 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.

SETCURRENT

Sets the length of a string or bit 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.

Error Conditions

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

Example

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

SIZE

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.

Error Conditions

No error condition can occur.

Example

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

Reference

.ASIZE (Internal call)

SUBSTR

Obtains a substring of a character or bit string, a substring, or an integer.

Format

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

Arguments

s is a string or bit variable that receives the substring.

expr is a character string, bit string, or integer expression.

starting index is an integer expression that specifies the position of the first character you want to return (one is the leftmost).

terminating index is an integer expression that specifies the position of the last character you want to return. If you do not specify a terminating index, the system assumes that the terminating index is the same as the starting index, and s will receive one character.

Error Conditions

The following error condition can occur:

AISOV	String overflow (if /T global compiler switch is used).

Example

```
SUBS := SUBSTR(STRING[N],3,5);
```

Notes

DG/L calls SUBSTR if expr is a character string and .USUBSTR if it is a bit string; otherwise, DG/L calls .ISUBSTR.

TBIT

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.

Error Conditions

No error condition can occur.

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

A call to TBIT which gives an index outside the legal range will return a value of False.

End of Chapter

Chapter 5
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.

Error Conditions

The following error code may be returned:

AISUB Subscript out of bounds.

Example

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

HBOUND

Obtains the upper bound of an array dimension.

Format

i := HBOUND (array name,dimension)

Arguments

i is an integer variable that receives the upper bound.

array name is an identifier 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.

Error Conditions

The following error code may be returned:

AIBND Illegal bound specification.

Example

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

LBOUND

Obtains the lower bound of an array dimension.

Format

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

Arguments

i is an integer variable that receives the lower bound.

array name is an identifier 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.

Error Conditions

The following error code may be returned:

AIBND Illegal bound specification.

Example

```
ILOWER := LBOUND(IARRAY 10,3);
```

SIZE

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.

Error Conditions

No error condition can occur.

Example

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

Reference

.ASIZE (internal call)

End of Chapter

Chapter 6
Command Line Handling

COMARG

Reads command line operands and switches.

Format

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

Arguments

file number is an integerexpression whose value is to be associated with the file in COM.CM format.

string is a string variable of up to 133 characters which 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.

Rules

COM.CM must be open before you call COMARG.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
EREOF	End of file.
ERRPR	Attempt to read a read-protected file.
ERFOP	Attempt to reference a file not opened.
ERLLI	Line limit exceeded (132 non-terminator characters).
ERPAR	Parity error.
ERFIL	File read error.
ERRD	Attempt to read into system area.
ERDIO	File accessible by direct block I/O only.
ERSIM	Simultaneous reads on same QTY line.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERMCA	Attempt to perform an MCA read on a channel where reading currently in progress.
ERCLO	QTY/MCA input terminated by channel close.

Example

Assume you typed:

```
MYPROG/C JOE/D HARRY
```

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

```
OPEN (0,"COM.CM");
.
.
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 JOE and ISWITCH receives the value of switch D. In the third call, ITEXT receives the string HARRY 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.

References

See Appendix C of the *RDOS Command Line Interpreter Reference Manual* for information on the command file COM.CM.

.RDL	(System call)
.RDS	(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 value associated with the file.

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

Rules

COM.CM must be open before you call RCOMARG.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	Attempt to reference an unopened unit.
ERSCP	File position error.

Example

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

Reference

.SPOS (System call)

End of Chapter

Chapter 7
Using the System Clock

GTIME

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

Error Conditions

No error condition can occur.

Example

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

References

.GDAY	(System call)
.GTOD	(System call)

STIME

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

Error Conditions

The following error condition can occur:

ERTIM	Illegal time of day.

Example

See Figure 7-1.

References

.SDAY	(System call)
.STOD	(System call)

```
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*/
```

Figure 7-1. Call to STIME

End of Chapter

Chapter 8
Handling Errors

ERETURN

Terminates a program and indicates an error.

Format

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

Arguments

error is an integer variable that contains one of the RDOS error status codes. See the *RDOS Reference Manual;* typically, it's a code returned by READERROR.

error label is a statement label in the higher program to which control transfers if the call to ERETURN cannot be executed.

Rules

Upon execution of a call to ERETURN, the system makes an unconditional return to the next higher level program. If the next higher level program is the CLI, one of the following occurs:

1. If the error status code is an RDOS error code, the appropriate message is displayed.
2. If a null error occurs (ERNUL), the CLI does not report a message.
3. If the error EREXQ is returned, the CLI takes the next command from disk file CLI.CM.
4. If you specify one of your own error status codes, i.e., if the CLI does not recognize the code, UNKNOWN ERROR CODE n is displayed, where n is your error status code in octal.

Error Conditions

No error condition can occur.

Example

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

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

Notes

This call is also documented in Chapter 15.

References

.ERTN (System call)

RDOS Reference Manual, "Error Message Reporting"

See the *RDOS Reference Manual* for a table of all RDOS error codes; see Appendix E of this manual for a list of all DG/L error codes.

ERPRINT

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

Format

ERPRINT;

Arguments

None.

Error Conditions

No error condition can occur.

Example

See Figure 8-1.

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	Non-fatal
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 RDOS or DG/L error codes.

```
TRY_AGAIN:    OPEN(0,"DATA1",OPNERR);
                .
                .
                .
OPNERR:        IF READERROR = ER$DLE  /*IF FILE DOES NOT EXIST*/
                  THEN ERPRINT
                  ELSE BEGIN
                         CREATE ("DATA1");  /*CREATE IT SEQUENTIAL*/
                         GO TO TRY_AGAIN;
                         END;
```

Figure 8-1. Call to ERPRINT

Rules

If the error was posted, ERRFATAL prints a full message.

Error Conditions

No error condition can occur.

Example

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

Reference

.BREAK (System call)

.ERTN (System call)

See the *RDOS Reference Manual* for a table of all RDOS error codes; see Appendix E of this manual for a list of DG/L error codes.

ERRINTERCEPT

Intercepts errors and calls an error handling routine.

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 integer 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	Meaning
Printing:	0B0	Suppress printing of error message
	1B0	Print error message before handling
Severity:	1B1	Handle errors fatal to process
	1B2	Handle errors fatal to task
	1B3	Handle non-fatal errors
Type:	1B4	Handle system errors
	1B5	Handle user error codes
	1B6	Handle user error text
	1B7	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.

ERRINTERCEPT (continued)

Rules

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 (error code,datum); procedure body
```

where error code and datum will be passed by ERRINTERCEPT.

Error Conditions

No error condition can occur.

Examples

See Figure 8-2.

Notes

ERRINTERCEPT without arguments removes any active ERRINTERCEPT.

After the procedure executes, control returns to the place where a non-fatal error occurred. If the error was fatal, the task or process will be killed.

ERRKILL

Prints an error message and kill 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	Non-fatal
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 RDOS or DG/L error code.

Error Conditions

No error condition can occur.

Example

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

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

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

Figure 8-2. Calls to ERRINTERCEPT

Notes

If no stack is available the following message is printed:

error number ERROR KILLED task number

where *task number* is 0 in single-task environment.

Otherwise, a full error message is printed.

Reference

See the *RDOS Reference Manual* for a table of all RDOS error codes; see Appendix E of this manual for a list of all DG/L error codes.

ERROR

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

Format

```
ERROR (message);
```

Arguments

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

Error Conditions

No error condition can occur.

Example

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

ERRTRAP

Traps errors without returning to the system.

Format

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

Arguments

error label is a statement label to which control transfers.

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.

	Code	Meaning
Printing:	0B0	Suppress printing of error message
	1B0	Print error message before handling
Severity:	1B1	Handle errors fatal to process
	1B2	Handle errors fatal to task
	1B3	Handle non fatal errors
Type:	1B4	Handle system errors
	1B5	Handle user error codes
	1B6	Handle user error text
	1B7	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.	

Rules

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

ERRTRAP is block structured, so you may redefine traps at lower block levels, or replace traps on the same block level.

Error Conditions

No error condition can occur.

Example

```
ERRTRAP (EOF,74006); /*TRAP EOF'S*/
```

Notes

ERRTRAP called without an argument removes any active ERRTRAP.

If the statement label is at a higher level block than the ERRTRAP, all ERRTRAPs at the same and intervening levels will be released. If they are on the same level, the trap is retained.

ERRUSER

Calls the error handler with an error code.

Format

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

Arguments

error code is an integer variable that receives a 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	Non-fatal
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 RDOS or DG/L error code.

error label is a statement label to which control is transferred.

Error Conditions

No error condition can occur.

Example

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

Notes

This call uses .RTER, the DG/L error handler, as a runtime routine.

Reference

See the *RDOS Reference Manual* for a table of all RDOS error codes; see Appendix E of this manual for a list of all DG/L error codes.

FPUERROR

Returns boolean and string values for FPU errors.

Formats

```
bool := FPUMANTISSA

bool := FPUUNDERFLOW

bool := FPUOVERFLOW

bool := FPUZDIVIDE

i:= FPUERROR
```

Arguments

bool is a boolean variable that can be set to check for each type of floating-point error. It returns true if an error has occured.

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

AEMAN	Mantissa overflow
AEUND	Underflow
AEOVF	Overflow
AEZDV	Zero divide

See Appendix E and ERRFATAL for the numeric codes.

Error Conditions

No error condition can occur.

Example

```
IF (BOOL := FPUUNDERFLOW) THEN
```

Notes

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

NOFPUTRAP

Ignores floating point errors.

Format

EXTERNAL INTEGER NOFPUTRAP;

Arguments

None.

Error Conditions

No error condition can occur.

Notes

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

NOMESSAGE

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

Format

EXTERNAL INTEGER NOMESSAGE;

Arguments

None.

Error Conditions

No error condition can occur.

Notes

If you want to use error messages for debugging but to suppress them during execution, you can bind an assembly language module of this form into your program at load time:

```
.EXTN NOMESSAGE
```

This will save an extra compilation.

See also SHORTMESSAGE; your program cannot, however, declare both.

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 error code.

severity is an integer variable that receives a severity code:

0 Fatal to process
1 Fatal to task
2 Non-fatal 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 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 program 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 will set the error code to -1. Any other value will prevent clearing.

Error Conditions

No error condition can occur.

Example

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

Reference

See the *RDOS Reference Manual* for a table of all RDOS error codes; see Appendix E of this manual for a list of all DG/L error codes.

SHORTMESSAGE

Is a declaration for printing out short error messages.

Format

```
EXTERNAL INTEGER SHORTMESSAGE;
```

Arguments

None.

Error Conditions

No error condition can occur.

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	Non-fatal
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

SHORTMESSAGE (continued)

Bits 8-15 (the last three octal places) contain the RDOS 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 bind in with your program after debugging.

See also NOMESSAGE; your program cannot, however, declare both.

Reference

See the *RDOS Reference Manual* for a table of all RDOS error codes; see Appendix E of this manual for a list of all DG/L error codes.

End of Chapter

Chapter 9
Managing Directories and Devices

CDIR

Creates a subdirectory.

Format

CDIR (subdirectory name *[,error label]);*

Arguments

subdirectory name is a string expression that specifies the name of the subdirectory you want to create.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal subdirectory name.
ERCRE	Attempt to create an existing subdirectory.
ERDSN	Directory specifier unknown.
ERDDE	Attempt to create a subdirectory in a subdirectory.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

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

Reference

.CDIR (System call)

CPART

Creates a secondary partition.

Format

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

Arguments

partition name is a string expression that specifies the name of the secondary partition.

size is an integer that specifies the number of contiguous disk blocks to allocate to the secondary partition.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal secondary partition name.
ERCRE	Attempt to create an existing secondary partition.
ERICB	Insufficient number of free contiguous disk blocks available.
ERDSN	Directory specifier unknown.
ERD2S	Directory too small (must have at least 60 (octal) blocks).
ERDDE	Attempt to create a secondary partition in a secondary partition (i.e., a tertiary partition).
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

CPART (continued)

Example

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

Notes

The minimum allocation is 60R8 blocks, with size rounded down to the nearest multiple of 20R8.

Reference

.CPAR (System call)

DIR

Changes the current default directory.

Format

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

Arguments

directory name is a string expression that specifies the name of the new default directory.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERDLE	Directory doesn't exist.
ERDNM	Device or directory not in system.
ERNMD	Insufficient number of extra buffers specified at SYSGEN time.
ERIDS	Illegal directory specifier.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
EROVF	System stack overflow due to excessive number of chained directory specifiers.

Example

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

Reference

.DIR (System call)

EQUIV

Assigns a temporary name to a device, permitting unit independence during the execution of your program.

Format

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

Arguments

new name is a string expression that specifies the new, temporary name to a global specifier.

old name is a string that specifies the current name of the device.

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

Rules

You must issue a call to EQUIV before you initialize the device.

You cannot assign a temporary name to a master device.

Error Conditions

The following error codes may be returned:

ERDSN	Directory specifier unknown.
ERDIU	Device in use (i.e., already initialized).
ERMPR	Address is outside address space (mapped systems only).

Example

```
EQUIV ("QDISK","GETDIR",IERR);
```

Notes

Temporary name assignments persist until you issue a disk bootstrap, an initialization, a release, or a new temporary name assignment.

Reference

.EQIV	(System call)

GETDIR

Obtains the name of the current directory.

Format

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

Arguments

s is a string variable that receives the name of the current directory.

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

Error Conditions

The following error codes may be returned:

ERRD	Attempted system overwrite.
ERMPR	Address is outside address space (mapped systems only).

Example

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

Reference

.GDIR	(System call)

GETMDIR

Obtains the name of the current master device.

Format

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

Arguments

s is a string variable that receives the name of the current master device.

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

Error Conditions

The following error codes may be returned:

ERRD	Attempted system overwrite.
ERMPR	Address is outside address space (mapped systems only).

Example

```
DIR ((GETMDIR)); /*GO TO THE MASTER DIRECTORY*/
```

Reference

.MDIR (System call)

GETSYS

Obtains the current system name.

Format

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

Arguments

s is a string variable that receives the name of the current system.

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

Error Conditions

The following error codes may be returned:

ERRD	Attempted system overwrite.
ERMPR	Address is outside address space (mapped systems only).

Examples

```
SSYS := GETSYS (IERR);

BOOT((GETSYS)); /*REBOOT THE SYSTEM*/
```

Reference

.GSYS (System call)

INIT

Initializes a directory, partition, or device.

Format

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

Arguments

directory name is a string expression that specifies the name of the directory, partition, or device you want to initialize.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERWRP	Device is write-protected.
ERDLE	Directory doesn't exist.
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERIBS	Device already initialized.
ERNMD	Insufficient number of extra buffers specified at SYSGEN time.
ERIDS	Illegal directory specifier.
ERLDE	Link depth exceeded.
ERMPR	Address is outside address space (mapped systems only).
ERSDE	Error detected in SYS.DR of non-master device.
ERDTO	Ten second disk timeout occurred.
ERENA	No linking allowed (N attribute).
EROVF	System stack overflow due to excess number of chained directory specifiers; this can occur only when you use links in the specifier string.

Example

```
INIT ("MT0",IERR);
```

Reference

.INIT (System call)

RELEASE

Releases a directory, partition, or device from current use.

Format

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

Arguments

directory name is a string expression that names a directory, partition, or device you want to release.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERDIU	Directory in use.
ERDNI	Directory not initialized.
EROPD	Released directory in use by other program.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERNIR	Attempted release of a unit containing an open file.

Example

```
RELEASE ("DP3",IERR);
```

Reference

.RLSE (System call)

End of Chapter

Chapter 10
Maintaining Files

Chapters 10 and 11 describe runtime routines for handling files. The DG/L runtime libraries allow you to create random access, contiguous access, and sequential access files. It provides routines for reading files into memory, and writing data to disk and tape.

The type of file you create will depend on your programming needs. The relative importance of flexible access and sequential order will be especially important considerations, as will the size of the file you expect to call into memory.

All disk files store data in 256-word blocks. However, the way you may organize data within the blocks varies considerably.

A *random file* has a variable number of 256-word blocks. It gives you freedom for locating a specific block and datum without requiring the system to search through each entry from the file's beginning. Use the call CRAND to create a random file.

A *contiguous file* allows random access, but is limited to a fixed number of blocks. Its advantage is that you can access each of its blocks with a single disk read or write, because each block occupies a fixed location on the disk. Random files ordinarily require several accesses to locate a block because they have a tree structure of disk pointers. Contiguous loading allows quick access to blocks and data. Use CCONT and NCONT to create a contiguous file.

A *sequentially-ordered file* is most useful when the order of the data is an important part of its processing and the file size must be flexible. Your program can access data in your current block or in the preceding or subsequent block with a single disk access, but cannot jump to another part of the file without reading it sequentially. Use CREATE to create a sequential file.

ATTRIBUTE

Obtains the attributes of an opened file.

Format

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

Arguments

file number is an integer expression that specifies an opened file.

attributes is an integer variable that receives the file's attributes.

device characteristics is an integer variable that receives the file's device characteristics.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	Attempt to get attributes of an unopened file.
ERDTO	Ten second disk timeout occurred.

Example

```
INTEGER ATTS,DEVCH;
.
.
ATTRIBUTE (1,ATTS,DEVCH,IERR);
```

Notes

See the *RDOS Reference Manual* for tables of bit-to-attribute and bit-to-characteristic correspondences.

Reference

.GTATR (System call)

CCONT

Creates a contiguous file of a specified 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.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERCRE	Attempt to create an existing file.
ERSPC	Out of disk space.
ERICB	Insufficient number of free contiguous disk blocks available (contiguous file only).
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
CCONT ("FILNM.DC",2000, IERR);
```

Notes

See also CRAND, CREATE, and NCONT.

Reference

.CCONT (System call)

CHATR

Changes the resolution attributes of an opened file.

Format

CHATR (file number,attributes *[,error label]);*

Arguments

file number is an integer expression that specifies the file whose attributes are to be changed.

attributes is an integer that specifies the new attributes.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERCHA	Illegal attempt to change file attributes.
ERFOP	Attempt to change attributes of an unopened file.
ERDTO	Ten second disk timeout occurred.

Example

```
ATTRIBUTE (1,ATTS,DEVS);
.
.
.
CHATR(1,ATTS OR 10R2);
```

Notes

See the *RDOS Reference Manual* for a table of the bit-to-attribute correspondences.

Reference

.CHATR (System call)

CHLAT

Changes link access attributes of an opened file.

Format

CHLAT (file number,attributes *[,error label]);*

Arguments

file number is an integer expression that specifies the number of the link file whose attributes are to be changed.

attributes is an integer expression that specifies the new attributes.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERCHA	Resolution entry is attribute protected.
ERFOP	Attempt to change attributes of an unopened file.
ERDTO	Ten second disk timeout occurred.

Example

```
CHLAT (2,IATTS,IERR);
```

Notes

When you open a link entry, you will have the access attributes of the resolution file. Those you specify in attributes will be added to your present copy.

For a table of bit-to-attribute correspondences, see .CHATR in the *RDOS Reference Manual.*

Reference

.CHLAT (System call)

CHSTATUS

Obtains the current directory status for an opened file.

Format

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

Arguments

file number is an integer expression that specifies the file.

status is an array that receives the 18 words of status information. See the RDOS Reference Manual for its contents.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	No file opened on the given channel.
ERRD	Attempted system overwrite.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
INTEGER ARRAY ISTAT [1:18];
.
.
CHSTATUS (3,ISTAT,IERR);
```

References

.UPDAT (System call) - To update the current file size.

.CHSTS (System call)

CRAND

Creates a random file of zero length.

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.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERCRE	Attempt to create an existing file.
ERSPC	Out of disk space.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

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

Notes

See also CCONT, CREATE, and NCONT.

Reference

.CRAND (System call)

CREATE

Creates a sequential file of zero length.

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.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERCRE	Attempt to create an existing file.
ERSPC	Out of disk space.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

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

Notes

See also CCONT, CRAND, and NCONT.

Reference

.CREATE (System call)

DELETE

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:

ERFNM	Illegal filename.
ERDLE	Attempt to delete a non-existing file.
ERDE1	Attempt to delete a permanent file.
ERDSN	Directory specifier unknown.
ERDIU	Directory in use.
ERLDE	Link depth exceeded.
ERFIU	File in use.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERMDE	Error detected in MAP.DR of non-master device.
ERDTO	Ten second disk timeout occurred.
ERENA	No linking allowed (N attribute).

Example

```
DELETE ("MUNG.DC",IERR);
```

Notes

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

Reference

.DELET (System call)

FILESIZE

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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	File not opened.
ERRD	Attempted system overwrite.
ERMPR	Address outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

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

Reference

.UPDAT (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.

Error Conditions

The following error codes may be returned:

ERUFT No channels are free.

Example

See Figure 10-1.

Reference

.GCHN (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 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.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERCRE	Link entry name already exists.
ERSPC	Insufficient disk space to create SYS.DR entry.
ERDSN	Directory specifier unknown.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

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

Reference

.LINK (System call)

```
OPEN (LISTFILE := GCHANNEL (CANTOPEN),(GETLIST), CANTOPEN);
 •
 •
 •
WRITE (LISTFILE, "NL");
```

Figure 10-1. Call to GCHANNEL

NCONT

Creates a contiguous file of a specified length without initializing all words to zero.

Format

NCONT (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.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERCRE	Attempt to create an existing file.
ERSPC	Out of disk space.
ERICB	Insufficient number of free contiguous disk blocks available.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
NCONT ("FILNM.DC",2000, IERR);
```

Notes

See CCONT, CRAND, and CREATE.

Reference

.NCONT (System call)

RENAME

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.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERCRE	Attempt to create an existing file.
ERDLE	Attempt to rename a non-existing file.
ERDE1	Attempt to rename a permanent file.
ERDIR	Files specified on different directories.
ERDSN	Directory specifier unknown.
ERFIU	File in use.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

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

Reference

.RENAM (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 18 words of file directory information. See the *RDOS Reference Manual* for its contents.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERDLE	File doesn't exist.
ERRD	Attempted system overwrite.
ERDNM	Device not in system.
ERDSN	Directory specifier unknown.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
INTEGER ARRAY FILUFD [1:18];
.
.
STATUS ("IOLINK",FILUFD,IERR);
```

Notes

For an open file, STATUS returns information as of the opening. CHSTATUS returns information updated to the call.

Reference

.STAT (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.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERDLE	Attempt to delete a non-existing file.
ERDSN	Directory specifier unknown.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERNLE	Not a link entry.
ERDTO	Ten second disk timeout occurred.

Example

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

Reference

.ULNK (System call)

End of Chapter

Chapter 11
File Input/Output

Using data from storage files requires a series of runtime calls for opening the file, reading/writing data, and closing the file. DG/L offers five methods for transferring data to and from disk and tape files.

1. *Block I/O:* BLKREAD and BLKWRITE read and write multiples of 256 words at a time. You may use block I/O with any but a sequentially organized file.

   Open/close calls: OPEN, APPEND, EOPEN, ROPEN; CLOSE.

2. *Sequential, delimited I/O:* LINEREAD and LINEWRITE read data (usually character strings) 133 bytes at a time, or up to the first terminating character (new line, carriage return, form feed, or NUL).

   Open/close calls: OPEN, APPEND, EOPEN, ROPEN; CLOSE.

3. *Sequential, unlimited I/O:* BYTEREAD and BYTEWRITE transfer data in a specified number of bytes. This flexible method allows you to specify the number of bytes you want to read or write.

   Open/close calls: OPEN, APPEND, EOPEN, ROPEN; CLOSE.

4. *Sequential, buffered, I/O:* DATAREAD and DATAWRITE handle data records by words, and allow you to specify the number of words you want to read and write. The read and write calls transfer data by accumulating one 256-word block at a time, and transferring a final partial block if necessary. The block transfer makes it possible to later read the file by any I/O method.

   Open/close calls: DATAOPEN and DATACLOSE only. You must close and reopen the file to go from reading to writing or writing to reading.

5. *Free-form magnetic tape I/O:* MTDIO transfers data to and from a magnetic tape or cassette unit. You may write up to 4096 variable length records from 2 to 4096 words long. You do not need to specify the number of words. You can space as well as read and write within a record file.

   Open/close calls: MTOPD only; closing unnecessary.

For another method of handling file I/O, see Chapter 13, *Cache Memory Management.*

APPEND

Opens a file for appending, starting writing at its end.

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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFNM	Illegal filename.
ERICD	Illegal command for device.
ERDLE	File doesn't exist.
ERUFT	No channels are free (returned by .GCHN).
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERFIU	File in use.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERENA	No linking allowed (N attribute).
ERDOP	Attempt to open a file that is already open.

Example

```
APPEND (13,"MYFIL",IERR);
```

Notes

The call to APPEND is identical to the call to OPEN except that APPEND allows you to open and extend a file that already exists. Also, you open the file specifically for appending. In writing to a line printer, APPEND will not generate an initial form feed, but OPEN will.

Reference

. APPEND (System call)

BLKREAD

Reads a series of blocks from a randomly or contiguously organized disk file.

Format

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

Arguments

file number is an integer expression that specifies the number of the randomly or contiguously organized disk file you want to read.

starting block is an integer expression that specifies the block where you want to start reading. (The first block is numbered 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 blocks you want to 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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERRPR	File is read protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
ERRD	Attempt to read into system area.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

See Figure 11-1.

Reference

. RDB (System call)

```
##
POINTER  FILPT;
   .
   .
OPEN (7,DATA.OL);

   .
   .
BLKREAD (7,5,FILPT,SEGM,IERR);
COMMENT READ A SEGMENT OF BLOCKS MEMORY AT FILPT;
##
```

Figure 11-1. Call to BLKREAD

BLKWRITE

Writes a series of blocks to a randomly or contiguously organized disk file.

Format

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

Arguments

file number is an integer expression that specifies the number of the randomly or contiguously organized disk file you want to write to.

starting block is an integer expression that specifies the first block you want written. (The first block is numbered 0.)

pointer is a pointer expression that points to the first address in memory 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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERWPR	File is write-protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

See Figure 11-2.

Reference

. WRB (System call)

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

Figure 11-2. Call to BLKWRITE

BYTEREAD

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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
EREOF	End of file.
ERRPR	Attempt to read a read-protected file.
ERFOP	Attempt to reference an unopened unit.
ERFIL	File read error.
ERRD	Attempted read into system area.
ERDIO	File accessible by direct block I/O only.
ERSIM	Simultaneous reads on same QTY line.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERMCA	Attempt to perform an MCA read on a channel where reading is currently in progress.
ERCLO	QTY/MCA input terminated by channel close.

Example

```
POINTER IPNT;
.

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

Reference

. RDS (System call)

BYTEWRITE

Writes a specified number of bytes to a file.

Format

BYTEWRITE (file number,pointer,count *[errorlabel];*

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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERWPR	Attempt to write a write-protected file.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERDIO	File accessible by direct block I/O only.
ERSIM	Simultaneous writes to the same QTY line.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERMCA	Attempt to issue an MCA write to a channel on which writing is currently in progress.
ERSRR	Incomplete MCA transmissions due to short receiver request.
ERCLO	MCA/QTY output terminated by channel close.
ERNMC	No outstanding receive request.

Example

See Figure 11-3.

Reference

.WRS	(System call)

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

Figure 11-3. Call to BYTEWRITE

CLOSE

Closes a file.

Format

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

Arguments

file number is an integer expression that specifies the number of the file you want to close.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	Attempt to reference an unopened unit.
ERDTO	Ten second disk timeout occurred.

Example

```
CLOSE (7,IERR);
```

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 is assigned in the DATAOPEN call.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERWPR	Attempt to write a write-protected file.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERDIO	File accessible by direct block I/O only.
ERSIM	Simultaneous writes to the same QTY line.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERMCA	Attempt to issue an MCA write to a channel on which writing is currently in progress.
ERSRR	Incomplete MCA transmissions due to short receiver request.
ERCLO	MCA/QTY output terminated by channel close.
ERNMC	No outstanding receive request.

Example

```
DATACLOSE (5);
```

Notes

See DATAOPEN, DATAREAD, and DATAWRITE.

Reference

.WRS (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, a random file will be created.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFNM	Illegal filename.
ERUFT	No channels are free (returned by .GCHN).
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERFIU	File in use due to .EOPEN.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERDOP	Attempted open of a tape file which is already open.

Example

```
DATAOPEN (6, "NOCAN.DO");
```

Notes

See DATACLOSE, DATAREAD, and DATAWRITE.

References

.OPEN	(System call)
.CRAND	(System call)

DATAREAD

Reads data 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 number of the file you want to read.

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 receives the number of words actually read.

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

Rules

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:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSVI	Not a randomly or contiguously ordered file.
EREOF	End of file.
ERRPR	Attempt to read a read-protected file.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space exhausted.
ERFIL	File read error.
ERRD	Attempt to read into system area.
ERDIO	File accessible by direct block I/O only.
ERSIM	Simultaneous reads on same QTY line.
EROVA	File not accessibly by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERMCA	Attempt to perform an MCA read on a channel where reading is currently in progress.
ERCLO	QTY/MCA input terminated by channel close.
AISIZ	Illegal size specified ($<=0$).

DATAREAD (continued)

Example

```
DATAREAD (3,ADDRESS(DATAARRAY),5,EOF3);
```

References

.RDS	(System call)
.RDB	(System call)
.SPOS	(System call)

DATAWRITE

Writes 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 which you want to write.

count is an integer expression that specifies the number of bytes you want written. If unspecified, the count will be 1.

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

Rules

You must close and reopen a file with DATACLOSE and DATAOPEN between DATAREAD and DATAWRITE.

DATAWRITE writes 256-word blocks.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERWPR	Attempt to write a write-protected file.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERDIO	File accessible by direct block I/O only.
ERSIM	Simultaneous writes to the same QTY line.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERMCA	Attempt to issue an MCA write to a channel on which writing is currently in progress.
ERSRR	Incomplete MCA transmissions due to short receiver request.
ERCLO	MCA/QTY output terminated by channel close.
ERNMC	No outstanding receive request.
AISIZ	Illegal size specified ($<=0$).

Example

```
DATAWRITE (4,ADDRESS(AREA1),2);
```

References

.WRS (System call)

.WRB (System call)

.SPOS (System call)

EOPEN

Exclusively opens a file at its beginning.

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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFNM	Illegal filename.
ERSPC	File space exhausted.
ERUFT	No channels are free (returned by .GCHN).
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERFIU	File in use due to .EOPEN.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERENA	No linking allowed (n attribute).
ERDOP	Attempted open of a tape file which is already open.

Example

```
EOPEN (6, "NOCAN.DO");
```

Notes

If the named file does not exist, EOPEN creates a randomly organized file.

References

.OPEN (System call)

.GTATR (System call)

.CRAND (System call)

FILEPOSITION

Computes a position within a file.

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 two-word address.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	Attempt to reference an unopened file.

Example

```
INTEGER ARRAY POSTN [1:2];
.
.
FILEPOSITION (2,POSTN);
```

Reference

.GPOS (System call)

LINEREAD

Reads a line from a file into memory.

Format

LINEREAD (file number,address,count *[,error label]);*

Arguments

file number is an integer expression that specifies the number of the file you want to read.

address is a pointer expression that points to the first address of the core buffer. The buffer should be 133 characters.

count is an integer variable that specifies the number of bytes you want read. It can be up to 133 characters, including the terminator. If the system detects an error or an end-of-file condition, count receives the number of bytes actually read.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
EREOF	End of file.
ERRPR	Attempt to read a read-protected file.
ERFOP	Attempt to reference an unopened unit.
ERLLI	Line limit (132 non-terminator characters) exceeded.

```
INTEGER IBYTE;
   .
   .
LINEREAD (5,ADDRESS(ARAYD),IBYTE,IERR);
IF IBYTE = 1 GO TO NOINPUT;

COMMENT BRANCH IF TERMINATOR IS THE ONLY CHARACTER;
```

Figure 11-4. Call to LINEREAD

ERPAR	Parity error.
ERFIL	File read error.
ERRD	Attempt to read into system area.
ERDIO	File accessible by direct block I/O only.
ERSIM	Simultaneous reads on same QTY line.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERCLO	QTY/MCA input terminated by channel close.

Example

See Figure 11-4.

Reference

.RDL (System call)

LINEWRITE

Writes a line from memory to a file.

Format

```
LINEWRITE (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 you want to write out.

count is an integer variable that specifies the number of bytes you want written. It can be up to 133 characters including the terminator. If the system detects an error, count receives the number of bytes actually written.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
EREOF	End of file when writing to a contiguous file.
ERWRP	Attempt to write a write-protected file.
ERFOP	Attempt to reference an unopened unit.
ERLLI	Line limit (132 characters) exceeded.
ERSPC	Disk space is exhausted.
ERDIO	File accessible by direct block I/O only.
ERSIM	Simultaneous writes to same QTY line.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERCLO	QTY output terminated by channel close.

Example

```
LINEWRITE(4,ADDRESS("HI THERE"),8);

/*WRITE "HI THERE" WITH NO TERMINATOR*/
```

Reference

.WRL (System call)

MTDIO

Performs free-form I/O to or from a magnetic tape or cassette unit.

Format

MTDIO (file number,command,address,status word
[[count,]error label]);

Arguments

file number is an integer expression that specifies the tape opened for direct I/O.

command is an integer expression whose bit values specify the command the unit is to perform.

address is a pointer expression that points to the first address to transmit or receive data.

status word is an integer variable that receives a code describing the status of the hardware if the return is normal; however, if a software error occurs, this argument receives a zero.

count is an integer variable that specifies the number of records you want spaced, read, or written (depends on command); this count is returned upon a premature end-of-file, and returns undefined if there is a system error.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device (i.e., improper open).
ERFOP	Attempt to reference an unopened unit.
ERSEL	Unit improperly selected (check status word).
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).

Example

See Figure 11-5.

Notes

See the *RDOS Reference Manual* for a table of bit-to-command correspondences.

Reference

.MTDIO (System call)

```
INTEGER ARRAY ARAY1 [1:100];
  .
  .
ICOUNT := HBOUND (ARAY1,1);
  .
  .
  .

MTDIO (3,150144K,ADDRESS(ARAY1),ISTAT,ICOUNT,IERR);
  /*WRITE 100 WORDS WITH EVEN PARITY*/
```

Figure 11-5. Call to HBOUND

MTOPD

Opens a magnetic tape or cassette unit for free-form I/O.

Format

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

Arguments

file number is an integer expression whose value will be associated with the file.

filename is a string expression that specifies the name of the file you want to open.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERDLE	File doesn't exist.
ERUFT	Attempt to use a channel already in use.
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDOP	Attempted open of a file that is already open.

Example

```
MTOPD (4,"MT0:3",IERR);
```

Reference

.MTOPD (System call)

OPEN

Opens a file.

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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFNM	Illegal filename.
ERDLE	File does not exist.
ERSPC	File space exhausted.
ERUFT	No channels are free (returned by .GCHN).
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERFIU	File in use due to .EOPEN.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERENA	No linking allowed (N attribute).
ERDOP	Attempted open of a tape file which is already open.

Example

```
OPEN (6,"MYFILE",IERR);
```

Notes

If the named file does not exist, a randomly organized file will be created.

References

.OPEN (System call)

.GTATR (System call)

.CRAND (System call)

POSITION

Repositions an open file.

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 another two-word entity with a two-word integer value that points to the byte where the file will be positioned.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	Attempt to reference an unopened unit.
ERSCP	File position error.

Example

```
INTEGER ARRAY POSTN [1:2];
.
.
POSITION (2,POSTN,IERR);
```

Notes

To reposition the file at the beginning, rewind it with the call

POSITION(file number, 0.0);.

The single-precision real will read as two consecutive words of zero data. You may use POSITION to restore a position saved by a call to FILEPOSITION.

Reference

.SPOS (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.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFNM	Illegal filename.
ERSPC	File space exhausted.
ERUFT	No channels are free (returned by .GCHN).
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERFIU	File in use due to .EOPEN.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERENA	No linking allowed (N attribute).
ERDOP	Attempted open of a tape file which is already open.

Example

```
ROPEN (6, "NOCAN.DO");
```

Notes

If the file does not exist, it will not be created.

References

.OPEN (System call)

.GTATR (System call)

End of Chapter

Chapter 12
Console Input/Output

.CONSOLE

Writes to the console without using the stack.

Format

JSR @.CONSOLE

Arguments

AC0 contains an integer that specifies the type of message you want to write:

0	Write a number plus two spaces.
-1	Write a line terminator.
not 0,-1	Write a string message.

AC1 contains the number if AC0 is set to zero.

AC2 contains the address of the string you want to write if the type is not zero or -1.

Error Conditions

No error condition can occur.

Example

```
        SUBZL 0,0
        JSR   STKOF
        .TXT  "STACK OVERFLOW"

STKOF:  MOV   3,2
        JSR   @.CONSOLE
```

Notes

This is an assembly language call only.

GETCHAR

Reads a single character from the operator's console without echoing it.

Format

i := GETCHAR *[(error label)]*

Arguments

i is an integer variable that receives the character from the console. The character is stored in ASCII format in the low byte, with zeroes in the high byte.

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

Error Conditions

The following error codes may be returned:

ERCID	Console not in system.

Example

```
ICHAR := GETCHAR (IERR);
```

Reference

.GCHAR	(System call)

GETCINPUT

Obtains the input console name.

Format

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

Arguments

s is a string variable that receives the name of the current input console. $TTI is background, $TTI1 is foreground console name.

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

Error Conditions

The following error codes may be returned:

ERRD	Attempt to read into a system area.
ERMPR	Address is outside address space (mapped systems only).

Example

```
STRING(6) CONNM;
.
.
CONNM := GETCINPUT (IERR);


OPEN (10,(GETCINPUT));

/* OPEN CONSOLE FOR READING */
```

Reference

.GCIN (System call)

GETCOUTPUT

Obtains the output console name, making a program portable.

Format

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

Arguments

s is a string variable that receives the name of the current output console. $TTO is background, $TTO1 is foreground.

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

Error Conditions

The following error codes may be returned:

ERRD	Attempt to read into a system area.
ERMPR	Address is outside address space (mapped systems only).

Example

```
OPEN (1,(GETCOUTPUT)); /* OPEN CONSOLE FOR
WRITING */
```

Reference

.GCOUT (System call)

ODIS

Disables console interrupts.

Format

ODIS;

Arguments

None.

Error Conditions

No error condition can occur.

Example

```
ODIS;
```

Notes

When the system is first booted up, it automatically enables the console interrupts CTRL-A, CTRL-C, and CTRL-F. When issued from the foreground, ODIS disables CTRL-A, CTRL-C, and the backgrond interrupt CTRL-F. When issued from the background, ODIS disables CTRL-C and CTRL-A.

To re-enable console interrupts, call OEBL.

Reference

.ODIS (System call)

OEBL

Enables console interrupts.

Format

OEBL;

Arguments

None.

Error Conditions

No error condition can occur.

Example

```
OEBL;
```

Notes

When the system is first booted up, it automatically enables the console interrupts CTRL-A, CTRL-C, and CTRL-F. The call ODIS disables interrupts in either the foreground or the background. OEBL restores interrupts in the ground in which it is issued--CTRL-A, CTRL-C, and CTRL-F in the foreground, CTRL-A and CTRL-C in the background.

Reference

.OEBL (System call)

PUTCHAR

Writes a single character to the console.

Format

PUTCHAR (character *[,error label]);*

Arguments

character is a one-word entity containing the ASCII character you want to output to the console, taken from the low-order byte of the word.

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

Error Conditions

The following error codes may be returned:

ERCID Console not in system.

Example

```
PUTCHAR (101R8,IERR);
COMMENT WRITE THE LETTER "A" TO THE
CONSOLE;
```

Reference

.PCHAR (System call)

READCHAR

Reads and echoes a single character typed from the console.

Format

i := READCHAR *[(error label)]*

Arguments

i is an integer variable that receives the character; the character is stored in ASCII format in the low-order byte, with zeroes in the high-order byte.

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

Error Conditions

The following error codes may be returned:

ERCID Console not in system.

Example

```
ICHAR := READCHAR (IERR);
```

References

.GCHAR (System call) - To read a character from the console.

.PCHAR (System call) - To write a character to the console.

WAITCHAR (for RTOS)

Waits for a character from the console.

Format

i := WAITCHAR (wait character, *[,error label])*

Arguments

Under RTOS, i is an integer variable that receives the device code of the keyboard transmitting the character. Under RDOS, i is an integer variable that receives the character in ASCII format in the low byte.

wait character, under RTOS, is either the console character that will ready the task or -1 which will terminate the wait and ready the suspended task.

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

Error Conditions

The following error codes may be returned:

ERSIM	A previous wait character request is outstanding.

Example

```
ICODE := WAITCHAR (14R8,IERR);
```

Notes

A call to this routine suspends the caller until the specified character is typed onto the console or another task reissues the call with wait character of -1 to terminate this keyboard wait. This routine may suspend only one task at any one moment.

Under RDOS, WAITCHAR is the same as GETCHAR, with the wait character ignored.

Reference

.WCHAR (System call)

See the *RTOS Manual,* Chapter 2, *Teletypewriter and Video Display Commands.*

End of Chapter

Chapter 13
Cache Memory Management

Cache Memory Management (CMM) is a flexible method for making large bodies of data accessible to a program. It proceeds by sequentially reading data from files into buffers in memory.

From the buffer, you can write the data back to the file, or store it in node arrays on a temporary disk file. You can address the data nodes and bring them back into memory for processing as you need them. Using CMM, you can transfer data in 64K blocks of words, in variable-size arrays, or a word at a time. The *DG/L Reference Manual* discusses the principles of CMM in detail.

To use CMM, you need a sequence of runtime calls to perform the following:

1. Create one or more buffer pools-- BUFFER.
2. Open a file for Cache Memory Management-- ACCESS.
3. Read sets or blocks of data into the buffers in memory-- WORDREAD.
4. Write sets or blocks of data from the buffers in memory-- WORDWRITE.
5. Store data in a node array on file zero-- STASH, NODEWRITE.
6. Read a word or node from file zero into a buffer-- FETCH, NODEREAD.
7. 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 array and the offset of the data from the node's first word. MINRES and NODESIZE return the necessary values.

ACCESS

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.

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	File Size
1	64K words
2	65-128K words
3	129-192K words
.	. .
.	. .
.	. .

If you do not specify an *element size,* the size is assumed to be 1.

ACCESS (continued)

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFNM	Illegal filename.
ERSPC	File space exhausted.
ERUFT	No channels are free (returned by .GCHN).
ERSEL	Unit improperly selected.
ERDNM	Device not in system.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERFIU	File in use due to .EOPEN.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERENA	No linking allowed (N attribute).
ERDOP	Attempted open of a tape file which is already open (N attribute).

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.

References

.CRAND (System call)

.OPEN (System call)

See the *DG/L Reference Manual,* Chapter 9.

BUFFER

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 first 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 buffers you want to create.

Error Conditions

The following error conditions may occur:

AIBFS	Insufficient buffers.
ERICM	Illegal system command.
ERICD	Illegal command for device.
ERMEM	Insufficient memory available.

Examples

```
BUFFER(BUFF,2100,7);
BUFFER(BUFF,2000,NUMBUF-4);
```

Notes

Mapped systems ignore pool size; unmapped systems ignore *virtual buffer.* If a mapped system has fewer than 31 + .VMIN blocks available, however, it will allocate the pool size.

The external integer NUMBUF gives the number of virtual memory buffers available. You may specify NUMBUF as *virtual buffer.*

Reference

See the *DG/L Reference Manual,* Chapter 9.

FETCH

Retrieves a word from a node array on file zero.

Format

i := FETCH ([*file address,*] offset)

Arguments

i is an integer expression variable that receives the word.

file address is an integer that specifies the beginning of the file array. If unspecified, the system assumes its value is zero. See MINRES.

offset is an integer expression that specifies the number of words beyond file address where the word is located.

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERRPR	File is read protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is enhausted.
ERFIL	File read error.
ERRD	Attempt to read into system area.
EROVA	File not accessible by direct I/O.
ERFIU	File is in use.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
NEXT := FETCH(N,1);
```

References

.RDB (System call)

See the *DG/L Reference Manual,* Chapter 9.

FLUSH

Writes out the contents of all modified buffers in the pool to disk.

Format

FLUSH (pointer);

Arguments

pointer is a pointer variable that points to the first address of the buffer pool (see BUFFER).

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERWPR	File is write-protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
EROVA	File not accessible by direct I/O.
ERFIU	File is in use.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
FLUSH(BUFF1);
```

Notes

You must use FLUSH to update CMM files; CLOSE will not modify disk files.

References

.WRB (System call)

.CLOSE (System call)

See the *DG/L Reference Manual,* Chapter 9.

HASHBACK

Brings the last used cache memory block back into memory.

Format

```
HASHBACK (pointer);
```

Arguments

pointer is a pointer variable that points to the first address in memory of the buffer pool (see BUFFER).

Error Conditions

No error condition can occur.

Example

```
HASHBACK(BUFF2);
```

Notes

Under mapped systems, this call guarantees that a block is in memory if another cache memory call follows the HASHREAD.

Reference

See the *DG/L Reference Manual,* Chapter 9.

HASHREAD

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 pointer variable that receives the address of the hash value block.

offset is an integer expression that specifies the location of the word in the block.

Rules

Do not call HASHREAD or HASHWRITE under multitasking.

Under extended memory, you can call HASHREAD(filenumber) to bring the last used buffer into core.

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERRPR	File is read protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
ERRD	Attempt to read into system area.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
HASHREAD(1,ELEMENT,BLKNUM,BLKOFF);
.
.
DATA := (BLKNUM+BLKOFF) - BI
```

References

.RDB (System call)

See the *DG/L Reference Manual,* Chapter 9.

HASHWRITE

Marks the current buffer as modified.

Format

HASHWRITE (pointer);

Arguments

pointer is a pointer variable that points to the first memory address of the buffer pool.

Rules

You must identify a modified buffer for it to be written out to disk under FLUSH.

Error Conditions

The following error conditions may occur:

ERFIU File is in use.

Example

HASHWRITE(BUFF1);

Reference

See the *DG/L Reference Manual,* Chapter 9.

MINRES

Is a declaration to specify, modify, or otherwise access the default lower bound of the cache memory file.

Format

EXTERNAL INTEGER MINRES;

Arguments

None.

Rules

If you are using FETCH and STASH, the value in EXTERNAL INTEGER MINRES must agree with your array declarations; FETCH and STASH use MINRES to determine the word offset from the beginning of the node. MINRES has a default value of -3.

Error Conditions

No error condition can occur.

Example

See Figure 13-1.

Reference

See the *DG/L Reference Manual,* Chapter 9.

```
BEGIN
  INTEGER ARRAY LOCARRAY[MINRES: MINRES + NODESIZE (EMPTR)];
    .
    .
    .
  END;
```

Figure 13-1. Call Using MINRES

NODEREAD

Reads a specified node from file zero into memory.

Format

NODEREAD (file address,array);

Arguments

file address is an integer expression that specifies the logical address in the file of the first word of the node.

array is an integer array that receives the words.

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERRPR	File is read protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
ERRD	Attempt to read into system area.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
AINOD	Invalid node specified.

Example

See Figure 13-2.

Reference

.RDB (System call)

See the *DG/L Reference Manual,* Chapter 9.

NODESIZE

Obtains the size of a node on file zero.

Format

i := NODESIZE (file address)

Arguments

i is an integer variable that receives the number of words in the node.

file address is an integer expression that specifies the address in the file of the node's first word.

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERRPR	File is read protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
ERRD	Attempt to read into system area.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
AINOD	Invalid node specified.

Example

See Figure 13-3.

References

.RDB (System call)

See the *DG/L Reference Manual,* Chapter 9.

```
INTEGER ARRAY IDATA[MINRES : MINRES + NODESIZE (ELEMENT)];
 .
 .
 .
NODEREAD(ELEMENT,IDATA);
```

Figure 13-2. Call to NODEREAD

```
BEGIN
   INTEGER ARRAY LOCARRAY [MINRES:  MINRES + NODESIZE (EMPTR)];
```

Figure 13-3. Call to NODESIZE

NODEWRITE

Writes a node from memory to file zero.

Format

NODEWRITE (file address,array);

Arguments

file address is an integer expression that specifies the location in the file you want to write to.

array is an integer array that contains the data you want written.

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERWPR	File is write-protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
AINOD	Invalid node specified.

Example

```
NODEWRITE (LOC,IARAY);
```

Notes

The first word of the array specifies the number of words that will be written.

References

.WRB (System call)

See the *DG/L Reference Manual,* Chapter 9.

STASH

Stores a word into a node array on file zero.

Format

STASH (word *[,file address]* ,offset);

Arguments

word is a variable containing the word you want to store.

file address is an integer expression that specifies the first address of the node array. See MINRES.

offset is an integer that specifies the position within the node where the word will be stored. If file address is unspecified, the offset will be from the beginning of the file.

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERWPR	File is write-protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
STASH(DATUM,N,0);
N := N + 1;
STASH(FETCH(P,$STAT) OR $DATABIT,P,$STAT);
/* SET $DATABIT AT OFFSET $STAT IN NODE P */
```

References

.WRB (System call)

See the *DG/L Reference Manual,* Chapter 9.

WORDREAD

Reads words from a file using cache modules.

Format

WORDREAD (file number,file address,pointer
[,count[,offset]]);

Arguments

file number is an integer expression whose value is associated with the file (see ACCESS).

file address is an integer expression that specifies the first address you want to read in the file.

pointer is a pointer variable that points to the first address in memory to receive the data.

count is an integer expression that specifies the number of words you want to read. If you omit *count,* the routine uses the value of the first word as the count; this is equivalent to NODEREAD.

offset is an integer expression that specifies the number of words beyond file address where you want reading to begin. If you omit *offset,* reading begins at file address.

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERRPR	File is read protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
ERRD	Attempt to read into system area.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
AINOD	Invalid node specified.

Example

See Figure 13-4.

References

.RDB (System call)

DG/L Reference Manual, Chapter 9.

```
ACCESS(1,CASH,BUF1,2000);
  .
  .
  .
WORDREAD(I,C,BUF1);
COMMENT OPENS A FILE AND READS CONTENTS INTO A BUFFER;
```

Figure 13-4. Call to WORDREAD

WORDWRITE

Writes words to a file using cache modules.

Format

WORDWRITE (file number,file address,pointer
[,count[,offset]]);

Arguments

file number is an integer expression whose value is associated with the file (see ACCESS).

file address is an integer expression that specifies the first address of the file you want to write to.

pointer is a pointer expression that points to the first address 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 as the count; this is equivalent to NODEWRITE.

offset is an integer expression that specifies the number of words beyond file address where you want to begin writing. If you omit *offset,* writing begins at file address.

Error Conditions

The following error conditions may occur:

ERFNO	Illegal channel number.
ERICD	Illegal command for device.
ERSV1	Not a randomly or contiguously organized file.
EREOF	End of file (for contiguous files only).
ERWPR	File is write-protected.
ERFOP	Attempt to reference an unopened unit.
ERSPC	Disk space is exhausted.
ERFIL	File read error.
EROVA	File not accessible by direct I/O.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
AINOD	Invalid node specified.

Example

See Figure 13-5.

References

.WRB (System call)

DG/L Reference Manual, Chapter 9.

```
WORDWRITE(1,2,ADDRESS(ARU),2,3)
  /* WRITE 2 WORDS, STARTING 2 WORDS PAST START OF ELEMENT 2 */
```

Figure 13-5. Call to WORDWRITE

End of Chapter

Chapter 14
INFOS System Routines

INFOS® is DGC's file management system. It allows you to create and manage files using random, sequential, indexed sequential, and database access modes. DG/L offers you an interface with the INFOS system.

The manuals cited in the Preface to this book describe the INFOS system, document the system calls, and contain tables of parameter packets and error messages.

To use INFOS system calls in DG/L, you must create parameter packets that specify conditions for processing. See the *INFOS System Programmer's Manual* and the *INFOS System Planning Manual* for a detailed description of the Volume Initialization Packet, File Definition Packet, and Process Packet that system calls require.

Once you have created the INFOS packets, DG/L provides four calls to interface with the system. INFOS initializes a volume; PINFOS calls the system to pre-open the volume, and resolve any parameter defaults. OINFOS opens the finished volume for execution after pre-opening, and INFOS calls the system to perform INFOS operations.

IINFOS
Initializes an INFOS volume.

Format

IINFOS (packet *[,error label]);*

Arguments

packet is an integer array that contains the initialization information.

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

Error Conditions

Error conditions depend on the INFOS volume.

Example

```
INTEGER ARRAY INFPK [0:15];
.
.
.
IINFOS (INFPK, IERR);
```

Reference

.INFOS --See the *INFOS System Programmer's Manual.*

INFOS

Calls the INFOS system.

Format

INFOS (channel number,function,process packet
[,error label]);

Arguments

channel number is an integer expression whose value is associated with the file by an OINFOS call.

function is an integer expression that specifies the INFOS function to be called.

process packet is an integer array containing the process packet that passes to the INFOS system call.

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

Error Conditions

Error conditions depend on the INFOS call.

Example

```
INTEGER ARRAY PKARR [0:20 ]; INTEGER FUN;
.
FUN:=OINFOS(PKARR);
.
.
INFOS(FUN,20,PKARR,IERR);
```

Reference

.INFOS --See the *INFOS System Programmer's Manual.*

OINFOS

Opens an INFOS file.

Format

i := OINFOS (file definition *[,error label]);*

Arguments

i is an integer variable that receives the channel number of the opened file.

file definition is an integer array that contains the file definition packet passed for the system call.

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

Rules

You must pre-open the file with a call PINFOS before actually opening it.

Error Conditions

INFOS Open Error Messages.

Example

```
INTEGER INFIL;
INTEGER ARRAY PKT [0:49];
.
.
.
INFIL:=OINFOS(PKT,IERR);
```

Notes

You must pre-open the file with a call to PINFOS before actually opening it.

Reference

.OINFOS --See the *INFOS System Programmer's Manual.*

PINFOS

Pre-opens an INFOS file.

Format

PINFOS (file definition *[,error label]);*

Arguments

file definition is an array that contains the file definition packet.

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

Error Conditions

INFOS Open Error Messages.

Example

```
PINFOS (PKT,IERR);
```

Notes

Pre-opening allows the system to resolve all parameters in the File Definition Packets and Volume Definition Table. INFOS assigns system-defined parameters for all those left unspecified.

Reference

.PINFOS --See the *INFOS System Programmer's Manual.*

End of Chapter

Chapter 15
Calling to and Returning from Programs

Just as Cache Memory Management uses disk I/O to make large bodies of data available to a program, program swaps and overlays allow you to run modular programs of virtually unlimited length by executing files stored on disk.

In a program swap or chain, each new segment replaces the user's entire memory space. Overlays, described in Chapter 16, replace only a portion of your user address space.

Swapping

In a program swap, the currently executing DG/L program suspends its execution and calls in another program file which resides on disk. The system brings in the new program, writing over the code of the program that called the swap.

Under normal termination, the swapped-in program returns control to the calling program. You may, however, provide for a return to the operating system by calling SYSRETURN.

If the swapped-in program generates a fatal error, the system will return control to the calling program. You can, of course, provide your own error procedure as outlined in Chapters 1 and 8.

A swapped-in program can also call for another swap. In this case, the error return or normal return will go to the first swapped-in program before it goes to the original program. If you want to call in more than one swap but do not to create a hierarchy among the swap files, use the call to CHAIN.

Chaining

In the CHAIN call, the file brought in from disk operates as a simple extension of the calling program. The system considers each segment of the program to be on the same level, and returns to the next higher level upon termination or error condition. Thus, chained program files work as sections rather than as subprograms.

Operating Instructions

Compile and assemble program files for swapping and chaining as you would any other DG/L program file; use RLDR to create a separate, executable file with an .SV extension for each file to be swapped or chained.

CHAIN

Terminates execution of the current program segment and calls another disk file for execution.

Format

CHAIN (filename, *[datum[,error label]]);*

Arguments

filename is a string expression that names the file you want to invoke.

datum is a variable containing one word you want to pass to the new file.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERSV1	File requires "S"ave attribute.
ERDLE	File does not exist.
ERCM3	Trying to push too many levels.
ERMEM	Attempt to allocate more memory than is available.
ERADR	Illegal starting address.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERUSZ	Too few channels defined at load time or SYSGEN time.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second timeout occurred.
ERENA	No linking allowed (N attribute).

Example

```
CHAIN ("FIL.SV",TDAT,IERR);
```

Reference

.EXEC (System call)

ERETURN

Terminates a swapped-in program and indicates an error.

Format

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

Arguments

error is an integer variable that receives one of the RDOS error status codes. See the *RDOS Reference Manual.* Typically, this is a code returned by READERROR.

error label is a statement label in the higher program to which control transfers if the ERETURN cannot be executed.

Rules

Upon execution of a call to ERETURN, unconditional return is made to the next higher level program. If the next higher level program is the CLI, one of the following occurs:

1. If the error status code is an RDOS error code, the appropriate message is not displayed at the output device.

2. If a null error occurs (ERNUL), the CLI does not report a message.

3. If the error EREXQ is returned, the CLI takes its next command from disk file CLI.CM.

4. 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.

Error Conditions

No error condition can occur.

Example

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

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

Notes

This call is also documented in Chapter 8.

References

.ERTN (System call)

See the *RDOS Reference Manual,* "Error Message Reporting."

SWAP

Transfers control to a program on disk file.

Format

SWAP (filename *[,datum[,error label]]);*

Arguments

filename is a string expression that names the program you want to transfer control to.

datum is a variable containing one word you want to pass to the new program.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERSV1	File requires "S"ave attribute.
ERDLE	File does not exist.
ERCM3	Trying to push too many levels.
ERMEM	Attempt to allocate more memory than is available.
ERADR	Illegal starting address.
ERDNM	Device code exceeds 77 octal (returned by .DEBL).
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERUSZ	Too few channels defined at load time or SYSGEN time.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERENA	No linking allowed (N attribute).

Example

```
SWAP ("FILE.SV",TDAT,IERR);
```

Reference

.EXEC (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.

Error Conditions

No error condition can occur.

Reference

.RTN (System call)

End of Chapter

Chapter 16
Using Overlays in a Single-Task Environment

Like swap files, overlays are program modules which are stored on disk and which your program calls into memory for execution. However, overlays differ from swaps in several important ways.

You must create the file specifically as an overlay file. It should be divided into numbered overlays, and it must have a .OL extension.

The overlay occupies only a portion of the user's memory space. The program that calls for an overlay remains resident in main memory, and always regains control when the overlay completes execution.

File Structure

An overlay file is a contiguous disk file, divided into segments called nodes. A file can have up to 256 nodes, which can be of different sizes. The nodes are divided into overlays; the overlays can be of any size up to 125 blocks, but within a given node all overlays must be of equal length. The nodes within the file are numbered from zero, as are the overlays within each node.

To reference and load an overlay, you must identify it by the node number/overlay number (byte packed). You do this through reference to a symbol established by RLDR. See Chapter 10 of the *DG/L Reference Manual* for explicit instructions.

DG/L Overlay Routines

To use an overlay file in a DG/L program, declare the overlay procedure EXTERNAL within the program and load the procedure along with the root program in RLDR. To call an overlay, you must open the overlay file, and load the specific overlay you want.

OVOPN opens an overlay file; OVLOAD brings it into memory. OVCLOSE closes the overlay file and allows another file of overlays to be loaded in the overlay area. The system always allocates an overlay area large enough for the largest overlay that RLDR encounters.

See Chapter 4 of the *RDOS Reference Manual* for a discussion of overlay files and overlay management, and Chapter 10 of the *DG/L Reference Manual* for information on overlay programming.

OVCLOSE

Closes an overlay file.

Format

OVCLOSE *[(error label)];*

Arguments

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	Attempt to reference a channel not in use.
ERDTO	Ten second disk timeout occurred.

Example

```
OVCLOSE (IERR);
```

Reference

.CLOSE (System call)

OVLOD

Loads an overlay file in a single-task environment.

Format

OVLOD (overlay *[,flag[,error label]]);*

Arguments

overlay gives the node number/overlay number in bytes through reference to a symbol defined by RLDR. See Chapter 10 in the *DG/L Reference Manual* for a definition of its use.

flag is an integer expression set to -1 if the overlay will be loaded unconditionally (whether it is already resident or not). Zero or no argument specifies a conditional loading.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
EREOF	End of file.
ERRPR	Attempt to read a read-protected file.
ERFOP	File not opened.
ERFIL	Read error.
EROVN	Illegal overlay number
EROVA	Overlay file is not a contiguous file.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
OVLOD (101,-1,IERR);
```

Reference

.OVLOD (System call)

OVOPN

Opens an overlay file in a single-task environment.

Format

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

Arguments

file number is an integer expression whose value is to be associated with the filename.

filename is a string that gives the name of the overlay file. It must have an .OL extension.

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

Rules

You must open an overlay file before you can access any overlay within.

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFNM	Illegal filename.
EREOF	End of virtual overlay file.
ERRPR	Attempt to read a read-protected virtual overlay file.
ERDLE	Non-existent file.
ERUFT	Attempt to use a channel which is already is use.
ERMEM	Insufficient memory to load virtual overlays.
ERFIL	File read error of virtual overlay file.
EROVA	Not a contiguous file (virtual overlays only).
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second timeout occurred.
ERENA	No linking allowed (N attribute).

Example

```
OVOPN (200,"CATCH9.OL",IERR);
```

Reference

.OVOPN (System call)

End of Chapter

Chapter 17
Foreground/Background Programming

EXBG (Mapped systems only)

Checkpoints a mapped background program; i.e., suspends one background program temporarily in order to execute a background program of a higher priority.

Format

EXBG (filename,priority *[,error label]);*

Arguments

filename is a string expression that gives the name of the background program you want to execute.

priority has one of the following integer values:

0 Give the checkpoint program the same priority as the old background program;
1 Give the checkpoint program the same priority as the current foreground program.

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

Rules

Only a foreground program in a mapped environment may issue the checkpoint call.

A checkpointed program resumes execution when the new background program terminates, or when you press a CTRL-C or CTRL-A at the keyboard. A keyboard interrupt produces the message CP INT on $TTO. When the checkpointed program is restored, the message CP RTN appears on $TTO.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERICM	Attempt to checkpoint in an unmapped system.
ERSV1	File requires "S"ave attribute.
ERDLE	File does not exist.
ERUFT	Not enough channels.
ERCM3	Trying to checkpoint a new background program.
ERMEM	Attempt to allocate more memory than is available.
ERDSN	Directory specifier unknown.
ERLDE	Link depth exceeded.
ERDNI	Directory not initialized.
ERUSZ	No room for UFT's.
ERMPR	Address is outside address space (mapped systems only).
ERNTE	Program to be checkpointed is not checkpointable, or attempt to create two outstanding checkpoints.
ERDTO	Ten second disk timeout occurred.

Example

```
EXBG ("PROG10.SV",0,IERR);
```

Reference

.EXBG (System call)

EXFG

Loads and executes a program in the foreground.

Format

EXFG (filename,priority *[,error label]);*

Arguments

filename is a string expression that gives the name of the program you want loaded and executed.

priority has one of the following integer values:

0 Give the foreground program a higher priority than the background program;
1 Give the foreground program the same priority as the background program.

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

Rules

You must issue a call to EXFG from a currently running background program.

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERSV1	File requires "S"ave attribute.
ERDLE	File does not exist.
ERUFT	Not enough channels.
ERMEM	Attempt to allocate more memory than is available.
ERADR	Illegal starting address.
ERDSN	Directory specifier unknown.
ERDNI	Directory not initialized.
ERFGE	Foreground already exists.
ERUSZ	No room for UFT's.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.

Example

```
EXFG ("FGPROG.SV",0,IERR);
```

Reference

.EXFG (System call)

FOREGROUND

Determines whether or not a foreground program is running.

Format

```
bool := FOREGROUND
```

Arguments

bool is a boolean variable that receives one of the following:

FALSE	A foreground program is not running;
TRUE	A foreground program is running.

Error Conditions

No error condition can occur.

Example

```
IF NOT FOREGROUND THEN EXFG ("CLI.SV", 1);
```

Reference

.FGND (System call)

GETGROUND

Determines if the current program is in the foreground or background.

Format

i := GETGROUND

Arguments

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

0 Current program is in background;
1 Current program is in foreground.

Error Conditions

No error condition can occur.

Example

```
IF GETGROUND = 0 THEN

        OPEN (0, "COM.CM");
        ELSE
                OPEN (0, "FCOM.CM");

/* SIMULATE OPEN (0,NAMEGROUND("COM.CM")); */
```

ICOMMON

Defines a message area in program space for interground communication.

Format

ICOMMON (pointer,length *[,error label]);*

Arguments

pointer is a pointer variable that receives the beginning of the area for sending or receiving messages. Each ground must have one area for receiving and one for sending messages.

length is an integer that gives the size of the communications area in words (no greater than 256 (decimal)).

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

Error Conditions

The following error codes may be returned:

ERCMS Communications area exceeds the program size or attempt to overwrite the system.
ERMPR Address is outside address space (mapped systems only).

Example

```
POINTER MESSAG;
.
.
ICOMMON (MESSAG,256,IERR);
```

Reference

.ICMN (System call)

NAMEGROUND

Retrieves an identifier which indicates if it is in the foreground.

Format

s := NAMEGROUND (name)

Arguments

s is a string variable that returns one of the following:

name	if the identifier is in the background.
Fname	if the identifier is in the foreground.

name is any string.

Error Conditions

No error condition can occur.

Example

```
OPEN(0,NAMEGROUND("COM.CM"));
```

Notes

This routine is useful for creating unique temporary filenames to prevent cross-ground interference, and for accessing COM.CM *[FCOM.CM]*.

RDCOMMON

Reads a message from the other ground's communication area.

Format

RDCOMMON (pointer,offset,count *[,error label]);*

Arguments

pointer is a pointer variable that points to the first address of the other ground's message area (see ICOMMON.)

offset is an integer expression that gives the word offset within the other ground's communications area where the message begins.

count is an integer expression that gives the number of words you want read (no greater than 256 decimal).

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

Error Conditions

The following error codes may be returned:

ERCMS	The size of the requested message exceeds the communications area size.
ERCUS	No communications area is defined in the other program.
ERMPR	Address is outside address space (mapped systems only).

Example

```
POINTER MESARY;
.
.
RDCOMMON (MESARY,100,100,IERR);

/* READ THE SECOND HALF OF THE MESSAGE*/
```

Reference

.RDCM (System call)

RDOPERATOR

Reads a message from the console.

Format

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

Arguments

s is a string variable that receives the message.

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

Rules

Either the foreground or background task may receive a message. Only one message may be outstanding at any one moment in the foreground and one in the background. The system always reads from the background.

The first character in the message must be a CTRL-E character (echoed as an exclamation point); if no system call to read an operator message was issued, a bell sounds when you press CTRL-E. The second character must be either an F or a B to indicate whether the foreground or the background program is to receive the message. Entering any character other than these will sound the console bell and no further text will be accepted until an F or B is typed.

The message then follows; its last character must be a carriage return, form feed or null. The full message (including the terminator) can be up to 132 characters long.

To cancel a message, type RUBOUT in column 2 instead of F or B.

Do not issue a call to RDOPERATOR if you are using INITOPCOM or the task message runtime routines TWROPERATOR and TRDOPERATOR (see Chapter 27).

Error Conditions

The following error codes may be returned:

ERICM	Operator messages not specified.
ERMPR	Address is outside address space (mapped systems only).

Example

```
MESSAGE := RDOPERATOR (IERR);
```

Reference

.RDOP (System call)

WRCOMMON

Writes a message into the other ground's communication area.

Format

WRCOMMON (message,offset,count *[,error label]);*

Arguments

message is a string expression that gives the message.

offset is an integer expression that gives the word offset within the other ground's communications area where the message begins.

count is an integer expression that gives the number of words you want to write (no greater than 256 decimal).

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

Error Conditions

The following error codes may be returned:

ERCMS	Message too large for communications area.
ERCUS	No communications area is defined in the other program.
ERMPR	Address is outside address space (mapped systems only).

Example

```
STRING MESSAG;
.
.
WRCOMMON (MESSAG,20,10,IERR);
```

Reference

.WRCM (System call)

WROPERATOR

Writes a message to the console.

Format

WROPERATOR (message *[,error label]);*

Arguments

message is a string expression that contains the message you want sent.

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

Rules

The message may exist in either the foreground or the background, but it is always written to the background console. Only one outstanding WRITE command may exist in the foreground and one in the background.

The message appears as two exclamation points, followed by F or B to indicate whether the origin of the message is in the foregound or the background, followed by the message and terminator.

Do not issue a call to WROPERATOR if you are using INITOPCOM or the task message runtime routines TWROPERATOR and TRDOPERATOR (see Chapter 27).

Error Conditions

The following error codes may be returned:

ERICM	Operator messages not specified.
ERMPR	Address is outside address space (mapped systems only).

Example

```
STRING(67) STRNG;
.
.
WROPERATOR (STRNG,IERR);
```

Reference

.WROP (System call)

End of Chapter

Chapter 18
User Device and Multiple Processor Routines

BOOT

Boots a new system, partition, or device.

Format

BOOT (unit name *[,error label]);*

Arguments

unit name is a string expression that gives the name of the system, partition, or device.

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

Error Conditions

The following error codes may be returned:

ERFNM	Illegal filename.
ERDLE	Unit name does not exist.
ERRTN	File RESTART.SV does not exist yet; data switches were all set for restarting without operator intervention.
ERDSN	Unknown specifier.
ERMPR	Address is outside address space (mapped systems only).
ERDTO	Ten second disk timeout occurred.
ERSFA	Spool file is active.

Example

```
BOOT ("DP0",IERR);
```

Notes

This call closes all open files in the currently executing system (both foreground and background), releases all directories and resets all system I/O.

Upon completion of the call to BOOT, HIPBOOT receives control and then passes it to a new operating system or to a stand-alone program on the specified device or in the specified file.

Reference

.BOOT (System call)

DUCLK

Initializes a non-DG/L clock service routine, and specifies the intervals.

Format

DUCLK *[(procedure,time,units[,error label])];*

Arguments

procedure is the name of the task that will run under the clock routine.

time is an integer that specifies the number of units between the times the task executes.

units is an integer that indicates the size of the units.

Code	Meaning
0	System units
1	Milliseconds
2	Seconds
3	Minutes

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

A call to DUCLK with no arguments removes a current user clock.

Error Conditions

The following error codes may be returned:

ERICD	Illegal command for device.
ERDNM	Device not in system.
ERIBS	A user clock already exists.
ERMPR	Address outside address space (mapped systems only).

A non-fatal error occurs from an illegal number conversion.

Example

```
DUCLK (SORT,2,4,IERR);
```

References

.DUCLK (System call)

.RUCLK (System call)

GETMCA

Gets the unit number of the MCA.

Format

i := GETMCA *[(error label)]*

Arguments

i is an integer variable that receives the number of the MCA.

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

Error Conditions

The following error codes may be returned:

ERICD	Improper device code.
ERDNM	Device not in system (i.e., no MCA was specified at SYSGEN time in this operating system).

Example

```
IUNIT := GETMCA (IERR);
```

Notes

There may be two MCAs (Multiprocessing Communication Adaptors) in a given system. A call to GETMCA returns the code of the first MCA and a call to GETMCA1 returns the code of the second MCA.

Reference

.GMCA (System call)

GETMCA1

Gets the unit number of the secondary MCA.

Format

i := GETMCA1 *[(error label)]*

Arguments

i is an integer variable that receives the number of the secondary MCA unit.

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

Error Conditions

The following error codes may be returned:

ERICD	Improper device code input to system call.
ERDNM	Device not in system (i.e., no MCA was specified at SYSGEN time in this operating system).

Example

```
IUNIT2 := GETMCA1(IERR);
```

Notes

There may be two MCAs (Multiprocessing Communication Adaptors) in a given system. A call to GETMCA returns the code of the first MCA and a call to GETMCA1 returns the code of the second MCA.

Reference

.GMCA (System call)

End of Chapter

Chapter 19
Multitask Programming in DG/L

The remaining chapters of this manual document runtime routines for multitask programs. This chapter gives an overview of the DG/L runtime calls that control the multitask environment. For a detailed presentation of multitasking, see Chapter 5 of the *RDOS Reference Manual,* "Multitask Programming." The *RDOS Reference Manual* describes the basic multitask concepts of initiating and queueing tasks, changing task states, and communicating with tasks.

Multitask programming is the most challenging and complex extension of your DGC 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. This condition is called *asynchronous execution.*

Multitask programs allow you to arrange these asynchronous executions 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.

Additionally, DG/L mulitasking has available all the memory-extending and program-extending features outlined earlier in this manual. With DG/L multitasking, you may use overlay programs, foreground/background programs, disk file I/O, and Cache Memory Management.

Tasks receive control of the CPU according to two criteria--their priority level and their state of readiness. The multitask environment uses CPU time efficiently by defining tasks as ready or suspended. The separate tasks execute concurrently, with each task taking control of the Central Processing Unit by turns.

When a task calls for input or output, the task manager defines it as waiting, and the task does not receive its time slice. Meanwhile, the I/O operation proceeds while other tasks use the CPU. 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 arrange for some to have a higher priority than others. If several tasks are ready to execute at once, the one with highest priority will receive 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 needs to 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.

The Multitask Runtime Environment

Figure 1-4 in Chapter 1 shows main memory for a multitask program. The compiler creates a single area of program code, and a single Process Global Table containing information about the program as a whole. Within the user stack, however, there are multiple stack spaces for the 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 in different executions of the same program.

Because the tasks execute concurrently, however, the RDOS task scheduler must have separate task state information about each task in a Task Control Block (TCB). RLDR creates the TCBs at load time.

Because tasks alternate in execution, each needs a TCB extender to preserve the contents of task-specific page zero locations and the floating point unit's contents and status. DG/L allocates a TCB extender when you initialize the task. When a task regains control of the CPU, the information in the TCB extender makes it possible to resume execution with the values the task had when it left off.

Creating a Task

You must code any part of your program that will run as a task as an external procedure. Like any procedure, the task code begins with the declaration

PROCEDURE task name *[(argument)];*

You will call the task by task name; you may also pass a one-word datum to the task as an *argument.* You must declare any other variables that tasks use in common as EXTERNAL variables in the program. See Chapter 1 of this manual, "External Data and the Internal Structure of Data," for directions.

Compiling

You must compile each task in a multitask program separately:

```
(EXTERNAL PROCEDURE)

DGL procedure
DGL procedure
DGL procedure
```

Each procedure is an entry point in your DG/L multitask program.

Loading

When you load your program, include the name of each task procedure in the command line, as in the following example:

```
RLDR TASK1 TASK2 TASK3 10/K
```

In this example, three separate procedures can run as tasks. The K switch directs the compiler to create a number (in octal) of TCBs (one by default). When this program runs, there can be up to ten tasks in any combination of activations of TASK1, TASK2, or TASK3 running at any given time.

See Chapter 11 of the *DG/L Reference Manual* for a full discussion of assembling and loading a DG/L program.

DG/L Multitasking Routines

The DG/L runtime libraries offer the full range of RDOS task calls. In addition, they contain 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 will need to execute a multitask program will 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, either through intertask communication or through the operator's console.

In addition to monitoring tasks, you may need to program the conditions of task execution in your code. DG/L routines for changing task states can ready, suspend, abort, or kill a task.

Initiating a Task

Under DG/L you have four options for initiating a task. The calls TASK and RUNTASK begin a task immediately; RUNTASK allows you to pass more than one argument to the task. See Chapter 20 for these calls. Using the queueing commands QTASK and OVQTASK, you can specify a relative delay before the task begins to run (see Chapter 23). If you initiate task/operator communications, you can add tasks to those executing by issuing commands from the console (see Chapter 27).

With any of these procedures, you may specify the stack size or allocate a default-size stack. With QTASK, you can specify when a task will begin to run, and the intervals of successive activations.

When you initiate a task, you give the task a specific identification number, a priority level, and a stack size. The runtime environment allocates it a stack space, instructs the system to associate a TCB with it, and enters it into the list of running tasks.

Task Identifiers

A number of DG/L task routines allow you to indicate the task you want affected by the call (see TIDABORT, TIDKILL, TIDREADY, etc.). To use these calls you must issue the call with the task's identification number.

The task identification number is actually a 16-bit number with a value in the range 1 through 255 inclusive. Any number of the active tasks may have an identifier of zero, but they cannot be referenced by number.

If your program gives the task an identifier less than 1 or greater than 255, there will not be an error condition. Instead, DG/L will use only the low byte of the integer. Algebraically, the value used for a task identifier from the positive integer I would be

```
MOD (I,256)
```

This conversion also applies in calls that specify the task identifier.

If you try to refer to non-existent task identifiers, you will create the error condition ERTID --"no such identifier exists."

Priority Levels

You can assign a priority level to a task when you initiate it. The level remains in effect until you change it with a call to PRIORITY. Zero is the highest priority, 255 the lowest. By default, a task has priority zero.

When you assign a task priority, you direct the task controller to give the CPU to the task before other ready tasks of lower priority. Tasks with lower priority 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 to guarantee it CPU time.

A task can change its own priority level or the priority level of another task: see PRIORITY, IDPRIORITY.

Stack Sizes

When you initiate a task, DG/L allocates it a portion of your program's stack space. By default, the task receives 400_8 words of memory. However, you may find that this standard size uses memory inefficiently, especially if the various tasks have quite different storage reqirements.

The calls which initiate a task give you the option of specifying the stack size. You may define the stack size for some tasks and allocate the default size for others in the same program.

It is important to define stack spaces carefully; once a task completes execution, its space becomes free for use by another task of the same or smaller stack size. 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.

When an allocation fails to find a large enough block of memory, the call merges contiguous blocks and tries to find a block large enough. If this fails there will be an error condition.

Calls that initiate tasks also give you the option of waiting for a stack space; RUNTASK, however, must find a stack immediately.

If you do not indicate the option of waiting for a default stack, you may include an assembly language error-handling routine in your code. Give the procedure name as an argument to the initiating call.

For information on error reporting and error handling, see "Error Handling" in Chapter 1 and the routines in Chapter 8 of this manual.

Stack Contents

The following information will help you define a stack size. The first three elements in the list must be present:

Area	Size (decimal)
Global Area	7 words
TCB Extender	26 words
Endzone	24 words
Input/Output Control Block:	93 words/IOCB
Task Stack Space:	
Frame Header	5 words
Arguments	Number of arguments received
Temporary Storage	Number of words given in SAVE n + total number of words needed for all subprograms and runtime routines which the procedure calls.

Task Control and Communications

You can alter the "round robin" execution of tasks by specifying a priority level when you initiate tasks. DG/L task routines also make it possible to control the flow of execution while the tasks are running. The calls in Chapter 22, "Changing Task States" and in Chapter 26, "Intertask Communication," enable you to override the way the system defines a task's readiness.

Suspending, Readying, Killing

Calls to SUSPEND, ASUSPEND, and TIDSUSPEND block a task's execution until another task executes a call to AREADY or TIDREADY. These calls permit you to block a task's execution while another task takes precedence.

Calls to KILL, TIDKILL, and AKILL remove tasks and free their stack spaces. These calls are useful where an error in one task affects other tasks.

You can also define conditions that block a task temporarily at a given point in the procedure. See Chapter 26, "Intertask Communications," for routines, a discussion, and examples.

Termination

When a task terminates, the runtime environment will release its stack space.

Because a multitask program executes asynchronously, it has no necessary end defined in its code. The program continues execution until the last task has finished.

Examples

The examples in Figures 19-1 and 19-2 illustrate some principles of multitask programming, and show how to use some DG/L multitasking routines.

Program Structure

The example in Figure 19-1 shows the structure of a multitask program and illustrates some of the combinations of initiating calls that you can use.

Each procedure that runs as a task is a separate routine, external to the others. MULT, MULT1, MULT2, and MULT3 all have separate entry points.

Unlike single-task programs, a multitask program theoretically has no "main program," only an initial routine. The initial routine, however, is still called ".MAIN".

The procedure MULT1 shows that any task can initiate another procedure as a task, as it does in the calls to TASK for MULT2 and MULT3.

MULT1 also calls MULT3 as a subroutine. In this call, the variables' storage will be in MULT1's stack space, with a stack frame for the subroutine.

When MULT1 initiates MULT3 as a task, however, the multitask initializer allocates a separate stack space to MULT3, and the new task takes its turns executing in alternation with MULT1. When MULT1 runs as a task, it does not return values to the initiating task.

```
BEGIN
 EXTERNAL PROCEDURE TASK, RUNTASK, QTASK, MULT1, MULT2, MULT3;
       .
       .
       .
      TASK (MULT1, argument);
       .
       .
      RUNTASK (MULT2, arguments);
       .
       .
      QTASK (MULT3, argument);
     END;


  PROCEDURE MULT1 (arguments);
     BEGIN
       MULT3;
       TASK (MULT2,argument);
       TASK (MULT3,argument);
       END;

  PROCEDURE MULT2 (argument);
       .
       .
       .
     END;

  PROCEDURE MULT3 (argument);
     BEGIN
       .
       .
       END;
```

Figure 19-1. Structure of a Multitask Program

Intertask Communication

The example in Figure 19-2 demonstrates the task communication calls TRANSMIT and RECEIVE. The program is a simple tasking routine to read lines from separate terminals and write them to a single disk file.

RECEIVE and TRANSMIT

In Figure 19-2, the calls RECEIVE and TRANSMIT prevent two tasks from writing to the same file at once. The first block of the program declares an EXTERNAL integer, FILECLEAR, as a message word. The message, "-1," will allow a task to proceed past the RECEIVE call. When one task receives the message, the message word becomes 0 until the task transmits it again. Any task that attempts to RECEIVE the 0 message will be suspended until the message word is set to a different value by another task's execution of TRANSMIT.

```
BEGIN
  EXTERNAL INTEGER OUTCHAN, FILECLEAR;
  EXTERNAL INTEGER (2) COUNT;
  INTEGER I, J;
  EXTERNAL PROCEDURE TRANSMIT, TASK, DELAY, READIT, IDSTATUS;

  COUNT := 0;
  TRANSMIT (FILECLEAR, -1, OK);

OK:  OPEN (OUTCHAN := 0,"0UTFILE");
     FOR I := 1 STEP 1 UNTIL 10 DO
     TASK (READIT,I,I,5,400,NOSTACK);
     J := 11
     GO TO CHECK;
NOSTACK: K := I;

CHECK:    READIT (J);
          DO BEGIN
          DELAY (5,2);  /* DELAY 5 SECONDS */

          FOR I := 1 STEP 1 UNTIL K DO
              IF IDSTATUS(I)  8 THEN
                   GO TO CONT;
          GO TO NOMORE;

CONT:        END;

NOMORE:      END;

     PROCEDURE READIT (ME);
       INTEGER ME;

     BEGIN
       INTEGER DATUM; STRING S;
       EXTERNAL INTEGER FILECLEAR, OUTCHAN;
       EXTERNAL INTEGER (2) COUNT;
       EXTERNAL PROCEDURE RECEIVE, TRANSMIT;

       OPEN (ME, "QTY:" || ME);

       DO BEGIN
           READ (ME,S,EOF);
           RECEIVE (FILECLEAR,DATUM);
           WRITE(OUTCHAN,S);
           COUNT := COUNT + 1;
           TRANSMIT (FILECLEAR,DATUM);
           END;

EOF:       END;
```

Figure 19-2. Example of a Multitask Program

The call to RECEIVE is part of the procedure READIT. Any task can read the string S from a console, but only one at a time can receive a -1 at FILECLEAR.

After the first task passes TRANSMIT, the next task with highest priority can receive the message, clear the message word, and proceed through the WRITE portion of the procedure whether or not there was an error on TRANSMIT.

TASK

The first call to TRANSMIT initializes the message word. Using the *error label* argument, it then transfers control to the label, OK, which starts the tasking program.

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. Each task has an identifier set at its value for I, and passes that number to the routine; each task takes a stack of 400 words.

Passing a Datum

Each activation of READIT has its own identifier and reads a string from a different console.

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 console to read, and gives the console a unique file number. The call to OPEN reads the QTY whose number corresponds to ME.

However, COUNT which records the number of WRITEs, is an EXTERNAL variable. The value of COUNT is not part of any of the tasks.

When the task executes the READ, it will be pended until the READ operation is complete. When it reaches RECEIVE, however, it will wait regardless of its readiness if another task is writing.

No Stack Error Handler

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, receives control. NOSTACK sets J to a value one greater than the highest initiated task. If stack space is available, J is set to 11.

DELAY and IDSTATUS at CHECK

The code at CHECK serves two purposes. First, it allows the main task to act as a task reading from a console. When the main task receives an end-of-file condition from the console, the subroutine terminates, and the main task continues with its second purpose.

CHECK then 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 8, no task with the specified ID is running.

If no tasks are running, then the program will terminate. Control passes from CHECK to an END at NOMORE. If tasks are running, execution continues at the top of the loop, delaying the main task five seconds.

Terminating

The program terminates when all the tasks have reached the END at EOF. When all ten tasks are complete, control passes to the end at NOMORE.

DG/L and Non-DG/L Tasks

The DG/L runtime environment handles both DG/L and non-DG/L tasks. Non-DG/L tasks are subject to restrictions that do not hold for DG/L tasks. They cannot use the floating point unit or the DG/L state variables .GP, .RP, and .RSP. On the NOVA machines, they cannot use the stack or the state variables .FP, .SP, and .SSE. On the ECLIPSE computer, however, they use the hardware stack freely.

You may, however, simulate a DG/L task by beginning a non-DG/L task procedure with a SAVE instruction (NOVA macro). This strategy will signal the system to save and restore the floating point unit's state and any other state variables stored in the TCB extender. See Chapter 1, "Writing Your Own Runtime Routine," for a discussion of non-DG/L routines.

End of Chapter

Chapter 20
Initiating Tasks in a Multitask Environment

RUNTASK

Executes a routine as a task.

Format

RUNTASK (procedure *[,arguments]* ,identifier, priority,stack size,key *[,error label]);*

Arguments

procedure is the name of the subroutine 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.

stack size is an integer expression that specifies the stack size you want to allocate to the subroutine when it executes as a task. Zero or no argument indicates the default-size stack of 400 (octal) words.

key is an integer variable that defines error handling and returns the routine's completion status:

Code	Meaning
-1	Intercept and report all errors
0	Wait for a non-zero key (set by WAITFOR); task is executing
1	Task successfully completed
= > 3	Error in task (error code + 3)

Set the key to a non-zero initial value.

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

Error Conditions

The following error codes may be returned:

ERNOT	No TCB available.
ERTID	A task with the requested ID (except 0) already exists.

Example

```
RUNTASK (SUBR1,10,2,0,1,IERR);
```

Notes

RUNTASK is the only task initiation call that allows you to pass arguments. The task must run immediately; you do not have the option of waiting for a stack.

Reference

.TASK (Task call)

TASK

Initiates a task and allocates storage for it.

Format

TASK (procedure *[,datum[,identifier[,priority [,stack size[,error label]]]]]);*

Arguments

procedure is the name of the DG/L or runtime subroutine you want to initiate as a task.

datum is a one-word variable or constant value--not an expression--which you want to pass to the procedure.

identifier is an integer expression that becomes the task number.

priority is an integer expression that gives the task's priority number.

stack size is an integer expression that gives the stack size you want to allocate to the program when it executes as a task. A positive value gives the size of the stack. Zero or no argument indicates the default size of 400 (octal) words. A negative value indicates waiting for a default-size stack.

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

Error Conditions

The following error codes may be returned:

ERNOT	No TCB's available.
ERTID	A task with the requested ID (except 0) already exists.

Example

```
TASK (SUBR9,LAST,15,4,1000,IERR);
```

Reference

.TASK	(Task call)

End of Chapter

Chapter 21
Obtaining Task-Related Information in a Multitask Environment

GETIDENTIFIER

Obtains the calling task's identification number.

Format

```
i := GETIDENTIFIER
```

Arguments

i is an integer variable that receives the task's identification number.

Error Conditions

No error condition can occur.

Example

```
I := GETIDENTIFIER;
FORMAT (0, "I AM A TASK <NL>", (GETIDENTIFIER));
```

GETPRIORITY

Obtains the calling task's priority.

Format

```
i := GETPRIORITY
```

Arguments

i is an integer variable that receives the task's priority number.

Error Conditions

No error condition can occur.

Example

```
I := GETPRIORITY;
```

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.

Error Conditions

The following error codes may be returned:

ERTID	You specified an identifier of zero, or no such task ID was found.

Example

```
PTASK := IDPRIORITY (5,IERR);


MYPRI := IDPRIORITY((GETIDENTIFIER));
    /* SIMULATE GETPRIORITY */
```

IDSTATUS

Obtains a specified task's status.

Format

i := IDSTATUS (identifier)

Arguments

i is an integer variable that receives the task's status as have one of the following values:

0	Ready;
1	Suspended by a .SYSTM call or .TRDOP;
2	Suspended by a .SUSP, .ASUSP, or .TIDS;
3	Waiting for a message to be sent or received;
4	Waiting for an overlay area;
5	Suspended by .ASUSP, .SUSP, or .TIDS and by .SYSTM;
6	Suspended by .XMTW or .REC or by .SUSP, .ASUSP, or .TIDS;
7	Waiting for an overlay area and suspended by .ASUSP, .SUSP, or .TIDS;
8	No task exists with this ID.

identifier is an integer expression that specifies the task's identification number.

Error Conditions

No error code is returned.

Example

```
I := IDSTATUS (124);
```

Reference

.IDST (Task call)

End of Chapter

Chapter 22
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.

Error Conditions

No error code is returned; if no task exists at the priority given, no action is taken.

Example

```
AKILL (4);
```

Reference

.AKILL (Task call)

AREADY

Readies all tasks of a given priority.

Format

```
AREADY (priority);
```

Arguments

priority is an integer expression that gives a task priority number.

Error Conditions

No error code is returned; if no task exists at the priority given, no action is taken.

Example

```
AREADY (PRI);
```

Reference

.ARDY (Task call)

ASUSPEND

Suspends all tasks of a given priority.

Format

```
ASUSPEND (priority);
```

Arguments

priority is an integer expression that gives a task priority number.

Error Conditions

No error code is returned; if no task exists at the priority given, no action is taken.

Example

```
ASUSPEND (2);
```

Reference

.ASUSP (Task call)

KILL

Kills the calling task.

Format

```
KILL;
```

Error Conditions

No error condition can occur.

Example

```
IERR: KILL;
      END;
```

Notes

If a kill-processing address exists for the task, then the task is raised to the highest priority and control goes to this address. Otherwise, control returns to the Task Scheduler, which allocates system resources to the highest priority task that is ready. See the *RDOS Reference Manual* for a discussion of kill processing.

Reference

.KILL (Task 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.

Error Conditions

No error condition can occur.

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 (Task call)

SUSPEND

Suspends the calling task.

Format

```
SUSPEND;
```

Error Conditions

No error condition can occur.

Example

```
SUSPEND;
```

Reference

.SUSP (Task call)

TIDABORT

Kills the 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:

ERTID	You specified a task ID of zero, or no such task ID was found.
ERABT	The task you specified was performing either QTY, MCA I/O, or a system read/write operator message call.

Example

```
TIDABORT (201,IERR);
```

Notes

The task where the call TIDABORT statement appears may kill either itself or some other ready or suspended task.

Outstanding operations performed by the task terminate and all system calls are killed. (See the error ERABT described above for the exceptions.)

TIDABORT does not release open channels used by the killed task, nor does it release any overlays.

Reference

.ABORT (Task call)

TIDKILL

Kills the 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.

Error Conditions

The following error codes may be returned:

ERTID	You specified a task ID of zero, or no such task ID was found.

Example

```
TIDKILL (99,IERR);
```

Reference

.TIDK (Task call)

TIDPRIORITY

Changes the priority of the 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 to 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.

Error Conditions

The following error codes may be returned:

ERTID	You specified a task ID of zero, or no such task ID was found.

Example

```
TIDPRIORITY (100,1,IERR);
```

Reference

.TIDP (Task call)

TIDREADY

Readies the 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 you want to ready.

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

Error Conditions

The following error codes may be returned:

ERTID	You specified a task ID of zero, or no such task ID was found.

Example

```
TIDREADY (ITSK,IERR);
```

Reference

.TIDR (Task call)

TIDSUSPEND

Suspends the 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 you want to suspend.

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

Error Conditions

The following error codes may be returned:

ERTID	You specified a task ID of zero, or no such task ID was found.

Example

```
TIDSUSPEND (55,IERR);
```

Reference

.TIDS	(Task call)

End of Chapter

Chapter 23
Queueing Tasks for Deferred or Periodic Execution

OVQTASK

Queues an overlay task for execution after a specified delay.

Format

```
OVQTASK (procedure,datum,identifier,priority,
  stack size,overlay,flag,start hour,start min
   [,start sec[,rerun count,rerun increment[,time units
   [,error handler[,error label]]]]]);
```

Arguments

procedure is a string expression that specifies the routine or subprogram you want to execute as a task.

datum is an expression whose one-word value passes to the procedure.

identifier is an integer expression that will be associated with the task.

priority is an integer expression that specifies the task's priority level, from 0 (highest) to 255.

stack size is an integer expression that indicates the size of the stack:

Code	Meaning
> 0	Specified size
0	Default size (400_8 words)
-1	Wait for default size
< -1	Queued task waits for stack

The stack includes the global area, extender and end zone.

overlay gives the node number/overlay number in bytes through reference to a symbol defined by RLDR. See Chapter 10 of the *DG/L Reference Manual* for a definition of its use. overlay in bytes.

flag is an integer set to -1 if the overlay containing the program (task-to-be) is to be loaded unconditionally. Zero specifies a conditional loading.

start hour, start min, and *start sec* are integer expressions that specify the delay before execution. A value of -1 for start hour indicates immediate execution.

rerun count is an integer or integer variable that specifies the number of times to execute the task. A value of -1 indicates unlimited execution.

rerun increment is an integer expression that specifies the number of time units between executions of the task.

time units is an integer expression that has one of the following values:

0	Real-Time Clock Frequency
1	Milliseconds
2	Seconds
3	Minutes (less than one day)
4	Hours (less than one day)

error handler is the name of an error procedure that receives control if no stack has been allocated.

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

Error Conditions

The following error codes may be returned:

ERQTS	Illegal information in User Task Queue Table.
ERQOV	.TOVL not loaded for an overlay queued task.
AINUM	Illegal number conversion.

Example

```
OVQTASK (ITSK,10,1,IERR);
```

Reference

.QTSK	(Task call)

QKILL

Removes a queued task from the active queue.

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.

Error Conditions

The following error codes may be returned:

ERTID	You specified a task ID of zero, or no such task ID was found.

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.

Reference

.DQTSK (Task 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 a string expression that specifies the routine or subprogram you want to execute as a task.

datum is an expression whose one-word value passes to the procedure.

identifier is an integer expression whose value will be associated with the task.

priority is an integer expression that specifies the task's priority level, from 0 (highest) to 255.

stack size is an integer that indicates the size of the stack:

Code	Meaning
> 0	Specified size
0	Default size (400_8 words)
-1	Wait for default size
< -1	Queued task waits for stack

The stack includes the global area, extender and end zone.

start hour, start min, and *start sec* are integers or integer variables that specify the delay before execution. A value of -1 for start hour indicates immediate execution.

rerun count is an integer or integer variable that specifies the number of times to execute the task. A value of -1 indicates unlimited execution.

rerun increment is an integer that specifies the number of time units between executions of the task.

time units is an integer that has one of the following values:

0	Real-Time Clock Frequency
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.

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

Error Conditions

The following error codes may be returned:

ERFNO	Illegal channel number.
ERFOP	Attempt to reference an unopened file.
ERQTS	Illegal information in User Task Queue Table.
AINUM	Illegal number conversion.

Example

```
QTASK (ITSK,I,1,12,0,13,24,IERR);
```

Reference

.QTSK (Task call)

End of Chapter

Chapter 24
Using Overlays in a Multitask Environment

TOVKILL

Kills an overlay task and releases the overlay.

Format

TOVKILL (overlay *[,error label]);*

Arguments

overlay gives the node number/overlay number in bytes through reference to a symbol defined by RLDR. See Chapter 10 of the *DG/L Reference Manual* for a definition of its use.

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

Error Conditions

The following error codes may be returned:

EROVN Invalid overlay number.

Example

```
TOVKILL (OVL1,IERR);
```

Notes

You cannot issue this command from the overlay you want to kill and release.

Reference

.OVKIL (Task call)

TOVLOAD

Loads an overlay in a multitask environment.

Format

TOVLOAD (overlay,flag *[,error label]);*

Arguments

overlay gives the node number/overlay number in bytes through reference to a symbol defined by RLDR. See Chapter 10 of the *DG/L Reference Manual* for a definition of its use.

flag is an integer set to -1 if the overlay is to be loaded unconditionally (whether it is resident or not). Zero specifies a conditional loading.

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

Error Conditions

The following error codes may be returned:

EROVN Invalid (non-existent) overlay number.
EROVA Overlay file is not a contiguous file.
ERDTO Ten second disk timeout occurred.

Example

```
TOVLOAD (OVL1,-1,IERR);
```

Notes

You must pair each overlay request with an eventual overlay release or the area will be reserved indefinitely.

Reference

.TOVLD (Task call)

TOVRELEASE

Releases an overlay in a multitask environment.

Format

TOVRELEASE (overlay *[,error label]);*

Arguments

overlay gives the node number/overlay number in bytes through reference to a symbol defined by RLDR. See Chapter 10 of the *DG/L Reference Manual* for a definition of its use.

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

Error Conditions

The following error codes may be returned:

EROVN	Invalid overlay number; the overlay area is not occupied by this user overlay.

Example

```
TOVRELEASE (OVL1,IERR);
```

Notes

You cannot issue this call from the overlay you want to release.

Reference

.OVREL (Task call)

End of Chapter

Chapter 25
Using Memory Management in a Multitask Environment

GALLOCATE

Globally reserves a number of words of memory 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 value signed negative 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.

Error Conditions

The following error codes may be returned:

ERMEM	Insufficient memory.
AICOR	Invalid core error.

Example

```
GALLOCATE (P1,100);
```

Notes

This call searches a chain of free blocks and reserves the first free block or the first free block of a specified size. Failing this, it merges contiguous blocks and takes any that fit. The block consists of size +2 words; offset -1 from the pointer contains the negative of the block's size.

GFREE

Globally frees a block of memory allocated by GALLOCATE in a multitask environment.

Format

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

Arguments

pointer is a pointer expression that points to the first memory address of the block.

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

Error Conditions

The following error codes may be returned:

AICOR	Invalid core error; block does not exist.

Example

```
GFREE (P1);
```

GMEMORY

Obtains the size of the largest, free, global memory block.

Format

```
i := GMEMORY
```

Argument

i is an integer variable that receives the number of words in the largest block.

Error Conditions

No error condition can occur.

Example

```
GALLOCATE (DATA,(GMEMORY),IERR);
```

End of Chapter

Chapter 26
Intertask Communication

You will use the calls in this chapter 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 the program that only one task at a time may execute, 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.

The program might define the message as a 1 to continue execution. The value 0 always suspends execution.

```
TRANSMIT (SYNCH,1);
.
.
.
RECEIVE (SYNCH, TEST);
.
.
.
critical region
.
.
.
TRANSMIT (SYNCH, TEST);
```

In this example, the program sets the value of TEST to 1. When a task enters the critical path, it receives the value 1 in its variable TEST. The message word is then cleared to zero.

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 TEST. 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.

WAITALL and WAITFOR; SIGNAL and CLEAR

These two routines, using the message routines SIGNAL and CLEAR, also suspend a task's execution, but in a more complex way. Instead of a simple zero/non-zero 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 WAITALL suspends will not continue execution until all bits in the signal word 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 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 3, 4, and 5 remain suspended. If task 2 or another executing task 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 will continue execution without waiting. If you again want to stop tasks at WAITALL or WAITFOR, you must provide a call to CLEAR to return the condition word to all zeroes.

WAITALL and WAITFOR do *not,* however, create an interlocking mechanism with SIGNAL and CLEAR. If you want to determine whether the message from SIGNAL was actually received, you must provide calls to TRANSMIT and RECEIVE prior to zeroing the message with CLEAR.

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 to pass the call to WAITFOR but not the call to WAITALL.

CLEAR

Clears the bits in a condition word.

Format

CLEAR (condition word *[,mask]);*

Arguments

condition word is an identifier. The argument cannot be an expression or integer constant.

mask is a 16-bit mask of bits you want cleared. If omitted, all 16 bits will be cleared.

Error Conditions

No error condition can occur.

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.

RECEIVE

Receives a one-word message from another task.

Format

RECEIVE (key name,message);

Arguments

key name is a variable that contains the message.

message is an integer variable that receives the message.

Error Conditions

No error condition can occur.

Example

RECEIVE (AREA9,IMSG);

Notes

After the message is received, it is cleared to all zeroes, barring any other task from executing RECEIVE until the task executes a TRANSMIT.

Reference

.REC (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 an identifier. The argument cannot be an expression or integer constant.

mask is a 16-bit mask of bits you want set. If omitted, all 16 bits will be set.

Error Conditions

No error condition can occur.

Example

SIGNAL (SYNCH,11R2);

Notes

This routine is used with CLEAR, WAITALL, and WAITFOR to synchronize tasks or pass single bits of information between tasks.

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.

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.

Error Conditions

The following error codes may be returned:

ERXMT	Message address is in use.
ERXMZ	Zero message word.

Example

```
TRANSMIT (ITSK10,IMSG,100);
```

Notes

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 one task previously issued a call for the message to RECEIVE, WTRANSMIT will not suspend the caller.

Reference

.XMT (Task 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 an identifier. The argument cannot be an expression or integer constant.

mask is a 16-bit mask you want tested. If omitted, all 16 bits will be tested.

Error Conditions

No error condition can occur.

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.

WAITFOR

Waits for any bit in a condition word signaled by SIGNAL to be true.

Format

WAITFOR (condition word *[,mask]);*

Arguments

condition word is an identifier. The argument cannot be an expression or integer constant.

mask is a 16-bit mask of bits you want tested. If omitted, all 16 bits will be tested.

Error Conditions

No error condition can occur.

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.

WTRANSMIT

Transmits a one-word message to a task and wait 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.

Error Conditions

The following error codes may be returned:

ERXMT	Message address is in use.
ERXMZ	Zero message word.

Example

```
WTRANSMIT (IADDR,IABC,1050);
```

Notes

While TRANSMIT simply deposits a message, WTRANSMIT deposits a message and suspends the caller. However, if one task previously issued a call for the message to RECEIVE, WTRANSMIT will not suspend the caller.

Reference

.XMTW (Task call)

End of Chapter

Chapter 27
Task/Operator Communications in a Multitask Environment

INITOPCOM

Initiates operator communications.

Call Formats

INITOPCOM *[(number of programs, number of queues [,error label])];*

Arguments

number of programs is an integer expression that gives the number of programs the program table can contain.

number of queues is an integer expression that specifies the maximum number of queue tables the queue area array may contain (up to 255).

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

Rules

Use the call without arguments if you are not going to issue the OPCOM commands RUN and QUE/DEQ. The short form call allows the operator to sample and change the status of tasks already existing, but not to initiate tasks. With the call made, the operator may issue commands to OPCOM through the console.

The long form of the call allows an operator to run and queue programs. The full repertoire of OPCOM commands are available once appropriate calls to OPPROG have set up the program table.

Error Conditions

The following error codes may be returned:

ERNOT	No TCB's available.
EROPN	Operator messages not specified.
ERDIU	Directory is in use; OPCOM already initialized.

Example

```
INITOPCOM (IPGNM,IQUE,IERR);

INITOPCOM (IERR);
```

Notes

When the operator issues a RUN or QUE command, OPCOM takes a free queue table from the queue table area and sets it up with information from the RUN or QUE command and from the appropriate program frame (the one corresponding to the program number given in the RUN or QUE command). The task-queueing facility of RDOS then initiates the task either once or many times at appropriate intervals.

You may call INITOPCOM only once, but may retry on errors.

If you plan to use an overlay with a call to OVPROG, you must open the overlay before initializing operator communication (see OVOPEN).

INITOPCOM brings in the error-handling routine ERRFATAL.

Reference

.IOPC (Task call)

OPPROG

Adds a program description to the program table.

Format

OPPROG (program name,program number,datum, identifier,priority,stack size *[,handler [,error label]]);*

Arguments

program name is a string expression that gives the name of a subroutine declared external in the calling program.

program number is an integer expression that gives the number by which the RUN or QUE commands can refer to the program.

datum is an expression whose one-word value passes to the subroutine.

identifier is an integer that is truncated from the left to 8 bits (MOD (identifier, 256) for positive numbers) and stored as part of the program description. When a program is run or queued, this identification number is used for the initiated task unless a task with this number already exists; in that case it uses an number of 0.

priority is an integer expression that is truncated from the left to 8 bits, stored as part of the program description and used to assign an initial priority to the initiated task unless the priority argument is specified in the RUN or QUE command that invokes this program.

stack size is an integer expression that gives the stack size you want to allocate to the program when it executes as a task. If you omit this argument, a default-size stack (400_8 words) is allocated to the task. Stack size includes the global area, extender and end zone.

handler is the name of an error procedure to pass control to if an error occurs when no stack has been allocated. You may specify ERRKILL, KILL, or your own procedure; otherwise ERRFATAL will be used.

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

Error Conditions

The following error codes may be returned:

ERFOP	File not open.
ERMEM	Insufficient memory to execute program.

Example

```
OPPROG (SUBR3,2,45,2,-1,IERR);
```

Notes

Once you make the long form call to INITOPCOM, the operator may begin issuing commands to OPCOM. However, RUN and QUE commands are not meaningful until you set up the appropriate program descriptions in the program table by using OPPROG. You may add as many program descriptions as are specified in *number of programs* in the call to INITOPCOM. Trying to add more than number of programs produces an error condition.

OVPROG

Adds an overlay program description to the program table.

Format

```
OVPROG (program name,program number,datum,
  identifier,priority,stack size,overlay,flag
  [,handler[,error label]]);
```

Arguments

program name is a string that gives the name of a subroutine declared external in the calling program.

program number is an integer that gives the number by which the RUN or QUE commands can refer to the program.

datum is a variable or constant whose one-word value passes to the subroutine.

identifier is an integer that is truncated from the left to 8 bits (MOD (identifier, 256) for positive numbers) and stored as part of the program description. When a program is run or queued, it uses this identification number for the initiated task unless a task with this number already exists; in that case it uses an identifier of 0.

priority is an integer expression that is truncated from the left to 8 bits, stored as part of the program description and used to assign an initial priority to the initiated task unless the priority argument is specified in the RUN or QUE command that invokes this program.

stack size is an integer expression that gives the stack size you want to allocate to the program when it executes as a task. Stack size includes the global area, extender, and end zone.

overlay gives the node/overlay number in bytes in which the program (task-to-be) resides. You must declare this name external in the program calling OVPROG and declare it in an overlay statement in the subroutine that resides in the overlay. See Chapter 10 of the *DG/L Reference Manual* for directions for referencing overlays.

flag is an integer expression set to -1 if the overlay is to be loaded unconditionally (whether it is resident or not). Zero specifies a conditional loading.

handler is the name of an error procedure to pass control to if an error occurs when no stack has been allocated. You may specify ERRKILL, KILL, or your own routine; otherwise, ERRFATAL will be used.

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

You must open the overlay file with a call to OVOPEN before making the call to INITOPCOM, or OVPROG will return an error. It is your responsibiliy to release the overlay area when the task is done executing.

Error Conditions

The following error codes may be returned:

ERFOP	File not open.
ERMEM	Insufficient memory to execute program.

Example

```
OVPROG (SUBR3,2,45,2,OV1,-1,IERR);
```

Notes

Once you make the long form call to INITOPCOM, the operator may begin issuing commands to OPCOM. However, RUN and QUE commands are not meaningful until you set up the appropriate program descriptions in the program table by using OPPROG. You may add as many program descriptions as are specified in number of programs in the call to INITOPCOM. Trying to add more than number of programs produces an error condition.

TRDOPERATOR

Reads a task message from the console.

Format

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

Arguments

s is a string variable that receives the message.

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

Error Conditions

The system may output two diagnostic messages to the console to indicate errors in messages intended for tasks:

Message	Meaning
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 error codes that may be returned at error label are:

ERNOT	No TCB's available.
ERMPR	Address is outside address space (mapped systems only).
EROPM	Operator messages not specified at SYSGEN time.

Example

```
IMESS := TRDOPERATOR (IERR);
```

Notes

A task issuing a call to TRDOPERATOR may reside in either the foreground or the background; more than one task within a program may issue an outstanding request for a task message.

Reference

.TRDOP (Task call)

TWROPERATOR

Writes a message to the console.

Format

TWROPERATOR (message *[,flag[,error label]]);*

Arguments

message is an string expression up to 128 characters long including a terminator.

flag is an integer value of 0 if you want the task's identification number printed with the message, and -1 or omitted if you want it suppressed.

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

Error Conditions

The following error codes may be returned:

ERMPR	Address is outside address space (mapped systems only).
EROPM	Operator messages not specified at SYSGEN time.

Example

```
TWROPERATOR ("FLAG IS 10",1,IERR);
```

Notes

The message appears in this form: first, two exclamation points and an F or a B indicating whether the calling task is in the foreground or background. Following this is the calling task's identification number if you set *flag* to 1, and the message itself.

Reference

.TWROP (Task call)

End of Chapter

Chapter 28
User/System Clock Commands

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 you want to suspend the task.

time units is an integer expression that indicates the unit of measurement with one of the following values:

0	Real-Time Clock Frequency
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.

Error Conditions

The error codes that may be returned at error label are:

ERDNM	Device not in system, or illegal device code.
ERIBS	No user clock defined.
AINUM	Illegal number conversion.

Example

```
DELAY (IPULS,0);
```

Reference

.DELAY (System call)

GETFREQUENCY

Gets the real-time clock frequency.

Format

```
i := GETFREQUENCY
```

Arguments

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

Code	Meaning
0	There is no real-time clock in this system.
1	Frequency is 10 HZ.
2	Frequency is 100 HZ.
3	Frequency is 1000 HZ.
4	Frequency is 60 HZ (line frequency).
5	Frequency is 50 HZ (line frequency).

Error Conditions

No error condition can occur.

Example

```
I := GETFREQUENCY;
```

Reference

.GHRZ (System call)

TUSERCLOCK

Defines a location to count clock pulses.

Format

TUSERCLOCK *[(clock location,time,time units [,error label])];*

Arguments

clock location is an integer variable that will contain the pulse count.

time is an integer expression that specifies the number of time units between clock pulses.

time units is an integer expression that indicates the unit measurement with one of the following values:

0	Real-Time Clock Frequency
1	Milliseconds
2	Seconds
3	Minutes

Giving no argument is the same as a zero.

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

Error Conditions

The following error codes may be returned:

ERDNM	Device not in system, or illegal device code.
ERIBS	No user clock defined.
AINUM	Illegal number conversion.

Example

```
TUSERCLOCK (CLOCK,100,1);
   /* INCREMENT EACH .1 SEC */
```

Notes

A call to TUSERCLOCK without arguments removes a current user clock.

References

.UCEX	(System call)
.RUCLK	(System call)
.DUCLK	(System call)

USERCLOCK

Defines a location to count clock pulses.

Format

USERCLOCK *[(clock location,time,time units [,error label])];*

Arguments

clock location is an integer variable that will contain the pulse count.

time is an integer expression that specifies the number of time units between clock pulses.

time units is an integer expression that indicates the unit measurement with one of the following values:

0	Real-Time Clock Frequency
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.

Error Conditions

The following error codes may be returned:

ERDNM	Device not in system, or illegal device code.
ERIBS	No user clock defined.
AINUM	Illegal number conversion.

Example

```
USERCLOCK (CLOCK,1,2);
   /* INCREMENT EACH SECOND */
```

Notes

A call to USERCLOCK without arguments removes a current user clock.

References

.UCEX	(System call)
.RUCLK	(System call)
.DUCLK	(System call)

End of Chapter

Chapter 29
Disabling and Enabling the Multitask Environment

MULTITASK

Enables the multitask environment.

Format

MULTITASK;

Error Conditions

No error condition can occur.

Notes

Task initiation automatically enables the multitask environment. Use this call to re-enable multitasking after a call to SINGLETASK.

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.

SINGLETASK

Disables the multitask environment.

Format

SINGLETASK;

Error Conditions

No error condition can occur.

Notes

Task initiation automatically enables the multitask environment. This call gives the calling task sole control of the CPU. It loses singletask status if it suspends itself for anything but a system call, or if it dies. To re-enable the multitask environment, use the call to MULTITASK.

End of Chapter

Appendix A
Alphabetic List of DG/L Runtime Routines

Routine	Function	Chapter
ACCESS	Opens a file for cache memory management, associates it with a buffer pool, and defines its element size.	13
ADDRESS	Obtains the address of a referenced datum.	2
AKILL	Kills all tasks of a given priority.	22
ALLOCATE	Reserves a number of words of memory, obtains their location, and initializes the words to zero.	2
APPEND	Opens a file for appending writing at its end.	11
AREADY	Readys all tasks of a given priority.	22
ARGCOUNT	Obtains the number of arguments that the current procedure received.	2
ASCII	See BYTE.	
ASUSPEND	Suspends all tasks of a given priority.	22
ATTRIBUTE	Obtains the attributes of an opened file.	10
BLKREAD	Reads a series of blocks from a randomly or contiguously organized disk file.	11
BLKWRITE	Writes a series of blocks to a randomly or contiguously organized disk file.	11
BOOT	Boots a new system, partition, or device.	18
BUFFER	Creates a buffer pool area in memory and returns its address.	13
BYTE	Obtains the integer value of a byte in a datum.	2
BYTEREAD	Reads a specified number of bytes from a file into memory.	11
BYTEWRITE	Writes a specified number of bytes from memory to a file.	11
CBIT	Clears a bit in a bit string.	4
CCONT	Creates a contiguous file of a specified length and initializes all words to zero.	10
CDIR	Creates a subdirectory.	9

Routine	Function	Chapter
CHAIN	Terminates execution of the current program segment and calls another disk file for execution.	15
CHATR	Changes the resolution attributes of an opened file.	10
CHLAT	Changes link access attributes of an opened file.	10
CHSTATUS	Obtains the current directory status for an opened file.	10
CLASSIFY	Obtains an integer value within a table of ranges.	2
CLEAR	Clears the bits in a condition word.	26
CLOSE	Closes a file.	11
COMARG	Reads command line operands and switches.	6
CONSOLE	Writes to the console without using the stack.	12
CPART	Creates a secondary partition.	9
CRAND	Creates a random file of zero length.	10
CREATE	Creates a sequential file of zero length.	10
DATACLOSE	Closes a file opened for word buffering.	11
DATAOPEN	Opens a file at its beginning for word buffering.	11
DATAREAD	Reads data from a file into a specified area of memory.	11
DATAWRITE	Writes a specified number of words from memory to a file.	11
DELAY	Suspends a task for a specified number of clock pulses.	28
DELETE	Deletes a disk file.	10
DIM	Determines the width of an array dimension.	5
DIR	Changes the current default directory.	9
DUCLK	Initializes a non-DG/L clock service routine and specifies the intervals.	18
EOPEN	Exclusively opens a file at its beginning.	11
EQUIV	Assigns a temporary name to a device, permitting unit independence during execution of your program.	9
ERETURN	Terminates a program and indicates an error.	8, 15

Routine	Function	Chapter
ERPRINT	Prints out the last error message if it is not yet printed.	8
ERRFATAL	Prints an error message without using the stack and returns to the system. If the error was posted, print the full message.	8
ERRINTERCEPT	Intercepts errors and calls an error handling routine.	8
ERRKILL	Prints an error message and kills the calling task.	8
ERROR	Creates a message for the DG/L error handler to write.	8
ERRTRAP	Traps errors without returning to the system.	8
ERRUSER	Calls the error handler with an error code.	8
EXBG	Checkpoints a mapped background program; suspends one background program temporarily in order to execute a background program of a higher priority.	17
EXFG	Loads and executes a program in the foreground.	17
FETCH	Retrieves a word from a node array on file zero.	13
FILEPOSITION	Computes the position of a file.	11
FILESIZE	Computes the current size of a file in bytes.	10
FLUSH	Writes out the contents of all modified buffers in the pool to disk.	13
FOREGROUND	Determines whether or not a foreground program is running.	17
FPUERROR	Returns boolean and string values for FPU errors.	8
FREE	Frees an allocated block of words and links it into the free chain.	2
GALLOCATE	Reserves globally a number of words of memory and returns a pointer in a multitask environment.	25
GCHANNEL	Obtains the number of an available channel.	10
GETCHAR	Reads a single character from the operator's console without echoing it.	12
GETCINPUT	Obtains the input console name.	12
GETCOUTPUT	Obtains the output console name, making a program portable.	12

Routine	Function	Chapter
GETDIR	Obtains the name of the current directory.	9
GETFREQUENCY	Obtains the real time clock frequency.	28
GETGROUND	Determines if the current program is in the foreground or background.	17
GETIDENTIFIER	Obtains the calling task's identification number.	21
GETLIST	Obtains the name of the current list file.	2
GETMCA	Gets the unit number of the MCA.	18
GETMCA1	Gets the unit number of the secondary MCA.	18
GETMDIR	Obtains the name of the current master device.	9
GETPRIORITY	Obtains the calling task's priority.	21
GETSYS	Obtains the current system name.	9
GFREE	Globally frees a block of memory allocated in a multitask environment.	25
GMEMORY	Obtains the size of the largest free global memory block.	25
GTIME	Obtains the current time.	7
HASHBACK	Brings the last used cache memory block back into memory.	13
HASHREAD	Reads a specified block of the hash file into memory.	13
HASHWRITE	Marks the current buffer as modified.	13
HBOUND	Obtains the upper bound of an array dimension.	5
ICOMMON	Defines a message area in program space for interground communication.	17
IDPRIORITY	Obtains a specified task's priority.	21
IDSTATUS	Obtains a specified task's status.	21
IINFOS	Initializes an INFOS volume.	14
INFOS	Calls the INFOS system.	14
INIT	Initializes a directory, partition, or device.	9
INITOPCOM	Initiates operator communications.	27
INDEX	Searches for a pattern of characters or bits in a string and obtains the position of the first character or bit.	3
.ISUBSTR	See SUBSTR.	

Routine	Function	Chapter
KILL	Kills the calling task.	22
LBOUND	Obtains the lower bound of an array dimension.	5
LENGTH	Obtains the current length of a string variable.	4
LINEREAD	Reads a line from a file into memory.	11
LINEWRITE	Writes a line to a file.	11
LINK	Creates a link entry in the current directory to a file in the same or in another directory.	10
MEMORY	Obtains the number of remaining words of storage available to the user.	2
MINRES	Declaration for specifying, modifying, or otherwise accessing the default lower bound of the cache memory file.	13
MTDIO	Performs free-form I/O to or from a magnetic tape or cassette unit.	11
MTOPD	Opens a magnetic tape or cassette unit for free-form I/O.	11
MULTITASK	Enables the multitask environment.	29
NAMEGROUND	Returns an identifier with an indication whether it is in the foreground.	17
NCONT	Creates a contiguous file of a specified length without initializing all words to zero.	10
NODEREAD	Reads a specified node from file zero into memory.	13
NODESIZE	Obtains the size of a node on file zero.	13
NODEWRITE	Writes a node from memory to file zero.	13
NOFPUTRAP	Declaration for ignoring floating point errors.	8
NOMESSAGE	Declaration for removing error messages and the error reporter from storage.	8
ODIS	Disables console interrupts.	12
OEBL	Enables console interrupts.	12
OINFOS	Opens an INFOS file.	14
OPEN	Opens a file.	11
OPPROG	Adds a program description to the program table.	27
OVCLOSE	Closes an overlay file in a singletask environment.	16

Routine	Function	Chapter
OVLOD	Loads an overlay file in a singletask environment.	16
OVOPN	Opens an overlay file in a singletask environment.	16
OVPROG	Adds an overlay program description to the program table.	27
OVQTASK	Queues an overlay task for execution after a specified delay.	23
PINFOS	Pre-opens an INFOS file.	14
POSITION	Repositions an open file.	11
PRIORITY	Changes the priority of a calling task.	22
PUTCHAR	Writes a single character to the console.	12
QKILL	Removes a queued task from the active queue.	23
QTASK	Queues a task for execution after a specified delay.	23
RANDOM	Obtains a random number from a pseudo-random sequence of integers in the range of 0 to (2 ↑ 16) - 1.	2
RCOMARG	Rewinds the file of command arguments to make the next COMARG argument number zero.	6
RDCOMMON	Reads a message from the other ground's communications area.	17
RDOPERATOR	Reads a message from the console.	17
RDSW	Reads the CPU console data switches.	2
READCHAR	Reads and echoes a single character typed from the console.	12
READERROR	Checks and reads information about errors.	8
RECEIVE	Receives a one-word message from another task.	26
RELEASE	Releases a directory, partition, or device from current use.	9
REM	Divides two integers and obtains an integer result and remainder.	3
RENAME	Changes the name of a file.	10
ROPEN	Opens a file for read access only.	11
ROTATE	Rotates a word a specified number of bit positions.	3

Routine	Function	Chapter
RUNTASK	Executes a routine as a task.	20
SBIT	Sets a bit in a bit string to 1.	4
SEED	Provides an initial number on which to base random number generation.	2
SETCURRENT	Sets the length of a string or bit variable.	4
SHIFT	Shifts a word a specified number of bit positions.	3
SHORTMESSAGE	Declaration for printing out short error messages.	8
SIGNAL	Signals a set of conditions.	26
SINGLETASK	Disables the multitask environment.	29
SIZE	Determines the number of elements in an array or the declared length of a string.	3, 5
STASH	Stores a word in a node array on file zero.	13
STATUS	Obtains the current directory status of a specified file.	10
STIME	Sets the time.	7
SUBSTR	Obtains a substring of a character or bit string or substring, or of an integer.	4
SUSPEND	Suspends the calling task.	22
SWAP	Transfers control to a program on disk file.	15
SYSRETURN	Terminates a program with no error.	15
SYSTEM	Provides a generalized interface to the system, allowing you to use all RDOS system calls.	2
TASK	Initiates a task and allocates storage for it.	20
TBIT	Tests a bit in a bit string.	4
TIDABORT	Kills the task specified by an identification number.	22
TIDKILL	Kills the task specified by an identification number.	22
TIDPRIORITY	Changes the priority of the task specified by an identification number.	22
TIDREADY	Readys the task specified by an identification number.	22
TIDSUSPEND	Suspends the task specified by an identification number.	22

Routine	Function	Chapter
TOVKILL	Kills an overlay task and releases the overlay.	24
TOVLOAD	Loads an overlay in a multitask environment.	24
TOVRELEASE	Releases an overlay in multitask environment.	24
TRANSMIT	Transmits a one-word message to a task.	26
TRDOPERATOR	Reads a task message from the console.	27
TUSERCLOCK	Defines a location to count clock pulses.	28
TWROPERATOR	Writes a message to the console.	27
UMUL	Multiplies two unsigned integers, adds another, and obtains the result and the overflow.	3
UNLINK	Deletes a link entry.	10
USERCLOCK	Defines a location to count clock pulses.	28
USUBSTR	See SUBSTR.	
WAITALL	Waits for all bits in a condition word signalled by SIGNAL to be true.	26
WAITCHAR	Waits for a character from the console (RTOS).	12
WAITFOR	Waits for any bit in a condition word signalled by SIGNAL to be true.	26
WORDREAD	Reads words from a file by using cache modules.	13
WORDWRITE	Writes words to a file by using cache modules.	13
WRCOMMON	Writes a message into the other ground's communications area.	17
WROPERATOR	Writes a message to the console.	17
WTRANSMIT	Transmits a one-word message to a task and waits for it to be received.	26
XCT1	Executes an instruction on the assembly level.	2
XCT2	Executes a two-word instruction on the assembly level.	2

End of Appendix

Appendix B
Alphabetic List of Mathematical Runtime Routines

Routine	Function
.ABS	Computes absolute value of single-precision real argument.
.AINT	Truncates fractional portion of single-precision real argument.
.AMAX	Selects largest single-precision number from list of single-precision real arguments.
.AMIN	Selects smallest single-precision number from list of single-precision real arguments.
.AMOD	Calculates remainder of single-precision quotient of single-precision real arguments.
.ASAC	Computes single-precision arcsine or arccosine of single-precision real argument.
.ATAN	Computes single-precision arctangent of single-precision real argument.
.ATAN2	Computes single-precision arctangent of quotient of two single-precision real arguments.
.DABS	Computes absolute value of double-precision real argument.
.DASAC	Computes double-precision arcsine or arccosine of double-precision real argument.
.DATAN	Computes double-precision arctangent of double-precision real argument.
.DATN2	Computes double-precision arctangent of quotient of two double-precision real arguments.
.DBLE	Converts single-precision real argument to double-precision real number.
.DEXP	Computes double-precision exponential for a double-precision real argument.
.DINT	Truncates fractional portion of double-precision real argument.
.DLOG	Computes double-precision natural logarithm of double-precision real argument.
.DLOG10	Computes double-precision base 10 logarithm of double-precision real argument.
.DMAX	Selects largest double-precision number from list of double-precision real arguments.
.DMIN	Selects smallest double-precision number from list of double-precision real arguments.
.DMOD	Calculates remainder of double-precision real quotient of two double-precision real arguments.
.DPCMP	Compares equality/inequality relationship of two double-precision real arguments, depending on the entry: .DLE, .DGE, .DLT, .DGT, .DNE, and .DEQ.
.DPOWR	Raises a double-precision real number to a double-precision power.
.DPURN	Interfaces to double-precision real floating point unit for function given by the entry: .DLF Load .DSF Store .DAD Add .DSB Subtract .DML Multiply .DDV Divide .DNG Negate .DAV Make positive

Routine	Function
.DPWRI	Raises double-precision real number to single-precision integer power.
.DSGN	Multiplies double-precision real argument by sign of another double-precision real argument.
.DSHCH	Computes double-precision hyperbolic sine or cosine of double-precision real argument.
.DSNCS	Computes double-precision sine or cosine of double-precision real argument.
.DSQRT	Computes double-precision square root of double-precision real argument.
.DTAN	Computes double-precision tangent of double-precision real argument.
.DTANH	Computes double-precision hyperbolic tangent of double-precision real argument.
EXP	Computes single-precision exponential of single-precision real argument.
FIX	Converts single-precision real argument to integer by truncation.
FLOAT	Converts single-precision integer argument to single-precision real number.
.IABS	Computes absolute value of single-precision integer argument.
.IDIV	Signed division of AC1 by AC2.
.IEV	Single-precision integer equivalence.
.IIM	Single-precision integer implication.
.IMOD	Calculates remainder of single-precision integer quotient of two single-precision integer arguments.
.IMUL	Signed multiplication of AC1 by AC2.
.IPWRI	Raises a single-precision integer number to a single-precision integer power.
.ISGN	Multiplies a single-precision integer argument by the sign of another single-precision integer argument.

Routine	Function
.JAB	Double-precision integer absolute value.
.JAD	Double-precision integer add.
.JAN	Double-precision integer and.
.JDV	Double-precision divide.
.JEQ	Double-precision integer equal.
.JEV	Double-precision integer equivalence.
.JGE	Double-precision integer greater than or equal.
.JGT	Double-precision integer greater than.
.JIM	Double-precision integer implication.
.JLE	Double-precision integer less than or equal.
.JLT	Double-precision integer less than.
.JML	Double-precision integer multiply.
.JMV	Moves double-precision integers.
.JNE	Double-precision integer not equal.
.JNG	Double-precision integer negate.
.JNT	Double-precision integer not.
.JOR	Double-precision integer or.
.JSB	Double-precision integer subtract.
.JXO	Double-precision integer exclusive or.
.LEQ	Logical equivalence.
.LIM	Logical implication.
LN	Computes natural logarithm of argument.
LOG10	Computes base 10 logarithm of argument.
MAX	Selects largest integer from list of single-precision integer arguments.
MIN	Selects smallest integer from list of single-precision integer arguments.

Routine	Function
.POWR	Raises a single-precision real number to a single-precision power.
.PWRI	Raises a single-precision real number to a single-precision integer power.
.SGN	Multiplies single-precision real argument by sign of another.
SIGN	Transfers sign of one single-precision real argument to another single-precision real argument.
.SNCS	Computes single-precision sine or cosine of single-precision real argument.
.SNGL	Converts double-precision real argument to single-precision real number.
SQRT	Computes single-precision square root of single-precision real argument.
TAN	Computes single-precision tangent of single-precision real argument.
TANH	Computes single-precision hyperbolic tangent of single-precision real argument.

Routine	Function
.TGE	String comparison routine.
.TLT	String comparison routine.
.TLE	String comparison routine.
.TGT	String comparison routine.
.TNE	String comparison routine.
.TEQ	String comparison routine.
ZRCMP	Title of binary file containing the following routines which compare either a single-precision or double-precision real number to zero.

	Single	Double
arg < 0	.FNP	.DNP
arg > 0	.FPZ	.DPZ
arg < 0	.FLZ	.DLZ
arg > 0	.FGZ	.DGZ
arg $= 0$	.FNZ	.DNZ
arg $= 0$	.FEZ	.DEZ

End of Appendix

Appendix C
Conversion Routines

Name	Function
.BLJ	Boolean to double-precision integer.
.CONCAT	Concatenates two strings, yielding result string.
.DBJ	Double-precision real to double-precision integer.
.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.

Name	Function
.RLJ	Real to double-precision integer.
.RLST	Real to string.
.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.
.STREAL	String to real.
.STDBL	String to double-precision real.
.TMV	Moves DG/L string to a DG/L string.
.TMVN	Moves string ending with a null to a DG/L string.

End of Appendix

Appendix D
Formatting Routines

FCHAR
Entry: .FCHAR

Purpose
To insert a character into the format output buffer.

Call
```
JSR   @.FCHAR
AC0   right adjusted character
      (AC0 = -1 forces out the buffer)
```

Notes
See XCHAR routine, FCHAR and XCHAR are alternative routines.

FOPB
Entry: .FCHAR

Purpose
To format output of a boolean value.

Call
```
JSR   @FOPB
AC1   address of boolean value
```

FOPD
Entry: .FOPD

Purpose
To format output of a double-precision real value.

Call
```
JSR   @.FOPD
AC1   address of double-precision real value
```

FOPI
Entry: .FOPI

Purpose
To format output of an integer value.

Call
```
JSR   @.FOPI
AC1   address of integer
```

FOPJ
Entry: .FOPJ

Purpose
To format output of a double-precision integer value.

Call
```
JSR   @.FOPJ
AC1   address of double-precision integer
```

FOPP
Entry: .FOPP

Purpose
To format output of a pointer value.

Call
```
JSR   @.FOPP
AC1   address of pointer
```

FOPR
Entry: .FOPR

Purpose
To format output of a single-precision real value.

Call
```
JSR   @.FOPR
AC1   address of single-precision real value
```

FOPS
Entry: .FOPS

Purpose
To format output of a character string.

Call
```
JSR   @.FOPS
AC1   address of string specifier/substring specifier
```

FOPU
Entry: .FOPU

Purpose
To format output of a bit string.

Call
```
JSR   @.FOPU
AC1   address of bit string specifier/substring specifier
```

FWRB
Entry: .FWRB

Purpose
To format writing of a boolean value.

Call
```
JSR   @.FWRB
AC1   address of boolean value
```

FWRD
Entry: .FWRD

Purpose
To format writing of a double-precision real number.

Call
```
JSR   @.FWRD
AC1   address of double-precision real
```

FWRI
Entries: .FWRI, .FWRP

Purpose
To format writing of a single-precision integer or pointer.

Call
```
JSR   @.FWRI(.FWRP)
AC1   address of integer(pointer)
```

FWRJ
Entry: .FWRJ

Purpose
To format writing of a double-precision integer.

Call
```
JSR   @.FWRJ
AC1   address of double-precision integer
```

FWRR
Entry: .FWRR, .FWRD

Purpose
To format writing of a single-precision or double-precision real number.

Call
```
JSR   @.FWRR (.FWRD)
AC1   address of single-precision (double-precision)
      real
```

FWRS
Entry: .FWRS

Purpose
To format writing of a string datum.

Call
```
JSR   @.FWRS
AC1   address of string specifier/substring specifier
```

FWRU
Entry: .FWRU

Purpose
To format writing of a bit string datum.

Call
```
JSR   @.FWRU
AC1   address of a bit string specifier/substring
      specifier
```

IFLE
Entry: .IFLE

Purpose
To initialize parameters for an unformatted read.

Call
```
JSR   @.IFLE
AC1   address of file number
AC0   address of error label specifier (optional)
```

IFMT
Entry: .IFMT

Purpose
To initialize parameters for formatted output.

Call
```
JSR     @.IMFT
AC1     address of file number
AC0     address of format string specifier
IARG2   address of error label specifier (optional)
```

Notes
Format must be a literal or a string ending with null.

IWRT
Entry: .IWRT

Purpose
To initialize parameters for write output.

Call
```
JSR   @.IWRT
AC1   address of file number
AC0   address of error label (optional)
```

TFLE
Entry: .TFLE

Purpose
To terminate the current unformatted read.

Call
```
JSR @.TFLE
```

TFMT
Entry: .TFLE

Purpose
To terminate the current formatted write.

Call
```
JSR @.TFMT
```

TFOP
Entry: .TFOP

Purpose
To terminate the current formatted output.

Call
```
JSR @.TFOP
```

TWRT
Entry: .TWRT

Purpose
To terminate the current write.

Call
```
JSR @.TWRT
```

URDB
Entry: .URDB

Purpose
To do an unformatted read of a boolean.

Call
```
JSR   @.URDB
AC1   address of boolean datum
```

URDD
Entry: .URDD

Purpose
To do an unformatted read of a double-precision real number.

Call
```
JSR   @.URDD
AC1   address of double-precision real number.
```

URDI
Entry: .URDI

Purpose
To do an unformatted read of an integer.

Call
```
JSR   @.URDI
AC1   address of integer
```

URDJ
Entry: .URDJ

Purpose
To do an unformatted read of a double-precision integer.

Call
```
JSR   @.URDJ
AC1   address of double-precision integer
```

URDP
Entry: .URDP

Purpose
To do an unformatted read of a pointer.

Call
```
JSR   @.URDP
AC1   address of pointer
```

URDR
Entry: .URDR

Purpose
To do an unformatted read of a single-precision real number.

Call
```
JSR   @.URDR
AC1   address of single-precision real number
```

URDS
Entry: .URDS

Purpose
To do an unformatted read of a string datum.

Call
```
JSR   @.URDS
AC1   address of string specifier
```

URDU

Entry: .URDU

Purpose

To do an unformatted read of a bit string datum.

Call

```
JSR   @.URDU
AC1   address of bit string specifier
```

XCHAR

Entry: .FCHAR

Purpose

To insert a character into the format output buffer.

Call

```
JSR   @.FCHAR
AC0   right-adjusted character
      (AC0 = -1 forces out the buffer)
```

Notes

XCHAR and FCHAR are alternative routines. XCHAR differs in that: it maintains the value LIST; it does not output when LIST is False and the routine maintains an output character CRCNT.

LIST and CRCNT should be declared as External Boolean and External Integer, respectively.

End of Appendix

Appendix E
DG/L Runtime Errors

The following table lists all DG/L runtime error codes 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. See routines such as ERRFATAL for the numeric equivalents of ECTRT, ECSAF, and ECSCB. Chapter 8 documents error-handling and error-reporting routines.

Mnemonic	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
AISIZ	11+ECTRT	ILLEGAL SIZE
AISTK	12+ECTRT	STACK OVERFLOW
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 F
DG/L Runtime Routines Implemented under Both RDOS and AOS

This appendix lists all routines in this manual that will run under DGC's Advanced Operating System (AOS) as well as RDOS. To write DG/L programs that can run under either system, use only these runtime calls. For additional calls available only under AOS, see the AOS edition of this manual.

Routine	Chapter
ACCESS	13
ADDRESS	2
AKILL	22
ALLOCATE	2
APPEND	11
AREADY	22
ARGCOUNT	2
ASCII	2
ASUSPEND	22
ATTRIBUTE	10
BLKREAD	11
BLKWRITE	11
BUFFER	13
BYTE	2
BYTEREAD	11
BYTEWRITE	11
CBIT	4
CCONT	10
CDIR	9
CHAIN	15
CLASSIFY	2
CLEAR	26
CLOSE	11
COMARG	6
.CONSOLE	12
CPART	9
CRAND	10
CREATE	10
DATACLOSE	11
DATAOPEN	11
DATAREAD	11
DATAWRITE	11
DELAY	28
DELETE	10
DIM	5
DIR	9
EOPEN	11
ERETURN	8, 15
ERPRINT	8
ERRFATAL	8
ERRINTERCEPT	8
ERRKILL	8
ERROR	8
ERRTRAP	8
ERRUSER	8
FETCH	13
FILEPOSITION	11
FILESIZE	10
FLUSH	13
FPUERROR	8
FREE	2
GALLOCATE	25
GCHANNEL	10
GETCINPUT	12
GETCOUTPUT	12
GETDIR	9
GETFREQUENCY	28
GETIDENTIFIER	21
GETLIST	2
GETPRIORITY	21
GFREE	25
GMEMORY	25
GTIME	7
HASHREAD	13
HASHWRITE	13
HBOUND	5
IDPRIORITY	21
IDSTATUS	21
INIT	9
INDEX	3
.ISUBSTR	4
KILL	22
LBOUND	5
LENGTH	4
LINEREAD	11
LINEWRITE	11
LINK	10

End of Appendix

Appendix G
DG/L Tables and Files

The tables and files in this appendix contain information that allows you to address error information, alter the runtime environment, and define assembly modules for a DG/L program.

Process Global Table

The Process Global Table contains information globally pertinent to a DG/L program or process. The page zero pointer .PT points to word zero of the table. The decimal displacements of the contents in this list are in ascending order from .PT.

Word	Mnemonic	Contents
0	GTFREE	Address of start of the global free chain (see GALLOCATE, GFREE).
1	MSKEY	not in Rev 3.4
2	UCKEY	Location of a user-defined clock (see TUSERCLOCK, USERCLOCK, DUCLK).
3	DSTK	Default stack size for user tasks (currently 200 (decimal) words; see TASK).
4	CONV1	Conversion constant for 1000HZ to system units.
5	CONV2	Conversion constant for seconds to system units.
6	CONV3	Conversion constant for minutes to system units.
7	CNTFL	Greatest channel number allowed.
8	WLIST	Save area for LIST value during formatted write.
9	VRNUM	Next free virtual (mapped) buffer.
10	VRADDR	Address of virtual cache memory buffer.
11	VRRTN	Save area for return pointer for .BUFMAP in .VBUFMAP (virtual only).
12	VRBUF	Save area for current buffer in .BUFMAP in .VBUFMAP (virtual only).
13	GMERG	GALLOCATE remerge indicator, used for communication between GALLOCATE and GFREE.
14	OPKEY	INITOPCOM key--0 if in use, -1 if not.
15	AWAIT	GALLOCATE wait key-- -1 if a task is waiting for memory (multitask only).
15	DATBF	Free data buffer chain (singletask only). See DATACLOSE, DATAOPEN, DATAREAD, DATAWRITE.
16	VRLCK	Cache memory buffer lock for multitasking; only one task is allowed to do CMM.
17	VRMAP	Previous block in window map--indicates whether remapping is necessary.
18	OVCHAN	Overlay channel number.
19	FTLCK	Table of file locks to indicate if individual channel numbers are in use-- CNTFL entries long.

The CNTFL words before .PT are FTBUF, a table of pointers to the cache memory buffers associated with each file number.

The CNTFL words before FTBUF are FTLBZ and FTWRL for each cache memory file: FTLBZ contains the element size and FTWRL indicates whether the associated file requires write line commands.

Runtime Global Area

The Runtime Global Area contains information specific to each task in a DG/L program. In the singletask environment, it occupies the locations immediately below the Process Global Table. In multitasking, it is allocated dynamically as tasks become active.

The decimal displacement in this list of its contents are given in ascending order from the page zero pointer .GP.

Word	Mnemonic	Contents
0	FIOCB	Pointer to current formatted I/O control block, chained to previously nested IOCBs.
1	AFREE	Start of the free chain (see ALLOCATE, FREE).
2	GPERB	Pointer to chain of ERRTRAP/ERRINTERCEPT declaration blocks (-1 if there is none).
3	GPERR	Error code from last detected error.
4	GPETX	Word address of last error message text.
5	GPECL	Address of caller of routine which caused error.
6	GPERA	Address of runtime routine which caused or detected the error.

Assembly Language Files

DG/L provides three files for use in creating external routines in assembly language.

DGLSYM.SR

This file includes definitions of all the DG/L runtime tables, symbolic arguments, symbolic error codes, switches, and fixed areas. See Figure G-1.

DGLMAC.SR

This file contains macro definitions that eliminate repetitive typing of code (see, for example, the macro TITLE) and allow you to use different hardware and operating systems with source file compatibility. See Figure G-2.

DGLPARAM.SR

This file supplies prototype assembly definitions of locations and symbols you can modify to effect changes in the actual runtime environment of your programs. See Figure G-3.

Using these symbols and macros, you will be able to create assembly language files that are readable and easy to maintain.

Changing Values in the Symbol Table

You may want to modify certain default values in the symbol table to meet your programming needs. By changing .DSTACK, you can specify a new default-size stack for multitasking; by changing .NMAX, you can reduce the size of the user stack that the initializer takes. You can alter the CMM values for least and maximum number of virtual buffers, .VMIN and .VMAX. Under RDOS, you have three methods available for changing values in the symbol table:

1. 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 highest address available for user memory.

   Assemble the module as you would an external variable, using MAC and RLDR. The resulting file will have an .SV extension.

2. Use OEDIT or SEDIT and the RLDR load map to locate the variable and change the value.

3. Use the debugger to change the file.

   .NMAX/ current value new value

   The system will display the current value; you key in the new one.

Remember that if you edit the symbol file, the new value will remain in effect until you modify it again.

```
;                ****************************
;                *                          *
;                * DG/L SYMBOL DEFINITIONS  *
;                *                          *
;                ****************************

                 .TITLE          DGLSYM

; MISCELLANEOUS SWITCHES

**       .NOCON  1
**       .IFE    .PASS
**       .IFE    ANSW!ABSW
**       AOSW=   1
**       IRMSW=  AOSW
**       OBSW=   1
**       ABSW=   1
**       BSW=    1
**       MBSW=   1
**       .ENDC
**       .IFE    AOSW
**       AOSW=   0
**       IRMSW=  AOSW
**       .LOC    1
**       .NREL   1
**       OBSW=   .
**       .ZREL
**       .NREL   1
**       .IFN    (OBSW<>.)
**       OBSW=   0
**       .ENDC   X
**       OBSW=   1
**[X]
**.ENDC
**.ENDC
ASW=     AOSW            ; AOS ASSEMBLY SWITCH
RSW=     1-ASW           ; RDOS ASSEMBLY SWITCH
ESW=     ABSW!AOSW       ; ECLIPSE ASSEMBLY SWITCH
NSW=     1-ESW           ; NOVA ASSEMBLY SWITCH

         ;STACK FIXED AREA DISPLACEMENTS FROM .FP

.DUSR    FAC0=   -4              ;SAVE REGISTER 0
.DUSR    FAC1=   FAC0+1          ;SAVE REGISTER 1
.DUSR    FAC2=   FAC1+1          ;SAVE REGISTER 2
.DUSR    FOFP=   FAC2+1          ;SAVE FOR OLD FRAME POINTER
.DUSR    FRTN=   FOFP+1          ;SAVE CARRY AND RETURN ADDRESS
.DUSR    FFLE=   FRTN+1          ;FIRST FREE LOCATION AFTER FRAME
.DUSR    FOLD=   FFLE            ;SAVE PREVIOUS CONTENTS OF DISPLAY[I]
.DUSR    FLEN=   FFLE-FAC0       ;LENGTH OF FIXED AREA

         ;BUFFER ZONE BETWEEN STACK TOP AND STACK END

**.DO    ESW
.DUSR    ENZON=  30      ; CBLKS(7) PSH (4) FAULT (5) RTER (2) ERRINT (3)
**.ENDC

**.DO    NSW
.DUSR    ENZON=  100             ;NO MAN'S LAND
**.ENDC
```

Figure G-1. DG/L Symbol Definitions

```
        ;RUNTIME DISPLACEMENTS OF CALLING ARGUMENTS FROM .FP
        ; FOR USER CALLABLE ROUTINES.

.DUSR   ARG0=   FAC1            ;RESULT OR ZERO-TH ARGUMENT
.DUSR   ARG1=   ARG0-1          ;FIRST ARGUMENT
.DUSR   ARG2=   ARG1-1          ;SECOND ARGUMENT
.DUSR   ARG3=   ARG2-1          ;THIRD ARGUMENT
.DUSR   ARG4=   ARG3-1          ;FOURTH ARGUMENT
.DUSR   ARG5=   ARG4-1          ;FIFTH ARGUMENT
.DUSR   ARG6=   ARG5-1          ;SIXTH ARGUMENT
.DUSR   ARG7=   ARG6-1          ;SEVENTH ARGUMENT
.DUSR   ARG8=   ARG7-1          ;EIGTH ARGUMENT
.DUSR   ARG9=   ARG8-1          ;NINTH ARGUMENT
.DUSR   ARG10=  ARG9-1          ;TENTH ARGUMENT
.DUSR   ARG11=  ARG10-1         ;ELEVENTH ARGUMENT
.DUSR   ARG12=  ARG11-1         ;TWELTH ARGUMENT
.DUSR   ARG13=  ARG12-1         ;THIRTEENTH ARGUMENT
.DUSR   ARG14=  ARG13-1         ;FOURTEENTH ARGUMENT

        ;RUNTIME DISPLACEMENTS OF CALLING ARGUMENTS FROM .FP
        ; FOR INTERNALLY CALLED ROUTINES.

.DUSR   IARG0=  FAC1            ;RESULT OR ZERO-TH ARGUMENT
.DUSR   IARG1=  IARG0-1         ;FIRST ARGUMENT
.DUSR   IARG2=  IARG1-1         ;SECOND ARGUMENT
.DUSR   IARG3=  IARG2-1         ;THIRD ARGUMENT
.DUSR   IARG4=  IARG3-1         ;FOURTH ARGUMENT
.DUSR   IARG5=  IARG4-1         ;FIFTH ARGUMENT

        ;FIRST TWO ARGS OF INTERNAL ROUTINE CALLS ARE PASSED
        ; IN AC1 AND AC0.

.DUSR   ACAR0=  1               ;FIRST ARGUMENT
.DUSR   ACAR1=  0               ;SECOND ARGUMENT

        ;FROM OLD .SP

.DUSR   ARGN=   1               ;LAST ARGUMENT
.DUSR   ARGN1=  ARGN+1          ;NEXT TO LAST ARGUMENT
.DUSR   ARGN2=  ARGN1+1         ;NEXT TO NEXT TO LAST ARGUMENT
.DUSR   ARGN3=  ARGN2+1         ;
.DUSR   ARGN4=  ARGN3+1 ;

        ;ARGUMENT TESTING

.DISS   SEA=    SUBR#   0,0,SNC         ;SKIP ON EVEN ARGS (ACS=FAC2,ACD=.FP)
.DISS   SOA=    SUBR#   0,0,SZC         ;SKIP ON ODD ARGS  (ACS=FAC2,ACD=.FP)
.DUSR   COA=    ADCR    3,2             ;SET CARRY ON ODD ARGS
                                        ;   (DESTROYS AC2, CALL AFTER SAVE)
**.IFE FLEN&1815
        LDA:    ;CHANGE ARGUMENT TESTS IN ALSYM
**.ENDC
        ;STRUCTURE OF STRING AND SUBSTRING SPECIFIERS

.DUSR   SSPTR=  0               ;!! BYTE POINTER TO STRING OR
                                ;POINTER TO PARENT STRING SPECIFIER
.DUSR   SSMAX=  SSPTR+1         ;MAXIMUM LENGTH OF STIRNG
.DUSR   SSLEN=  SSMAX+1         ;CURRENT LENGTH OF STRING DATUM
.DUSR   SSIND=  SSMAX           ;STARTING INDEX OF SUBSTRING (NEGATIVE)
```

Figure G-1. DG/L Symbol Definitions (continued)

```
        ;STRUCTURE OF LABEL PARAMETER SPECIFIERS

.DUSR   LBLAD=  0               ;ADDRESS CORRESPONDING TO LABEL
.DUSR   LBLFP=  LBLAD+1         ;FRAME POINTER OF PASSER OF LABEL
.DUSR   LBLBD=  LBLFP+1         ;BLOCKLEVEL(LABEL)-BLOCKLEVEL(CALL)
.DUSR   LBLSP=  LBLBD+1         ;STACK POINTER OF PASSER OF LABEL
                                ; PRESENT IF LBLBD CONTAINS COMPLIMENT
                                ; NUMBER OF LEVEL DIFFERENCE


        ;STRUCTURE OF ARRAY DOPE VECTOR BOUND PAIRS

.DUSR   DVLBD=  -2              ;OFFSET FROM DIMENSION COUNT TO FIRST LBOUND
.DUSR   DVHBD=  DVLBD+1         ;OFFSET FROM DIMENSION COUNT TO FIRST HBOUND


        ;STRUCTURE OF CACHE I/O BUFFER POOLS
        ;  ALSO DATA WORD BUFFERING

.DUSR   BFLNK=  0               ;LINK TO FIRST BUFFER !!
                                ;   WORD OR BYTE ADDRESS OF BUFFER
.DUSR   CMLCK=  1               ;BUFFER POOL LOCK WHEN NON-VIRTUAL !!
.DUSR   CMKEY=  CMLCK+1         ;@ADDRESS OF CM LOCK USED (CMLCK OR VRLCK IN .PT)
.DUSR   CMCHN=  CMKEY+1         ;CHANNEL NUMBER FOR REQUEST
.DUSR   CMFPT=  CMCHN+1         ;FILE ELEMENT ADDRESS FOR REQUEST
                                ;   WORD ADDRESS OF BUFFER
.DUSR   CMOFF=  CMFPT+1         ;ELEMENT OFFSET FOR REQUEST
                                ;   END OF BUFFER
.DUSR   CMLBZ=  CMOFF+1         ;ELEMENT SIZE FOR FILE REQUESTED
.DUSR   CMCAD=  CMLBZ+1         ;CORE ADDRESS FOR DATA MOVEMENT
.DUSR   CMMOD=  CMCAD+1         ;READ/WRITE MODE SWITCH
.DUSR   CMBN=   CMMOD+1         ;BLOCK NUMBER REFERENCED
                                ;   BLOCKS IN THE FILE
.DUSR   CMFBW=  CMBN+1          ;WORD OFFSET IN FIRST BLOCK FOR MOVE
.DUSR   CMCNT=  CMFBW+1         ;COUNT OF WORDS TO MOVE
.DUSR   CMBCN=  CMCNT+1         ;PARTIAL WORD COUNT FOR BLOCK TO MOVE
                                ;   BYTES IN LAST BLOCK
.DUSR   CMUSP=  CMBCN+1         ;SAVE FOR CALLER'S USP
.DUSR   CMBCT=  CMUSP           ;   NUMBER OF BYTES OR BLOCK NUMBER
.DUSR   CMRTN=  CMUSP+1         ;SAVE FOR RETURN ADDRESS
.DUSR   CMCH=   CMRTN           ;   1//CHANNEL NUMBER
.DUSR   CMSIZE= CMRTN+1         ;SIZE OF BUFFER POOL HEADER

.DUSR   BFLNK=  0               ;LINK TO NEXT BUFFER (-1 TERMINATES) !!
.DUSR   BFCHN=  BFLNK+1         ;STATUS/CHANNEL, 1B0=MOD, +INF.=NOT IN USE
.DUSR   BFBN=   BFCHN+1         ;BLOCK NUMBER HELD IN BUFFER
.DUSR   BFCAD=  BFBN+1          ;(CORE,LOGICAL) ADDRESS OF BUFFER
.DUSR   HSIZE=  BFCAD+1         ;SIZE OF BUFFER DESCRIPTOR
```

Figure G-1. DG/L Symbol Definitions (continued)

```
        ;STRUCTURE OF I/O CONTROL BLOCK

.DUSR   FBFSZ=  67.             ;133 CHARACTER FORMAT LINE BUFFER
**.DO   ASW
.DUSR   FMPSZ=  12.             ;AOS I/O PACKET AREA SIZE
**.ENDC

.DUSR   IOLBL=  0               ;I/O ERROR LABEL SPECIFIER !!
.DUSR   IOCSP=  IOLBL+4         ;SP AT WHICH WE AUTOMAGICALLY RELEASE IOCB
.DUSR   ACTFL=  IOCSP+1         ;ACTIVE CHANNEL NUMBER FOR FORMAT
.DUSR   LBUFF=  ACTFL+1         ;PREVIOUS BUFFER LINK
.DUSR   IORTN=  LBUFF+1         ;RETURN ADDRESS FROM .NPIC ETC.
.DUSR   IOADR=  IORTN+1         ;ADDRESS OF DATA WORD
.DUSR   IOCNT=  IOADR+1         ;COUNT OF DATA WORDS IN ARRAY
.DUSR   FMRCR=  IOCNT+1         ;FORMAT RECURSION INDICATOR
.DUSR   FMWRL=  FMRCR+1         ;WRITE LINE INDICATOR (IF NO-ZERO)
.DUSR   FMSBP=  FMWRL+1         ;CURRENT FORMAT START (RESCAN) POINT
.DUSR   FMWBP=  FMSBP+1         ;CURRENT FORMAT WORKING BYTE POINTER
.DUSR   FMEND=  FMWBP+1         ;END OF FORMAT (FMWBP>=FMEND)
.DUSR   FOPIC=  FMEND+1         ;FORMAT OUTPUT INTEGER COUNT
.DUSR   FOPFC=  FOPIC+1         ;FORMAT OUTPUT FRACTION COUNT
.DUSR   FOPEC=  FOPFC+1         ;FORMAT OUTPUT EXPONENT COUNT
.DUSR   FOPTC=  FOPEC+1         ;FORMAT OUTPUT TOTAL PICTURE COUNT
.DUSR   FOPSN=  FOPTC+1         ;FORMAT OUTPUT PICTURE SIGN
.DUSR   FOPDP=  FOPSN+1         ;FORMAT OUTPUT PERIOD INDICATOR
.DUSR   FOPRD=  FOPDP+1         ;DEFAULT RADIX FOR NPIC
.DUSR   BFSBP=  FOPRD+1         ;OUTPUT BUFFER STARTING BYTE POINTER
.DUSR   BFEBP=  BFSBP+1         ;END OF FORMATTED OUTPUT BUFFER
.DUSR   BFWBP=  BFEBP+1         ;WORKING BYTE POINTER INTO BUFFER
**.DO   ASW
.DUSR   FMPKT=  BFWBP+1         ;ADDRESS OF AOS I/O PACKET
.DUSR   FMPKA=  FMPKT+1         ;AOS I/O PACKET AREA
.DUSR   FMBUF=  FMPKA+FMPSZ     ;START OF FORMAT OUTPUT BUFFER
**.ENDC
**.DO   RSW
.DUSR   FMBUF=  BFWBP+1         ;START OF FORMAT OUTPUT BUFFER
**.ENDC
.DUSR   IOEND=  FMBUF+FBFSZ     ;SIZE OF I/O CONTROL BLOCK

.DUSR   RMODE=  FMWRL           ;UNFORMATTED READ MODE
.DUSR   RSPTR=  FMSBP           ;UNFORMATED READ STRING POINTER


        ;ERROR CODE WORD FORMAT

.DUSR   ECSEV=  3B1             ;ERROR CODE SEVERITY

        .DUSR   ECSFP=  0B1             ;FATAL TO PROCESS
        .DUSR   ECSFT=  1B1             ;FATAL TO TASK
        .DUSR   ECSNF=  2B1             ;NON FATAL
        .DUSR   ECSAF=  3B1             ;ABSOLUTELY FATAL

.DUSR   ECTYP=  3B3             ;ERROR CODE TYPE

        .DUSR   ECTSY=  0B3             ;SYSTEM ERROR
        .DUSR   ECTUE=  1B3             ;USER ERROR CODE
        .DUSR   ECTUT=  2B3             ;USER ERROR TEXT (NO CODE)
        .DUSR   ECTRT=  3B3             ;RUNTIME ERROR
```

Figure G-1. DG/L Symbol Definitions (continued)

```
.DUSR   ECSCD=  17B7            ;ERROR SUB CODE

        .DUSR   ECSCB=  1B7     ;LOW BIT OF SIB CODE (USE FOR FORMING ERR CODE)

.DUSR   ECCOD=  377             ;ERROR CODE


        ;ERROR MASK WORD FORMAT

.DUSR   EMPRT=  1B0             ;(0) SUPPRESS PRINTING OF ERROR MESSAGES
                                ;(1) PRINT ERROR MESSAGE BEFORE HANDLING
.DUSR   EMSEV=  7B3             ;SEVERITY TO BE HANDLED DISCRIPTOR

        .DUSR   EMSFP=  1B1             ;HANDLE FATAL TO PROCESS ERRORS
        .DUSR   EMSFT=  1B2             ;HANDLE FATAL TO TASK ERRORS
        .DUSR   EMSNF=  1B3             ;HANDLE NON FATAL ERRORS

.DUSR   EMTYP=  17B7            ;TYPE TO BE HANDLED DISCRIPTOR

        .DUSR   EMTSY=  1B4             ;HANDLE SYSTEM ERRORS
        .DUSR   EMTUE=  1B5             ;HANDLE USER ERROR CODES
        .DUSR   EMTUT=  1B6             ;HANDLE USER ERROR TEXT
        .DUSR   EMTRT=  1B7             ;HANDLE RUNTIME ERROR

.DUSR   EMCOD=  377             ;HANDLE THIS IS ERROR CODE
                                ; ANY CODE IF SET TO 377

        ;ERROR PROCESS BLOCK FORMAT

.DUSR   EPLNK=  0               ;(-1) LINK TO NEXT ERR PROCESS BLOCK
.DUSR   EPBMK=  EPLNK+1         ;ERROR PROCESS MASK
.DUSR   EPBPA=  EPBMK+1         ;ADDRESS OF BLOCK PROCESSER ROUTINE
.DUSR   EPBDL=  EPBPA+1         ;ADDRESS OF BLOCK REMOVE ROUTINE
.DUSR   EPBSP=  EPBDL+1         ;STACK POINTER AT TIME OF EPB CREATE
.DUSR   EPBSZ=  EPBSP+1         ;BASIC EPB SIZE


        ;STRUCTURE OF RUNTIME GLOBAL AREA

.DUSR   FIOCB=  0               ;CURRENT FORMATTED I/O CONTROL BLOCK !!
.DUSR   AFREE=  FIOCB+1         ;ALLOCATE FREE BLOCK CHAIN !!
.DUSR   GPERB=  AFREE+1         ;(-1) POINTER TO ERR PROCESS BLK CHAIN
.DUSR   GPERR=  GPERB+1         ;(-1) LAST ERROR CODE
.DUSR   GPETX=  GPERR+1         ;(-1) WORD ADDRESS OF ERROR MESSAGE
.DUSR   GPECL=  GPETX+1         ;ADDR CALLER OF ROUTINE IN WHICH ERR OCCURED
.DUSR   GPERA=  GPECL+1         ;ADDR IN RUNTIME OF ERROR
.DUSR   GPEND=  GPERA+1         ;SIZE OF RUNTIME GLOBAL AREA
```

Figure G-1. DG/L Symbol Definitions (continued)

```
        ;PROCESS GLOBAL TABLE

                                        ;( ) INITIAL VALUE
.DUSR   GTFREE= 0                       ;START OF GLOBAL FREE CHAIN !! USED FOR KEY
.DUSR   MSKEY=  1                       ;(<>0) CONSOLE MESSAGE KEY !!
.DUSR   UCKEY=  MSKEY+1                 ;USER CLOCK KEY
.DUSR   DSTK=   UCKEY+1                 ;DEFAULT STACK SIZE
.DUSR   CONV1=  DSTK+1                  ;CONVERSION 1000HZ => SYSTEM UNITS
.DUSR   CONV2=  CONV1+1                 ;CONVERSION SEC -> SYSTEM UNITS
.DUSR   CONV3=  CONV2+1                 ;CONVERSION MIN -> SYSTEM UNITS
.DUSR   CNTFL=  CONV3+1                 ;COUNT OF FILE NUMBERS
.DUSR   WLIST=  CNTFL+1                 ;SAVE 'LIST'VALUE FOR WRITE
.DUSR   VRNUM=  WLIST+1                 ;(0) NEXT FREE VIRTUAL BUFFER
.DUSR   VRADDR= VRNUM+1                 ;ADDRESS OF VIRTUAL CM WINDOW
.DUSR   VRRTN=  VRADDR+1                ;SAVE FOR RETURN FOR .BUFMAP
.DUSR   VRBUF=  VRRTN+1                 ;SAVE FOR CURRENT BUFFER IN .BUFMAP
.DUSR   GMERG=  VRBUF+1                 ;(-1) GALLOCATE REMERGE INDICATOR
.DUSR   OPKEY=  GMERG+1                 ;(-1) INIT CPCOM KEY
.DUSR   AWAIT=  OPKEY+1                 ;(-1) GALLOCATE WAIT KEY
.DUSR   DATBF=  AWAIT                   ;(-1) FREE DATA BUFFER CHAIN (NON-TASKING ONLY)
.DUSR   VRLCK=  AWAIT+1                 ;(-1) CM TASKING LOCK FOR VIRTUAL MAPPING
.DUSR   VRMAP=  VRLCK+1                 ;(-1) PREVIOUS CONTENTS OF WINDOW MAP
.DUSR   OVCHAN= VRMAP+1                 ;(-1) OVERLAY CHANNEL NUMBER

        .IFN    RSW

.DUSR   FTLCK=  OVCHAN+1                ;TABLE OF FILE LOCKS
.DUSR   PTFIX=  OVCHAN+1                ;SIZE OF FIXED PROCESS GLOBAL TABLE

        .ENDC   SKZ

.DUSR   ARGNM=  OVCHAN+1                ;CURRENT CLI ARGUMENT #
.DUSR   CONPAD= ARGNM+1                 ;ADDRESS OF CONSOLE I/O PACKET
.DUSR   FTLCK=  CONPAD+1                ;TABLE OF FILE LOCKS
.DUSR   PTFIX=  CONPAD+1                ;SIZE OF FIXED PROCESS GLOBAL TABLE

[SKZ]

                        ;NOTE: THE FOLLOWING ARE DEFINED IN CONTFL.SR
;.CNTFL=        16.
;.DUSR  FTBUF=  -.CNTFL                 ;TABLE OF BUFFER POOL MEMBERSHIPS
;.DUSR  FTLBZ=  FTBUF-.CNTFL            ;TABLE OF CACHE FILE ELEMENT SIZES
;.DUSR  FTWRL=  FTBUF-.CNTFL            ;TABLE OF WRITE LINE INDICATORS
;;.DUSR FTLCK=  FTCHN+.CNTFL            ;TABLE OF FILE LOCKS


        .IFN    ASW

.DUSR   TPC=    ?TPC
.DUSR   TAC0=   ?TAC0
.DUSR   TAC1=   ?TAC1
.DUSR   TAC2=   ?TAC2
.DUSR   TAC3=   ?TAC3
.DUSR   TSYS=   ?TSYS
.DUSR   TLNK=   ?TLNK
.DUSR   TUSP=   ?TUSP
.DUSR   TKLAD=  ?TKAD
.DUSR   TELN=   ?TELN
.DUSR   TSP=    ?TSP
.DUSR   TFP=    ?TFP
.DUSR   TSL=    ?TSL
.DUSR   TSO=    ?TSO
.DUSR   TPRST=  ?TSTAT
```

Figure G-1. DG/L Symbol Definitions (continued)

```
;FLAGS

.DUSR   TSUPN=  ?TSUF

.DUSR   CSP=    41

        .ENDC

        .IFN    RSW

.DUSR   .NEWLN= 15
.DUSR   ERAC=   2
.DUSR   FRAC=   0

        .ENDC   SKERR

.DUSR   .NEWLN= 12
.DUSR   ERAC=   0
.DUSR   FRAC=   2

[SKERR]

        ;RUNTIME TCB-EXTENDER (MULTI-TASKING)
                                ;OCCURS AFTER .GP (.ISTK,RUNTASK)
.DUSR   E.GP=   1               ;GLOBAL AREA POINTER
.DUSR   E.RP=   E.GP+1          ;RETURN TEMPORARY
.DUSR   E.DP=   E.RP+1          ;DISPLAY POINTER (!! FORTRAN 5 COMPATIBILITY)
        .DO     ESW
.DUSR   E.FPS=  E.DP+1          ;FLOATING POINT STATUS !!
.DUSR   E.FPU=  E.FPS+2         ;SPACE FOR FPAC !!
.DUSR      E.SP=        E.FPU+1         ;INITIAL .SP
.DUSR      E.SSE=       E.SP+1          ;INITIAL .SSE
        .ENDC
        .DO     NSW
.DUSR   E.FPU=  E.DP+1          ;SPACE FOR FPAC AND TEMP !!
.DUSR   E.FPS=  E.FPU+8.        ;FLOATING POINT STATUS !!
.DUSR      E.SP=        E.FPS+1         ;STACK POINTER
.DUSR      E.FP=        E.SP+1          ;FRAME POINTER
.DUSR      E.SSE=       E.FP+1          ;END OF STACK POINTER
        .ENDC
.DUSR      E.AC0=       E.FPU+4         ;TEMP FOR AC0,AC1 (.ISTK,.TASK)
.DUSR      E.AC1=       E.AC0+1         ;
.DUSR   E.SSR=  E.FPU+16.       ;MIN .SSE FOR F5 COMPATABILITY
.DUSR   E.ZREL= E.SSR+1         ;ZREL ADDRESS OF PROGRAM
.DUSR   E.RTN=  E.ZREL+1        ;INSTRUCTION FOR RETURN
.DUSR   E.QUE=  E.RTN+1         ;QUEUE ARRAY ADDRESS
.DUSR   E.SIZ=  E.QUE+1         ;SIZE OF TCB-EXTENDER


        ;TASK STATUS BITS (TPRST)

.DUSR   TBSYS=  180             ;SYSTEM WAIT BIT !!
.DUSR   TBSUSP= 181             ;TASK SUSPEND BIT !!
.DUSR   TBXMT=  182             ;XMT,REC, OVERLAY BIT !!
.DUSR   TBRDUP= 183             ;READ OPERATOR BIT


        ;QUEUE ARRAY EXTENSION FOR DG/L
        .IFN    RSW

.DUSR   QORG=   QTLN            ;FLAG FOR QUEUE ORIGIN
.DUSR   QZREL=  QORG+1          ;TASK ZREL ADDRESS
.DUSR   QMEM=   QZREL+1         ;STACK SIZE
.DUSR   QNOSTK= QMEM+1          ;NO STACK ERROR HANDLER
.DUSR   .QLEN=  QNOSTK+1        ;SIZE OF DG/L QUEUE ARRAY
```

Figure G-1. DG/L Symbol Definitions (continued)

```
        .ENDC   QUE

.DUSR   ?DNST=  ?DCI+1          ;NO STACK HANDLER FOR AOS TDP
.DUSR   ?DZREL= ?DNST+1         ;TASK ZREL ADDRESS
.DUSR   ?QLNK=  ?DLNK-1         ;DG/L TDP CHAIN POINTER
.DUSR   QNUM=   ?DCC
.DUSR   QSH=    ?DSH
.DUSR   QSMS=   ?DSMS
.DUSR   QRR=    ?DCI
.DUSR   QAC2=   ?DAC2
.DUSR   QMEM=   ?DSSZ
.DUSR   QZREL=  ?DZREL
.DUSR   QPC=    ?DPC
.DUSR   QNOSTK= ?DNST
.DUSR   .QLEN=  ?DXLTH          ;DEFINE QUEUE (TDP) LENGTH
[QUE]



        ;STRUCTURE OF THE FPU STATUS REGISTER

.DUSR   FPANY=  180             ;BIT ZERO INDICATES SOME ERROR
.DUSR   FPOVF=  181             ;EXPONENT OVERFLOW
.DUSR   FPUNF=  182             ;EXPONENT UNDERFLOW
.DUSR   FPDVZ=  183             ;ATTEMPTED DIVISION BY ZERO
.DUSR   FPMOF=  184             ;MANTISSA OVERFLOW
.DUSR   FPGTZ=  185             ;C(FPAC) POSITIVE
.DUSR   FPEQZ=  186             ;C(FPAC) ARE TRUE ZERO
.DUSR   FPLTZ=  187             ;C(FPAC) NEGATIVE
.DUSR   FPIND=  1813            ;FPU INTERRUPTS DISABLED
.DUSR   FPPPM=  1814            ;PARALLEL FPU OPERATION
.DUSR   FPDMD=  1815            ;FPU IN DIAGNOSTIC MODE

        ;MISC.

.DUSR   BZREL=  50              ;BEGINNING OF ZREL

        .IFN    RSW

.DUSR   MKWRL=  DCCGN+DCTO+DCNAF+DCRAT+DCPCK+DCLAC+DCLTU  ;0 IF .WRL == .WRS

        .ENDC   SKY

.DUSR   MKWRL=  ?CST+?CSFF+?CRAT+?CLCC+?CLT+?CFF        ;0 IFF .WRL == .WRS

[SKY]

.DUSR   TASKING= 0              ;TASKING SWITCH (RTER)
.DUSR   DGL=    1               ;DGL SWITCH (USED IN RINIT)
.DUSR   CVSTART= FFLE           ;START OF CV* SAVE LOCS
.DUSR   CVEND=  CVSTART+7       ;END OF CV* SAVE LOCS
```

Figure G-1. DG/L Symbol Definitions (continued)

```
        ;RUNTIME ERROR MESSAGE CODES

.DUSR   AICOR=  1+ECTRT+ECSAF   ;INVALID POINTER OR CORE
.DUSR   AISUB=  2+ECTRT         ;SUBSCRIPT OUT OF BOUNDS
.DUSR   AINUM=  3+ECTRT         ;ILLEGAL NUMBER CONVERSION
.DUSR   AIFMT=  4+ECTRT         ;MISSING FORMAT OUTPUT FIELD
.DUSR   AISET=  5+ECTRT         ;PROBABLE LINEREAD OVERFLOW
.DUSR   AEOBA=  6+ECTRT         ;REQUEST FOR ODD-BYTE STRING ADDRESS
.DUSR   AINOD=  7+ECTRT         ;INVALID NODE SIZE
.DUSR   AIBFS=  10+ECTRT        ;INSUFFICIENT BUFFER POOL ALLOCATION
.DUSR   AISIZ=  11+ECTRT        ;ILLEGAL SIZE
.DUSR   AISTK=  12+ECTRT        ;STACK OVERFLOW

.DUSR   AESQT=  20+ECTRT        ;ILLEGAL ARGUMENT TO SQRT
.DUSR   AEEXP=  21+ECTRT        ;ILLEGAL ARGUMENT TO EXP
.DUSR   AELOG=  22+ECTRT        ;ILLEGAL ARGUMENT TO LOG
.DUSR   AEPWR=  23+ECTRT        ;ILLEGAL EXPONENTIATION
.DUSR   AEASC=  24+ECTRT        ;ILLEGAL ARGUMENT TO ASIN/ACOS
.DUSR   AEATN=  25+ECTRT        ;ILLEGAL ARGUMENT TO ATAN2
.DUSR   AEINT=  26+ECTRT        ;ILLEGAL ARGUMENT TO INT

.DUSR   AEFPU=  50+ECTRT        ;FPL ERROR
                                ;FPL ERR SUB-CODES MAY BE ORED
                                ;TOGETHER
.DUSR   AEMAN=  AEFPU+(1*ECSCB) ;MANTISSA OVERFLOW
.DUSR   AEUND=  AEFPU+(2*ECSCB) ;FPL UNDERFLOW
.DUSR   AEOVF=  AEFPU+(4*ECSCB) ;FPL OVERFLOW
.DUSR   AEZDV=  AEFPU+(10*ECSCB);ZERO DIVIDE

.DUSR   AIBND=  60+ECTRT        ;ILLEGAL BOUND SPECIFICATION
.DUSR   AISOV=  70+ECTRT        ;STRING OVERFLOW

                .EOT
```

Figure G-1. DG/L Symbol Definitions (continued)

```
;*****************************************************************
;
;                DG/L MACRO DEFINITIONS
;
;*****************************************************************

                .TITLE          DGLMACS

        ;GENERAL PURPOSE MACRO DEFINITIONS

        .MACRO  MKST            ;PUSH ARGUMENT LIST TO STACK
        .EXTD   .SP
        LDA     2,MODEL+1       ;
        ADD     3,2             ;
        STA     2,.SP           ;
        LDA     2,MODEL         ;
        ADD     3,2             ;
        STA     2,AT0,3         ;
%
        .MACRO  TOP             ;TOP OF STACK INTO AC
**      .EXTD   .SP
**      LDA     1,@.SP          ;
%

        .MACRO  ARG?            ; CHECK TO SEE IF ARG PRESENT IN CALL
**      .DO     (1==IARG0)      ; RETURNS TO MACRO + 1 IF ARG NOT
        ZCALL   .ARI0           ; PRESENT IN CALL TO ROUTINE
**      .ENDC   SK1
**      .DO     (1==IARG1)
        ZCALL   .ARI1
**      .ENDC   SK1
        ZCALL   .ARCK
        1
**[SK1]
%

        .MACRO  GBIT            ; GET BIT ADDRESSED IN ACS,ACD INTO CARRY
**      SZB     1,2
**      MOVO    0,0,SKP
**      MOVZ    0,0
%

        .MACRO  PBIT            ; PUT BIT IN CARRY INTO BIT ACS,ACD
**      MOV#    0,0,SNC
**      BTZ     1,2
**      MOV#    0,0,SZC
**      BTO     1,2
%
        .DO     ESW

        .MACRO  XMUL            ; HARDWARE MUL ON ECLIPSE
**      MUL
%

        .MACRO  XDIV            ; HARDWARE DIV ON ECLIPSE
**      DIV
%

        .ENDC
```

Figure G-2. DG/L Macro Definitions

```
        .DO     NSW
        .MACRO  SAVE            ;SAVE STATE OF STACK
**      ZSTA    3,.ND1          ; SAVE RETURN ADDRESS
**      .DO     ↑1==0           ; IF ZERO FRAME ALLOCATED
**      .EXTD   .SAV0
**      JSR     @.SAV0          ; GO DO SAVE
**      .ENDC   ZERO            ; IF MUST SAVE SPACE ON STACK
**      .EXTD   .SAV1
**      JSR     @.SAV1          ; GO DO SAVE WITH ALLOCATE OF ↑1 WORDS
**      ↑1                      ;
**(ZERO)
%
        .MACRO  RTN             ;RETURN STATE OF STACK
**      .EXTD   .RETN
**      JMP     @.RETN          ;
%
        .MACRO  PSH             ;PUSH C(ACS->ACD) ONTO STACK (PSH ACS,ACD)
**      .EXTD   .SP
**      .IFE    ↑1<=↑2
**      LDA:    ;ERROR IN PSH
**      .ENDC   DONE
**      .PUSH   I
**      I=      ↑1
**      .DO     ↑2-↑1+1
**      ISZ     .SP             ;
**      STA     1,@.SP          ;
**      I=      I+1
**      .ENDC
**      I=      .POP
**(DONE)
%
        .MACRO  POP             ;POP TOP OF STACK INTO AC (POP AC,AC)
**      .EXTD   .SP
**      .IFE    ↑1>=↑2
**      LDA:    ;ERROR WITH POP
**      .ENDC   DONE
**      .PUSH   I
**      I=      ↑1
**      .DO     ↑1-↑2+1
**      LDA     I,@.SP          ;
**      DSZ     .SP             ;
**      I=      I-1
**      .ENDC
**      I=      .POP
**(DONE)
%
.MACRO  MK.STK
**      .EXTD   .MKSTK
**      JSR     @.MKSTK
%
        .MACRO  ADI             ;ADD IMMEDIATE TO AC
**      .DO     ↑1              ;
**      INC     ↑2,↑2           ;
**      .ENDC                   ;
%
        .MACRO  SBI             ;SUBTRACT IMMEDIATE FROM AC
**      NEG     ↑2,↑2           ;
**      .DO     ↑1-1            ;
**      INC     ↑2,↑2           ;

**      .ENDC                   ;
**      COM     ↑2,↑2           ;
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
        .MACRO  HXL             ;HEX SHIFT LEFT OF AC
**      .DO     ↑1              ;
**      .DO     4               ;
**      MOVZL   ↑2,↑2           ;
**      .ENDC                   ;
**      .ENDC                   ;
%
        .MACRO  HXR             ;HEX SHIFT RIGHT OF AC
**      .DO     ↑1              ;
**      .DO     4               ;
**      MOVZR   ↑2,↑2           ;
**      .ENDC                   ;
**      .ENDC                   ;
%
        .MACRO  LDB             ;LOAD BYTE INTO AC (LDB 2,0)
**      .IFE    (↑1==2)&(↑2==0)
**      .EXTD   .LDB
**      JSR     @.LDB
**      (-FAC0-↑1)B7+(-FAC0-↑2)
**.ENDC LDB
**      .EXTD   .GETBT
**      JSR     @.GETBT         ;
**[LDB]
%
        .MACRO  STB             ;STORE BYTE IN AC (STB 2,0)
**      .IFE    (↑1==2)&(↑2==0)
**      .EXTD   .STB
**      JSR     @.STB
**      (-FAC0-↑1)B7+(-FAC0-↑2)
**.ENDC STB
**      .EXTD   .PUTBT
**      JSR     @.PUTBT         ;
**[STB]
%
        .MACRO  XCH             ;EXCHANGE ACCUMULATORS (USES AC2)
**      .IFN    (↑1==2)!(↑2==2)
**      LDA:    ;ERROR WITH XCH USING AC2 AS TEMP
**      .ENDC
**      MOV     ↑1,2
**      MOV     ↑2,↑1
**      MOV     2,↑2
%


        .MACRO  XMUL            ; EMULATION OF ECLIPSE 'MUL'
**      ZSTA    3,.ND1
**      XCALL   .MUL.
%

        .MACRO  XDIV            ; EMULATION OF ECLIPSE 'DIV'
**      ZSTA    3,.ND1
**      XCALL   .DIV.
%

        .MACRO  DIVS            ; SIGNED DIVIDE
**      PSH     3,3
**      ZCALL   .DIV
**      POP     3,3
%
.MACRO  DIVX
**      MOVL#   1,1,SZC         ;EXTEND SIGN, CHECK FOR 0 SIGN
**      ADC     0,0,SKP         ;SIGN = 1, AC0 <- -1
**      SUB     0,0             ;SIGN = 0, AC0 <- 0
**      DIVS                    ;SIGNED DIVIDE
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
.MACRO  DAD
**      .DO     (↑1==3)!(↑2==3)
**:     ;ERROR IN DAD MACRO
**      .ENDC DAD
**      .EXTD   D.AD
**      JSR     @D.AD
**      ↑1B7+↑2
**[DAD]
%

.MACRO  DSB
**      .DO     (↑1==3)!(↑2==3)
**:     ;ERROR IN DSB MACRO
**      .ENDC   DSB
**      .EXTD   D.SB
**      JSR     @D.SB
**      ↑1B7+↑2
**[DSB]
%


.MACRO  MSP
**      .DO     (↑1==3)
**:     ;ERROR IN MSP MACRO
**      .ENDC   .MSP
**      .EXTD   .MSP↑1
**      JSR     @.MSP↑1
**[MSP]
%

.MACRO  CLM
**      .DO     (↑1==3)!(↑2==3)
**:     ;ERROR IN CLM MACRO
**      .ENDC   CLM
**      .EXTD   .CLM
**      JSR     @.CLM
**      ↑1B7+↑2
**[CLM]
%
        .MACRO  SGE             ;SKIP ON GREATER THAN OR EQUAL TO
**      ADDOR   ↑1,↑1           ;  NOTE: FLIPS BIT ZERO
**      ADDOR   ↑2,↑2           ;
**      SUBZ#   ↑2,↑1,SNC       ;
%
        .MACRO  SGT             ;SKIP ON GREATER THAN
**      ADDOR   ↑1,↑1           ;  NOTE: FLIPS BIT ZERO
**      ADDOR   ↑2,↑2           ;
**      SUBZ#   ↑1,↑2,SZC       ;
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
        .MACRO  XOR             ;EXCLUSIVE OR
**      .DO     ↑1>↑2           ;  IF ↑1>↑2 THEN USE AC2 AS TEMP
**      .IFN    (↑1==2)!(↑2==2)
**      LDA:    ;ERROR WITH MACRO XOR
**      .ENDC
**      MOV     ↑2,2            ;
**      ANDZL   ↑1,2            ;
**      ADD     ↑1,↑2           ;
**      SUB     2,↑2            ;
**      .ENDC
**      .IFG    ↑2-↑1+1         ;  IF ↑1=<↑2 THEN USE AC3 AS TEMP
**      .IFN    (↑1==3)!(↑2==3)
**      LDA:    ;ERROR WITH MACRO XOR
**      .ENDC
**      MOV     ↑2,3            ;
**      ANDZL   ↑1,3            ;
**      ADD     ↑1,↑2           ;
**      SUB     3,↑2            ;
**      .ENDC
%
        .MACRO  IOR             ;INCLUSIVE OR
**      .DO     ↑1<=↑2          ;   USE AC3
**      .IFN    (↑1==3)!(↑2==3)
**      LDA:    ;ERROR IN MACRO IOR
**      .ENDC
**      COM     ↑1,3            ;
**      AND     3,↑2            ;
**      ADC     3,↑2            ;
**      .ENDC
**      .DO     ↑1>↑2           ;   USE ↑1
**      COM     ↑1,↑1           ;
**      AND     ↑1,↑2           ;
**      ADC     ↑1,↑2           ;
**      .ENDC
%
        .MACRO  BLM             ;BLOCK MOVE
**      E.N.T=  E.N.T+1
**      NEGC    1,1             ;
**BΔE.N.T:      LDA     0,0,2           ;  NOTE USES AC0, ASSUMES AC1 > 0
**      STA     0,0,3           ;       PRESERVES CARRY
**      INC     2,2             ;
**      INC     3,3             ;
**      INC     1,1,SZR         ;
**      JMP     BΔE.N.T         ;
%

        .MACRO  ELEF
**      .DO     (.ARGCT<>3)
**:     ILLEGAL ELEF MACRO CALL
**      .ENDC   XXX
**      .DO     (↑1==↑3)
**:     ILLEGAL ELEF MACRO CALL
**      .ENDC   XXX
**      XLDA    ↑1,= ↑2
**      ADD     ↑3,↑1
**[XXX]
**%
        .ENDC
```

Figure G-2. DG/L Macro Definitions (continued)

```
        .DO     NSW
.MACRO  ANC                     ;AND WITH COMPLEMENTED SOURCE
**      COM     ↑1,↑1           ;
**      AND     ↑1,↑2           ;
**      COM     ↑1,↑1           ;
%
        .ENDC
.MACRO  HDATA
**      .PUSH   .RDX
**      .RDX    16
**H=    0
**      .DO     .ARGCT
**H=    H+1
**0↑H
**      .ENDC
**      .RDX    .POP
%

        .MACRO  CHAN2           ;AC2 <- RDOS CHANNEL FOR FILE
        LDA     2,@ARG0,3       ;
** ;    JSR     @.NIOPREP       ;ALTERNATE FOR LOGICAL CHANNELS
** ;    JMP     ERET            ;
%
                .IFE    DGL     ;NOT USED FOR PHYSICAL CHANNELS
.MACRO  DYNAMIC                 ;DYNAMICALLY OPEN A FILE
**      E.N.T=  E.N.T+1
**SMΔE.N.T:     .SYSTM                  ;
**      .GCHNL                  ;
**      JMP     EMΔE.N.T+1              ;
**      .SYSTM                  ;
**      ↑1      CPU             ;
**      JMP     CHΔE.N.T                ;
**      JMP     EMΔE.N.T+2              ;
**USΔE.N.T:     ERUFT                   ;
**CHΔE.N.T:     LDA     3,USΔE.N.T              ;
**      SLB#    2,3,SNR         ;
**EMΔE.N.T:     JMP     SMΔE.N.T                ;
%
                .ENDC

        .MACRO  FTBUF           ;TABLE OF BUFFER POOL MEMBERSHIPS
.FTBUF,↑1%

        .MACRO  FTLBZ           ;TABLE OF CACHE FILE ELEMENT SIZES
.FTLBZ,↑1%

        .MACRO  FTWRL           ;TABLE OF WRITE LINE INDICATORS
.FTLBZ,↑1%


        .DO     ASW

        .MACRO  FTCHN           ;TABLE OF ACS CHANNEL EQUIVALENTS
.FTCHN,↑1%

        .ENDC


;       .MACRO  FTLCK           ;TABLE OF FILE LOCKS
; .FTLCK,↑1%
```

Figure G-2. DG/L Macro Definitions (continued)

```
        ;FLOATING POINT MACRO DEFINITIONS

        .DO     NSW

        .MACRO  LOAF            ;WAIT FOR FPU TO FINISH
**      SKPBZ   FPU             ;
**      JMP     .-1             ;
%
        .MACRO  FAS             ;ADD SINGLE
**      LOAF                    ;
**      DOA     1,FPU1          ;
%
        .MACRO  FSBS            ;SUBTRACT SINGLE
**      LOAF                    ;
**      DOAS    1,FPU1          ;
%
        .MACRO  FMLS            ;MULTIPLY SINGLE
**      LOAF                    ;
**      DOAP    1,FPU1          ;
%
        .MACRO  FDVS            ;DIVIDE SINGLE
**      LOAF                    ;
**      DOAC    1,FPU1          ;
%
        .MACRO  FATS            ;ADD SINGLE FROM TEMP
**      LOAF                    ;
**      DOC     0,FPU1          ;
%
        .MACRO  FSBTS           ;SUBTRACT SINGLE FROM TEMP
**      LOAF                    ;
**      DOCS    0,FPU1          ;
%
        .MACRO  FMLTS           ;MULTIPLY SINGLE FROM TEMP
**      LOAF                    ;
**      DOCP    0,FPU1          ;
%
        .MACRO  FDVTS           ;DIVIDE SINGLE FROM TEMP
**      LOAF                    ;
**      DOCC    0,FPU1          ;
%
        .MACRO  FLODS           ;LOAD FPAC SINGLE
**      LOAF                    ;
**      DOBP    1,FPU1          ;
%
        .MACRO  FSTRS           ;STORE FPAC SINGLE
**      LOAF                    ;
**      DOBS    1,FPU1          ;
**      LOAF                    ;
%
        .MACRO  FMFT            ;MOVE FPAC TO TEMP
**      LOAF                    ;
**      NIOP    FPU2            ;
%
        .MACRO  FMTF            ;MOVE TEMP TO FPAC
**      LOAF                    ;
**      NIOC    FPU2            ;
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
        .MACRO  FSCAL           ;SCALE FPAC TO C(REG)
**      LOAF                    ;
**      DOB     ↑1,FPU2         ;
%
        .MACRO  FLDEX           ;REPLACE EXPONENT WITH C(REG)
**      LOAF                    ;
**      DOBC    ↑1,FPU2         ;
%
        .MACRO  FCOM            ;NEGATE FPAC
**      LOAF                    ;
**      NIOC    FPU1            ;
%
        .MACRO  FNORM           ;NORMALIZE FPAC
**      LOAF                    ;
**      NIOS    FPU2            ;
%
        .MACRO  FPLS            ;MAKE FPAC POSITIVE
**      LOAF                    ;
**      NIOP    FPU1            ;
%
        .MACRO  FAD             ;ADD DOUBLE
**      LOAF                    ;
**      DOA     ↑1,FPU2         ;
%
        .MACRO  FSBD            ;SUBTRACT DOUBLE
**      LOAF                    ;
**      DOAS    ↑1,FPU2         ;
%
        .MACRO  FMLD            ;MULTIPLY DOUBLE
**      LOAF                    ;
**      DOAP    ↑1,FPU2         ;
%
        .MACRO  FDID            ;DIVIDE DOUBLE
**      LOAF                    ;
**      DOAC    ↑1,FPU2         ;
%
        .MACRO  FATD            ;ADD DOUBLE FROM TEMP
**      LOAF                    ;
**      DOC     0,FPU2          ;
%
        .MACRO  FSBTD           ;SUBTRACT DOUBLE FROM TEMP
**      LOAF                    ;
**      DOCS    0,FPU2          ;
%
        .MACRO  FMLTD           ;MULTIPLY DOUBLE FROM TEMP
**      LOAF                    ;
**      DOCP    0,FPU2          ;
%
        .MACRO  FDVTD           ;DIVIDE DOUBLE FROM TEMP
**      LOAF                    ;
**      DOCC    0,FPU2          ;
%
        .MACRO  FLODD           ;LOAD FPAC DOUBLE
**      LOAF                    ;
**      DOBP    ↑1,FPU2         ;
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
        .MACRO  FSTRD           ;STORE FPAC DOUBLE
**      LOAF                    ;
**      DOBS    ↑1,FPU2         ;
**      LOAF                    ;
%
        .MACRO  FLST            ;LOAD STATUS REGISTER WITH C(REG)
**      LOAF                    ;
**      DOA     ↑1,FPU          ;
%
        .MACRO  FRST            ;READ/CLEAR STATUS INTO REG
**      LOAF                    ;
**      DIAC    ↑1,FPU          ;
**      LOAF                    ;
%
        .MACRO  FCLRS           ;CLEAR FPAC SINGLE
**      LOAF                    ;
**      NIOS    FPU1            ;
%
        .MACRO  FCLRD           ;CLEAR FPAC DOUBLE
**      LOAF                    ;
**      NIOS    FPU1            ;
%
        .MACRO  FRHW            ;READ FPAC HIGH WORD
**      LOAF                    ;
**      DIA     ↑1,FPU1         ;
**      LOAF                    ;
%

        .ENDC

.MACRO  TITL            ;.TITLE AND SHOW ASSEMBLY ENVIRONMENT
**      E.N.T=  0
%

; WHICH OPERATING SYSTEM AND CPU?

        .DO ASW
.MACRO  TITL
                .TITLE  ↑1      ; ACOS  ECLIPSE DG/L
**      OBJCT   A↑1
%
        .ENDC   OS
        .DO RSW&ESW
.MACRO  TITL
                .TITLE  ↑1      ; RDOS ECLIPSE DG/L
**      OBJCT   E↑1
%
        .ENDC OS
        .DO RSW&NSW
.MACRO  TITL
                .TITLE  ↑1      ; RDOS NOVA    DG/L
**      OBJCT   N↑1
%
        .ENDC OS
        ;UNKNOWN
.MACRO  TITL
                .TITLE  ↑1      ; OS?? CPU???? DG/L
**      OBJCT   OOPS$↑1
%

[OS]
```

Figure G-2. DG/L Macro Definitions (continued)

```
.MACRO  TITL
**              .TXTM   1       ;ALWAYS PACK LEFT TO RIGHT
**
**.DO '↑2'<>''           ;;IF SECOND ARG EXISTS
**      .PUSH   .LOC            ;;PRESERVE .
**      NREL                    ;;GO NREL
**      .ENT    ↑2              ;;CREATE AN ENTRY SYMBOL
**  ↑2:                         ;;AND DEFINE IT
**      .LOC    .POP            ;;RESTORE .
**.ENDC
%


        .MACRO  TITLE
**
**      TITL    ↑1 ↑2
**      .NOCON  1
**      NREL
%

        .MACRO  E.ND
**
**      LPOOL           ;DEPOSIT REMAINING LITERALS
**
**      .PUSH   .NOCON
**      .NOCON  1
**      .PUSH   .RDXO
**      .RDXO   10

**      .ZREL
ZRELSIZE=.
**      .NREL
**.DO RSW
NRELSIZE=.
**.ENDC
**.DO ASW
UNRLSIZE=.
**      .NREL   1
SNRLSIZE=.
**.ENDC
**      .RDXO   .POP
**      .NOCON  .POP
**
**              .END    ↑1
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
        .MACRO  OBJCT
**
**.PUSH .NOCON
**.NOCON 1
**
**.DO ASW!OBSW
                .OB     ↑1.OB
**.ENDC RSW
                .RB     ↑1.RB
**[RSW]
**
**.NOCON .POP
%


        .MACRO  UNREL           ;(DEFINITELY) UNSHARED NREL
**
**      .NREL
%

        .MACRO  SNREL           ;(POSSIBLY) SHARED NREL
**
**.DO ASW!OBSW
**      .NREL   1
**.ENDC RSW
**      .NREL                   ;(CAN'T BE SHARED)
**[RSW]
%

        .MACRO  NREL            ;DEFAULT CASE
**
**      SNREL                   ;DEFAULT WILL BE SHARED
**
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
; DEFINE THE ENTRY MACROS

.MACRO  ENTRY
**      E.N.T=  E.N.T+1
**      I=      0
**      .DO     .ARGCT
**      I=      I+1
**      .ENT    ↑I
**      .ENDC
**      NREL
;**    .ZREL
**      I=      0
**      .DO     .ARGCT
**      I=      I+1
**↑I:
**      .ENDC
;**    E.ΔE.N.T
;**    NREL
;**E.ΔE.N.T:
%

.MACRO  ZENTRY
**      E.N.T=  E.N.T+1
**      I=      0
**      .DO     .ARGCT
**      I=      I+1
**      .ENT    ↑I
**      .ENDC
**      .ZREL
**      I=      0
**      .DO     .ARGCT
**      I=      I+1
**↑I:
**      .ENDC
**      E.ΔE.N.T
**      NREL
**E.ΔE.N.T:
%

.MACRO  NENTRY
**      E.N.T=  E.N.T+1
**      I=      0
**      .DO     .ARGCT
**      I=      I+1
**      .ENT    ↑I
**      .ENDC
**      NREL
**      I=0
**      .DO     .ARGCT
**      I=      I+1
**↑I:
**      .ENDC
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
;DEFINE THE WORD DEFINING MACROS

.MACRO  ZWORD
**      .PUSH   .LOC
**      .ZREL
**      I=      0
**      .DO     .ARGCT
**      I=      I+1
**↑I:   .BLK    1
**      .ENDC
**      .LOC    .POP
%

.MACRO  UNWORD
**      .PUSH   .LOC
**      I=      0
**      UNREL
**      .DO     .ARGCT
**      I=      I+1
**↑I:   .BLK    1
**      .ENDC
**      .LOC    .POP
%

.MACRO  SNWORD
**      .PUSH   .LOC
**      I=      0
**      SNREL
**      .DO     .ARGCT
**      I=      1+1
**↑I:   .BLK    1
**      .ENDC
**      .LOC    .POP
%

.MACRO  ZENT
**      .PUSH   .LOC
**      .ZREL
**      I=      0
**      .DO     .ARGCT
**      I=      I+1
**      .ENT    ↑I
**↑I:   .BLK    1
**      .ENDC
**      .LOC    .POP
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
.MACRO  UNENT
**      .PUSH   .LOC
**      I=      0
**      UNREL
**      .DO     .ARGCT
**      I=      I+1
**      .ENT    ↑I
**↑I:   .BLK    1
**      .ENDC
**      .LOC    .POP
%

.MACRO  SNENT

**      .PUSH   .LOC
**      I=      0
**      SNREL
**      .DO     .ARGCT
**      I=      I+1
**      .ENT    ↑I
**↑I:   .BLK    1
**      .ENDC
**      .LOC    .POP
%


; DEFINE VARIOUS TASKING MACROS

        .MACRO  SCH.EDULER      ;ENTER SCHEDULER STATE
**      .DO     RSW
**      .EXTN   EN.SCHED,.TSAVE
**      EN.SCHED
**      .TSAVE
**.ENDC SCH
**      SAVE    0               ;NO STACK TEMPORARIES
**      ?SURTM
**[SCH]
%

        .DO     ASW

        .MACRO  ?ABORT
**      E.N.T=  E.N.T+1
**      ELEF    0,A.ΔE.N.T              ;POINT TO KILL CODE
**      ?IDGOTO                 ;REDIRECT TASKING CODE SEQUENCE
**      JMP     B.ΔE.N.T
**      JMP     B.ΔE.N.T+1
**A.ΔE.N.T:     ?KILL
**B.ΔE.N.T:
%

        .MACRO  RE.SCHED
**      ?SCHED
%

        .MACRO  ER.SCHED
**      .EXTD   .FP
**      LDA     3,.FP
**      DSZ     FRTN,3
**      ?SCHED
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
        .MACRO  .SINGL
**      ?DRSCH                  ;DISABLE RESCHEDULING
**      JMP     .+1             ;IGNORE ANY ERRORS
%

        .MACRO  .MULTI
**      ?ERSCH                  ;ENABLE RESCHEDULING
**      JMP     .+1             ;IGNORE ANY ERRORS
%

        .ENDC


        .MACRO  LNTCB           ; LOAD AC WITH NUMBER OF TCBS
**      .DO     RSW
**      .EXTN   USTAD           ; ADDRESS OF RDOS UST
**      LDA     1,@LIT[,.GADD USTAD,USTCH]
**      HXR     2,1             ; GET OUT OT HIGH BYTE
**.ENDC NTCB
**      ELDA    1,UST+USTTL     ; GET NUMBER OF TCBS
**[NTCB]
%

        .MACRO  LTCB            ;LOAD AC WITH CURRENT TCB ADDR (MUST BE 2 OR 3)
**      .DO     RSW
**      .EXTD   CTCB
**      LDA     1,CTCB          ;AC(1) => CURRENT TCB
**.ENDC TCB
**      ELDA    1,UST+USTCT     ;AC(1) => CURRENT TCB
**[TCB]
%

        .MACRO  LTID            ;LOAD AC WITH TASK ID
**      .DO     RSW
**      LDA     1,TID,2         ;AC(1) <- TID
**.ENDC TIDL
**      LDA     1,?TIDPR,2              ;AC(1) <- TIDPR
**      HXR     2,1             ;ADS HAS IT IN LEFT BYTE OF TIDPR
**[TIDL]
%

        .MACRO  LTELN           ;LOAD TASK EXTENDER
**      .DO     RSW
**      LDA     1,TELN,2        ;AC(1) <- EXTENDER
**.ENDC ELN
**      LDA     1,?TELN,2       ;AC(1) <- EXTENDER
**[ELN]
%


        .MACRO  LSTAT           ;LOAD AC WITH TASK STATUS FROM TCB
**      .DO     RSW
**      LDA     1,TPRST,2       ;AC(1) <- STATUS
**.ENDC STAT
**      LDA     1,?TSTAT,2      ;AC(1) <- STATUS
**[STAT]
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
        .MACRO  SSTAT           ;STORE TASK STATUS FROM AC INTO TCB
**      .DO     RSW
**      STA     ↑1,TPRST,↑2     ;STORE STATUS
**.ENDC STAT
**      STA     ↑1,?TSTAT,↑2    ;STORE STATUS
**[STAT]
%

.MACRI  LPRI
**      .DO     RSW
**      LDA     ↑1,TPRST,↑2     ;AC(↑1) <- STATUS/PRIORITY
**      .ENDC   PRI
**      LDA     ↑1,?TIDPR,↑2    ;AC(↑1) <- TID/PRIORITY
**[PRI]
%

.MACRO  USTA
**      .DO     RSW
**      STA     ↑1,↑2           ;LOCAL STORE
**      .ENDC   STX
**      ESTA    ↑1,↑2           ;UNSHARED STORE FROM SHARED
**[STX]
%

.MACRO  ULDA
**      .DO     RSW
**      LDA     ↑1,↑2           ;LOCAL LOAD
**      .ENDC   LDX
**      ELDA    ↑1,↑2           ;LOAD FROM UNSHARED
**[LDX]
%

.MACRO  UJMP
**      .DO     RSW
**      JMP     ↑1              ;LOCAL JUMP
**      .ENDC   JMX
**      EJMP    ↑1              ;JUMP THRU UNSHARED
**[JMX]
%

.MACRO  UDSZ
**      .DO     RSW
**      DSZ     ↑1              ;LOCAL DECREMENT
**      .ENDC   DSX
**      EDSZ    ↑1              ;DECREMENT IN UNSHARED
**[DSX]
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
; SOME MACROS TO DEFINE EXTERNALS USING .EXTD AND .EXTN

.MACRO  D.LDA
.MACRO  LDA↑1
**      .EXTD   .↑1
**      LDA     _↑1,.↑1
_%
.MACRO  LDI↑1
**      .EXTD   .↑1
**      LDA     _↑1,@.↑1
_%
%

        D.LDA   SP
        D.LDA   FP
        D.LDA   SSE
        D.LDA   SOV
        D.LDA   GP
        D.LDA   PT
        D.LDA   RP
        D.LDA   DP

.MACRO  D.STA
.MACRO  STA↑1
**      .EXTD   .↑1
**      STA     _↑1,.↑1
_%
.MACRO  STI↑1
**      .EXTD   .↑1
**      STA     _↑1,@.↑1
_%
%

        D.STA   SP
        D.STA   FP
        D.STA   SSE
        D.STA   SOV
        D.STA   GP
        D.STA   PT
        D.STA   RP
        D.STA   DP
```

Figure G-2. DG/L Macro Definitions (continued)

```
.MACRO  D.JSR
.MACRO  ↑1
**      .EXTD   ↑2
**      JSR     @↑2
_%
%

.MACRO  D.XTN
.MACRO  ↑1
**      .EXTN   ↑2
**      ↑2
_%
%

        D.XTN   R.T.N           .RTN.
        D.XTN   SO.CHK          .SOCHK
        D.XTN   A.TERM          .ATERM
        D.XTN   A.INIT          .AINIT
        D.XTN   ST.BE           .STBE
        D.XTN   FRE.MEMORY      .FREMEMORY
        D.XTN   G.MEMORY        .GMEMORY
        D.XTN   MES.SAGE        .MESSAGE
        D.XTN   FTRP.           F.TRP

        D.JSR   G.FREE          .GFREE
        D.JSR   T.FREE          .TFREE
        D.JSR   1.STK           .ISTK
        D.JSR   G.ALLOCATE      .GALLOCATE
        D.JSR   T.ALLOCATE      .TALLOCATE
        D.JSR   CON.SOLE        .CONSOLE
        D.JSR   CR.DIS          .CRDIS
        D.JSR   CPY.S           .CPYS
        D.JSR   T.MVN           .TMVN
        D.JSR   T.MV            .TMV

.MACRO  RT.ERR
**      .EXTD   .RTER
**      .DO     .ARGCT==1
**      XLDA    ERAC,= ↑1
**      .ENDC
**      .DO     .ARGCT>1
:       WRONG NUMBER OF ARGUMENTS TO RT.ERR
**      .ENDC
**      JSR     @.RTER
%


.MACRO  ASSUME
**      .IFE    ↑1
**:     ;VIOLATED ASSUMPTION THAT ↑1
**.ENDC
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
.MACRO  ZCALL
**      .EXTD   ↑1
**      JSR     @↑1
%

.MACRO  NCALL
**      .EXTD   ↑1
**.DO   NSW
**      ZCALL   .ACALL
**      ↑1
**.ENDC
**.DO   ESW
**      EJSR    ↑1
**.ENDC
%

.MACRO  XCALL
**      .EXTN   ↑1
**      ↑1
%

.MACRO  ZLDA
**      .EXTD   ↑2
**      LDA     ↑1,↑2
%

.MACRO  ZLDI
**      .EXTD   ↑2
**      LDA     ↑1,@↑2
%

.MACRO  ZSTA
**      .EXTD   ↑2
**      STA     ↑1,↑2
%
```

Figure G-2. DG/L Macro Definitions (continued)

```
.MACRO  ZSTI
**      .EXTD   ↑2
**      STA     ↑1,@↑2
%

.MACRO  ZISZ
**      .EXTD   ↑1
**      ISZ     ↑1
%

.MACRO  ZISZI
**      .EXTD   ↑1
**      ISZ     @↑1
%

.MACRO  ZDSZ
**      .EXTD   ↑1
**      DSZ     ↑1
%

.MACRO  ZDSZI
**      .EXTD   ↑1
**      DSZ     @↑1
%

.MACRO  ZJMP
**      .EXTD   ↑1
**      JMP     ↑1
%

.MACRO  ZJMPI
**      .EXTD   ↑1
**      JMP     @↑1
%

                .EOT
```

Figure G-2. DG/L Macro Definitions (continued)

```
;  DGLPARAM
;
; PARAMETER TAPE FOR CHANGES

                .TITLE  DGLPARAM
                .ENT .MSTK,  .DSTACK, .XFIT, MINRES, .NMAX
                .ENT    .CNTFL, .FTBUF, .FTLBZ
                .ENT    .UMEM,.VMIN
                .NREL

        ;ERROR MESSAGE WRITER

  ;     .EXTN   NOMESSAGE       ;REMOVES MESSAGE WRITER (CA 300. WORDS)

  ;     .EXTN   SHORTMESSAGE    ;BRINGS IN SHORT MESSAGE WRITER
                                ;  (SAVES CA. 260 WORDS)

;       .ENT    EOFER           ;ERROR CODE FOR END OF FILE
;       EOFER:  AIEOF+NONFATAL

        ;VIRTUAL CM PACKAGE

  ;     .EXTD   NUMBUF          ;BRINGS IN VIRTUAL CM (CA 25. WORDS)

        .UMEM:  0               ;NUMBER OF 1K BLOCKS NEEDED FOR USER PROGRAM
                                ;  0 MEANS NO VIRTUAL REQUESTED
        .VMIN:  1               ;MIN. EXTRA VIRTUAL BLOCKS REQUIRED TO GO VIRTUAL

        ;TASKING PARAMETERS

.MSTK:  0               ;STACK SIZE FOR .MAIN
.DSTACK: 200.           ;DEFAULT STACK SIZE IS (MEMORY-.DSTACK)/ # TASKS
.XFIT:  0               ;NON-ZERO: GALLOCATE EXACT FIT INSTEAD OF FIRST FIT

        ;STUFF.LB PARAMETERS

.NMAX:  077777          ;MAX AMOUNT OF CORE TAKEN

;  .ENT  .LLOC          ;0 FOR NOT FREEING UP THE INITIALIZER
;  .LLOC:  0            ;  (MAKES REENTRANT UNDER RTOS)

;       .ENT    .SPREAD
        .ZREL

;.SPREAD=        1       ;NON-ZERO => SPACE, TAB, COMMA, SEMICOLON, CR USED AS
                        ;       DELIMITERS IN STRING READS; OTHERWISE ONLY CR

        .NREL
```

Figure G-3. DG/L Parameter Definitions

```
        ;EXTERNAL DATA AREAS

;       .ENT    COM1,COM2,COM3,COM4,COM5,COM6,COM7,COM8,COM9,COM10
;       .NREL

;  COM1:        .BLK    4               ;EXTERNAL INTEGER, BOOLEAN, OR REALS
;  COM2:        .BLK    4
;  COM3:        .BLK    4
;  COM4:        .BLK    4
;  COM5:        .BLK    4
;  COM6:        .BLK    4
;  COM7:        .BLK    4
;  COM8:        .BLK    4
;  COM9:        .BLK    4
;  COM10:       .BLK    4

        ;EXTERNAL DECLARATION TEMPLATES

;       .ENT    ESTRING,EPOINTER,EARRAY
;       .NREL

;       EPOINTER:       .PTR            ;EXTERNAL POINTER
;       .PTR:   .BLK    1

;       ESTRING:        .STR*2          ;EXTERNAL STRING
;                       MAXLENGTH
;                       CURLENGTH
;       .STR:           .TXT    ""
;       MAXLENGTH=      0
;       CURLENGTH=      0
; 2 DIMENSION EXTERNAL INTEGER ARRAY

;       LBND2                   ;LOWER BOUND, DIM 2
;       HBND2                   ;UPPER BOUND, DIM 2
;       LBND1                   ;LOWER BOUND, DIM 1
;       HBND1                   ;UPPER BOUND, DIM 1
;       2                       ;NUMBER OF DIMENSIONS
;       EARRAY: .BLK    (HBND2-LBND2+1)*(HBND1-LBND1+1)
;       HBND1=  0
;       LBND1=  0
;       HBND2=  0
;       LBND2=  0

;               .ENT    .VMAX

;       N=0             ;NUMBER OF 1024 WORD BUFFERS TO LEAVE FOR INFOS
;       .VMAX:  N*1024  ;0 => INFOS NOT USED, LEAVE ALL SPACE FOR USER STACK
;                       ; NON-0 => INFOS USED, LEAVE BUFFERS FOR HYPERSPACE

;               .ENT    .SLEF

;       .SLEF=  0       ;0 -> DO NOT ENABLE LEF MODE ON PROGRAM INITIATION

;               .ENT    .RLEF

;       .RLEF=  0       ;0 -> DO NOT DISABLE LEF MODE ON PROGRAM TERMINATION
```

Figure G-3. DG/L Parameter Definitions (continued)

```
        ;I/O PARAMETERS (OPSYS.LB)

                .ZREL
MINRES: -3              ;LOWER BOUND OF A TREE NODE
                .NREL
.CNTFL= 16.             ;NUMBER OF CHANNELS


; DSTSZ
;
; DEFAULT STRING SIZE FOR STRING EXPRESSION TEMPORARIES

;               .ENT    .STSZ
;               .ZREL

;.STSZ: STSZ.
;
;               .NREL
;
;STSZ.: 32.

                ;DO NOT CHANGE THE FOLLOWING
.FTBUF= -.CNTFL
.FTLBZ= -2*.CNTFL

                .END
```

Figure G-3. DG/L Parameter Definitions (continued)

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. Commands, calls, and acronyms are in uppercase letters (e.g., CREATE); all others are lowercase.

DataGeneral

Software Documentation Remarks Form

How Do You Like This Manual?

Title _____ No. _____

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

Who Are You?

- ☐ EDP Manager
- ☐ Senior System Analyst
- ☐ Analyst/Programmer
- ☐ Operator
- ☐ Other _____

What programming language(s) do you use? _____

_____

How Do You Use This Manual?

(List in order: 1 = Primary use)

- ____ Introduction to the product
- ____ Reference
- ____ Tutorial Text
- ____ Operating Guide
- ____ _____

Do You Like The Manual?

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the topic order easy to follow?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you everything you need to know?

Comments?

(Please note page number and paragraph where applicable.)

From:

Name _____ Title _____ Company _____

Address _____ Date _____

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP

STAPLE