IAS System Library Routines Reference Manual

Order Number: AA-5580C-TC

Operating System and Version: IAS Version 3.4

DDIF	IAS	VAX C
DEC	MASSBUS	VAXcluster
DEC/CMS	PDP	VAXstation
DEC/MMS	PDT	VMS
DECnet	RSTS	VR150/160
DECUS	RSX	VT
DECwindows	ULTRIX	
DECwrite	UNIBUS	
DIBOL	VAX	digital

This document was prepared using VAX DOCUMENT, Version 1.2

Contents

TABLES

Preface

Manual Objectives

The *IAS System Library Routines Reference Manual* describes the use and function of the system library routines that may be called from MACRO-11 assembly language programs.

Intended Audience

This manual is intended for use by experienced MACRO-11 assembly language programmers, IAS system managers, and applications programmers.

Document Structure

Chapter 1 presents a general description of the services provided by the system library routines and their functional relationships.

Chapter 2 describes the use and function of the register handling routines.

Chapter 3 describes the use and function of the arithmetic routines.

Chapter 4 describes the use and function of the input data conversion routines.

Chapter 5 describes the use and function of the output data conversion routines.

Chapter 6 describes the use and function of the output formatting routines.

Chapter 7 describes the use and function of the dynamic memory management routines.

Chapter 8 describes the use and function of the virtual memory management routines.

Chapter 9 summarizes the calling sequences of the system library routines.

Appendix A presents a cross-reference system bibliography of other manuals that describe routines available to IAS system users.

Appendix B describes a routine that enables a program to access modules in a universal library as if they were files.

Associated Documents

The following manuals are prerequisite sources of information for readers of this manual:

- The *PDP-11 MACRO-11 Language Reference Manual*
- The *IAS Task Builder Manual*
- The manuals referred to in Appendix A

Readers should also refer to the *IAS Documentation Directory and Master Index* for descriptions of other documents associated with this manual.

1 Introduction

The routines described in this manual were written to provide commonly needed capabilities for DIGITAL-supplied utilities. We supply documentation for them because the routines are general enough to be used regularly by most MACRO-11 programmers. Note, however, that the basic functionality of the routines described in this manual cannot be changed because of the potentially widespread effect it might have on our system utilities.

The system library routines can be called by MACRO-11 assembly language programs to perform the following services:

- Save and restore register contents to enable transfers of control between the calling program and called subroutines
- Perform integer and double-precision multiplication and division
- Convert ASCII input data to internal binary and Radix-50 format
- Convert internal binary and Radix-50 data to ASCII output data
- Convert and format output data to produce text for a readable printout or display
- Manage the dynamic memory space available to the task that requires a small-to-moderate amount of resident memory for data
- Manage memory and disk file storage to accommodate tasks that require large amounts of memory for data that must be transferred between memory and a disk work file

This manual describes the procedures for calling the library routines from within the source program, the output that is returned to the executing task, and the interaction between the library routines and the executing task.

The system library routines interface with each other to perform their various services. For example, the data conversion routines call the arithmetic routines to perform the required multiplication and division. All library routines preserve the contents of the calling task's registers, generally by calling the appropriate register handling routine to do the following:

- Save register contents on the stack
- Subsequently restore the contents of the registers
- Return control to the calling task

The data conversion and format control functions performed by the Edit Message Routine require calls to the output data conversion routines, which in turn call other routines.

The virtual memory management routines function as an automatic control system to allocate and deallocate memory, maintain page addresses and status, and swap pages between memory and disk storage to accommodate large amounts of data in a limited amount of physical (dynamic) memory.

The system library routines communicate with the calling task by means of registers where output is returned or by settings of the C bit in the Condition Code of the Processor Status Word. The calling task can usually determine whether a requested service was successfully performed by examining the output register or registers or by testing the C bit setting when control is returned from the library routine. Exceptions to this procedure are described in the detailed discussions of given routines.

The system library routines are supplied to users as object code in the following files:

- The system library file (SYSLIB.OLB), which contains the following routines:
  - Register handling routines (described in Chapter 2)
  - Arithmetic routines (described in Chapter 3)
  - Input and output data conversion routines (described in Chapters 4 and 5)
  - Output formatting routines (described in Chapter 6)
  - Dynamic memory allocation and release routines (described in Chapter 7)
  - Universal library access routines (described in Appendix B)
- The memory management routines file (VMLIB.OLB), which contains the dynamic and virtual memory management routines.

At task-build time, the Task Builder will automatically search the system library file for any referenced routines. However, the VMLIB.OLB file must be specified at task-build time if a task has referenced the dynamic memory initialization routine (described in Chapter 7) or any of the virtual memory management routines (described in Chapter 8 of this manual).

A summary of each procedure for using the system library routines is given in Chapter 9. This is quick-reference material provided for the MACRO-11 assembly language programmer who has become familiar with the detailed procedures that are explained in Chapters 2 through 8.

Additional Executive and I/O routines available to IAS system users are described in other manuals. See the *IAS Documentation Directory and Master Index* for more information.

If the task that includes system library routines also references a position-independent resident library, it is possible that program section names might conflict. Routines included in a task cannot reside in the same program section as routines referenced in the position-independent resident library. Table 1–1 lists the program section names and the system library routines that reside in each program section. If your task includes a routine that uses a program section listed in Table 1–1 and the task also references a position-independent resident library routine that uses the same program section, the Task Builder generates a fatal error. To determine how to include the code in your task and avoid a conflict of program section names, refer to the *IAS Task Builder Manual*.

Table 1–1 Program Section Names for SYSLIB Routines

SYSLIB Routines		
Program Section Name	**Module Name**	**Routine Name(s)**
.BLK.	CATB	$CDTB
		$COTB
	CAT5	$CAT5
	CBTA	$CBDAT
		$CBDMG
		$CBDSG
		$CBOMG
		$CBOSG
		$CBTA
		$CBTMG
	CDDMG	$CDDMG
	CVTUC	$CVTUC
	C5TA	$C5TA
	EDDAT	$DAT
		$TIM
	OD2CT	.DD2CT
		.OD2CT
	SAVAL	$SAVAL
	SAVVR	$SAVVR
PUR$D	CAT5B (data)	$CAT5B
	EDTMG (data)	$EDTMG
PUR$I	CAT5B (instruction)	$CAT5B
	EDTMG (instruction)	$EDTMG
$$RESL	SAVRG	$SAVRG
	SAVR1	.SAVR1
$$RESM	ARITH	$DIV
		$MUL
	DARITH	$DDIV
		$DMUL

2 Register Handling Routines

The system library contains the following register handling routines:

- Save All Registers Routine ($SAVAL), which saves and subsequently restores Registers 0 through 5
- Save Registers 3–5 Routine ($SAVRG), which saves and subsequently restores Registers 3 through 5
- Save Registers 0–2 Routine ($SAVVR), which saves and subsequently restores Registers 0 through 2
- Save Registers 1–5 Routine (.SAVR1), which saves and subsequently restores Registers 1 through 5

The register handling routines function as coroutines to enable control swapping between themselves, a subroutine, and the original caller of the subroutine. The register handling routines are also called by other routines in the system library, as noted throughout this manual.

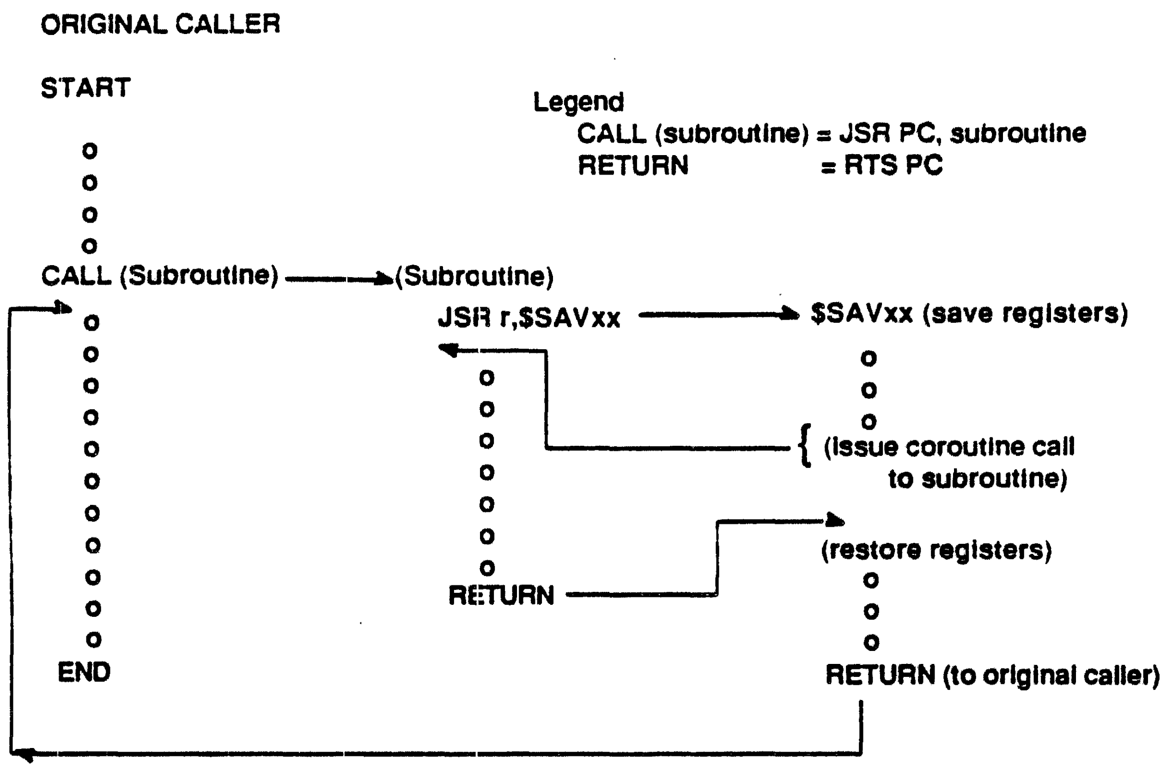
To illustrate the effect of using the register handling routines, assume the following situation:

1. An original caller calls a subroutine.
2. The subroutine calls a register handling coroutine.
3. The coroutine preserves (pushes onto the stack) the contents of the specified registers and issues a coroutine call back to the subroutine.
4. The subroutine executes to completion, then a return instruction is executed to swap control back to the coroutine.
5. The coroutine restores (pops from the stack) the initial contents of the registers and returns to the original caller.

Figure 2–1 illustrates the control swapping function performed by the register handling routines.

The register handling routines are called by other routines in the system library, as noted in this manual.

Figure 2–1 Control Swapping of the Register Handling Routines



$SAVAL

The $SAVAL routine saves and subsequently restores Registers 0 through 5 for a subroutine. The $SAVAL routine functions as a coroutine that swaps control between itself, a subroutine, and the original caller.

To call the $SAVAL routine, include the following Jump to Subroutine instruction in your subroutine:

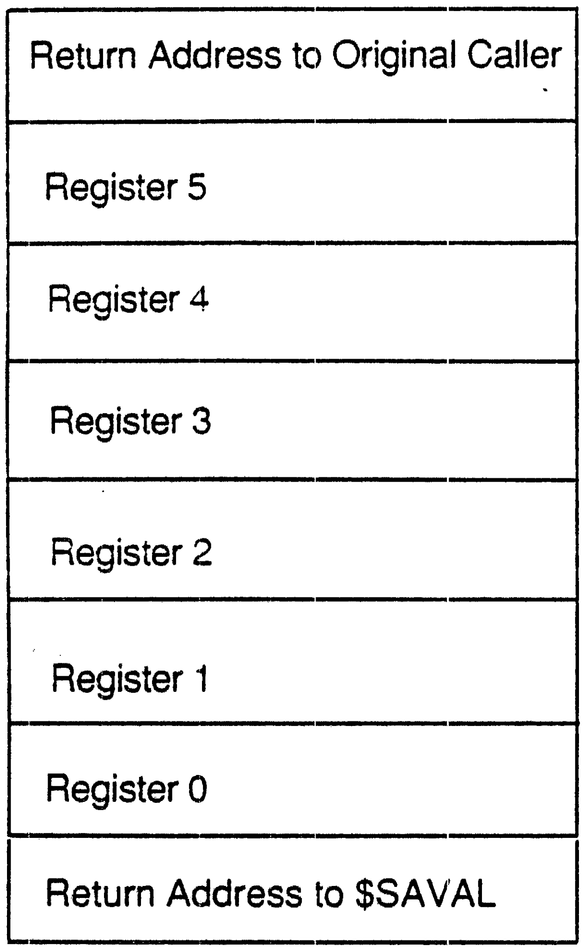
```
JSR PC,$SAVAL
```

The subroutine must return control to the $SAVAL routine with a RETURN source statement.

On entry to the $SAVAL routine, the program stack contains the return address to the original caller and the return address of the subroutine. The $SAVAL routine pushes the contents of registers 4 through 0 to the stack.

The $SAVAL routine moves the subroutine return address to the position following the contents of Register 0 and moves the current contents of R5 to the stack above the contents of R4.

The $SAVAL routine issues a coroutine call, in the form CALL @(SP)+, to swap control back to the subroutine. The coroutine call replaces the subroutine return address with the return address to the $SAVAL routine. When control returns to the subroutine the stack pointer points to $SAVAL's return address. The stack contains the following:



The subroutine executes until a RETURN (RTS PC) instruction is executed, which swaps control back to the $SAVAL routine. The contents of R0 through R5 are restored (popped from the stack) and the $SAVAL routine returns, by means of an RST PC instruction, to the original caller.

NOTE: For $SAVAL to work properly (that is, return control to the original caller), the routine that calls $SAVAL must itself have been invoked by the CALL instruction (that is, JSR PC, subroutine).

$SAVRG—Save registers 3–5

The $SAVRG routine saves and subsequently restores Registers 3 through 5 for a subroutine. The $SAVRG routine functions as a coroutine that swaps control between itself, a subroutine, and the original caller.

To call the $SAVRG routine, the subroutine must contain the following Jump to Subroutine instruction:

```
        JSR R5,$SAVRG
```

The subroutine must return control to the $SAVRG routine with a RETURN source statement.

On entry to the $SAVRG routine, the program stack contains the return address to the original caller and the contents of R5 of the original caller. The $SAVRG routine pushes the contents of registers 4 and 3 to the stack, then pushes the current contents of R5 (return address to the subroutine) to the stack.

DESCRIPTION

The $SAVRG routine copies the original contents back into R5 and issues a coroutine call in the form CALL @(SP)+, to swap control back to the subroutine. The coroutine call replaces the subroutine's return address with the return address to the $SAVRG routine. When control returns to the subroutine, the stack pointer points to $SAVRG's return address. The stack contains the following:

Return Address to Original Caller
Register 5 contents of Original Caller
Register 4
Register 3
Return Address to $SAVRG

The subroutine executes until a RETURN (RTS PC) instruction is executed; this swaps control back to the $SAVRG routine. The contents of Registers 3 through 5 are restored (popped from the stack) and the $SAVRG routine RETURNs via an RTS PC instruction to the original caller.

NOTE: For $SAVRG to work properly (that is, return control to the original caller), the routine that calls $SAVRG must itself have been invoked by the CALL instruction (that is, JSR PC, subroutine).

$SAVVR—Save registers 0–2

The $SAVVR routine saves and subsequently restores Registers 0 through 2 for a subroutine. The $SAVVR routine functions as a coroutine that swaps control between itself, a subroutine, and the original caller.

To call the $SAVVR routine, the subroutine must contain the following Jump to Subroutine instruction:

```
JSR R2,$SAVVR
```

DESCRIPTION

On entry to the $SAVVR routine, the program stack contains the return address to the original caller and the contents of Register 2 of the original caller. The $SAVVR routine pushes the contents of Registers 1 and 0 to the stack, then pushes the current contents of Register 2 (the return address to the subroutine) to the stack.

The $SAVVR routine copies the original contents back into Register 2 and issues a coroutine call, in the form CALL @(SP)+, to swap control back to the subroutine. The coroutine call replaces the subroutine's return address with the return address to the $SAVVR routine. When control returns to the subroutine, the stack pointer points to $SAVVR's return address. The stack contains the following information:

Return Address to Original Caller
Register 2 contents of Original Caller
Register 1
Register 0
Return Address to $SAVVR

The subroutine executes until a return instruction (RTS PC) is executed; this swaps control back to the $SAVVR routine. The contents of Registers 0 through 2 are restored (popped from the stack) and the $SAVVR routine returns, by means of the RTS PC instruction, to the original caller.

.SAVR1—Save registers 1–5

The .SAVR1 routine saves and subsequently restores Registers 1 through 5 for a subroutine. The .SAVR1 routine functions as a coroutine that swaps control between itself, a subroutine, and the original caller.

To call the .SAVR1 routine, the subroutine must contain the following Jump to Subroutine instruction:

```
JSR R5,.SAVR1
```

The subroutine must return control to the .SAVR1 routine with a RETURN source statement.

DESCRIPTION

On entry to the .SAVR1 routine, the program stack contains the return address to the original caller and the contents of Register 5 of the original caller. The .SAVR1 routine pushes the contents of Registers 4, 3, 2, and 1, and the current contents of Register 5 (the return address to the subroutine) to the stack.

The .SAVR1 routine copies the original contents back into Register 5 and issues a coroutine call, in the form CALL @(SP)+, to swap control back to the subroutine. The coroutine call replaces the subroutine return address with the return address to the .SAVR1 routine. When control returns to the subroutine, the stack pointer points to .SAVR1's return address. The stack contains the following information:

Return Address to Original Caller
Register 5 contents of Original Caller
Register 4
Register 3
Register 2
Register 1
Return Address to .SAVR1

The subroutine executes until a return instruction (RTS PC) is executed; this swaps control back to the .SAVR1 routine. The contents of Registers 1 through 5 are restored (popped from the stack) and the .SAVR1 routine returns, by means of the RTS PC instruction, to the original caller.

NOTE: For .SAVR1 to work properly (that is, return control to the original caller), the routine that calls .SAVR1 must itself have been invoked by the CALL instruction (that is, JSR PC, subroutine).

3 Arithmetic Routines

The system library contains four arithmetic routines that perform unsigned integer multiplication and division. This chapter describes the use and function of the following types of arithmetic routines:

1 Integer Arithmetic Routines

   The following routines perform arithmetic operations on 16-bit unsigned integer values:

   - The Integer Multiply Routine ($MUL), which multiplies integer values
   - The Integer Divide Routine ($DIV), which divides integer values

2 Double-Precision Arithmetic Routines

   The following routines perform double-precision arithmetic operations:

   - The Double-Precision Multiply Routine ($DMUL), which multiplies an unsigned double-precision value by a single-precision multiplier to produce a double-precision product
   - The Double-Precision Divide Routine ($DDIV), which divides an unsigned double-precision dividend by an unsigned single-precision divisor to produce a double-precision result

$MUL—Integer Multiply Routine

The $MUL routine multiplies two single-word unsigned integer input values to produce an unsigned double-word product.

FORMAT

CALL $MUL

INPUT

multiplier
In Register 0: a single-word unsigned integer

multiplicand
In Register 1: a single-word unsigned integer

OUTPUT

product (high-order)
In Register 0: the high-order part of the result

product (low-order)
In Register 1: the low-order part of the result

DESCRIPTION

The $MUL routine preserves Registers 2 through 5 of the calling task. It does not return any error indications to the caller.

EXAMPLE

The following source statements call the $MUL routine to perform multiplication and store the results in the buffer WORK:

```
WORK:   .BLKW   2             ; OUTPUT BUFFER

        MOV     #1200,R0      ; PUTS MULTIPLIER IN REGISTER 0
        MOV     #36,R1        ; PUTS THE MULTIPLICAND IN REGISTER 1
        CALL    $MUL          ; CALLS $MUL ROUTINE
        MOV     R0,WORK       ; SAVES HIGH-ORDER PART OF RESULT
        MOV     R1,WORK+2     ; SAVES LOW-ORDER PART OF RESULT
```

$DIV—Integer divide routine

The $DIV routine performs unsigned integer division.

FORMAT

CALL $DIV

INPUT

dividend
In Register 0: an unsigned integer

divisor
In Register 1: an unsigned integer

OUTPUT

quotient
In Register 0: the quotient

remainder
In Register 1: the remainder

DESCRIPTION

The $DIV routine preserves Registers 2 through 5 of the calling task. It does not return any error indications to the caller.

EXAMPLE

The following source statements call the $DIV routine to perform division and store the results in Registers 0 and 1:

```
FRACTN: .WORD   1               ; BUFFER FOR REMAINDER

        MOV     #36.,R0         ; SET DIVIDEND
        MOV     #8.,R1          ; SET DIVISOR
        CALL    $DIV            ; DIVIDE
        MOV     R1,FRACTN       ; SAVE REMAINDER
```

Double-Precision Multiply Routine—$DMUL

The $DMUL routine multiplies an unsigned double-precision value by an unsigned single-precision value to produce an unsigned double-precision product.

FORMAT

CALL $DMUL

INPUT

multiplier
In Register 0: an unsigned single-precision magnitude value

multiplicand (high-order)
In Register 2: the high-order part of an unsigned double-precision magnitude value

multiplicand (low-order)
In Register 3: the low-order part of the unsigned double-precision magnitude value

OUTPUT

product (high-order)
In Register 0: the high-order part of the product

product (low-order)
In Register 1: the low-order part of the product

DESCRIPTION

The $DMUL routine preserves Registers 4 and 5 of the calling task, clears the C bit, and destroys the contents of Registers 2 and 3 upon return to the caller. The $DMUL routine does not return any error indications to the caller.

EXAMPLE

The following source statements call the $DMUL routine to multiply the number stored in Registers 2 and 3 by 127_{10} and store the result in Registers 0 and 1:

```
MOV     R5,R2           ; HIGH-ORDER PART OF MULTIPLICAND
MOV     R4,R3           ; LOW-ORDER PART OF MULTIPLICAND
MOV     #127.,R0        ; MULTIPLIER
CALL    $DMUL           ; MULTIPLY BY 127.
```

$DDIV—Double-precision divide routine

The $DDIV routine divides an unsigned double-precision integer dividend by an unsigned single-precision (15-bit) divisor to produce an unsigned double-precision result.

FORMAT

CALL $DDIV

INPUT

divisor
In Register 0: an unsigned double-precision integer

dividend (high-order)
In Register 1: the high-order part of an unsigned single-precision integer

dividend (low-order)
In Register 2: the low-order part of an unsigned single-precision integer

OUTPUT

remainder
In Register 0: the remainder

quotient (high-order)
In Register 1: the high-order part of the quotient

quotient (low-order)
In Register 2: the low-order part of the quotient

DESCRIPTION

The $DDIV routine preserves the contents of Registers 3 through 5 of the calling task. The $DDIV routine does not return any error conditions to the caller.

EXAMPLE

The following source statements call the $DDIV routine to perform division and store the results in Registers 0, 1, and 2:

```
DVD:    .BLKW  2               ; BUFFER TO STORE HIGH-ORDER OF DIVIDEND
QUOT:   .BLKW  2               ; BUFFER TO STORE HIGH-ORDER OF QUOTIENT
RMAIN:  .BLKW  1               ; BUFFER FOR REMAINDER
        MOV    #150,R0         ; PUT DIVISOR IN REGISTER 0
        MOV    DVD,R1          ; SET UP HIGH-ORDER PART OF DIVIDEND
        MOV    DVD+2,R2        ; SET UP LOW-ORDER PART OF DIVIDEND
        CALL   $DDIV           ; CALL $DDIV ROUTINE
        MOV    R1,QUOT         ; PUT HIGH-ORDER PART OF QUOTIENT IN BUFFER
        MOV    R2,QUOT+2       ; PUT LOW-ORDER PART OF QUOTIENT IN BUFFER
        MOV    R0,RMAIN        ; PUT REMAINDER IN RMAIN
```

4 Input Data Conversion Routines

The input data conversion routines accept ASCII data as input and convert it to the specified numeric representation. The following three types of routines perform input data conversion:

- ASCII to binary double-word conversion routines, which accept ASCII decimal or octal input numbers and convert them to double-word binary numbers
- ASCII to binary conversion routines, which accept ASCII decimal or octal input numbers and convert them to single-word binary numbers
- ASCII to Radix-50 conversion routines, which accept the Radix-50 set of ASCII characters as input and convert them to Radix-50 internal format

4.1 ASCII to Binary Double-Word Conversions

The following system library routines convert ASCII input numbers to double-word binary numbers:

- The Decimal to Binary Double-Word Routine (.DD2CT), which accepts ASCII decimal numbers as input and converts them to double-word binary format
- The Octal to Binary Double-Word Routine (.OD2CT), which accepts ASCII octal numbers as input and converts them to double-word binary format

4.2 ASCII to Binary Conversions

The following routines convert unsigned ASCII input numbers to single-word unsigned binary numbers:

- The Decimal to Binary Conversion Routine ($CDTB), which accepts ASCII decimal numbers as input and converts them to single-word binary format
- The Octal to Binary Conversion Routine ($COTB), which accepts ASCII octal numbers as input and converts them to single-word binary format

These routines call the Integer Multiply Routine ($MUL) to perform the multiplication required for the conversion.

4.3 ASCII to Radix-50 Conversions

The following routines convert ASCII alphanumeric input characters to 16-bit Radix-50 values:

- The ASCII to Radix-50 Conversion Routine ($CAT5), which accepts input characters from the ASCII character Radix-50 subset and converts them to Radix-50 format[1]

[1] See the *PDP-11 MACRO-11 Language Reference Manual* for a complete listing of the Radix-50 character set and ASCII equivalents.

- The ASCII with Blanks to Radix-50 Conversion Routine ($CAT5B), which accepts input characters from the ASCII character Radix-50 subset and blank characters and converts them to Radix-50 format[1]

Both routines call the Integer Multiply Routine ($MUL) to perform the multiplication required for the conversion.

.DD2CT—Decimal to binary double-word routine

The .DD2CT routine converts a signed ASCII decimal number string to a double-length (2-word) signed binary number.

FORMAT

CALL .DD2CT

INPUT

output address

In Register 3: the address of the 2-word output field where the converted number is to be stored

number input characters

In Register 4: the number of characters in the string to be converted

input string address

In Register 5: the address of the character string to be converted

OUTPUT

binary result (high-order)

In word 1 of the output field: the high-order 16 bits of the converted number

binary result (low-order)

In word 2 of the output field: the low-order 16 bits of the converted number

Condition code

C bit	=	Clear if conversion was successful
C bit	=	Set if an illegal character was found and conversion was incomplete

DESCRIPTION

The .DD2CT routine accepts leading plus (+) or minus (−) signs and a trailing period (.) in the string to be converted. A preceding pound sign (#) forces octal conversion; a pound sign and a period in the same input string is invalid. The numbers 0 to 9 are acceptable characters in the decimal number string itself. Any other characters in the string will cause the .DD2CT routine to terminate the conversion procedure. The value range of a decimal number to be converted is -2^{31} to $+2^{31} - 1$.

The .DD2CT routine saves and restores all of the calling task's registers.

EXAMPLE

The following source statements call the .DD2CT routine to convert an ASCII decimal number string (pointed to by buffer ICHR), store the binary result in the address pointed to by buffer BOUT, and check the results upon return:

```
ICHR:   .ASCII /1234567./
        .EVEN
BOUT:   .BLKW   2

        MOV     #BOUT,R3      ; GET ADDRESS OF THE 2-WORD OUTPUT FIELD
        MOV     #10,R4        ; GET THE NUMBER OF INPUT CHARACTERS
        MOV     #ICHR,R5      ; GET ADDRESS OF THE INPUT CHARACTER STRING
        CALL    .DD2CT        ; CONVERT THE STRING
        BCS     100$          ; BRANCH IF C BIT SET (CONVERSION WAS NOT SUCCESSFUL)
           .
           .                  ; PROGRAM CONTINUES
           .
100$:   CALL ERR              ; CALL ROUTINE TO OUTPUT ERROR MESSAGE
```

.OD2CT—Octal to binary double-word routine

The .OD2CT routine converts an ASCII octal number string to a double-length (2-word) binary number.

FORMAT

CALL .OD2CT

INPUT

output address

In Register 3: the address of the 2-word output field in which the converted number is to be stored

number input characters

In Register 4: the number of characters in the string to be converted

input string address

In Register 5: the address of the character string to be converted

OUTPUT

binary result (high-order)

In word 1 of the output field: the high-order 16 bits of the converted number

binary result (low-order)

In word 2 of the output field: the low-order 16 bits of the converted number

Condition Code

C bit	=	Clear if conversion was successful
C bit	=	Set if an illegal character was found and conversion was incomplete

DESCRIPTION

The .OD2CT routine accepts leading plus (+) or minus (–) signs and a trailing period (.) in the string to be converted. A preceding pound sign (#) is accepted but unnecessary; a pound sign and a period in the same input string is invalid. A trailing period forces decimal conversion. (This is because the .OD2CT routine is an entry point in the .DD2CT routine, which converts decimal number strings to binary double-word values.) Acceptable characters in the octal number string itself are the numbers 0 to 7.

The .OD2CT routine terminates the conversion process if you use any other characters in the ASCII octal number string.

The value range of an octal number you can convert is -2^{31} to $+2^{31} - 1$.

The .OD2CT routine saves and restores all of the calling task's registers.

EXAMPLE

The following source statements call the .OD2CT routine to convert an ASCII octal number string (pointed to by buffer ICHR), store the binary result in the address pointed to by buffer BOUT, and check the results upon return:

```
ICHR:    .ASCII /2461357/
         .EVEN
BOUT:    .BLKW    2

         MOV      #BOUT,R3      ; GET ADDRESS OF THE 2-WORD OUTPUT
         MOV      #7,R4         ; GET THE NUMBER OF INPUT CHARACTERS
         MOV      #ICHR,R5      ; GET ADDRESS OF THE INPUT CHARACTER STRING
         CALL     .OD2CT        ; CONVERT THE STRING
         BCS      100$          ; BRANCH IF C BIT SET (INPUT STRING
            .                   ;   CONVERSION WAS NOT SUCCESSFUL)
            .                   ; IF C BIT CLEAR, CONVERSION WAS SUCCESSFUL
            .                   ;   AND THE PROGRAM CONTINUES
100$:    CALL ERR               ; CALL ROUTINE TO OUTPUT ERROR MESSAGE
```

$CDTB—Decimal to binary conversion routine

The $CDTB routine converts an unsigned ASCII decimal number to binary format.

FORMAT

CALL $CDTB

INPUT

input buffer address

In Register 0: the address of the first byte of the ASCII decimal character string to be converted

OUTPUT

next byte address

In Register 0: the address of the next byte of the input buffer

binary number

In Register 1: the converted number

terminator

In Register 2: the terminating character of the input buffer

DESCRIPTION

The numbers 0 to 9 are valid characters in the input decimal number. All other input characters are invalid and are not converted by this routine. The end of a string of numbers must be marked by a terminating character, which can be any ASCII character except the numbers 0 to 9. Examples of terminating characters are a blank, tab character, alphabetic character, and special symbol. Leading blanks and tab characters are ignored.

The maximum value of a decimal number that can be converted by the $CDTB routine is 65,535. Numbers of greater value will cause indeterminate results since the $CDTB routine does not check the value range of an input number. Also, the routine does not return a significant Condition Code setting to the calling task.

Because the $CDTB routine returns the address of the next byte in the input buffer to the calling task, you can convert successive strings by setting up a processing loop back to the CALL $CDTB statement (see the example for this routine).

$CDTB calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task.

NOTE: You can determine, in the task, whether an input string was successfully converted by testing the contents of Register 2. If the contents are other than the expected terminating character, the conversion was incomplete because the routine found an invalid character in the input string.

EXAMPLE

The following source statements define a processing loop, using the $CDTB routine, to convert a series of ASCII decimal character strings to binary numbers. This example uses the tab character as the terminating character of each string and the space character as the terminating character of the input buffer. If converted successfully, the binary numbers will be stored in the buffer BNUM:

```
IBUF:    .ASCII /123/<11>/4567/<11>/89/<11>/87654/<40>
         .EVEN
BNUM:    .BLKW    4          ; BUFFER FOR CONVERTED NUMBERS
    .
    .
    .
         MOV     #BNUM,R4    ; GET THE OUTPUT BUFFER ADDRESS
         MOV     #IBUF,R0    ; SET UP INPUT BUFFER ADDRESS
LOOP:    CALL    $CDTB       ; CONVERT THE STRING
         MOV     R1,(R4)+    ; SAVE CONVERTED STRING
         CMP     #11,R2      ; COMPARE ASCII TAB (HT) VALUE TO TERMINATING
                             ;   CHARACTER RETURNED IN REGISTER 2
         BEQ     LOOP        ; IF EQUAL, STRING SUCCESSFULLY CONVERTED,
                             ;   GO BACK THROUGH LOOP TO CONVERT NEXT INPUT
                             ;      STRING POINTED TO BY REGISTER 0
         CMP     #40,R2      ; COMPARE SPACE VALUE (40) WITH TERMINATING
                             ;   CHARACTER IN REGISTER 2
         BEQ     10$         ; IF EQUAL, CONTINUE PROGRAM (ALL INPUT
                             ;   HAS BEEN CONVERTED SUCCESSFULLY)
         JMP     ERR         ; IF NOT EQUAL, ILLEGAL CHARACTER IN INPUT
                             ;   STRING CAUSED CONVERSION TO TERMINATE; HENCE
                             ;      INPUT IS ERRONEOUS; GO TO ERROR ROUTINE
10$:                         ; PROGRAM CONTINUES
```

$COTB—Octal To Binary Conversion Routine

The $COTB routine converts an unsigned ASCII octal number to binary format.

FORMAT

CALL $COTB

INPUT

input buffer address
In Register 0: the address of the first byte of the ASCII octal character string to be converted

OUTPUT

next byte address
In Register 0: the address of the next byte of the input buffer

binary number
In Register 1: the converted number

terminator
In Register 2: the terminating character of the input buffer

DESCRIPTION

The characters 0 to 7 are valid in the input octal number. The maximum value of an octal number that can be converted by the $COTB routine is 177777. The end of a string must be marked by a terminating character, which can be any ASCII character except the numbers 0 to 7. Examples of terminating characters are a blank, tab character, alphabetic character, and special symbol. Leading blanks and tab characters are ignored.

$COTB calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task.

NOTE: You can determine, in the task, whether an input string was successfully converted by testing the contents of Register 2. If the contents are other than the expected terminating character, the conversion was incomplete because the routine found an invalid character in the input string.

EXAMPLE

The following source statements define a processing loop, using the $COTB routine, to convert a series of ASCII octal character strings to binary numbers. The example uses the tab character as the terminating character of each string and the space character as the terminating character of the input buffer. If converted successfully, the binary numbers will be stored in the buffer BNUM:

```
IBUF:   .ASCII /012/<11>/3456/<11>/76/<11>/54321/<40>
        .EVEN
BNUM:   .BLKW  4             ; BUFFER FOR CONVERTED STRINGS
   .
   .
   .
         MOV    #BNUM,R4     ; GET OUTPUT BUFFER ADDRESS
         MOV    #IBUF,R0     ; SET UP INPUT BUFFER ADDRESS
LOOP:    CALL   $COTB        ; CONVERT THE STRING
         MOV    R1,(R4)+     ; SAVE CONVERTED STRING
         CMP    #11,R2       ; COMPARE ASCII TAB (HT) VALUE TO TERMINATING
                             ;    CHARACTER RETURNED IN REGISTER 2
         BEQ    LOOP         ; IF EQUAL, STRING SUCCESSFULLY CONVERTED,
                             ;    GOES BACK THROUGH LOOP TO CONVERT NEXT INPUT
                             ;       STRING POINTED TO BY REGISTER 0
         CMP    #40,R2       ; COMPARES SPACE VALUE (40) WITH TERMINATING
                             ;    CHARACTER IN REGISTER 2
         BEQ    10$          ; IF EQUAL, CONTINUES PROGRAM (ALL INPUT
                             ;    HAS BEEN CONVERTED SUCCESSFULLY)
         JMP    ERR          ; IF NOT EQUAL, ILLEGAL CHARACTER IN INPUT
                             ;    STRING CAUSED CONVERSION TO TERMINATE; HENCE
                             ;       INPUT IS ERRONEOUS; GOES TO ERROR ROUTINE
10$:                         ; PROGRAM CONTINUES
```

$CAT5—ASCII to Radix-50 Conversion Routine

The $CAT5 routine converts up to three ASCII characters to a 16-bit Radix-50 value.

FORMAT

CALL $CAT5

INPUT

input buffer address

In Register 0: the address of the first character in the ASCII string you want to convert

period disposition flag

In Register 1, one of the following values:

R1	=	0 if the period is a terminating character
R1	=	1 to specify that the period is a valid character to be converted to Radix-50

OUTPUT

next input character

In Register 0: the address of the next character of the input string

Radix-50 value

In Register 1: the converted Radix-50 value

terminator

In Register 2: the terminating character or the invalid character that caused termination

Condition Code

C bit	=	Clear if conversion was complete
C bit	=	Set if conversion was incomplete

DESCRIPTION

The following characters are valid in the ASCII string to be converted:

- The alphabetic characters A to Z
- The numeric characters 0 to 9
- The dollar sign ($) and period (.)

For complete conversion, the string must contain three valid characters. If the string contains fewer than three valid characters, the $CAT5 routine will convert them but will set the C bit to indicate an incomplete conversion. Invalid characters cause the $CAT5 routine to terminate conversion. In this case, the output will be the valid character or characters and trailing blank or blanks, in binary format.

A blank character (space) in the ASCII character string causes the $CAT5 routine to terminate. If you include blanks as valid characters in the string, call the $CAT5B routine to do the conversion.

Since the address of the next character in the input string is returned in Register 0, you can convert successive strings by resetting Register 1 and setting up a processing loop back to the CALL $CAT5 statement.

The $CAT5 routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task.

NOTE: You can determine, in the task, whether conversion was complete by testing the C bit in the Condition Code or the contents of Register 2.

EXAMPLE

The following source statements define a subroutine that calls the $CAT5 routine to convert ASCII input data to Radix-50 format:

```
ASDAT:  .ASCII /ABC.DEF.HIJ./  ; STRINGS TO BE CONVERTED
        .EVEN
RAD5:   .BLKW 3.               ; OUTPUT BUFFER
        .EVEN
        MOV    #RAD5,R4        ; GET OUTPUT ADDRESS
        MOV    #3,R5           ; SET LIMIT TO LOOP
        MOV    #ASDAT,R0       ; SET UP THE ADDRESS OF THE FIRST ASCII CHARACTER
1$:     CLR    R1              ; SPECIFY THAT PERIOD IS CONVERSION TERMINATOR
        CALL   $CAT5           ; CONVERT ASCII RADIX-50
        BCC    2$              ; BRANCH IF C BIT IS CLEAR (CONVERSION COMPLETE)
        JMP    INER            ; JUMP TO INPUT ERROR ROUTINE IF
                               ;    C BIT IS SET (CONVERSION INCOMPLETE)
2$:     MOV    R1,(R4)+        ; STORE CONVERTED CHARACTER
        DEC    R5
        BGT    1$              ; PROCESS NEXT STRING
   .
   .
   .
```

$CAT5B—ASCII with Blanks to Radix-50 Conversion Routine

The $CAT5B routine converts an ASCII 3-character string, including blank characters, to a 16-bit Radix-50 value.

FORMAT

CALL $CAT5B

INPUT

input buffer address
In Register 0: the address of the first character in the ASCII string you want to convert

period disposition flag
In Register 1, one of the following values:

R1	=	0 if the period is a terminating character
R1	=	1 to specify that the period is a valid character to be converted to Radix-50

OUTPUT

next input character
In Register 0: the next character of the input string

Radix-50 value
In Register 1: the converted Radix-50 value, one to three characters in length

terminator
In Register 2: the terminating character or the invalid character that caused termination

Condition Code

C bit	=	Clear if conversion was complete
C bit	=	Set if conversion was incomplete

DESCRIPTION

The following characters are valid in the ASCII string to be converted:

- The alphabetic characters A to Z
- The numeric characters 0 to 9

- The dollar sign ($), period (.), and blank (space)

For complete conversion, the string must contain three valid characters. If the string contains fewer than three valid characters, the $CAT5B routine will convert them but will set the C bit to indicate an incomplete conversion. Invalid characters cause the $CAT5B routine to terminate conversion. In this case, the output will be the valid character or characters and trailing blank or blanks, in binary format.

Since the address of the next character in the input string is returned in Register 0, you can convert successive strings by resetting Register 1 and setting up a processing loop back to the CALL $CAT5B statement.

$CAT5B calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task.

NOTE: You can determine, in the task, whether conversion was complete by testing the C bit in the Condition Code or the contents of Register 2.

EXAMPLE

The following source statements call the $CAT5B routine to convert a 3-character ASCII string to Radix-50 format:

```
INSTR:  .ASCII  /IND/         ; ASCII INPUT STRING
        .BYTE   15            ; STRING TERMINATOR
        .EVEN
        MOV     INSTR,R0      ; POINT TO THE ASCII INPUT STRING
        MOV     #1,R1         ; SPECIFY PERIOD IS VALID CHARACTER
        CALL    $CAT5B        ; CONVERT IT TO RADIX-50
        BCC     10$           ; WERE CHARACTERS CONVERTED?
        CMPB    #15,R2        ; NO -- WAS TERMINATOR A <CR> ?
        BEQ     10$           ; EQ -- YES
        CALL    SERR          ; NO, CALL SYNTAX ERROR ROUTINE
10$:                          ; PROGRAM CONTINUES
```

5 Output Data Conversion Routines

The output data conversion routines convert internally stored numeric data to ASCII characters. The following four groups of routines convert output data:

- Binary to decimal conversion routines, which convert binary data to one of the following formats:
  - 2-digit day date, in the range 01 to 31
  - 5-digit unsigned decimal magnitude number
  - 5-digit signed decimal number
  - Decimal number up to nine digits in length
- Binary to octal conversion routines, which convert binary numbers to one of the following octal numbers:
  - 6-digit unsigned octal magnitude number
  - 6-digit signed octal number
  - 3-digit octal number
- A general-purpose binary conversion routine that converts binary data to ASCII format. Note that the preceding conversion routines format their output according to internally-defined conversion parameters. The $CBTA routine allows you to determine the format of the output by specifying the conversion parameters. You can call this routine directly, or you can call it indirectly when you use the binary to decimal or octal routines. These routines pass predefined conversion parameters to the $CBTA routine.
- A Radix-50 to ASCII conversion routine, which converts a Radix-50 value to a 3-character ASCII string

The output data routines described in this chapter are called by the Edit Message Routine ($EDMSG; described in Chapter 6) to convert data to be formatted for output to printers or display devices.

The following four system library routines convert internally formatted binary numbers to external ASCII decimal format:

- Binary Date Conversion Routine ($CBDAT), which converts an internally stored binary date to a 2-digit decimal number
- Convert Binary to Decimal Magnitude Routine ($CBDMG), which converts an internally stored binary number to a 5-digit unsigned ASCII decimal magnitude value
- Convert Binary to Signed Decimal Routine ($CBDSG), which converts an internally stored binary number to a 5-digit signed ASCII decimal number
- Convert Double-Precision Binary to Decimal Routine ($CDDMG), which converts a double-precision, unsigned binary number to an ASCII decimal number of nine or fewer digits

These routines use predefined conversion parameters that are passed to the general-purpose conversion routine ($CBTA), which performs the actual binary to ASCII conversion.

Note that these routines do not add an extra space for the minus sign (–) to the predefined field-width parameter. If you are converting a negative number, expect that one of the spaces in the output area will be used for the minus sign.

$CBDAT—Binary date conversion routine

The $CBDAT routine converts an internally stored binary date to a 2-digit unsigned decimal number.

FORMAT

CALL $CBDAT

The $CBDAT routine uses the following predefined conversion parameters:

Radix	=	10
Field width	=	2 characters
Sign flag	=	UNSIGNED

INPUT

output address
In Register 0: the starting address of the output area that will store the converted 2-byte date

input date
In Register 1: the date (a binary value in the range 01 to 31)

zero suppression indicator
In Register 2, one of the following values:

R2	=	0 to specify suppression of leading zeros in the converted date (the date will be left-justified)
R2	=	Nonzero to specify no suppression of leading zeros

OUTPUT

converted date
In the specified output area: the converted day date (in ASCII decimal format)

next output address
In Register 0: the next available address (the pointer to the location following the last digit stored)

DESCRIPTION

The $CBDAT routine pushes the predefined conversion parameters on the stack. It then passes the conversion parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBDAT routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task, and destroys the contents of Registers 1 and 2. The $CBDAT routine does not return any error conditions to the caller.

EXAMPLE

The following source statements call the $CBDAT routine to convert a binary date in the buffer BDAT and store the converted date in the buffer ASDAT:

```
ASDAT: .BLKB  2            ; OUTPUT BUFFER
       .EVEN
BDAT:  .WORD  1            ; INPUT -- BINARY DATE
       MOV    #ASDAT,R0    ; PUTS THE ADDRESS OF OUTPUT AREA IN REGISTER 0
       MOV    BDAT,R1      ; PUTS THE BINARY DATE, AT BDAT, IN REGISTER 1
       CLR    R2           ; CLEARS REGISTER 2 TO ZERO TO SPECIFY THAT LEADING
                           ;    ZEROS ARE TO BE SUPPRESSED
       CALL   $CBDAT       ; CALLS THE $CBDAT ROUTINE
```

$CBDMG—Convert binary to decimal magnitude routine

The $CBDMG routine converts an internally stored binary number to a 5-digit unsigned ASCII decimal magnitude number.

FORMAT

CALL $CBDMG

The $CBDMG routine uses the following predefined conversion parameters:

Radix	=	10
Field width	=	5 characters
Sign flag	=	UNSIGNED

INPUT

output address

In Register 0: the starting address of the output area that will contain the converted 5-digit number

input number

In Register 1: the unsigned binary number you want to convert

zero suppression indicator

In Register 2, one of the following values:

R2	=	0 to specify suppression of leading zeros in the converted number (the number will be left-justified)
R2	=	Nonzero to specify no suppression of leading zeros

OUTPUT

result

In the specified output area: the converted number, a maximum of five digits in length

next output address

In Register 0: the next available address in the output area (the pointer to the location following the last digit stored)

DESCRIPTION

The $CBDMG routine pushes the predefined conversion parameters on the stack. It then passes the conversion parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBDMG routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task. It destroys the contents of Registers 1 and 2. The $CBDMG routine does not return error conditions to the caller.

EXAMPLE

The following source statements call the $CBDMG routine to convert a binary number stored in the buffer $IEXT and store the converted 5-digit ASCII decimal magnitude number in the buffer .TEXT:

```
.TEXT:  .BLKB   5               ; OUTPUT BUFFER
        .EVEN
$IEXT:  .WORD   2765.           ; INPUT VALUE
        MOV     #.TEXT,R0       ; GET OUTPUT BUFFER
        MOV     $IEXT,R1        ; GET BINARY VALUE
        CLR     R2              ; SUPPRESS ZEROS
        CALL    $CBDMG          ; CONVERT TO ASCII (DECIMAL)
```

$CBDSG—Convert binary to signed decimal routine

The $CBDSG routine converts an internally stored binary number to a 5-digit signed ASCII decimal number.

FORMAT
CALL $CBDSG

The $CBDSG routine uses the following predefined conversion parameters:

Radix	=	10
Field width	=	5 characters
Sign flag	=	SIGNED

INPUT

output address
In Register 0: the starting address of the output area that will store the converted 5-digit number

input number
In Register 1: the binary number to be converted

zero suppression indicator
In Register 2, one of the following values:

R2	=	0 to suppress leading zeros in the converted number (the output number will be left-justified)
R2	=	Nonzero to specify no suppression of leading zeros

OUTPUT

result
In the specified output area: the converted number, a maximum of five digits in length

next output address
In Register 0: the next available address in the output area (the pointer to the location following the last digit stored)

DESCRIPTION

The $CBDSG routine automatically pushes the predefined conversion parameters on the stack. It then passes the conversion parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBDSG routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task, and does not save the contents of Registers 1 or 2. The $CBDSG routine does not return error conditions to the caller.

EXAMPLE

The following source statements call the $CBDSG routine to convert a binary value stored in the buffer F.ERR and store the converted 5-digit ASCII decimal number in the buffer ER2NUM:

```
ER2:     .ASCII  $I/O ERROR CODE:$ ; ERROR MESSAGE
ER2NUM:  .BLKB   5                 ; OUTPUT BUFFER
         .EVEN
FILERR:  MOVB    F.ERR(R0),R1      ; GET ERROR CODE TO CONVERT
         MOV     #ER2NUM,R0        ; POINT TO OUTPUT BUFFER
         CLR     R2                ; SUPPRESS LEADING ZEROS
         CALL    $CBDSG            ; CONVERT ERROR CODE
         MOVB    #'.,(R0)+         ; PUT IN DECIMAL POINT
```

$CDDMG—Convert double-precision binary to decimal routine

The $CDDMG routine converts a double-precision, unsigned binary number to an unsigned ASCII decimal number, up to nine digits, less than or equal to 65,536x10^4. If the number contains more than nine digits, the routine inserts a string of five ASCII asterisk symbols in the output area.

FORMAT
CALL $CDDMG

INPUT

output address
In Register 0: the starting address of the output area

input address
In Register 1: the address of the 2-word input area containing the double-precision number

zero suppression indicator
In Register 2, one of the following values:

R2	=	0 to specify suppression of leading zeros in the converted date (the date will be left-justified)
R2	=	Nonzero to specify no suppression of leading zeros

NOTE: If the five most significant digits are zeros, they will be suppressed automatically, regardless of the setting of the suppression indicator.

OUTPUT

result
In the output area: the converted ASCII number

next output address
In Register 0: the pointer to the next available address in the output storage area

NOTE: If the number was converted successfully, the output area will contain from four to nine digits. If the conversion attempt results in a decimal number greater than 65,536x10^4 or longer than nine digits, the $CDDMG routine prints a string of five ASCII asterisks in the output area.

DESCRIPTION

The $CDDMG routine performs the following actions:

- Calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task

- Calls the $DDIV routine to perform the double-precision division
- Calls the $CBTA routine to perform the actual ASCII conversion
- Destroys the contents of Registers 1 and 2

EXAMPLE

The following source statements call the $CDDMG routine to convert a double-precision number, pointed to by the buffer DPWRD, and store the converted ASCII decimal number in the buffer ASDN:

```
ASDN:  .BLKB  9.          ; OUTPUT BUFFER
       .EVEN
DPWRD: .BLKW  2           ; INPUT BUFFER
       MOV    #ASDN,R0    ; PUTS ADDRESS OF OUTPUT AREA IN REGISTER 0
       MOV    #DPWRD,R1   ; PUTS STARTING ADDRESS OF DOUBLE-
                          ;    PRECISION INPUT WORD IN REGISTER 1
       MOV    #4.,R2      ; PUTS NONZERO IN REGISTER 2 (SETS THE ZERO
                          ;    INDICATOR FLAG TO 1) TO SPECIFY
                          ;       THAT LEADING ZEROS ARE NOT TO
                          ;          BE SUPPRESSED
       CALL   $CDDMG      ; CALLS THE $CDDMG ROUTINE
       CMPB   #' *,ASDN   ; COMPARES AN ASCII ASTERISK SYMBOL WITH
                          ;    A BYTE OF THE CONVERTED NUMBER
       BNE    10$         ; IF NOT EQUAL, CONVERSION WAS SUCCESSFUL
                          ;    AND PROGRAM CONTINUES
       JMP    ERR         ; IF EQUAL, JUMP TO ERROR ROUTINE ERR (MORE
                          ;    THAN NINE DIGITS WERE CONVERTED AND THE
                          ;       OUTPUT DATA IS INVALID)
10$:
```

NOTE: The source statements also check the results and call an error routine if $CDDMG was not successful.

The following three routines convert internally formatted binary numbers to external ASCII octal format:

- Convert Binary to Octal Magnitude Routine ($CBOMG), which converts an internally stored binary number to a 6-digit unsigned ASCII octal magnitude number
- Convert Binary to Signed Octal Routine ($CBOSG), which converts an internally stored binary number to a 6-digit signed ASCII octal number
- Convert Binary Byte to Octal Magnitude Routine ($CBTMG), which converts an internally stored binary byte to a 3-digit unsigned ASCII octal number

These routines pass predefined conversion parameters to the general-purpose conversion routine ($CBTA), which performs the actual binary to ASCII conversion.

Note that these routines do not add an extra space for the minus sign (–) to the predefined field-width parameter. If you are converting a negative number, expect that one of the spaces in the output area will be used for the minus sign.

$CBOMG—Convert binary to octal magnitude routine

The $CBOMG routine converts an internally stored binary number to a 6-digit unsigned ASCII octal magnitude number.

FORMAT

CALL $CBOMG

The $CBOMG routine uses the following predefined conversion parameters:

Radix	=	8
Field width	=	6 characters
Sign flag	=	UNSIGNED

INPUT

output address

In Register 0: the starting address of the output area in which the converted 6-digit number is to be stored

input number

In Register 1: the binary number you want to convert

zero suppression indicator

In Register 2, one of the following values:

R2	=	0 to specify suppression of leading zeros in the converted number (the number will be left-justified)
R2	=	Nonzero to specify no suppression of leading zeros

OUTPUT

result

In the specified output area: the converted number, a maximum of six digits in length

next output address

In Register 0: the next available address in the output area (the pointer to the location following the last digit stored)

The $CBOMG routine does not return any error conditions to the caller.

DESCRIPTION

The $CBOMG routine pushes the predefined conversion parameters on the stack. It then passes the conversion parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBOMG routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task, and destroys the contents of Registers 1 and 2.

EXAMPLE

The following source statements call the $CBOMG routine to convert a binary number stored in the buffer BNUM and store the converted 6-digit ASCII octal magnitude number in the buffer OCOUT:

```
OCOUT: .BLKB 6          ; OUTPUT BUFFER
       .EVEN
BNUM:  .WORD 162710     ; INPUT VALUE
       MOV   #OCOUT,R0  ; PUTS THE STARTING ADDRESS OF THE OUTPUT AREA IN REGIS'
       MOV   BNUM,R1    ; PUTS THE BINARY NUMBER TO BE CONVERTED IN REGISTER 1
       MOV   #1,R2      ; PUTS THE VALUE 1 IN REGISTER 2 (SETS THE ZERO
                        ;   INDICATOR FLAG TO 1) TO SPECIFY THAT
                        ;      LEADING ZEROS ARE NOT TO BE SUPPRESSED
       CALL  $CBOMG     ; CALLS THE $CBOMG ROUTINE
```

$CBOSG—Convert binary to signed octal routine

The $CBOSG routine converts an internally stored binary number to a 6-digit signed ASCII octal number.

FORMAT

CALL $CBOSG

The $CBOSG routine uses the following predefined conversion parameters:

Radix	=	8
Field width	=	6 characters
Sign flag	=	SIGNED

INPUT

output address

In Register 0: the starting address of the output area in which the converted 6-digit number will be stored

input number

In Register 1: the binary number to be converted

zero suppression indicator

In Register 2, one of the following values:

R2	=	0 to specify suppression of leading zeros in the converted number (the output number will be left-justified)
R2	=	Nonzero to specify no suppression of leading zeros

OUTPUT

result

In the specified output area: the converted signed number, a maximum of six digits in length

next output address

In Register 0: the next available address in the output area (the pointer to the location following the last digit stored)

The $CBOSG routine does not return error conditions to the caller.

DESCRIPTION

The $CBOSG routine pushes the predefined conversion parameters on the stack. It then passes the conversion parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBOSG routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task, and destroys the contents of Registers 1 and 2.

$CBTMG—Convert binary byte to octal magnitude routine

The $CBTMG routine converts an internally stored binary byte to a 3-digit ASCII unsigned octal number.

FORMAT

CALL $CBTMG

The $CBTMG routine uses the following predefined conversion parameters:

Radix	=	8
Field width	=	3 characters
Sign flag	=	UNSIGNED

INPUT

output address

In Register 0: the starting address of the output area in which the converted 3-digit number will be stored

input binary byte

In Register 1 (low-order byte): the binary byte to be converted

zero suppression indicator

In Register 2, one of the following values:

R2	=	0 to specify suppression of leading zeros in the converted number (the number will be left-justified)
R2	=	Nonzero to specify no suppression of leading zeros

OUTPUT

result

In the specified output area: the converted number, a maximum of three digits in length

next output address

In Register 0: the next available address in the output area (the pointer to the location following the last digit stored

The $CBTMG routine does not return error conditions to the caller.

DESCRIPTION

The $CBTMG routine pushes the predefined conversion parameters on the stack. It then passes the conversion parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary byte.

The $CBTMG routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task, and destroys the contents of Register 2. In addition, $CBTMG clears the high-order byte of Register 1 (the low-order byte is unchanged).

EXAMPLE

The following source statements call the $CBTMG routine to convert a binary number stored in the buffer TBUF and store the converted 3-digit ASCII octal number in the buffer BOUT:

```
BOUT:   .BLKB   3               ; OUTPUT BUFFER
        .EVEN
TBUF:   .BYTE   177             ; INPUT BUFFER
        .EVEN
        MOV     #BOUT,R0        ; ADDRESS OUTPUT BUFFER
        MOVB    TBUF,R1         ; GET BINARY CODE
        MOVB    #1,R2           ; SPECIFY NO ZERO SUPPRESSION
        CALL    $CBTMG          ; CONVERT THE BINARY NUMBER TO OCTAL
```

$CBTA—General purpose binary to ASCII conversion routine

The $CBTA routine converts internally stored binary numbers to ASCII decimal or octal numbers when called by the binary-to-decimal and binary-to-octal conversion routines.

[illegible]$CBTA

INPUT

output address

In Register 0: the starting address of the output area in which the converted ASCII number will be stored

input value

In Register 1: the binary value to be converted

conversion parameters

In Register 2, the following options:

Bits 0 – 7	(Low byte.) Must contain the conversion radix (2 to 16 decimal).
Bit 8	Must contain the unsigned flag (= 0) if unsigned value to be converted; or must contain the sign flag (= 1) if signed value to be converted. (The minus sign is not counted in the output field width when you convert a negative signed number. The $CBTA routine will use a space in the output buffer for the minus sign.)
Bit 9	Zero suppression flag = 0; or nonzero suppression flag = 1.
Bit 10	Blank fill flag = 1 to specify replacement of leading zeros with blanks (only if nonzero suppression flag = 1). Blank fill flag = 0 to specify no replacement of leading zeros (if bit 9 = 1). (When the zero suppression flag = 0, the blank fill flag is ignored.)
Bits 11 – 15	Must contain a numeric value from 1 to 32 specifying the field width. If you convert a negative signed number, remember to add a space in the field width for the minus sign.

OUTPUT

result

In the specified output area: the converted number, from 1 to 32 digits in length

next output address

In Register 0: the next available address in the output area (the pointer to the location following the last digit stored)

The $CBTA routine does not return any error conditions to the caller.

DESCRIPTION

The $CBTA routine converts internally stored values according to the user-defined conversion parameters, which the calling routine passes as an input argument in Register 2.

Note that the $CBTA routine does not add an extra space for the minus sign (–) to the predefined field-width parameter. If you are converting a negative number, expect that one of the characters in the output area will be used for the minus sign.

The $CBTA routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the caller, and calls the $DIV routine to perform the required division. The $CBTA routine also destroys the contents of Registers 1 and 2.

EXAMPLE

The following source statements set the conversion parameters, expressed in the number 15012_8, which will determine the format of the output by $CBTA. The statements call the $CBTA routine to convert a binary value in Register 3 and store the ASCII result in buffer CASTR:

```
CASTR:  .BLKB   32.             ; OUTPUT BUFFER
        .EVEN
        MOV     R0,-(SP)        ; SAVE REGS FOR CONVERT CALL
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        MOV     #CASTR,R0       ; ADDRESS TO CONVERT INTO
        MOV     R3,R1           ; VALUE TO CONVERT
        MOV     #15012,R2       ; 3-DIGIT, NO ZERO SUPPRESSION
        CALL    $CBTA           ; CONVERT BINARY TO ASCII
```

In this example, the binary expression of the value in Register 2 (0001101000001010) specifies that the output will have the the following conversion parameters:

Conversion radix	=	10_{10}
Sign flag	=	0 (unsigned value)
NOSUP flag	=	1 (no zero suppression)
Blank fill flag	=	0 (no replacement of leading zeros with blanks)
Field width	=	3

$C5TA—Radix-50 to ASCII conversion routine

The $C5TA routine converts an internally stored 16-bit Radix-50 value to an ASCII character string.

FORMAT

CALL $C5TA

INPUT

output address

In Register 0: the address that will point to the first byte of the converted string

Radix-50 word

In Register 1: the Radix-50 value you want to convert

OUTPUT

next output address

In Register 0: the address of the next byte after the last character stored in the output area

result

In the specified output area: the converted ASCII 3-character string, stored in a maximum of three consecutive bytes

The $C5TA routine does not return error conditions to the caller. It destroys the contents of Registers 1 and 2 and does not use Registers 3 through 5.

EXAMPLE

The following source statements call the $C5TA routine to convert a Radix-50 number stored in the buffer CRNTS and store the ASCII string result in the buffer SCRPTR:

```
CRNTS:  .RAD50  /GEN/           ; RADIX VALUE
SCRPTR: .BLKB   3               ; OUTPUT BUFFER
        .EVEN

        MOV     #SCRPTR,R0      ; SET OUTPUT BUFFER ADDRESS
        MOV     CRNTS,R1        ; GET RADIX VALUE
        CALL    $C5TA           ; CONVERT IT
```

6 Output Formatting Routines

The output formatting routines convert internally stored data to external ASCII characters and format the converted characters to produce readable output. The five output formatting routines are as follows:

- The Uppercase Text Conversion Routine ($CVTUC), which converts lowercase ASCII text to uppercase
- The Date String Conversion Routine ($DAT), which converts a 3-word binary date to a 9-character ASCII output string
- The Alternate Date String Conversion Routine ($DAT), which converts a date to a user-defined ASCII format up to 25 characters long
- The Time Conversion Routine ($TIM), which converts the binary time to an ASCII output string
- The Edit Message Routine ($EDMSG), which converts internally stored data to the user-specified type of ASCII data (alphanumeric, octal, decimal) and formats the converted data to produce meaningful output for printing or display

$CVTUC—Uppercase text conversion routine

The $CVTUC routine converts lowercase ASCII text to uppercase. The routine performs a byte-by-byte transfer of the input ASCII character string, converting all lowercase alphabetic characters to uppercase, and transferring all uppercase characters unchanged to the output string.

FORMAT

CALL $CVTUC

INPUT

input address
In Register 0: the address of the text string to be converted

output address
In Register 1: the address of the output area for the uppercase string

number input bytes
In Register 2: the number of bytes in the string to be converted

NOTE: The number of bytes cannot be stated as 0. A statement of 0 will cause $CVTUC to fail.

OUTPUT

result
In the output area: the converted string

next input address
In Register 0: a pointer to the next available address in the input area

next output address
In Register 1: a pointer to the next available address in the output area

DESCRIPTION

The $CVTUC routine converts all ASCII alphabetic characters in the input string to uppercase. Any other characters are moved from the input area to the output area in their sequential positions. You can specify the input area address as the output area address (R0 = R1) when the $CVTUC routine is called. If you specify this at the outset, Register 0 and Register 1 will be left pointing to the character following the string. The $CVTUC routine converts lowercase alphabetic characters to uppercase where they occur in the input area. The original lowercase contents of the input area are destroyed.

$CVTUC destroys the contents of Register 2 and does not use Registers 3 through 5 of the calling task.

EXAMPLE

The following source statements call the $CVTUC routine to convert an ASCII string to uppercase:

```
MACNAM: .BLKW   3               ; WORK BUFFER
        MOV     #MACNAM,R0      ; POINT TO WORK BUFFER
        MOV     #6,R2           ; SAVE STRING COUNTER
        MOV     R0,R1           ; POINT TO OUTPUT ADDRESS
        CALL    $CVTUC          ; DO THE CONVERSION
    .
    .
    .
```

(In this example, the converted string will be stored in the buffer MACNAM because R0 = R1.)

$DAT—Date string conversion routine

The $DAT routine converts the 3-word internal binary date to the standard 8- or 9-character ASCII output format. $DAT formats the date for output as follows:

day-month-year

FORMAT

CALL $DAT

INPUT

output address

In Register 0: the address of the output area that will store the converted date

input address

In Register 1: the address of the 3-word input area that will store the binary date

date values

The input area must contain the following values:

Word 1	=	Last two digits of year
Word 2	=	A 2-digit number from 01 to 12 (month of year)
Word 3	=	A 2-digit number from 01 to 31 (day of month)

OUTPUT

date

In the output area: the 8- or 9-character date string in the following format:

dd-mmm-yy

dd	Day (one character for 1 to 9 and two characters for 10 to 31)
mmm	Month (first three letters)
yy	Year (last two digits)

next output address

In Register 0: the address of the next available location in the output area

next input address

In Register 1: the next address (input R1 + 6) of the input area

DESCRIPTION

The $DAT routine uses and might destroy the contents of Register 2. The calling task should save any critical value contained in Register 2 before calling the $DAT routine.

$DAT calls the $SAVRG routine to save and restore the contents of Registers 3 through 5 of the calling task.

EXAMPLE

The following source statements call the $DAT routine to convert the binary date stored in buffer DATBUF and store the formatted ASCII output in the buffer EDTBUF:

```
DATBUF: .WORD   87.             ; YEAR
        .WORD   11.             ; MONTH
        .WORD   01.             ; DAY
EDTBUF: .BLKB   9.              ; OUTPUT BUFFER
        .EVEN
START:
        MOV     #EDTBUF,R0      ; OUTPUT FROM CONVERSION
        MOV     #DATBUF,R1      ; GET INPUT BUFFER
        CALL    $DAT            ; CONVERT DATE TO STANDARD ASCII FORMAT
```

After execution, the output buffer contains the following information:

```
        1-NOV-87
```

$DAT—Alternate date string conversion routine

The Alternate Date Routine ($DAT), accessed by the SYSLIB module INTDAS, converts the binary date in a format not dependent upon the DIGITAL-standard date format (dd-mmm-yy). The calling sequence is the same as for the standard format $DAT routine.

FORMAT

CALL $DAT

INPUT

output address
In Register 0: the address of the output area that will store the converted date

input address
In Register 1: the address of the 3-word input area that will store the binary date

date values
In the input area, the following definitions:

Word 1	=	Last two digits of year
Word 2	=	A 2-digit number from 01 to 12 (month of year)
Word 3	=	A 2-digit number from 01 to 31 (day of month)

OUTPUT

date
In the output buffer: the converted and formatted string (up to 25 characters), determined by your definitions of the logical names SYS$DATE_FORMAT and SYS$MONTH_nn

next output address
In Register 0: the address of the next available location in the output area

next input address
In Register 1: the next address (input R1 + 6) of the input area

DESCRIPTION

The alternate $DAT routine is contained in the module INTDAS, which has been inserted into SYSLIB with entry points deleted. To include the INTDAS module in your task image, you must explicitly request it in one of the following ways:

- Before building the task, invoke the Librarian Utility (LBR) and enter the following command line to include the module INTDAS in the task:

```
LB:[1,1]SYSLIB/LB:INTDAS
```

- Insert the module EDDAT without entry points, and INTDAS with entry points, into SYSLIB by entering the following command sequence:

```
> LBR
LBR> EDDAT=LB:[1,1]SYSLIB.OLB/EX:EDDAT
LBR> INTDAS=LB:[1,1]SYSLIB.OLB/EX:INTDAS
LBR> LB:[1,1]SYSLIB.OLB/RP/-EP=EDDAT
LBR> LB:[1,1]SYSLIB.OLB/RP=INTDAS
LBR> CTRL/Z
> PIP INTDAS.OBJ;*/DE,EDDAT.OBJ;*
```

The alternate $DAT routine's calling sequence remains the same as for the standard $DAT routine, but the logical name SYS$DATE_FORMAT contains the following character formats:

Argument	Effect
DD	Print 2-digit day of month with leading zero
ZD	Print 2-digit day of month with leading zero suppressed
MM	Print 2-digit month number with leading zero
ZM	Print 2-digit month number with leading zero suppressed
YY	Print 2-digit year with leading zero
ZY	Print 2-digit year with leading zero suppressed
MMM	Print alphabetic month (not necessarily three characters long)

You can use additional characters (other than the uppercase letters D, Z, M, and Y) in SYS$DATE_FORMAT as delimiters. If SYS$DATE_FORMAT is not defined, you get the DIGITAL-standard date format (dd-mmm-yy) by default. SYS$DATE_FORMAT can have a maximum length of 16 characters.

The logical SYS$MONTH_nn (where nn is 01 to 12) provides the alphabetic month to be printed when the mmm attribute is used. If SYS$MONTH_nn is not defined, you get the DIGITAL-standard 3-letter month abbreviations (mmm) by default. SYS$MONTH_nn can have a maximum length of 12 characters.

Logical translation is done in standard order. A local terminal assignment can override a system-wide assignment, which permits the same program to produce output in the individual user's own language or preferred format.

There are two limitations to the alternate date routine. First, using it necessitates more output buffer space than the traditional format because the output produced can be as long as 25 characters. The standard $DAT routine, however, produces eight or nine characters. Second, the new module can be linked with many, but not all, existing programs. An example of a program that cannot use this routine is one that performs operations on the resulting output string, expecting it to be in the format produced by the standard routine.

The INTDAS module contains the routines $DAT and $TIM. The $TIM routine has not been modified; it produces the standard time format, as described in Time Conversion Routine ($TIM).

The $DAT routine uses and might destroy the contents of Register 2. The calling task should save any critical value contained in Register 2 before calling the $DAT routine.

$DAT calls the $SAVRG routine to save and restore the contents of Registers 3 through 5 of the calling task.

EXAMPLE

Assume that you have replaced the SYSLIB module INTDAS into your library with entry points and are ready to run a program that calls the $DAT routine. Your definition, at the system prompt, of the logical names SYS$DATE_FORMAT and SYS$MONTH_nn will determine the output of the $DAT routine when it executes, as shown in the following examples:

```
DEFINE SYS$DATE_FORMAT = "MMM ZD, 19YY"
DEFINE SYS$MONTH_11 = "November"
       Output: November 1,1987
DEFINE SYS$DATE_FORMAT = "DD.MMM.YY"
DEFINE SYS$MONTH_11 = "XI"
       Output: 01.XI.87

SYS$DATE_FORMAT = "ZD/MM/YY"

       Output:  1/11/87
```

$TIM—Time conversion routine

The $TIM routine converts the binary time, in a standard format, to an ASCII output string in the form:

HH:MM:SS.S

The $TIM routine converts and formats the time for output in one of the following forms:

hour
hour:minute
hour:minute:second
hour:minute:second.fraction

FORMAT

CALL $TIM

INPUT

The standard format for $TIM input values is shown in the following table:

Word	Significance	Output Format	Value Range
WD1	Hour-of-Day	HH	0 to 23
WD2	Minute-of-Hour	MM	0 to 59
WD3	Second-of-Minute	SS	0 to 59
WD4	Tick-of-Second	.S	Depends on clock frequency
WD5	Ticks-per-Second	.S	Depends on clock frequency

output address
In Register 0: the address of the output area that will store the converted time

input address
In Register 1: the starting address of the input area that stores the time values

parameter count
In Register 2, the parameter count, where:

R2	=	0 or 1, to specify that the hour (word 1) is to be converted in the format HH
R2	=	2, to specify that the hour and minute (words 1 and 2) are to be converted in the format HH:MM
R2	=	3, to specify that the hour, minute, and second (words 1, 2, and 3) are to be converted in the format HH:MM:SS
R2	=	4 or 5, to specify that the hour, minute, second, and tick are to be converted in the format HH:MM:SS.S (where .S = tenth of second)

NOTE: For HH, the $TIM routine always returns two characters for all values specified.

OUTPUT

next output address
In Register 0: the address of the next available location in the output area

next input address
In Register 1: the address of the next word in the input area

time string
In the specified output area: the converted time string

DESCRIPTION

The $TIM routine calls the $SAVRG routine to preserve Registers 3 through 5 of the calling task. The contents of Registers 0 and 1 are updated and returned to the calling task. The $TIM routine destroys the contents of Register 2 (the parameter count). It also calls the following routines:

- The $DIV routine, which performs the division required to convert binary values to ASCII format
- The $CBDAT routine, which actually performs the time conversion, two digits at a time

The $TIM routine does not check the validity of the input data.

EXAMPLE

The following source statements call the $DAT and $TIM routines to convert time values to the standard formats:

Assume a program contains an input block, an output block, and source statements. For example:

```
BDBLK:  .WORD   87.            ; YEAR
        .WORD   11.            ; MONTH
        .WORD   01.            ; DAY
        .WORD   10.            ; HOUR
        .WORD   15.            ; MINUTES
        .WORD   35.            ; SECONDS
        .WORD   xx
        .WORD   x
DTBLK:  .BLKB   20.
        MOV     #DTBLK,R0      ; PUTS ADDRESS OF OUTPUT AREA IN REGISTER 0
        MOV     #BDBLK,R1      ; PUTS ADDRESS OF INPUT BINARY DATE AREA IN REGISTER
        CALL    $DAT           ; CALLS THE $DAT ROUTINE
        MOVB    #11,(R0)+      ; PUTS TAB AFTER DATE IN OUTPUT BUFFER
                               ;    REGISTER 0 NOW CONTAINS NEXT ADDRESS IN DTBLK FR
                               ;       REGISTER 1 NOW CONTAINS ADDRESS OF NEXT WORD
                               ;          HOUR 10) IN BDBLK FROM $DAT
        MOV     #3.,R2         ; SPECIFIES THE HH:MM:SS FORMAT FOR CONVERTED TIME
        CALL    $TIM           ; CALLS THE $TIM ROUTINE
```

After execution, the output buffer will contain the following information:

```
1-NOV-87        10:15:35
```

The time and date fields are left-justified.

$EDMSG—Edit message routine

The $EDMSG routine converts internally stored data to ASCII decimal, octal, or alphanumeric characters, and controls the layout of the converted characters. You can use the $EDMSG routine to produce printed or displayed text in meaningful, readable formats.

FORMAT

CALL $EDMSG

INPUT

output address
In Register 0: the starting address of the output block

input address
In Register 1: the address of the input string

argument block address
In Register 2: the starting address of the argument block

input string
The input string contains the editing directives and ASCII text that determine data conversion and format control for the $EDMSG routine. The directives must be in one of the following formats:

- %l
- %nl
- %Vl

The directives have the following effects:

Directive	Effect
%	A delimiter that identifies an editing directive to the $EDMSG routine.
n	An optional repeat count (decimal number) specifying the number of times the editing operation is to be repeated by the $EDMSG routine. If n = 0 or is not specified, a repeat count of 1 is assumed.
V	Specifies that the repeat count is a value in the next word in the task's argument block. If the value is 0, a repeat count of 1 is assumed.
l	An alphabetic letter specifying one of the editing operations to be performed by the $EDMSG routine, as shown in Table 6–1.

Input strings can contain ASCII text as well as editing directives. Any number of directives can appear in an input string. Input strings must be in ASCIZ format.

argument block (ARGBLK)
The argument block contains the binary data to be converted, the addresses of ASCII and extended ASCII characters, or the address of a double-precision value.

Prior to calling the $EDMSG routine, set up the appropriate argument block, as follows:

- For $EDMSG to move ASCII or extended ASCII characters to the output block, the argument block must contain the address of the ASCII characters.
- For $EDMSG to convert a binary byte to octal format, the argument block must contain the address of the binary byte.
- For $EDMSG to convert binary values, the argument block must contain the values.
- For $EDMSG to perform filename string conversion, the argument block must contain the following information:

Word 1	=	Radix-50 file name
Word 2	=	Radix-50 file name
Word 3	=	Radix-50 file name
Word 4	=	Radix-50 file type
Word 5	=	Binary version number

- For $EDMSG to convert a binary date, the argument block must contain the following information:

Word 1	=	Year (last two digits)
Word 2	=	Number (01 to 12) of month
Word 3	=	Day of month (01 to 31)

NOTE: $EDMSG does not check the validity of the date values. If you specify erroneous date values, output results will be unpredictable.

- For $EDMSG to convert binary time, the argument block must contain the following information:

Word 1	=	Hour-of-day (0 to 23)
Word 2	=	Minute-of-hour (0 to 59)
Word 3	=	Second-of-minute (0 to 59)
Word 4	=	Tick-of-second (depends on clock frequency)
Word 5	=	Ticks-per-second (depends on clock frequency)

output block (OUTBLK)
The output block in which $EDMSG is to store output

OUTPUT

converted data
In the output block: the converted/formatted data

next byte
In Register 0: the address of the next available byte in the output block (the $EDMSG routine clears this byte to provide a null-terminated (ASCIZ) string)

output length

In Register 1: the number of bytes transferred to the output block (the count does not include the null-terminating byte)

next argument address

In Register 2: the address of the next argument in the argument block

Table 6–1 describes the editing directives for the $EDMSG routine.

Table 6–1 $EDMSG Routine Editing Directives

Directive	Form	Operation
A (ASCII[1] string)	%A	Move the ASCII character from address in ARGBLK to OUTBLK.
	%nA	Move the next n ASCII characters from address in ARGBLK to OUTBLK.
	%VA	Use the value in the next word in ARGBLK as repeat count and move the specified number of ASCII characters from address in ARGBLK to OUTBLK.
B (binary byte to octal conversion)	%B	Convert the next binary byte from address in ARGBLK to unsigned octal number and store result in OUTBLK.
	%nB	Convert the next n binary bytes from address in ARGBLK to octal numbers and store results in OUTBLK; insert space between numbers.
	%VB	Use the value in the next word in ARGBLK as the repeat count, convert the specified number of binary bytes from address in ARGBLK to octal numbers, and store results in OUTBLK; insert space between numbers.
D (binary to signed decimal conversion, 0 suppress)	%D	Convert the binary value in the next word in ARGBLK to signed decimal and store result in OUTBLK.
	%nD	Convert the next n binary values in ARGBLK to signed decimal and store results in OUTBLK; insert tab between numbers.
	%VD	Use the value in the next word in ARGBLK as repeat count, convert the specified number of binary values to signed decimal, and store results in OUTBLK; insert tab between numbers.
E (extended ASCII[1])	%E	Move the extended ASCII character from the address in ARGBLK to the OUTBLK.
	%nE	Move n extended ASCII characters from the address in ARGBLK to OUTBLK.
	%VE	Use the value in the next word in ARGBLK as repeat count and move the specified number of ASCII characters from the address in ARGBLK to OUTBLK.
F (form feed)	%F	Insert a form-feed character in OUTBLK.
	%nF	Insert n form-feed characters in OUTBLK.
	%VF	Use the value in the next word in ARGBLK as repeat count and insert specified number of form-feed characters in OUTBLK.
I (include ASCIZ string)	%I	Use the next value in ARGBLK as the address of an ASCIZ string to be logically included in the format string at this point.

[1]Extended ASCII characters consist of the printable characters in the 7-bit ASCII code. If nonprintable characters appear in an ASCII input string, the E directive replaces them with a space, while the A directive transfers the nonprintable characters to the output block.

Table 6–1 (Cont.) $EDMSG Routine Editing Directives

Directive	Form	Operation
M (binary to decimal magnitude conversion, 0 suppress)	%M	Convert the binary value in the next word in ARGBLK to decimal magnitude with leading zeros suppressed and store the result in OUTBLK.
	%nM	Convert the next n binary values in ARGBLK to decimal magnitude with leading zeros suppressed and store the results in OUTBLK; insert tab between numbers.
	%VM	Use the value in the next word in ARGBLK as repeat count, convert the specified number of binary values to decimal magnitude with leading zeros suppressed, and store the results in OUTBLK; insert tab between numbers.
N (new line-carriage return/line feed)	%N	Insert CR and LF characters in OUTBLK.
	%nN	Insert n CR and LF characters in OUTBLK.
	%VN	Use the value in the next word in ARGBLK as repeat count and insert the specified number of CR and LF characters in OUTBLK.
O (binary to signed octal conversion)	%O	Convert the binary value in the next word in ARGBLK to signed octal and store the result in OUTBLK.
	%nO	Convert the next n binary values in ARGBLK to signed octal and store the results in OUTBLK; insert tab between numbers.
	%VO	Use the value in the next word in ARGBLK as repeat count, convert the specified number of binary values to signed octal, and store the results in OUTBLK; insert tab between numbers.
P (binary to unsigned octal magnitude conversion, no 0 suppress)	%P	Convert the binary value in the next word in ARGBLK to octal magnitude with no leading zeros suppressed and store the result in OUTBLK.
	%nP	Convert the next n binary values in ARGBLK to octal magnitude with no leading zeros suppressed and store the results in OUTBLK; insert tab between numbers.
	%VP	Use the value in the next word in ARGBLK as repeat count, convert the specified number of binary values to octal magnitude with no leading zeros suppressed, and store the results in OUTBLK; insert tab between numbers.
Q (binary to octal magnitude conversion, 0 suppress)	%Q	Convert the binary value in the next word in ARGBLK to octal magnitude with leading zeros suppressed and store the result in OUTBLK.
	%nQ	Convert the next n binary values in ARGBLK to octal magnitude with leading zeros suppressed and store the results in OUTBLK; insert tab between numbers.
	%VQ	Use the value in the next word in ARGBLK as repeat count, convert the specified number of binary values to octal magnitude with leading zeros suppressed, and store the results in OUTBLK; insert tab between numbers.
R (Radix-50 to ASCII)	%R	Convert the Radix-50 value in the next word in ARGBLK to ASCII and store the result in OUTBLK.
	%nR	Convert the next n Radix-50 values in ARGBLK to ASCII and store the results in OUTBLK.
	%VR	Use the value in the next word in ARGBLK as repeat count, convert the specified number of Radix-50 values to ASCII, and store the results in OUTBLK.
S (space)	%S	Insert a space in OUTBLK.
	%nS	Insert n spaces in OUTBLK.

Table 6–1 (Cont.) $EDMSG Routine Editing Directives

Directive	Form	Operation
	%VS	Use the value in the next word in ARGBLK as repeat count and insert the specified number of spaces in OUTBLK.
T (double-precision binary to decimal conversion)	%T	Convert the double-precision unsigned binary value at the address in ARGBLK to decimal and store result in OUTBLK.
	%nT	Convert the next n double-precision binary values starting at the address in ARGBLK to decimal and store results in OUTBLK; insert tab between numbers.
	%VT	Use the value in the next word in ARGBLK as repeat count, convert the specified number of double-precision binary values starting at the address in ARGBLK to decimal, and store the results in OUTBLK; insert tab between numbers.
U (binary to decimal magnitude conversion, no 0 suppress)	%U	Convert the binary value in ARGBLK to decimal magnitude with no leading zeros suppressed and store result in OUTBLK.
	%nU	Convert the next n binary values in ARGBLK to decimal magnitude with no leading zeros suppressed and store results in OUTBLK; insert tab between numbers.
	%VU	Use the value in the next word in ARGBLK as repeat count, convert the specified number of binary values to decimal magnitude with no leading zeros suppressed and store results in OUTBLK; insert tab between numbers.
X (filename string conversion)	%X	Convert Radix-50 filename string in ARGBLK to ASCII string in format name.typ; convert octal version number, if present, to ASCII and store results in OUTBLK.
	%nX	Convert next n Radix-50 filename strings in ARGBLK to ASCII strings in format name.typ; convert octal version numbers, if present, to ASCII and store results in OUTBLK; insert tab between strings.
	%VX	Use the value in the next word in ARGBLK as repeat count, convert specified number of Radix-50 filename strings to ASCII strings in format name.typ; convert octal version numbers, if present, to ASCII and store results in OUTBLK; insert tab between strings.
Y (date conversion)	%Y	Convert the next three binary words in ARGBLK to ASCII date in format dd-mmm-yy and store in OUTBLK. For this directive, a repeat is acceptable but will be ignored.
Z (binary time conversion)	%0Z or %1Z	Convert binary hour-of-day in the next word of ARGBLK to ASCII and store in OUTBLK in format HH.
	%2Z	Convert the binary hour-of-day and minute-of-hour in the next two words of ARGBLK to ASCII and store in OUTBLK in format HH:MM.
	%3Z	Convert the binary hour-of-day, minute-of-hour, and second-of-minute in the next three words of ARGBLK to ASCII and store in OUTBLK in format HH:MM:SS.
	%4Z or %5Z	Convert the binary hour-of-day, minute-of-hour, second-of-minute, and ticks-of-second or ticks-per-second in the next five words of ARGBLK to ASCII and store in OUTBLK in format HH:MM:SS.S, where .S = tenth of second.
< (define <stack>byte field)	%n<	Insert n ASCII spaces followed by a field mark (NUL) in OUTBLK to define a fixed-length byte field. The output pointer will point to the first space.
> (locate field mark)	%n>	Increment the OUTBLK pointer until a field mark (NUL) is located or the n repeat count is exceeded.

DESCRIPTION

The $EDMSG routine converts internally formatted data, in an argument block, to external format and stores it in the calling task's output block. The editing performed by the $EDMSG routine is specified by user directives within an input string. Any nonediting directive characters are simply copied into the output block. Output strings are in ASCIZ format.

The $EDMSG routine calls the output data conversion routines (also described in Chapter 5) to convert binary data to the specified external format. See the detailed descriptions of individual conversion routines for specific output formats.

The $EDMSG routine scans the input string, character-by-character. If it encounters nondirective (or "unknown" directive) characters, it transmits them directly to the task's output block. When the $EDMSG routine finds a percent sign (%) delimiter, it interprets the character(s) following the delimiter. If it encounters a data conversion directive, the $EDMSG routine accesses the argument block, converts the specified data, and transmits it to the output block. If a format control directive is encountered, the routine generates the specified control(s) and transmits the data to the output block. If the percent sign delimiter is not followed by a valid operator, or if multiple delimiters are found, the $EDMSG routine transmits the first delimiter (and any subsequent delimiters not followed by a valid directive character) to the output block.

NOTE: You can call an appropriate output routine to output the converted/formatted data.

$EDMSG calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task.

EXAMPLE

1 The following source statements call the $EDMSG routine to format the data stored in ARGBLK, as specified by the directives in buffer ISTRING:

```
ISTRING:    .ASCIZ /%F%12S***TEXT***%3N%8S%VD%2N%12S***END****/
            .EVEN
ARGBLK:     .WORD 3.          ; NUMBER OF VALUES TO CONVERT
            .WORD 99.         ; VALUES
            .WORD -37.        ;    TO
            .WORD 137.        ;      FORMAT
OUTBLK:     .BLKB 100.        ; OUTPUT BLOCK
START:      MOV #OUTBLK,R0    ; SET UP ADDRESS OF OUTPUT
            MOV #ISTRING,R1   ; SET UP ADDRESS OF INPUT
            MOV #ARGBLK,R2    ; SET UP ARGUMENT BLOCK
            CALL $EDMSG       ; DO THE FORMATTING
```

The editing directives shown in this example have the following effects:

Directive	Effect
%F	Insert a form feed in OUTBLK (start a new page).
%12S	Insert 12 spaces in OUTBLK and move the ASCII string to OUTBLK (indent the first line 12 spaces and insert the header ***TEXT***).
%3N	Insert three pairs of CR-LF characters in OUTBLK (generate two blank lines).
%8S	Insert eight spaces in OUTBLK (indent the next line eight spaces).

Directive	Effect
%VD	Use the first value (3) in ARGBLK as the repeat count and convert the next three binary values in ARGBLK to signed decimal; store each value, followed by a tab, in OUTBLK (output three signed decimal numbers set up in columns).
%2N	Insert two pairs of CR-LF characters in OUTBLK (generate one blank line).
%12S	Insert 12 spaces at the beginning of a line in OUTBLK and move the ASCII string to OUTBLK (indent 12 spaces and insert the text "**END****).

The example produces the following output:

```
            ***TEXT***
        99     -37     137
            ***END****
```

2 The following example calls the $EDMSG routine to convert the data stored in IBLK, as specified by the formatting directives in the buffer INSTR:

```
INSTR:      .ASCIZ  /%F%5S***F. TREVISANI WORK REPORT FROM %Y TO %Y***/
            .EVEN
IBLK:       .WORD 87.         ; YEAR
            .WORD 8.          ; 8TH MONTH (AUG)
            .WORD 22.         ; DAY
            .WORD 87.         ; YEAR
            .WORD 9.          ; 9TH MONTH (SEP)
            .WORD 16.         ; DAY
PRBLK:      .BLKB 100.        ; OUTPUT BLOCK

BEGIN:      MOV #PRBLK,R0     ; SET UP ADDRESS OF OUTPUT
            MOV #INSTR,R1     ; SET UP ADDRESS OF INPUT
            MOV #IBLK,R2      ; SET UP ARGUMENT BLOCK
            CALL $EDMSG       ; DO THE CONVERSION
```

The editing directives in the example have the following effects:

Directive	Effect
%F	Insert a form feed in PRBLK (start a new page).
%5S	Insert five spaces in PRBLK and move ASCII string to PRBLK (indent the line five spaces and output the header ***F. TREVISANI WORK REPORT FROM).
%Y	Convert the next three words in IBLK to formatted date and store in PRBLK followed by ASCII text (insert 22-AUG-89 TO in header line).
%Y	Convert next three words in IBLK to formatted date and store in PRBLK followed by ASCII text (insert 16-SEP-89*** in header line).

The above example produces the following output:

```
***F. TREVISANI WORK REPORT FROM 22-AUG-89 TO 16-SEP-89***
```

7 Dynamic Memory Management Routines

The dynamic memory management routines enable manual management of the space in a task's free dynamic memory. The free dynamic memory consists of all memory extending from the assembled code of the task to the highest virtual address owned by the task, excluding resident libraries.

Initially, these routines allocate free dynamic memory as one large block, from the highest available memory address downward. Subsequent memory block allocations are made within the available memory blocks. Available memory blocks are maintained as a linked list of blocks in ascending order, pointed to by a 2-word listhead. Each free memory block contains a 2-word control field, where:

- The first word contains the address of the next available block, or 0 if there is not another block
- The second word contains the size of the current block

Memory allocation is either on a first-fit or best-fit basis. Allocation is always made from the top of the selected available dynamic memory block. The second word of the block is adjusted to reflect the new size of the current block of available dynamic memory. As memory blocks are allocated completely, they are removed from the free memory list.

When memory blocks are deallocated (released), they are returned to the free memory list. The released memory blocks are relinked to the free memory list in ascending address order. If possible, released memory blocks are merged with adjacent memory blocks to form a single, large block of free dynamic memory.

The following three routines perform dynamic memory management functions:

- Initialize Dynamic Memory Routine ($INIDM), which initializes the task's free dynamic memory
- Request Core Block Routine ($RQCB), which allocates blocks of memory in the free dynamic memory
- Release Core Block Routine ($RLCB), which releases (deallocates) previously allocated memory blocks in the executing task's free dynamic memory

To use the dynamic memory management routines, provide the following information in the source program:

- A 2-word free memory listhead in the following format:

```
FREEHD:    .BLKW 2
```

- The appropriate call and argument(s) for the given routine.

Before building the task, invoke the Librarian Utility (LBR) and enter the following command line to include the modules INIDM and EXTSK in the task:

```
LB:[1,1]VMLIB/LB:INIDM:EXTSK
```

$INIDM—Initialize dynamic memory routine

The $INIDM routine establishes the initial state of the free dynamic memory available to the executing task. The free dynamic memory consists of all memory extending from the end of the task code to the highest virtual address used by the task, excluding resident libraries.

FORMAT

CALL $INIDM

INPUT

free memory listhead
In the program's data section: a 2-word memory listhead

free memory listhead address
In Register 0: the address of the free memory listhead

OUTPUT

first address
In Register 0: the first address in the task

last address
In Register 1: the address following the task image (last available address in the free dynamic memory)

memory size
In Register 2: the size of the free dynamic memory

DESCRIPTION

The $INIDM routine performs the following actions:

- Rounds the free dynamic memory base address to the next 4-byte boundary
- Initializes the free dynamic memory as a single large block of memory
- Computes the total size of the free dynamic memory
- Sets the outputs in Registers 0 and 1 and returns to the calling task

Registers 3 through 5 are not used.

After initializing dynamic memory, your task can call the Request Core Block Routine ($RQCB) to allocate memory blocks in the dynamic memory and the Release Core Block Routine ($RLCB) to release the allocated blocks.

EXAMPLE

The following source statements call the $INIDM routine to initialize a block of free dynamic memory and save the first address of the task in Register 0:

```
$FREEHD::       .BLKW   2          ; FREE MEMORY LISTHEAD
    .
    .
    .
        MOV     #$FREEHD,R0        ; SET ARG FOR FREE MEM HEAD
        CALL    $INIDM             ; INITIALIZE MEMORY
```

$RQCB—Request core block routine

The $RQCB system library routine determines whether there is enough space available in the free dynamic memory to satisfy an executing task's memory allocation request. If memory is available, the $RQCB routine allocates the requested memory block.

FORMAT

CALL $RQCB

INPUT

free memory listhead address

In Register 0: the address of the free memory listhead

block size

In Register 1: the size (number of bytes) of the memory block to be allocated, where:

R1	=	A value greater than or equal to 0, to specify best-fit allocation
R1	=	A value less than 0, to specify first-fit allocation (the value is negated to determine block size)

OUTPUT

block address

In Register 0: the dynamic memory address of the allocated block

block size

In Register 1: the actual size of the allocated block (requested size rounded to next 2-word boundary)

Condition Code

C bit	=	Clear if allocation is successful
C bit	=	Set if allocation is not successful

The $RQCB routine calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling task. Register 2 is destroyed.

EXAMPLE

The following source statements call the $RQCB routine to allocate a block of dynamic memory and store the memory address in Register 0:

```
$FREEHD::       .BLKW  2        ; FREE MEMORY LISTHEAD
    .
    .
    .
        MOV     #$FREEHD,R0     ; GET ADDRESS OF FREE CORE POOL
        MOV     #512.,R1        ; SIZE OF BLOCK TO BE ALLOCATED
        NEG     R1              ; NEGATE TO SPECIFY FIRST FIT
        CALL    $RQCB           ; REQUEST CORE BLOCK
```

$RLCB—Release core block routine

The $RLCB system library routine releases a block of previously allocated dynamic memory to the free memory list. The memory addresses determine the order of the memory list.

FORMAT

CALL $RLCB

INPUT

free memory listhead address
In Register 0: the address of the free memory listhead

block size
In Register 1: the size (number of bytes) of the block to be released

output address
In Register 2: the memory address of the block to be released

OUTPUT

released block
In the free memory list: the released dynamic memory block

DESCRIPTION

The $RLCB routine searches the free memory list until it finds the proper address slot and then merges the released block into the list. If possible, the released memory block is merged with adjacent blocks already in the free memory list.

The $RLCB routine calls the $SAVRG routine to save and subsequently restore Registers 3 through 5 of the calling task. Register 0 is unchanged, while the contents of Registers 1 and 2 are destroyed.

EXAMPLE

The following source statements call the $RLCB routine to release a block of memory, stored in buffer FREEHD, to the free memory listhead:

```
FREEHD::.BLKW   2               ; FREE MEMORY LISTHEAD
REFHD:  .WORD   0               ; REFERENCE LISTHEAD
    .
    .
    .
        MOV     REFHD,R2        ; GET ADDRESS OF ENTRY
        MOV     #4,R1           ; GET SIZE OF ENTRY
        MOV     #FREEHD,R0      ; SET ADDRESS OF LISTHEAD
        CALL    $RLCB           ; RELEASE CORE BLOCK
```

8 Virtual Memory Management Routines

The virtual memory management routines perform memory allocation and deallocation by paging to and from disk file storage to accommodate tasks that require more memory than that available in the task's free dynamic memory at any given time. That is, the routines allow you to bring pages into memory when they are needed, hold them there until they are no longer needed, swap the pages out, and reallocate their memory space to other pages. These routines do not require the memory management hardware and are not related to memory management directives.

The virtual memory management routines perform the following major functions:

- Virtual memory initialization
- Dynamic memory allocation
- Virtual memory allocation
- Page management

Although you can call the individual virtual memory management routines, it is more efficient to use them as automatic control systems by calling only the following key routines:

- The Initialize Virtual Memory Routine ($INIVM), which initializes the task's dynamic memory and the disk work file
- The virtual memory allocation routines Allocate Virtual Memory Routine ($ALVRT) and Allocate Small Virtual Block Routine ($ALSVB), which manage the allocation of large and small page blocks to enable page swapping to and from dynamic memory
- The following page management routines:
  - The Convert and Lock Page Routine ($CVLOK), which converts a virtual address to a dynamic memory address and sets a lock byte in the memory page to prevent its being swapped out of memory until it is no longer needed
  - The Unlock Page Routine ($UNLPG), which clears the lock byte in a memory-resident page so that it can be released and its memory space reallocated to another page
  - The Convert Virtual to Real Address Routine ($CVRL), which converts a virtual address to a dynamic memory address
  - The Write-Marked Page Routine ($WRMPG), which sets the "written into" flag of memory pages

8.1 Using the Virtual Memory Management Routines

To call the virtual memory management routines, provide the appropriate call arguments and statements in the source program.

Your task should contain an error-handling routine and symbolic error codes.

At task-build time, specify the file and the virtual memory management modules required by the task.

8.1.1 User Error—Handling Requirements

Four virtual memory management routines detect fatal error conditions. These routines require a user-written error-handling routine, entitled $ERMSG. In conjunction with the $ERMSG routine, you should include definitions of three global error codes and one global severity code in the task. The symbolic error codes are as follows:

Global	Error
E$R4	Used by the $ALBLK routine when there is no dynamic memory available for allocation
E$R73	Used by the $RDPAG and $WRPAG routines when a work file I/O error occurs during an attempt to swap pages between resident memory and disk storage
E$R76	Used by the $ALVRT routine when there is no virtual storage available for allocation
S$V2	(Severity code) Used by the four routines cited above to denote a fatal error that must be corrected before task execution can resume

When a fatal error occurs, the detecting routine sets up the following input arguments:

Register 1	=	Low byte: error code High byte: severity code (always S$V2)
Register 2	=	Argument block address

and issues the following call:

```
        CALL $ERMSG
```

Note that most of the virtual memory management routines interact, directly or indirectly, with one of the four routines that call $ERMSG (see the General Block Diagram for each routine). The only exceptions, which do not result in a call to $ERMSG, are the following routines:

$EXTSK
$FNDPG
$WRMPG
$LCKPG
$UNLPG

These five routines indicate error conditions by setting the Condition Code C bit. Your error-handling operations for these routines should respond to the Condition Code C bit. However, these routines might have to link with the error routine $ERMSG. Therefore, you must define the global symbols and an $ERMSG routine in your task whenever you use a virtual memory management routine. If you have not defined the error-handling routine within the task, the undefined global symbol diagnostic message will be generated at task-build time.

A typical error-handling routine would print a message to indicate the specific error condition, close all files (including the work file), and exit.

Example

The following source statements illustrate a user-written error-handling routine that can be called by a virtual memory management routine:

```
ER60:     .ASCIZ <15>/ACNT--Workfile - dynamic memory exhausted/
ER61:     .ASCIZ <15>/ACNT--Workfile - IO error or ADDR past EOF/
FILOPN:   .BYTE  0                ; FILE OPEN FLAG.  0 = NO, 1 = YES
          .EVEN
GENFLG:   .WORD  0                ; GENERAL FLAG WORD
; 1       BIT 0 - VIRTUAL FILE OPEN.  1 = OPEN, 0 = CLOSED
; 2       BIT 1 - ALLOCATE VIRTUAL BLOCK ERROR FLAG, 1 = ERROR
$ERMSG::  BIS    #2,GENFLG        ; SET ALLOCATE BLOCK ERROR
          CMPB   #E$R4,R1         ; DYNAMIC MEMORY ERROR?
          BNE    ERM2             ; NO
          MOV    #ER60,R0         ; YES, GET MESSAGE
          BR     ERROR
ERM2:     CMPB   #E$R73,R1        ; I/O ERROR OR ADDRESS PAST EOF?
          BNE    ERM3             ; NO
          MOV    #ER61,R0         ; YES, GET MESSAGE
          BR     ERROR
ERM3:                             ; ERROR-HANDLING ROUTINE
    .
    .
    .
EXIT:     TSTB   FILOPN           ; IS ACCOUNT FILE OPEN?
          BLE    10$              ; NO
          CALL   CLOSE            ; ROUTINE TO CLOSE ACCOUNT FILE

10$:      BIT    #1,GENFLG        ; WORK FILE OPEN?
          BEQ    15$              ; NO
          CALL   CLOSEV           ; ROUTINE TO CLOSE VIRTUAL FILE
                 .
15$:             .
                 .
ERROR:                            ; ERROR MESSAGE OUTPUT ROUTINE
```

NOTE: Generally, the error-handling routine should not attempt to return to the virtual memory management routine that detected the fatal error because no meaningful output would result.

1.2 Task-Building Requirements

There are two versions of the virtual memory management routines: the statistical version and the nonstatistical version. Each version consists of 12 program modules, each containing one or more routines, and a data storage module. Individual routines in the virtual memory management routines library can reference other routines. The relationship of the modules and routines in the library is shown in Table 8–1.

Table 8–1 Contents of the Virtual Memory Management Library File

Module Name			
Statistical	Nonstatistical	Routine Name	Routines Referenced
ALBLK	ALBLK	$ALBLK	$GTCOR, $EXTSK, $WRPAG
ALSVB	ALSVB	$ALSVB	$ALVRT, $WRMPG, $CVRL, $ALBLK, $RQVCB, $FNDPG, $RDPAG
CVRS	CVRL	$CVRL	$FNDPG, $ALBLK, $RDPAG
EXTSK	EXTSK	$EXTSK	(none)

Table 8–1 (Cont.) Contents of the Virtual Memory Management Library File

Module Name			
Statistical	Nonstatistical	Routine Name	Routines Referenced
FNDPG	FNDPG	$FNDPG	(none)
GTCOS	GTCOR	$GTCOR	$EXTSK,[1] $WRPAG
INIDM[2]	INIDM[2]	$INIDM	$EXTSK
INIVS	INIVM	$INIVM	$ALBLK, $GTCOR, $EXTSK, $WRPAG
MRKPG	MRKPG	$LCKPG	$FNDPG
		$UNLPG	$FNDPG
		$WRMPG	$FNDPG
RDPAS	RDPAG	$RDPAG	(none)
		$WRPAG	
RQVCB	RQVCB	$RQVCB	(none)
VMUTL	VMUTL	$CVLOK	$CVRL, $LCKPG, $FNDPG, $ALBLK, $RDPAG
VMDAS	VMDAT	Global data storage module	

[1]The Extend Task Routine ($EXTSK) is called by the $GTCOR routine, but only if GTCOS, the statistical version of $GTCOR, has been defined and initialized in your source program at task-build time.

[2]The INIDM module is a dynamic memory management module (see Chapter 7) that is normally used with the virtual memory management routines.

Four modules in the statistical version of the routines set up or maintain statistics of the use of the work file and memory. These modules and their associated statistical data fields are as follows:

- The INIVS module, which initializes the following three double-word fields:
  - The total work file access field ($WRKAC)
  - The work file read count field ($WRKRD)
  - The work file write count field ($WRKWR)

  Each of these fields is a double-word integer contained in the global data storage module (VMDAS) for the statistical version of the routines.
- The CVRS module, which maintains the count of total work file accesses in the $WRKAC field.
- The RDPAS module, which maintains a total of the work file reads in the $WRKRD field and a total of the work file writes in the $WRKWR field.
- The GTCOS module, which maintains a count of the total amount of free dynamic memory in the $FRSIZ single-word field. This field must be defined and initialized in the source program.

The statistical version of the virtual memory management routines does not automatically report these statistics. It is your responsibility to provide for the output of the statistical data in the fields described above if the statistical version of the routines is used.

To use the statistical routines, specify at task-build time the virtual memory management routines library file, the names of all statistical modules whose routines will be used at task-execution time, and the name of the global data storage module. The only optional modules are ALSVB and INIDM.

The following specifications identify all modules of the statistical version of the routines:

```
LB:[1,1]VMLIB/LB:ALBLK:ALSVB:ALVRT:CVRS:EXTSK:FNDPG:GTCOS
LB:[1,1]VMLIB/LB:INIVS:MRKPG:RDPAS:RQVCB:VMUTL:INIDM:VMDAS
```

The nonstatistical routines use the global data storage module VMDAT. To use the nonstatistical routines, you specify at task-build time the virtual memory management routines library file, the names of all nonstatistical modules whose routines will be used at task-execution time, and the name of the global data storage module. The only optional modules are ALSVB and INIDM.

The following specifications identify all modules of the nonstatistical version of the routines:

```
LB:[1,1]VMLIB/LB:ALBLK:ALSVB:ALVRT:CVRL:EXTSK:FNDPG:GTCOR
LB:[1,1]VMLIB/LB:INIVM:MRKPG:RDPAG:RQVCB:VMUTL:INIDM:VMDAT
```

$INIVM—Virtual memory initialization routine

The $INIVM routine initializes the task's free dynamic memory, sets up the page address control list, and initializes your disk work file to enable memory-to-disk page swapping. Disk work file capacity is 64K words.

FORMAT

CALL $INIVM

INPUT

$FRHD block

In your source program: define and initialize a 2-word field named $FRHD. To define the field, include the following code in your source program:

```
$FRHD:: .BLKW 2.
```

To initialize the field, store the starting address of the free dynamic memory in $FRHD.

globals

In your source program: four global symbols as follows:

W$KLUN	Logical unit number (LUN) to be used for the work file. You must assign this LUN to a disk device.
W$KEXT	Work file extension size (in blocks). A negative number indicates that the extend should first be requested as a contiguous allocation of disk blocks. A positive number indicates that the extend need not be contiguous.
N$MPAG	Fast page search page count. If there is sufficient dynamic memory to allocate the number of pages specified, N$MPAG will set aside 512 words of dynamic memory to speed up the searching of memory-resident pages.
$WRKPT	Store the address of the FDB in the word $WRKPT before calling $INIVM.

memory address

In Register 1: the highest address of the task's free dynamic memory

OUTPUT

Condition Code

Initialization succeeded if both of the following conditions exist:

C bit	=	Clear
Register 0	=	0

Initialization failed if the following conditions exist:

C bit	=	Set
Register 0	=	One of the following values:
		–2 to indicate work file open failure
		–1 to indicate work file mark-for-deletion failure

NOTE: Before calling the $INIVM routine, the task can call the $INIDM routine (see Chapter 7), which returns the last address of dynamic memory and the total size of dynamic memory.

Also, you can examine the FCS error code at offset F.ERR in the work file FDB. The address of the FDB is stored in the word $WRKPT.

DESCRIPTION

Starting at the high address of the calling task's free dynamic memory, the $INIVM routine clears control fields and the page address control listhead. The $INIVM routine then sets up the heading for a new page address control list and calls the Allocate Block Routine ($ALBLK) to allocate a memory page block for the control list. The $INIVM routine calls the $ALBLK routine to allocate a page block for the first memory page for the calling task, and links the first allocated page to the page control list.

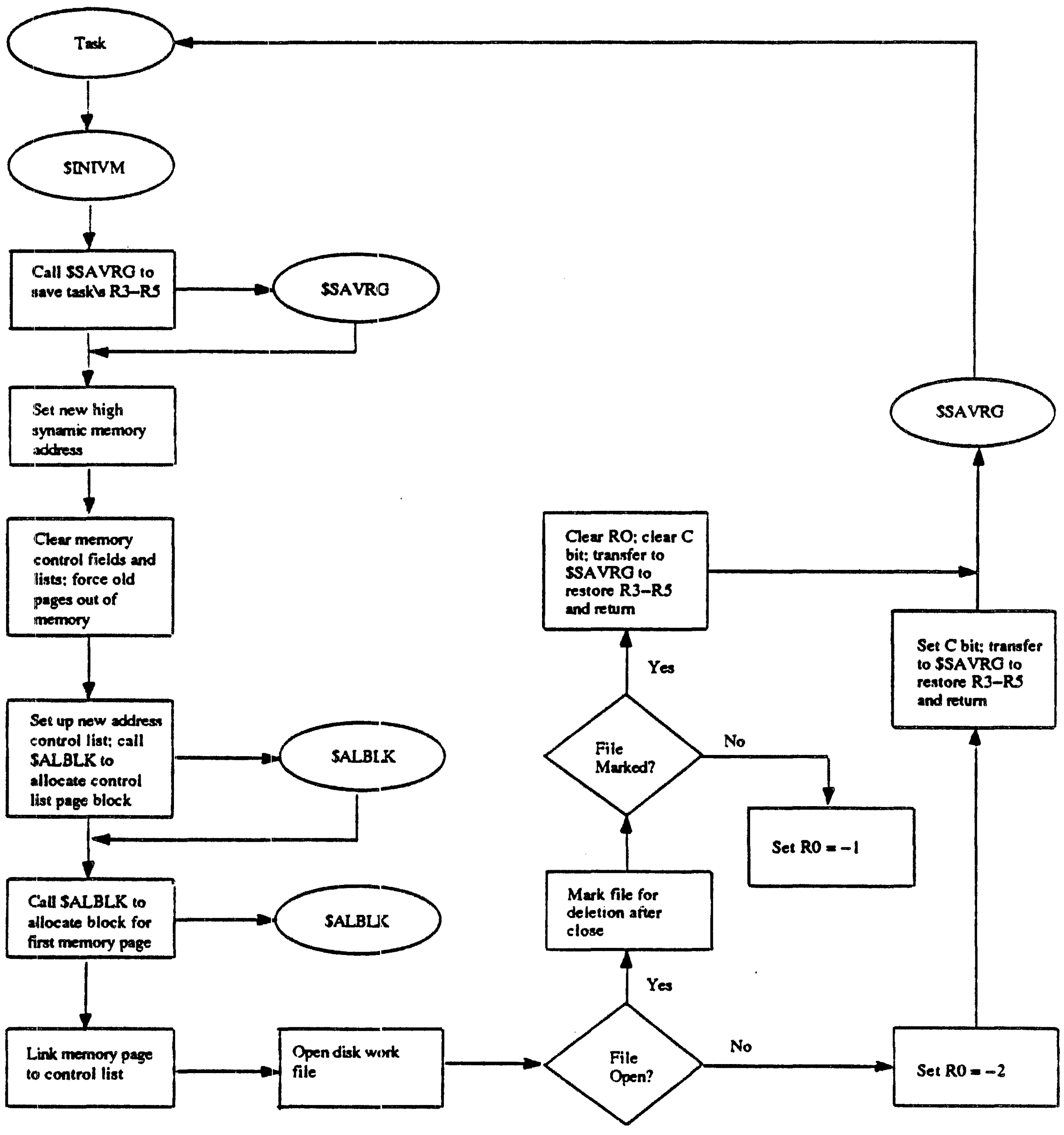
The $INIVM routine initializes (opens) your disk work file. If the file is opened successfully, the $INIVM routine attempts to mark it for deletion. This ensures that the file will be deleted automatically when it is closed, or if the task terminates abnormally or exits.

NOTE: The work file can be closed by the operation CLOSE$ $WRKPT.

The $INIVM routine destroys the contents of Registers 0 through 2. Whether or not the initialization is successful, it transfers control to the $SAVRG routine, which restores Registers 3 through 5 and returns to the calling task.

The interaction of the $INIVM routine with your task and the Allocate Block Routine ($ALBLK) is shown in Figure 8–1.

Figure 8–1 General Block Diagram of the $INIVM Routine


Task
$INIVM
Call $SAVRG to save task\s R3–R5
$SAVRG
Set new high synamic memory address
Clear memory control fields and lists; force old pages out of memory
Set up new address control list; call $ALBLK to allocate control list page block
$ALBLK
Call $ALBLK to allocate block for first memory page
$ALBLK
Link memory page to control list
Open disk work file
File Open?
Yes
No
Mark file for deletion after close
File Marked?
Yes
No
Clear R0; clear C bit; transfer to $SAVRG to restore R3–R5 and return
Set R0 = –1
Set R0 = –2
Set C bit; transfer to $SAVRG to restore R3–R5 and return
$SAVRG

EXAMPLE

The following source statements call the $INIVM routine to initialize free dynamic memory and then call $WRKPT to close the work file. In this example, the $INIDM routine provides the required free memory address in Register 1:

```
E$R4     ==    4               ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73    ==    73              ; WORK FILE I/O ERROR
E$R76    ==    76              ; WORK FILE EXCEEDED
S$V2     ==    302             ; SEVERITY 2
W$KLUN   ==    4               ; WORK FILE LUN
N$MPAG   ==    20              ; FAST PAGE SEARCH PAGE COUNT
W$KEXT   ==    24              ; WORK FILE EXTENSION SIZE (BLOCKS)
$WRKPT:  .WORD 0               ; ADDRESS OF FDB
$FRHD::  .BLKW 2               ; FREE MEMORY LISTHEAD
$FRSIZ:: .BLKW 1               ; SIZE COUNT FOR FREE MEMORY
GENFLG:  .WORD 0               ; GENERAL WORD FLAG
         ; 1   BIT 0 - VIRTUAL FILE OPEN   - 1 = OPEN, 0 = CLOSED

         MOV   #$FRHD,R0       ; SET ARG FOR FREE MEMORY HEAD
         CALL  $INIDM          ; INITIALIZE MEMORY
         MOV   R2,$FRSIZ       ; SET ARG FOR SIZE
         CALL  $INIVM          ; INITIALIZE WORK FILE SUBSYSTEM
    .
    .
    .
         CLOSE$ $WRKPT         ; CLOSE VIRTUAL WORK FILE
         BIC   #1,GENFLG       ; CLEAR WORK FILE OPEN FLAG
         RTS   PC
```

The core allocation routines manage the allocation and deallocation of space in the free dynamic memory of the executing task. The core allocation routines are as follows:

- The Allocate Block Routine ($ALBLK), which provides the interface between the executing task and the other core allocation routines. That is, the executing task is provided all the services of the core allocation routines by simply calling the $ALBLK routine, or those routines that call the $ALBLK routine.
- The Get Core Routine ($GTCOR), which is always called by the $ALBLK routine to perform the necessary processing to allocate the requested memory space from the free dynamic memory.
- The Request Core Block Routine ($RQCB), which is called by the $GTCOR routine to allocate the requested memory space if it is available in the free dynamic memory.
- The Write Page Routine ($WRPAG), which is called by the $GTCOR routine to transfer memory pages to your disk work file to free enough memory space to satisfy the memory allocation request.
- The Release Core Block Routine ($RLCB), which is called by the $GTCOR routine to release space previously allocated to a memory page that has been transferred to the disk work file.

In addition to the five core allocation routines mentioned above, there is a sixth routine called the Extend Task Routine ($EXTSK), which is accessed by the statistical module GTCOS. The $EXTSK routine is called by the $GTCOR routine to extend the size of the task region, thus making enough memory available in the free dynamic memory to satisfy the allocation request.

Do not confuse the statistical module GTCOS with the nonstatistical module GTCOR. Both of these modules are called by references to the entry point $GTCOR. $GTCOR calls $EXTSK only when you include the statistical module GTCOS at task-build time. If you do not include GTCOS, the $GTCOR routine uses the nonstatistical module GTCOR by default.

$ALBLK—Allocate block routine

The $ALBLK routine determines whether a block of memory storage can be allocated from the free dynamic memory. If so, the $ALBLK routine clears (zeros) the allocated block and returns the resident memory address of the block to the calling task. If there is insufficient space in the free dynamic memory, the requested block cannot be allocated.

FORMAT

CALL $ALBLK

INPUT

block size

In Register 1: the size (number of bytes less than or equal to 512_{10}) of the memory storage block to be allocated

error code

In the task: the definitions for the following global symbols:

E$R4
S$V2

OUTPUT

block address

In Register 0: the dynamic memory address of the allocated, cleared block

error response

If allocation is unsuccessful, $ALBLK produces the following output:

In Register 1: sets the error/severity codes E$R4 and S$V2
In Register 2: saves the address of the argument block $FRHD (free memory header)

The $ALBLK routine then calls the user $ERMSG routine.

DESCRIPTION

The $ALBLK routine calls the Get Core Routine ($GTCOR) to allocate the requested memory block, as follows:

- Request allocation from the free dynamic memory
- If the request is not met, attempt to extend the task region to increase the size of the free dynamic memory

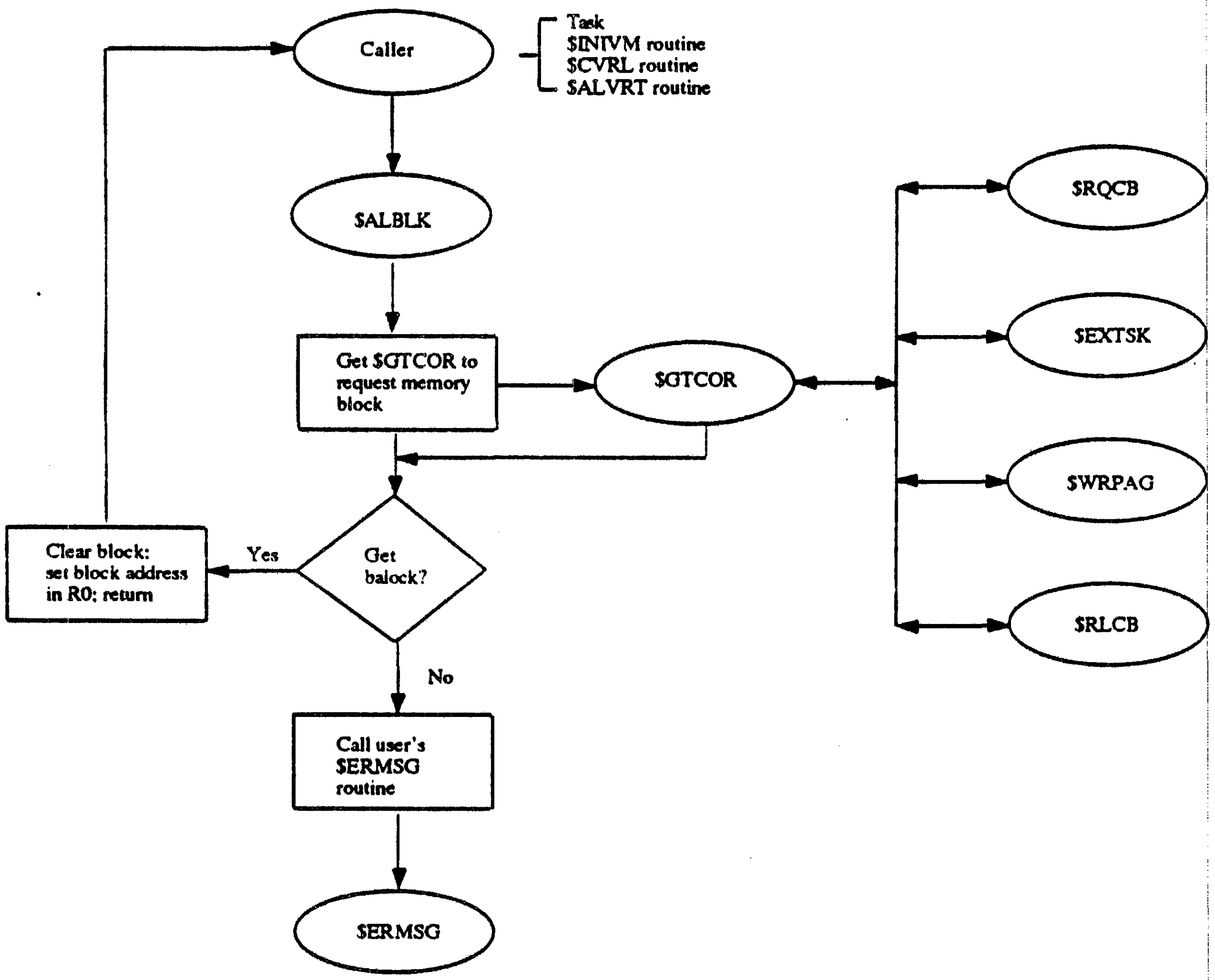
- If the task cannot be extended, swap unlocked pages from memory storage to disk to deallocate memory space for reallocation

In addition to being called by the user task, the $ALBLK routine is called by the following virtual memory management routines:

- Initialize Virtual Memory Routine ($INIVM), which calls $ALBLK to allocate initial blocks of dynamic memory to enable page swapping between disk and memory storage
- Convert Virtual to Real Address Routine ($CVRL), which calls $ALBLK to allocate a block of dynamic memory for a virtual page block
- Allocate Virtual Memory Routine ($ALVRT), which calls $ALBLK to allocate a memory page block for a virtual page block that is to be swapped from memory to disk storage

Figure 8–2 shows the interaction of the $ALBLK routine with a user task and other virtual memory management routines.

Figure 8–2 General Block Diagram of the $ALBLK Routine



EXAMPLE

The following source statements call the $ALBLK routine to allocate a 4-byte block of memory and store the address of the block in buffer REFHD:

```
E$R4     ==     4             ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73    ==     73            ; WORK FILE I/O ERROR
E$R76    ==     76            ; WORK FILE EXCEEDED
S$V2     ==     2             ; SEVERITY 2
REFHD:   .BLKW  2             ; REFERENCE LISTHEAD
         MOV    R1,-(SP)      ; SAVE VIRTUAL ADDRESS OF REFERENCE
         MOV    #4,R1         ; GET SIZE OF BLOCK
         CALL   $ALBLK        ; ALLOCATE CORE BLOCK
         MOV    R0,@REFHD+2   ; LINK REAL ADDRESS TO OLD LAST BLOCK ADDRESS
         MOV    R0,REFHD+2    ; SET NEW LAST BLOCK ADDRESS
         MOV    (SP)+,2(R0)   ; RECORD VIRTUAL ADDRESS OF REFERENCE
```

$GTCOR—Get core routine—nonstatistical module GTCOR

The $GTCOR routine (defined in the nonstatistical module GTCOR) attempts to allocate requested dynamic memory blocks in the following ways:

- Allocate memory from the currently available space in the free dynamic memory
- Swap unlocked page blocks from dynamic memory to disk, freeing previously allocated memory space for reallocation

FORMAT

CALL $GTCOR

INPUT

block size

In Register 1: the size (number of bytes less than or equal to 512_{10}) of the dynamic memory block to be allocated

OUTPUT

block address

In Register 0: the memory address of the dynamic memory block, if allocated

Condition Code

C bit	=	Clear if the allocation was successful
C bit	=	Set if the allocation failed

DESCRIPTION

$GTCOR calls the Request Core Block Routine ($RQCB; described in Chapter 7) to determine whether enough free dynamic memory space is currently available to satisfy the allocation request. If so, the $GTCOR routine returns the memory address of the resident block to the caller.

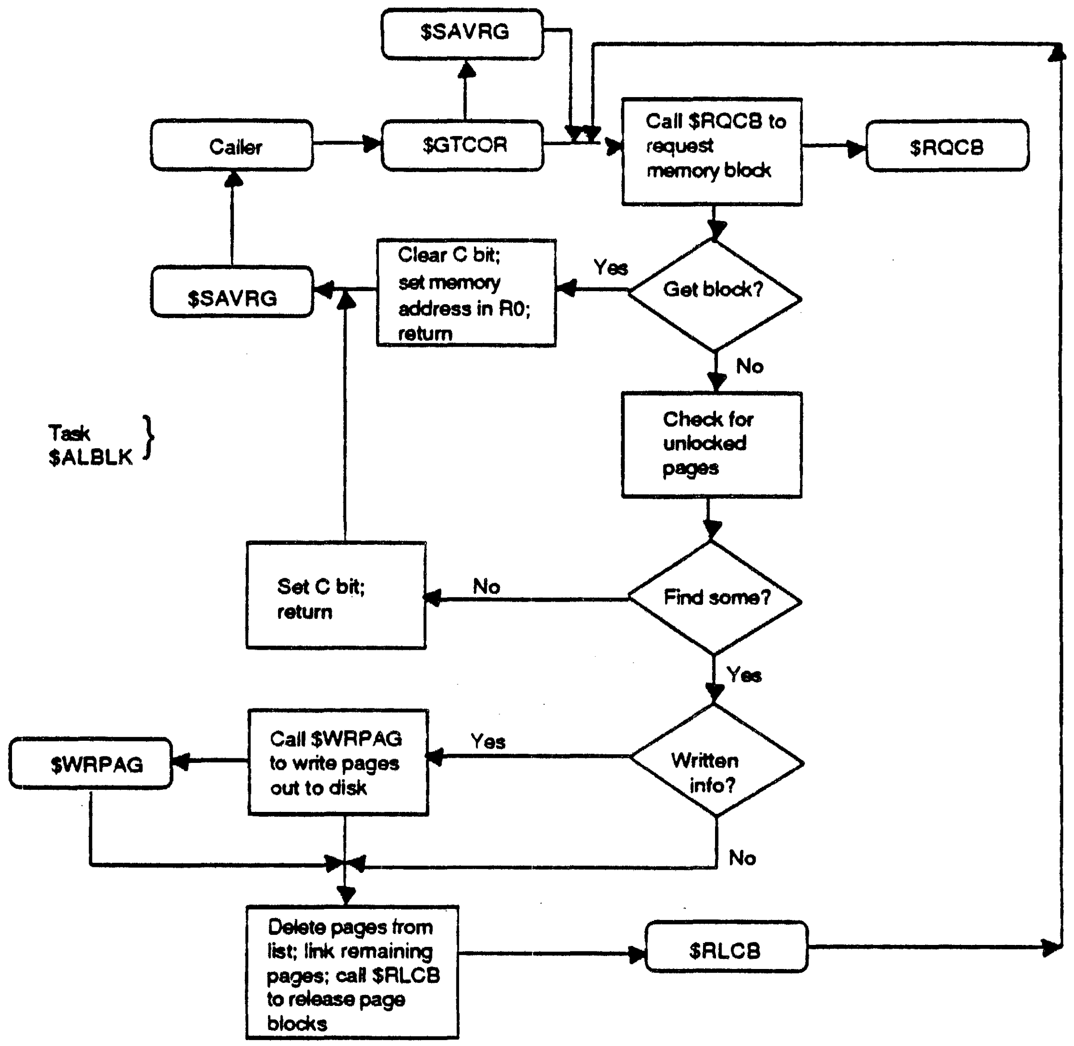
If the $RQCB routine cannot allocate the requested block from the current free dynamic memory, the $GTCOR routine searches for the unlocked pages currently resident in memory. If any unlocked pages are found, the least recently used (LRU) page is released and its memory space is allocated to the new page. If an unlocked page cannot be found, $GTCOR sets the C bit, indicating that it failed to find an unlocked page, and returns control to the caller.

When an LRU page is found, the $GTCOR routine checks the page to see if it has been written into. If so, the Write Page Routine ($WRPAG) is called to write the page to the disk work file. The Release Core Block Routine ($RLCB; described in Chapter 7) is called to release the page and the Request Core Block Routine ($RQCB) is called to allocate the page. The memory address of the allocated page returns in Register 0 to the caller. If the $GTCOR routine does not obtain sufficient memory for the requested block, it sets the C bit in the Condition Code and returns control to the caller. $GTCOR calls the $SAVRG routine to save and restore Registers 3 through 5 of the caller.

The $GTCOR routine is always called by the Allocate Block Routine ($ALBLK).

Figure 8–3 shows the interaction of the $GTCOR routine with other system library and virtual memory management routines.

Figure 8–3 General Block Diagram of the $GTCOR Routine (Nonstatistical Module GTCOR)


$SAVRG
Caller
$GTCOR
Call $RQCB to request memory block
$RQCB
Clear C bit; set memory address in R0; return
Yes
Get block?
$SAVRG
No
Task $ALBLK
Check for unlocked pages
Set C bit; return
No
Find some?
Yes
$WRPAG
Call $WRPAG to write pages out to disk
Yes
Written info?
No
Delete pages from list; link remaining pages; call $RLCB to release page blocks
$RLCB

EXAMPLE

The following source statements call the $GTCOR routine to allocate a memory block of one byte plus the length of the header:

```
E$R4     ==     4               ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73    ==     73              ; WORK FILE I/O ERROR
E$R76    ==     76              ; WORK FILE EXCEEDED
S$V2     ==     302             ; SEVERITY 2
LENGTH: .BLKW   1               ; LENGTH OF RECORD JUST READ
HDSZ:   .BLKW   1               ; LENGTH OF HEADER
        .EVEN

        MOV     #1,R0           ; SET LENGTH TO ONE BYTE
        MOV     R0,LENGTH       ; REMEMBER THE LENGTH
        ADD     #HDSZ,R1        ; ADD HEADER LENGTH
        ADD     R0,R1           ; ADD ALLOWANCE FOR MODIFICATIONS
        CALL    $GTCOR          ; ALLOCATE SPACE
```

$GTCOR—Get core routine—statistical module GTCOS

The $GTCOR routine (accessed by the statistical module GTCOS) attempts to allocate requested dynamic memory blocks in one of the following ways:

- Allocate memory from the currently available space in the free dynamic memory
- Extend the task region, increasing the size of the free dynamic memory to accommodate the allocation request
- Swap unlocked page blocks from dynamic memory to disk, which frees previously allocated memory space for reallocation

FORMAT

CALL $GTCOR

INPUT

block size

In Register 1: the size (number of bytes less than or equal to 512_{10}) of the dynamic block memory to be allocated

OUTPUT

address

In Register 0: the memory address of the dynamic block, if allocated

Condition Code

C bit	=	Clear if the allocation was successful
C bit	=	Set if the allocation failed

DESCRIPTION

The Request Core Block Routine ($RQCB; described in Chapter 7) is called to determine whether enough free dynamic memory space is currently available to satisfy the allocation request. If so, the $GTCOR routine returns the memory address of the resident block to the caller.

If the requested block cannot be allocated from the current free dynamic memory, the $GTCOR routine calls the Extend Task Routine ($EXTSK) to determine whether the task region can be extended to make available the requested space in the free dynamic memory. If so, the $GTCOR routine returns the memory address to the caller.

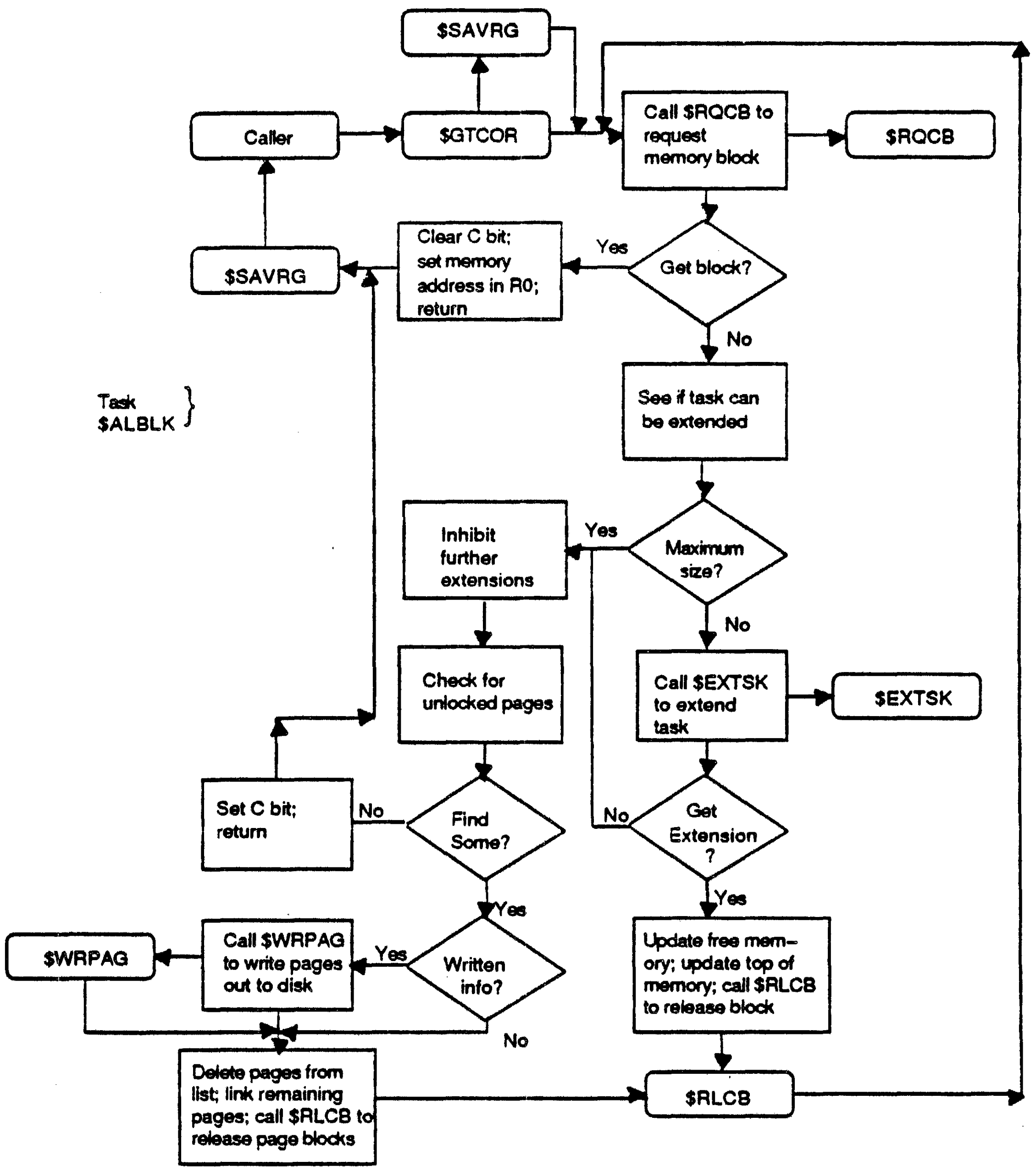
If the task region cannot be extended, the $GTCOR routine searches for unlocked pages currently resident in memory. If any unlocked pages are found, the least recently used (LRU) page is released and its memory space is allocated to the new page.

When an LRU page is found, the $GTCOR routine checks the page to see if it has been written into. If so, the Write Page Routine ($WRPAG) is called to write the page to the disk work file. The Release Core Block Routine ($RLCB; described in Chapter 7) is called to release the page and the Request Core Block Routine ($RQCB) is called to allocate the page. The memory address of the allocated page is returned in Register 0 to the caller. If the $GTCOR routine is not able to obtain sufficient memory for the requested block, it sets the C bit in the Condition Code and returns control to the caller. The $GTCOR routine calls the $SAVRG routine to save and subsequently restore Registers 3 through 5 of the caller.

The $GTCOR routine is always called by the Allocate Block Routine ($ALBLK).

Figure 8–4 shows the interaction of the $GTCOR routine with other system library and virtual memory management routines.

Figure 8–4 General Block Diagram of the $GTCOR Routine (Statistical Module GTCOS)


$SAVRG
Caller
$GTCOR
Call $RQCB to request memory block
$RQCB
Clear C bit; set memory address in R0; return
Yes
Get block?
$SAVRG
No
Task
$ALBLK
See if task can be extended
Inhibit further extensions
Yes
Maximum size?
No
Check for unlocked pages
Call $EXTSK to extend task
$EXTSK
Set C bit; return
No
Find Some?
No
Get Extension ?
Yes
Yes
$WRPAG
Call $WRPAG to write pages out to disk
Yes
Written info?
Update free memory; update top of memory; call $RLCB to release block
No
Delete pages from list; link remaining pages; call $RLCB to release page blocks
$RLCB

$EXTSK—Extend task routine

The $EXTSK routine extends the current region of the task to increase the amount of available memory for allocation. It extends the task region by the specified size rounded to the next 32-word boundary.

FORMAT

CALL $EXTSK

INPUT

block size

In Register 1: the size (number of bytes less than or equal to 512_{10}) of the memory storage block to be allocated

OUTPUT

extension size

In Register 1: the actual extension size (requested size rounded to next 32-word boundary)

Condition Code

C bit	=	Clear if extension was successful
C bit	=	Set if extension failed

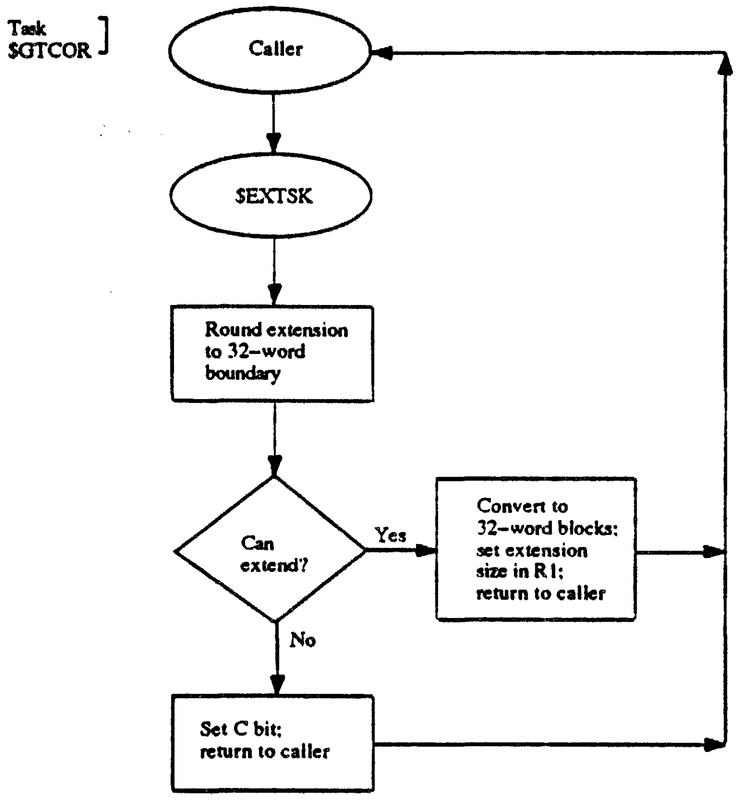
DESCRIPTION

The $EXTSK routine is called by the Get Core Routine ($GTCOR) when there is insufficient space in the current free dynamic memory to satisfy a memory block allocation request. The $EXTSK routine rounds the requested extension size to the next 32-word boundary. If there is enough memory space available, $EXTSK extends the task region, returning the total amount of the extension, in Register 1, to the $GTCOR routine. It preserves all other registers of the caller. If it cannot extend the task region, the $EXTSK routine sets the C bit in the Condition Code and returns to the $GTCOR routine.

While you can call the $EXTSK routine directly, the routine is also called by the Initialize Dynamic Memory Routine ($INIDM), described in Chapter 7.

Figure 8–5 shows the interaction of the $EXTSK routine with the $GTCOR routine (in statistical module GTCOS).

Figure 8–5 General Block Diagram of the $EXTSK Routine


Task
$GTCOR
Caller
$EXTSK
Round extension to 32–word boundary
Can extend?
Yes
No
Convert to 32–word blocks; set extension size in R1; return to caller
Set C bit; return to caller

EXAMPLE

The following source statements call the $EXTSK routine to extend the amount of memory available to the task:

```
T$KINC  ==      256.            ; TASK INCREMENT
T$KMAX  ==      0               ; MAXIMUM SIZE OF TASK
P$TADDR:.WORD   0               ; NEXT FREE ADDRESS
FRHD:   .BLKW   2               ; FREE MEMORY LISTHEAD

        CALL    $SAVRG          ; SAVE NONVOLATILE REGISTERS
10$:
        MOV     R1,-(SP)        ; SAVE BYTE COUNT
        MOV     #FRHD,R0        ; GET ADDRESS OF FREE CORE POOL
        CALL    $RQCB           ; REQUEST CORE BLOCK
        BCC     60$             ; IF C BIT CLEAR, SPACE IS ALLOCATED
        MOV     #P$TADDR,R3     ; GET POINTER TO NEXT FREE ADDRESS
        MOV     (R3),R2         ; GET NEXT FREE ADDRESS
        CMP     R2,#T$KMAX      ; IS TASK AT MAXIMUM ALLOWABLE SIZE?
        BHIS    17$             ; IF TASK HIGHER OR SAME, YES
        MOV     #T$KINC,R1      ; GET TASK INCREMENT (IN BYTES)
        CALL    $EXTSK          ; EXTEND THE TASK
        BCS     ERR$            ; IF C BIT SET, EXTENSION FAILED
        ADD     R1,FRHD         ; ADD INCREMENT TO POOL
        ADD     R1,(R3)         ; UPDATE TOP OF MEMORY
        BR      47$             ; RELEASE BLOCK TO POOL
17$:
        MOV     #-1,(R3)        ; BLOCK FURTHER ATTEMPTS TO EXTEND TASK
47$:
        MOV     #FRHD,R0        ; GET ADDRESS OF FREE CORE POOL
        CALL    $RLCB           ; RELEASE MEMORY
        MOV     (SP)+,R1        ; RESTORE BYTE COUNT
        BR      10$             ; BEGIN AGAIN
60$:
        INC     (SP)+           ; CLEAN STACK, LEAVE C BITS INTACT
        RTS     PC
```

$WRPAG—Write page routine

The $WRPAG routine transfers a memory page to the disk work file.

FORMAT

CALL $WRPAG

INPUT

page address

In Register 2: the dynamic memory address of the page to be transferred to disk

error code

In the task: the definitions for the following global symbols:

E$R73
S$V2

OUTPUT

Condition Code

C bit	=	Clear if transfer succeeded
C bit	=	Set if transfer failed

error response

If transfer is not successful, $WRPAG produces the following output:

In Register 1: sets the error/severity codes E$R73 and S$V2

The $WRPAG routine then calls the user $ERMSG routine.

DESCRIPTION

The $WRPAG routine is called by the Get Core Routine ($GTCOR) to transfer to your disk work file a resident memory page that has been written into.

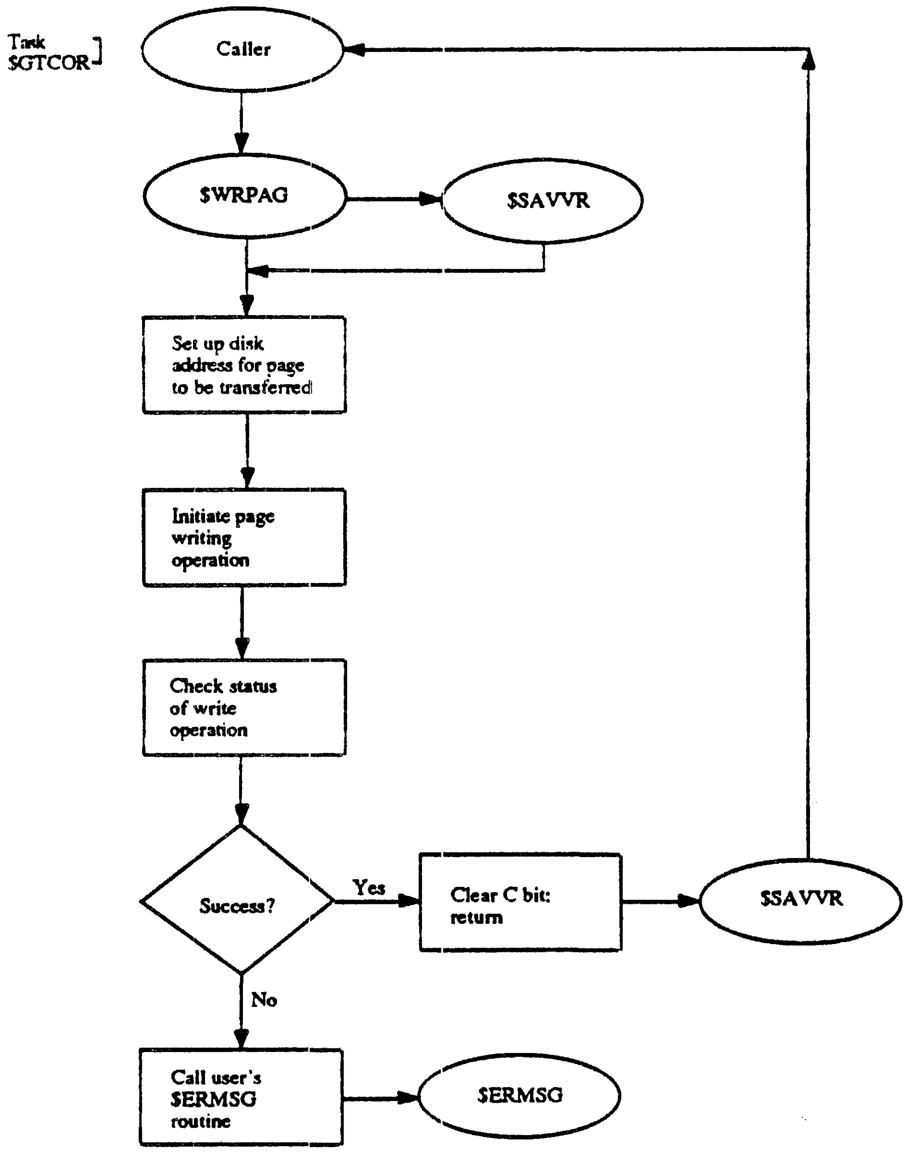
The $WRPAG routine calls the $SAVVR routine to save and subsequently restore Registers 0 through 2 of the caller. The routine then performs the following actions:

- Sets up the disk work file address of the page to be transferred
- Initiates the page-writing operation
- Checks the status of the write operation

- Indicates a successful transfer (clears the C bit in the Condition Code) and returns control to the $SAVVR routine, or calls your $ERMSG routine if a fatal work file I/O error prevented the page transfer

Figure 8–6 shows the interaction of the $WRPAG routine with the $GTCOR routine.

Figure 8–6 General Block Diagram of the $WRPAG Routine


Task
$GTCOR
Caller
$WRPAG
$SAVVR
Set up disk address for page to be transferred
Initiate page writing operation
Check status of write operation
Success?
Yes
No
Clear C bit: return
$SAVVR
Call user's $ERMSG routine
$ERMSG

EXAMPLE

The following source statements call the $WRPAG routine to transfer a memory page from buffer P$GNXT to the disk work file:

```
E$R4     ==     4              ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73    ==     73             ; WORK FILE I/O ERROR
E$R76    ==     76             ; WORK FILE EXCEEDED
S$V2     ==     302            ; SEVERITY 2
P$GNXT: .WORD   0              ; NEXT PAGE WORK FILE

        MOV     R4,R5          ; SAVE PREDECESSOR
        MOV     P$GNXT,R4      ; GET NEXT PAGE

        MOV     R4,R2          ; SET UP BUFFER FOR TRANSFER
        CALL    $WRPAG         ; WRITE OUT PAGE INTO DISK WORK FILE
```

8.2 Virtual Memory Allocation Routines

Virtual memory allocation routines manage the allocation of disk and memory storage to enable page swapping from the free dynamic memory to your disk work file. The three virtual memory allocation routines are as follows:

- Allocate Virtual Memory Routine ($ALVRT), which allocates disk and memory page blocks, maintains page control and address tables, and interfaces with the executing task and the cor allocation and page management routines.
- Allocate Small Virtual Block Routine ($ALSVB), which allocates small page blocks of disk and memory storage within large page blocks to enable efficient use of storage. The $ALSVB routine interfaces with the $ALVRT routine and page management routines to ensure address and status control of small pages in memory and disk storage.
- Request Virtual Core Block Routine ($RQVCB), which manages page-block allocation on your disk work file when it is called by the $ALVRT routine.

$ALVRT—Allocate virtual memory routine

The $ALVRT routine determines whether a page block of virtual storage can be allocated on your disk work file. If so, the $ALVRT routine allocates an equal amount of memory storage, updates page control and address tables, and returns the disk and memory addresses of the allocated page blocks to the caller. If the $ALVRT routine cannot allocate the requested storage, the error and severity codes E$R76 and S$V2 are stored in Register 1 and the user's $ERMSG routine is called.

FORMAT

CALL $ALVRT

INPUT

block size

In Register 1: the number of bytes to be allocated

NOTE: The maximum size of a page block is 512_{10} bytes.

OUTPUT

memory address

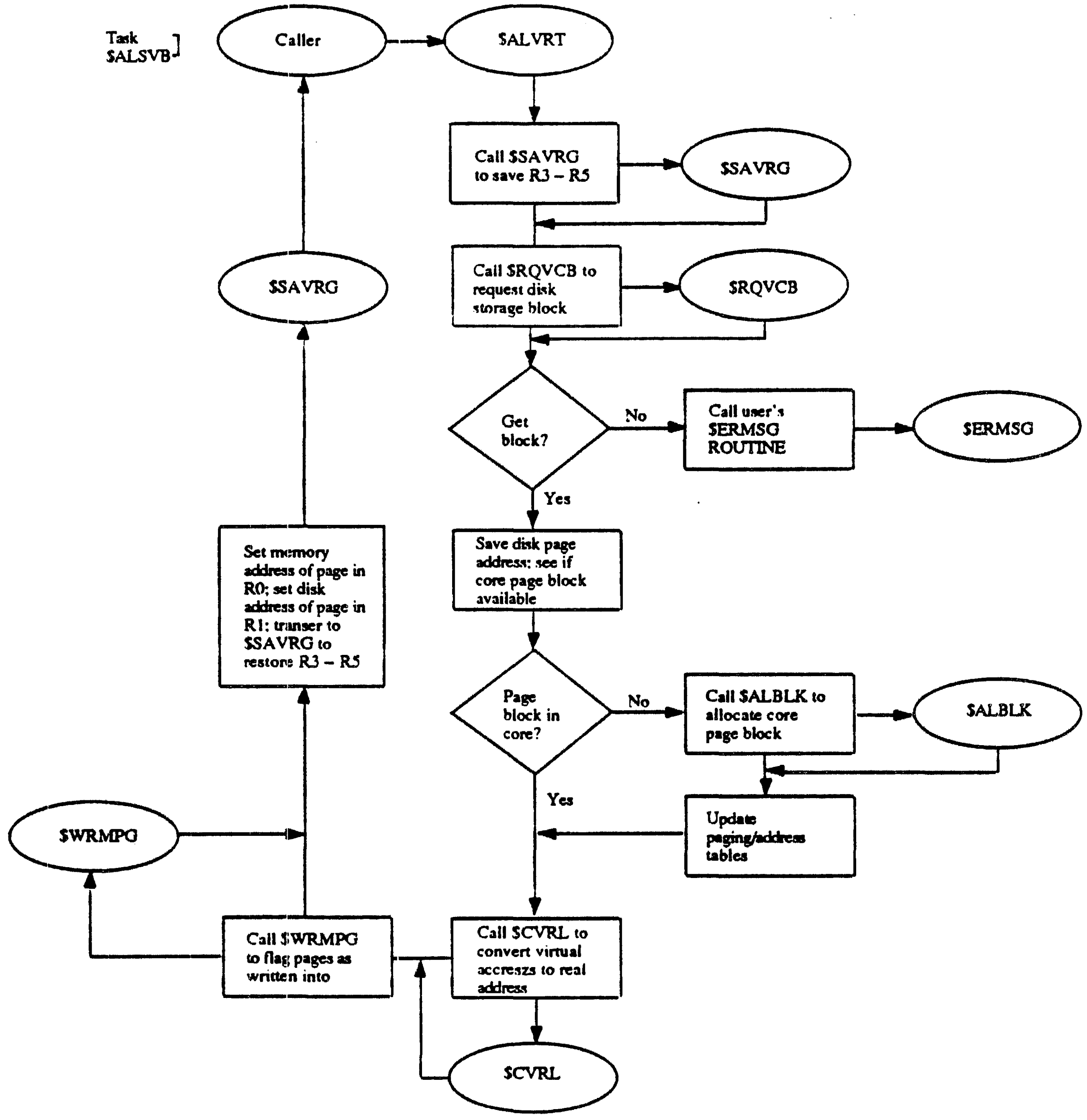
In Register 0: the memory address of the allocated page block
In Register 1: the virtual address of the allocated page block

DESCRIPTION

The $ALVRT routine calls the Request Virtual Core Block Routine ($RQVCB) to determine whether the requested storage can be allocated on the disk work file. If not, a fatal error is signalled and the $ALVRT routine calls your $ERMSG routine. If it can allocate the disk storage, the $RQVCB routine returns the disk page block address to the $ALVRT routine, which determines whether a page block of space is available in memory. If not, the Allocate Block Routine ($ALBLK) is called to allocate a page block. The $ALVRT routine then calls the Convert Virtual to Real Address Routine ($CVRL) to convert the virtual address to a memory address.

The $ALVRT routine calls the Write-Marked Page Routine ($WRMPG) to set the "written into" flag of the memory page. It also calls the $SAVRG routine to save and restore Registers 3 through 5 of the calling routine. Although you can call the $ALVRT routine directly, it is also called automatically by the Allocate Small Virtual Block Routine ($ALSVB). Figure 8–7 shows the interaction of the $ALVRT routine with your task and other virtual memory management routines.

Figure 8–7 General Block Diagram of the $ALVRT Routine


Task $ALSVB
Caller
$ALVRT
Call $SAVRG to save R3 – R5
$SAVRG
Call $RQVCB to request disk storage block
$RQVCB
$SAVRG
Get block?
No
Call user's $ERMSG ROUTINE
$ERMSG
Yes
Save disk page address; see if core page block available
Set memory address of page in R0; set disk address of page in R1; transer to $SAVRG to restore R3 – R5
Page block in core?
No
Call $ALBLK to allocate core page block
$ALBLK
Yes
Update paging/address tables
$WRMPG
Call $WRMPG to flag pages as written into
Call $CVRL to convert virtual accreszs to real address
$CVRL

EXAMPLE

The following source statements call the $ALVRT routine to allocate a page block of virtual memory on a disk file. In this example, the statements save the contents of Registers 0 and 2 before calling $ALVRT:

```
E$R4     ==     4               ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73    ==     73              ; WORK FILE I/O ERROR
E$R76    ==     76              ; WORK FILE EXCEEDED
S$V2     ==     302             ; SEVERITY 2
TEMP1:   .WORD  0               ; TEMPORARY BUFFER FOR VIRTUAL MEMORY
TEMP2:   .WORD  0               ; TEMPORARY BUFFER FOR VIRTUAL MEMORY
A.LEN:   .BLKW  1               ; LENGTH OF VIRTUAL ELEMENT

         MOV    R0,TEMP1        ; SAVE POINTER IN INPUT BUFFER
         MOV    R2,TEMP2        ; SAVE NUMBER OF BYTES IN BUFFER
         MOV    A.LEN,R1        ; LENGTH OF VIRTUAL ELEMENT TO REGISTER 1
         CALL   $ALVRT          ; ALLOCATE VIRTUAL BLOCK
```

$ALSVB—Allocate small virtual block routine

The $ALSVB routine allocates small page blocks within large page blocks of disk and memory storage. Thus, the routine accommodates variable user allocation size requirements and minimizes wasted storage space.

The $ALSVB routine initially allocates a large page block, then performs suballocation of requested small blocks within the large block. When the space within a large block is exhausted, a new large block is allocated by the $ALSVB routine.

FORMAT

CALL $ALSVB

INPUT

memory block

In the source program: a large memory block defined as follows:

```
                N$DLGH == 512.
```

NOTE: Normally, 512 is the size of a large memory block. In any case, it must be less than or equal to 512_{10}.

page block size

In Register 1: the size of the page block to be allocated, where:

R1	=	Zero (0) to force the allocation of a large virtual page block on the first call to $ALSVB
R1	=	A value less than or equal to 512_{10} specifying the size, in bytes, of the small page to be allocated

OUTPUT

In Register 0: the dynamic memory address of the allocated page block

virtual address

In Register 1: the virtual address of the allocated block

DESCRIPTION

When a small page block is to be allocated within an existing large page block, the $ALSVB routine calls the Convert Virtual to Real Address Routine ($CVRL) to do the following:

- Locate the allocated large page, if it is memory-resident (if it is not resident, read the page from disk to memory)
- Convert the virtual page address to a memory page address

- Transfer the large page block from disk into the large memory page block

The $ALSVB routine calls the Write-Marked Page Routine ($WRMPG) to set the "written into" flag of the allocated memory page.

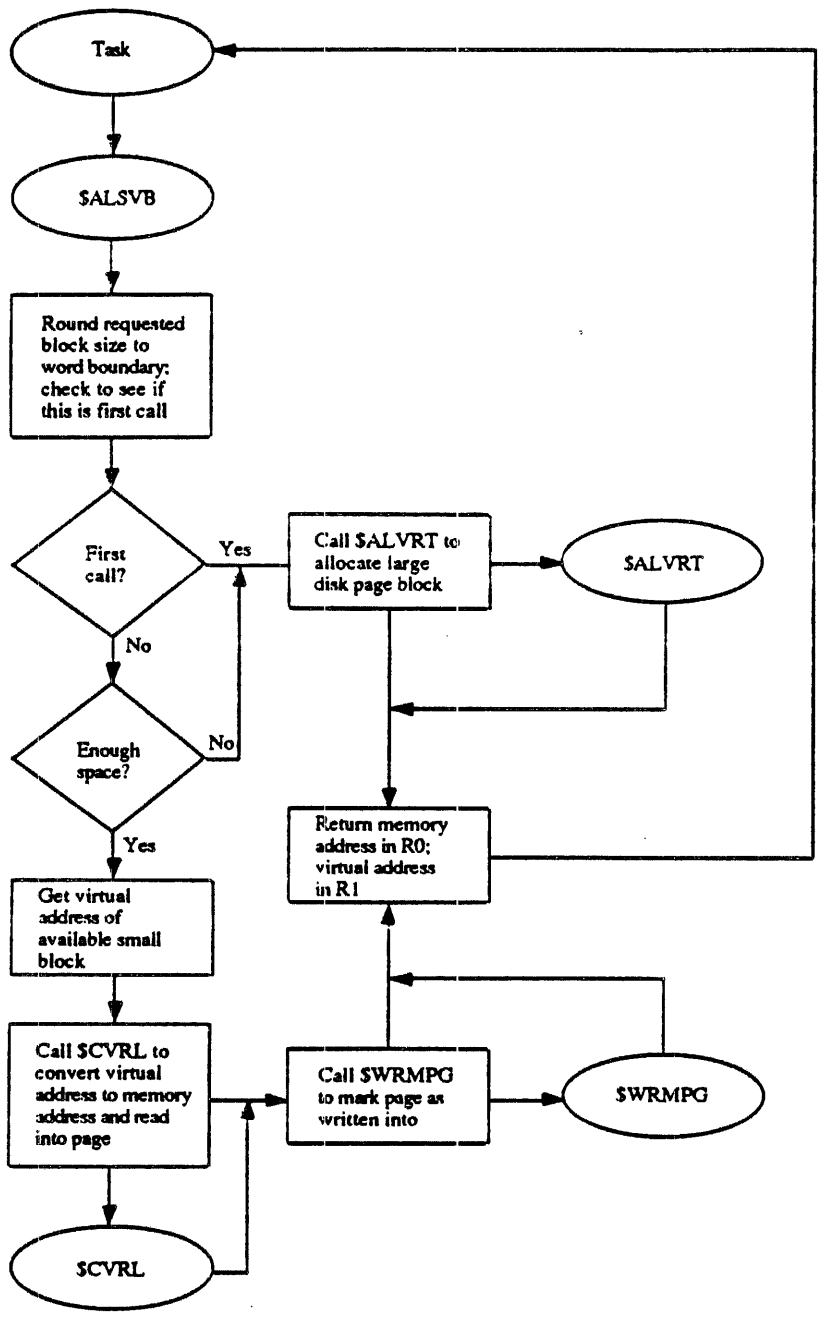
When a large page block is to be allocated, the Allocate Virtual Memory Routine ($ALVRT) is called to do the following:

- Allocate the disk and dynamic memory of the requested large page block
- Convert the virtual address to a memory address
- Transfer the large block, if necessary, from disk to dynamic memory
- Set the "written into" flag of the allocated page block

The $ALSVB routine destroys the contents of Register 2 and preserves the contents of Registers 3 through 5.

Figure 8–8 shows the interaction of the $ALSVB routine with other virtual memory management routines.

Figure 8–8 General Block Diagram of the $ALSVB Routine


Task
$ALSVB
Round requested block size to word boundary; check to see if this is first call
First call?
Yes
No
Call $ALVRT to allocate large disk page block
$ALVRT
Enough space?
No
Yes
Return memory address in R0; virtual address in R1
Get virtual address of available small block
Call $CVRL to convert virtual address to memory address and read into page
Call $WRMPG to mark page as written into
$WRMPG
$CVRL

EXAMPLE

The following source statements call the $ALSVB routine to allocate a block of memory within a larger block:

```
E$R4      ==     4                 ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73     ==     73                ; WORK FILE I/O ERROR
E$R76     ==     76                ; WORK FILE EXCEEDED
S$V2      ==     302               ; SEVERITY 2
N$DLGH    ==     512.              ; LARGE BLOCK SIZE
P$GSIZ    ==     24                ; SIZE OF CURRENT PAGE
          MOV    #P$GSIZ,R5        ; GET PAGE SIZE
          MOV    R5,-R1            ; COPY SIZE OF TABLE
          ASL    R1                ; CONVERT TO BYTES
          CALL   $ALSVB            ; ALLOCATE VIRTUAL MEMORY
          MOV    R1,(R4)+          ; SAVE VIRTUAL ADDRESS
```

$RQVCB—Request virtual core block routine

The $RQVCB routine manages page-block allocation on your disk work file. The $RQVCB routine is called by the Allocate Virtual Memory Routine ($ALVRT) when your task has requested allocation of a page block of a maximum of 512_{10} bytes in length.

The $RQVCB routine is not a user-called routine.

DESCRIPTION

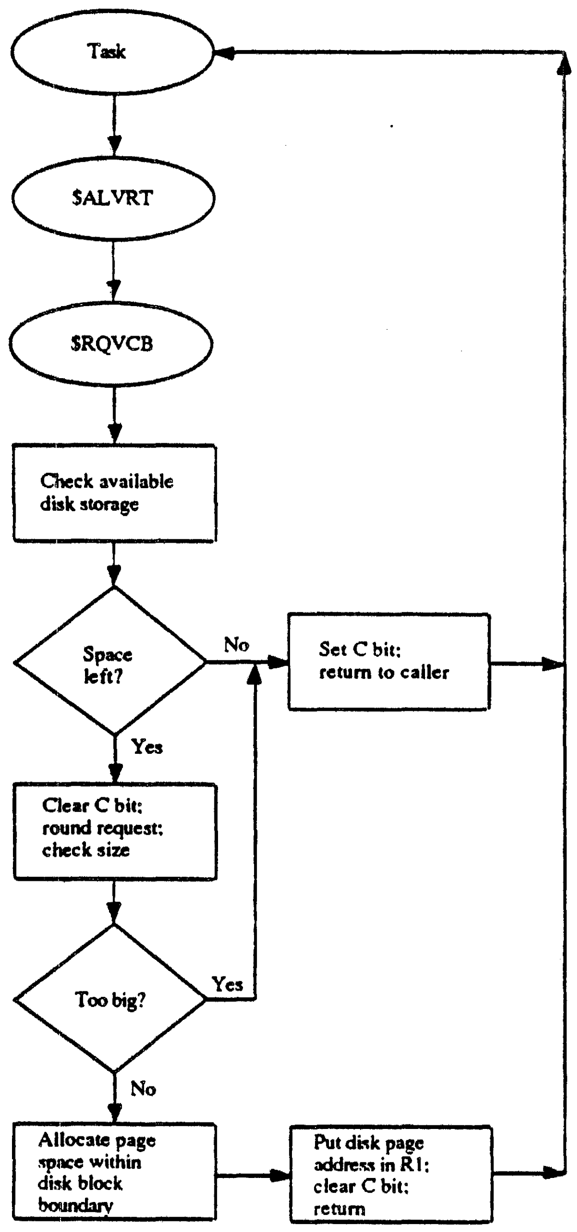
The $RQVCB routine rounds the requested number of bytes up to the nearest word. If the rounded value crosses a disk block boundary, the $RQVCB routine allocates the page block beginning at the next disk block.

If allocation is successful, the $RQVCB routine clears the C bit in the Condition Code and returns the disk address of the allocated page to the $ALVRT routine.

If allocation is not successful, the $RQVCB routine sets the C bit in the Condition Code and returns control to the $ALVRT routine. The following conditions can prevent allocation:

- There is no more disk storage space available.
- A page block size greater than 512_{10} bytes has been requested.

Figure 8–9 General Block Diagram of the $RQVCB Routine


Task
$ALVRT
$RQVCB
Check available disk storage
Space left?
No
Set C bit; return to caller
Yes
Clear C bit; round request; check size
Too big?
Yes
No
Allocate page space within disk block boundary
Put disk page address in R1; clear C bit; return

The page management routines perform the processing required to control page swapping between dynamic memory and disk file storage. This processing includes address conversion; page location; page transfer from disk to memory; and page status handling such as timestamping, flagging as "written into," and locking and unlocking memory pages.

The page management routines are as follows:

- The Convert and Lock Page Routine ($CVLOK), which converts a virtual address to a dynamic memory address and locks the page in memory when called by your task
- The Convert Virtual to Real Address Routine ($CVRL), which converts a virtual address to a dynamic memory address when called by one of the following:
  - User task
  - Allocate Virtual Memory Routine ($ALVRT) when a new disk page has been allocated
  - Convert and Lock Page Routine ($CVLOK) when a page address is to be converted and the page is to be locked in memory
- Read Page Routine ($RDPAG), which is called by the $CVRL routine to transfer a page from your disk work file to dynamic memory
- Find Page Routine ($FNDPG), which determines whether a virtual page is resident in dynamic memory when called by one of the following:
  - $CVRL routine
  - Lock Page Routine ($LCKPG)
  - Unlock Page Routine ($UNLPG)
  - Write-Marked Page Routine ($WRMPG)
- Write-Marked Page Routine ($WRMPG), which sets the "written into" flag of memory pages when called by a user or by the $ALVRT and $ALSVB virtual memory allocation routines
- Lock Page Routine ($LCKPG), which is called by the $CVLOK routine and a user task to set a lock byte in a memory page to prevent its being swapped from memory to the disk file
- Unlock Page Routine ($UNLPG), which is called by a user task to clear a lock byte in a memory page to allow it to be swapped to disk storage to free memory space for reallocation

$CVLOK—Convert and lock page routine

The $CVLOK routine performs the following functions:

- Converts a virtual address to a memory address
- Locks the page in memory

FORMAT

CALL $CVLOK

INPUT

virtual address
In Register 1: the virtual address you want to convert

OUTPUT

converted memory address
In Register 0

virtual address
In Register 1

Condition Code

C bit	=	Clear if the address was converted and the page locked
C bit	=	Set if address conversion or page locking failed

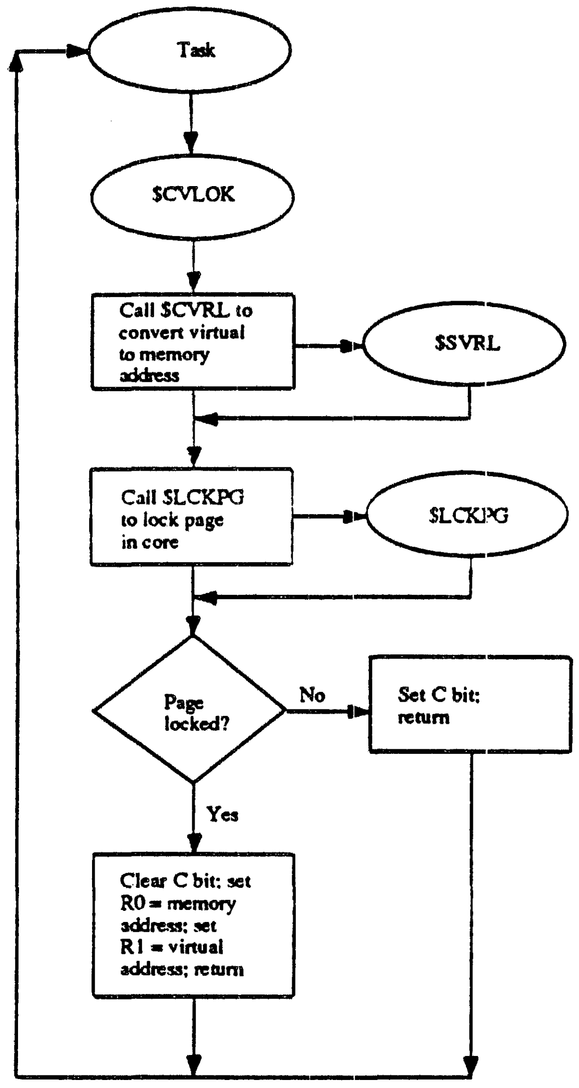
DESCRIPTION

The $CVLOK routine calls the following routines:

- The Convert Virtual to Real Address Routine ($CVRL) to convert the virtual address to a memory address
- $CVRL to preserve the contents of Registers 3 through 5
- The Lock Page Routine ($LCKPG) to lock the page in memory

$CVLOK also preserves the contents of Register 2.

Figure 8–10 General Block Diagram of the $CVLOK Routine


Task
$CVLOK
Call $CVRL to convert virtual to memory address
$SVRL
Call $LCKPG to lock page in core
$LCKPG
Page locked?
No
Set C bit; return
Yes
Clear C bit; set R0 = memory address; set R1 = virtual address; return

EXAMPLE

The following source statements call the $CVLOK routine to convert a virtual address from the listhead to a dynamic memory address in TEMP1 and then an error routine in case the conversion fails:

```
E$R4     ==      4              ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73    ==      73             ; WORK FILE I/O ERROR
E$R76    ==      76             ; WORK FILE EXCEEDED
S$V2     ==      302            ; SEVERITY 2
TEMP1:   .WORD   0              ; TEMPORARY STORAGE FOR VIRTUAL MEMORY
LISTHD:  .BLKW   1              ; LISTHEAD LOCATION

         MOV     LISTHD,R1      ; MOVE VIRTUAL ADDRESS
         CALL    $CVLOK         ; CONVERT, STORE REAL ADDRESS IN REGISTER 0
         BCS     LCKERR         ; ERROR
         MOV     R0,TEMP1       ; SAVE IN TEMPORARY BUFFER

LCKERR:  MOV     #ERR55,R0      ; GET ERROR MESSAGE
         BR      ERROR          ; GET ERROR ROUTINE
```

$CVRL—Convert virtual to real address routine

The $CVRL routine converts a virtual address to a dynamic memory address. Virtual address units are words and dynamic memory addresses are bytes.

FORMAT

CALL $CVRL

INPUT

virtual address
In Register 1: the virtual address you want to convert

OUTPUT

memory address
In Register 0: the converted memory address

DESCRIPTION

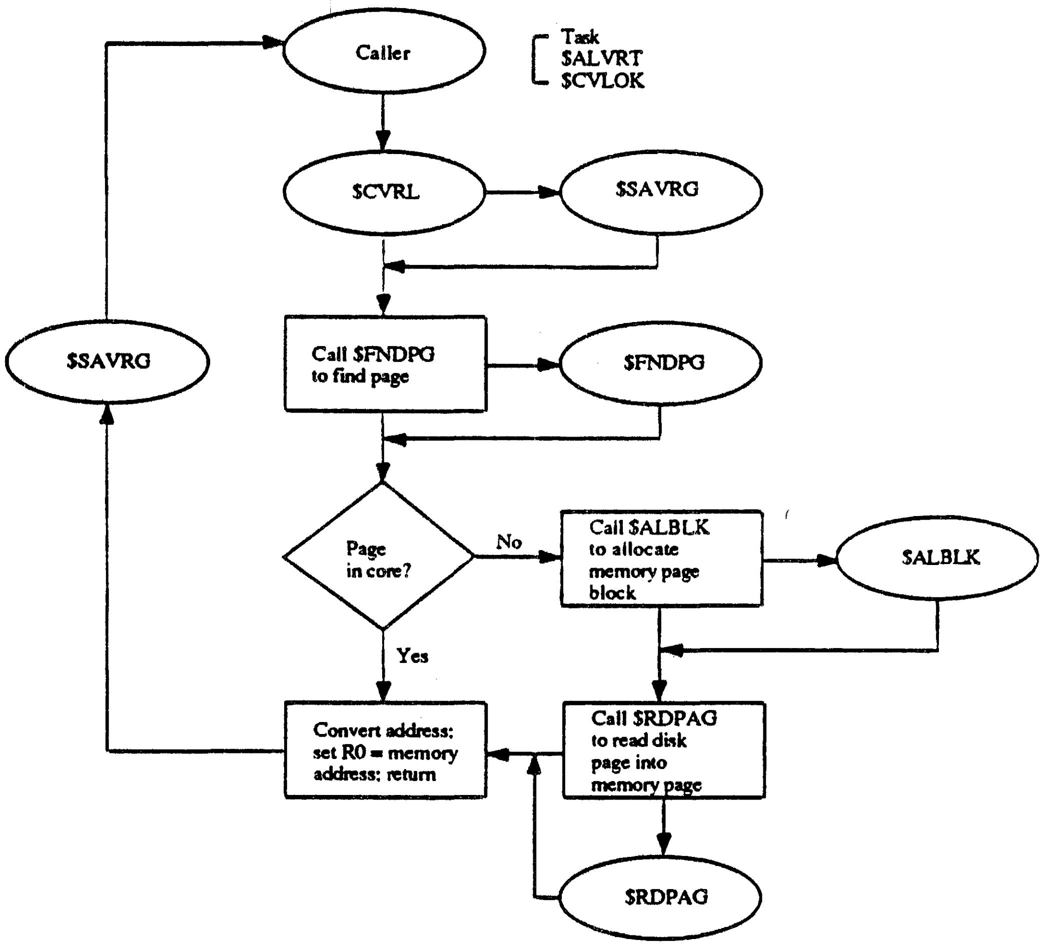
The $CVRL routine can be called directly in the task or indirectly by the following routines:

- Allocate Virtual Memory Routine ($ALVRT) when a new disk page has been allocated
- Convert and Lock Page Routine ($CVLOK) when the executing task has specified that a virtual address is to be converted to a memory address and the page is to be locked in memory

The $CVRL routine calls the Find Page Routine ($FNDPG) to determine whether the specified page is resident in memory. If so, the virtual address is converted to a memory address, which is returned to the caller. If the page is not in memory, $CVRL calls the Allocate Block Routine ($ALBLK) to allocate a memory page block. The $CVRL routine then calls the Read Page Routine ($RDPAG) to transfer the disk page into dynamic memory. The page address is then converted to a memory address. The memory address of the specified word in the page is stored in Register 0, and control is transferred to the $SAVRG routine, which restores Registers 3 through 5 and returns to the caller.

The $CVRL routine leaves Register 1 unchanged. It destroys the contents of Register 2.

Figure 8–11 General Block Diagram of the $CVRL Routine


Caller
Task
$ALVRT
$CVLOK
$CVRL
$SAVRG
Call $FNDPG to find page
$FNDPG
$SAVRG
Page in core?
No
Yes
Call $ALBLK to allocate memory page block
$ALBLK
Convert address; set R0 = memory address; return
Call $RDPAG to read disk page into memory page
$RDPAG

EXAMPLE

The following source statements call the $CVRL routine to convert a virtual address in Register 1 to a dynamic memory address and store the result in Register 0:

```
E$R4     ==     4             ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73    ==     73            ; WORK FILE I/O ERROR
E$R76    ==     76            ; WORK FILE EXCEEDED
S$V2     ==     302           ; SEVERITY 2
P$GSIZ   ==     24            ; SIZE OF CURRENT PAGE
P$GADR:  .BLKW  1             ; ADDRESS OF CURRENT PAGE

         MOV    #P$GADR,R1    ; GET PAGE ADDRESS
         MOV    R1,R5         ; SAVE VIRTUAL ADDRESS
         TST    R1            ; IS REQUEST ON BLOCK BOUNDARY?
         BNE    20$           ; IF NO, BLOCK ALREADY EXISTS
         MOV    #P$GSIZ,R1    ; CREATE A PAGE BUFFER
         CALL   $ALBLK        ; ALLOCATE STORAGE SPACE
         MOV    R5,R1         ; RESTORE VIRTUAL ADDRESS
20$:     CALL   $CVRL         ; CONVERT TO REAL ADDRESS
```

$RDPAG—Read page routine

The $RDPAG routine transfers a disk page from the work file to the dynamic memory.

FORMAT

CALL $RDPAG

INPUT

page address

In Register 0: the disk address of the page you want to transfer

OUTPUT

Condition Code

C bit	=	Clear if transfer succeeds
C bit	=	Set if transfer fails

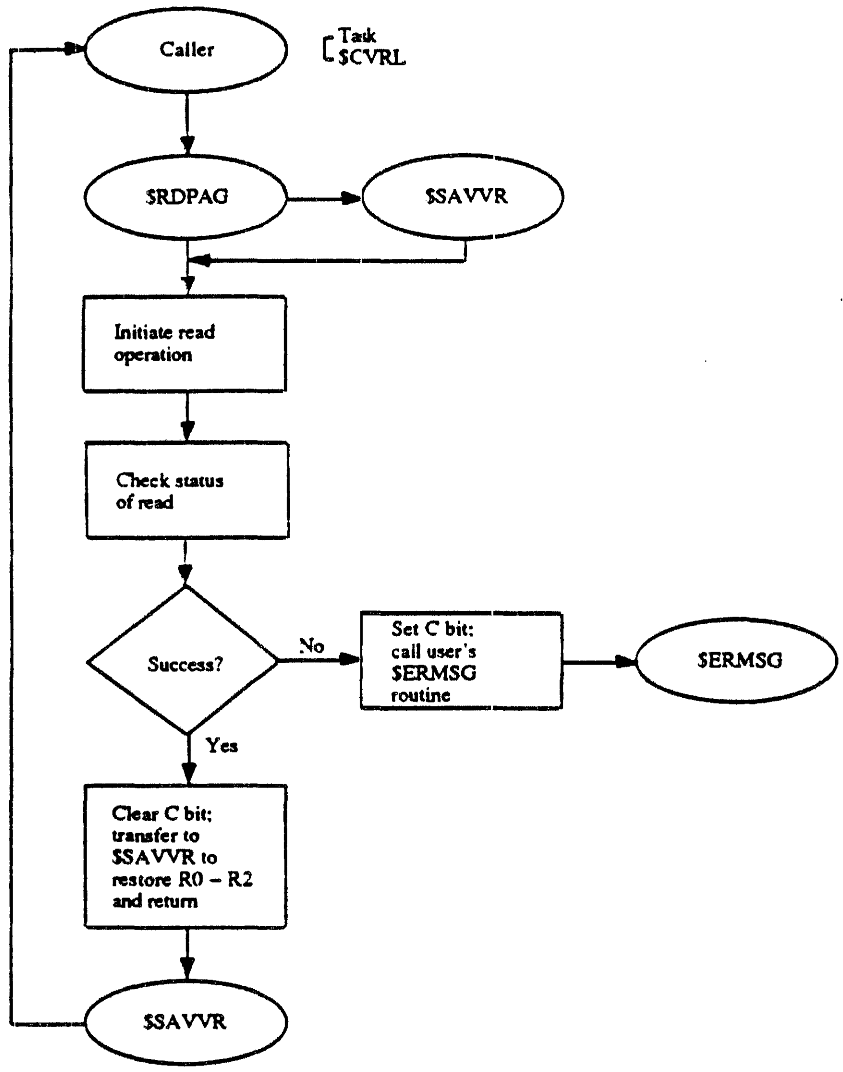
DESCRIPTION

The $RDPAG routine is called by the Convert Virtual to Real Address Routine ($CVRL) when a disk page is to be transferred to dynamic memory. The $RDPAG routine then does the following:

- Sets up the address of the page to be transferred
- Initiates the page-reading operation
- Checks the status of the read operation
- Calls the $SAVVR routine to save and subsequently restore the caller's Registers 0 through 2

The interaction of the $RDPAG routine with the task and the $CVRL routine is shown in Figure 8–12.

Figure 8–12 General Block Diagram of the $RDPAG Routine


Caller
Task
$CVRL
$RDPAG
$SAVVR
Initiate read operation
Check status of read
Success?
No
Set C bit: call user's $ERMSG routine
$ERMSG
Yes
Clear C bit; transfer to $SAVVR to restore R0 – R2 and return
$SAVVR

EXAMPLE

The following source statements allocate a page in buffer P$GSIZ and call the $RDPAG routine to read the virtual page address into core memory:

```
E$R4     ==      4                ; INSUFFICIENT WORK FILE DYNAMIC MEMORY
E$R73    ==      73               ; WORK FILE I/O ERROR
E$R76    ==      76               ; WORK FILE EXCEEDED
S$V2     ==      302              ; SEVERITY 2
P$GSIZ   ==       24              ; SIZE OF PAGE
P$GBLK: .BLKW    100.             ; RELATIVE BLOCK NUMBER
LISTHD: .BLKW    1.               ; LISTHEAD LOCATION
PAGLS:  .BLKW    1.               ; ADDRESS OF PAGE LIST

$CVRT:  SAVRG                     ; SAVE NONVOLATILE REGISTERS
        MOV      R1,R5            ; COPY VIRTUAL ADDRESS
        SWAB     R5               ; POSITION BLOCK NUMBER TO LOW BYTE
        CALL     $FNDPG           ; SEARCH FOR PAGE
        BCC      10$              ; IF C BIT CLEAR, PAGE IN CORE.
        MOV      #P$GSIZ,R1       ; GET SIZE OF PAGE BUFFER
        CALL     $ALBLK           ; ALLOCATE MEMORY
        MOV      PAGLS,R4         ; GET ADDRESS OF PAGE LIST
        BEQ      5$               ; IF EQ NONE
        CLR      R2               ; SET FOR MOVB WITH NO EXTEND
        BISB     R5,R2            ; GET RELATIVE BLOCK NUMBER
        ASL      R2               ; CONVERT TO WORD OFFSET
        ADD      R2,R4            ; COMPUTE LIST ADDRESS
        MOV      R0,(R4)          ; STORE ADDRESS OF PAGE
5$:                               ;
        MOVB     R5,P$GBLK(R0)    ; SET RELATIVE BLOCK NUMBER
        CALL     $RDPAG           ; READ PAGE INTO CORE
    .
    .
    .
```

$FNDPG—Find page routine

The $FNDPG routine searches an internal page address list to determine whether a virtual page has already been transferred into an allocated memory page block.

FORMAT

CALL $FNDPG

INPUT

virtual page address

In Register 1: the address of the page being searched for

OUTPUT

block address

In Register 0: the memory page block address where the page is resident

Condition Code

C bit	=	Clear if page is resident
C bit	=	Set if page was not found

DESCRIPTION

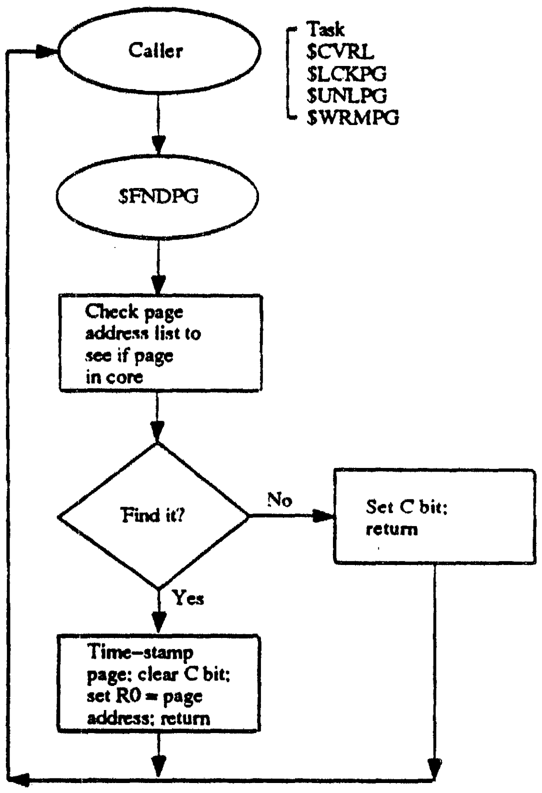
The $FNDPG routine is called by the following virtual memory management routines:

- Convert Virtual to Real Address Routine ($CVRL) when a virtual address is to be converted to a memory address
- Lock Page Routine ($LCKPG) when a memory page is to be locked in core memory
- Unlock Page Routine ($UNLPG) when a locked memory page is to be unlocked
- Write-Marked Page Routine ($WRMPG) when the "written into" flag is to be set in a memory page

The $FNDPG routine determines whether the specified page is resident in the task's dynamic memory. If so, the page is timestamped, its page block address is set in Register 0, the C bit in the Condition Code is cleared, and control returns to the caller. If the page is not resident in memory, the $FNDPG routine sets the C bit in the Condition Code and returns control to the caller. $FNDPG does not change the contents of Register 1.

The interaction of the $FNDPG routine with a user task and the page management routines is shown in Figure 8–13.

Figure 8–13 General Block Diagram of the $FNDPG Routine


Caller
Task
$CVRL
$LCKPG
$UNLPG
$WRMPG
$FNDPG
Check page address list to see if page in core
Find it?
No
Set C bit; return
Yes
Time–stamp page; clear C bit; set R0 = page address; return

EXAMPLE

The following source statements call the $FNDPG routine to verify that a page address, stored in buffer P$GADR, exists in core memory. The example then calls $ALBLK to allocate the page block:

```
P$GADR: .WORD    0               ; VIRTUAL PAGE ADDRESS
P$GSIZ  ==       24              ; SIZE OF PAGE

        CALL     $SAVRG          ; SAVE NONVOLATILE REGISTERS
        MOV      P$GADR,R1       ; GET PAGE ADDRESS
        CALL     $FNDPG          ; SEARCH FOR PAGE
        BCC      10$             ; IF CLEAR, PAGE IN CORE
        MOV      #P$GSIZ,R1      ; GET SIZE OF PAGE BUFFER
        CALL     $ALBLK          ; ALLOCATE MEMORY
                 .
10$:             .
                 .
```

$WRMPG—Write-marked page routine

The $WRMPG routine sets the "written into" flag of the specified page in dynamic memory.

FORMAT

CALL $WRMPG

INPUT

virtual page address
In Register 1: the address of the page for which the flag is being set

OUTPUT

Condition Code

C bit	=	Clear if the page was write-marked successfully
C bit	=	Set if the specified memory page was not resident in the task's free dynamic memory

DESCRIPTION

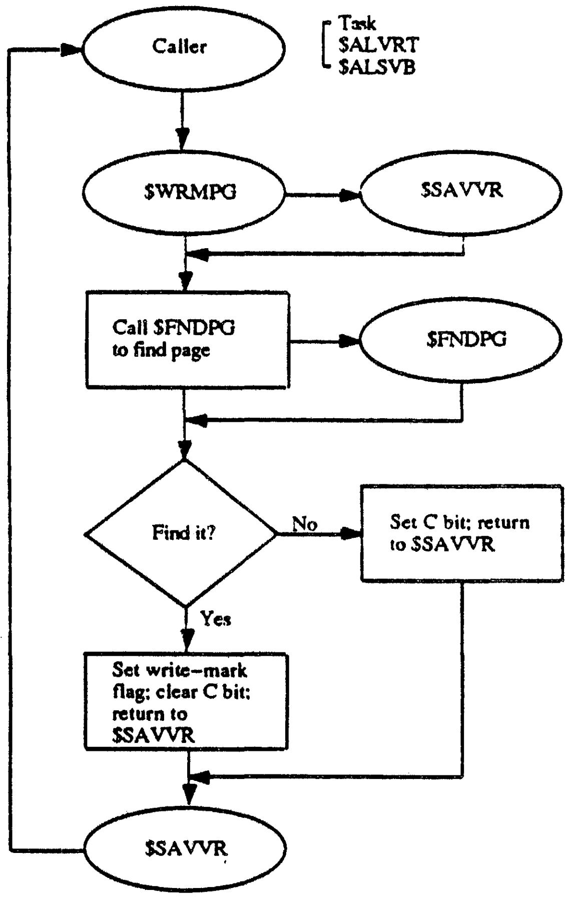
The $WRMPG routine is called by the following virtual memory management routines:

- Allocate Virtual Memory Routine ($ALVRT) when a disk page has been allocated in dynamic memory
- Allocate Small Virtual Block Routine ($ALSVB) when a small page block has been allocated within a large page block

$WRMPG calls the Find Page Routine ($FNDPG) to determine whether the specified page is resident in the task's memory. If not, the C bit in the Condition Code is set and control is transferred to the $SAVVR routine to restore Registers 0 through 2 and return to the caller. If the page is resident in memory, its "written into" flag is set, the C bit in the Condition Code is cleared, and control is transferred to the $SAVVR routine to restore Registers 0 through 2 and return to the caller.

The interaction of the $WRMPG routine with the caller and virtual memory management routines is shown in Figure 8–14.

Figure 8–14 General Block Diagram of the $WRMPG Routine



EXAMPLE

The following source statements call the $WRMPG routine to mark a page and then call an error routine in case $WRMPG is not successful:

```
TEMP1:  .WORD   0       ; TEMPORARY STORAGE FOR VIRTUAL MEMORY
FREECT: .BLKW   1       ; NUMBER OF AVAILABLE PAGE ENTRIES
ER58:   .ASCIZ   <15>/ACNT--Work file - page mark /
        .EVEN

        MOV     TEMP1,R1        ; SET $WRMPG ARGUMENT
        MOV     R5,TEMP1        ; MOVE PREV PAGE ADDRESS TO VIRTUAL MEMORY
        MOV     @R0,@R3         ; UPDATE PREV VIRTUAL ADDRESS PAGE POINTER
        INC     FREECT          ; INCREMENT NUMBER OF PAGES AVAILABLE
        CALL    $WRMPG          ; MARK PAGE "WRITTEN INTO"
        BCS     WRMERR          ; ERROR
    .
    .
    .
WRMERR: MOV     #ER58,R0        ; GET ERROR MESSAGE
        BR      ERROR           ; GET ERROR ROUTINE
```

$LCKPG—Lock page routine

The $LCKPG routine sets a lock byte in a memory-resident page to prevent its being swapped from dynamic memory to the disk work file.

FORMAT

CALL $LCKPG

INPUT

virtual page address
In Register 1: a virtual address in the page to be locked in dynamic memory

OUTPUT

Condition Code

C bit	=	Clear if the page was locked in memory
C bit	=	Set if the page was not found

DESCRIPTION

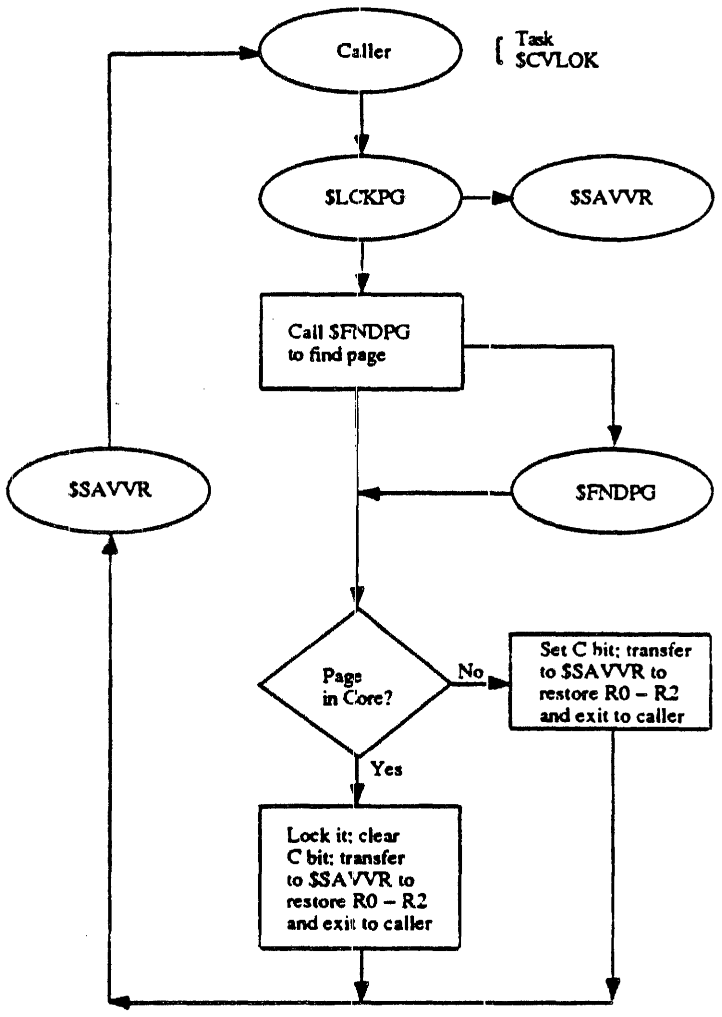
The $LCKPG routine can be called by a user task or by the Convert and Lock Page Routine ($CVLOK).

$LCKPG calls the Find Page Routine ($FNDPG) to determine whether the memory page is resident. If so, the page lock byte is set, the C bit in the Condition Code is cleared, and control is transferred to the $SAVVR routine to restore Registers 0 through 2 and return to the caller.

If the specified page is not in memory, the $LCKPG routine sets the C bit in the Condition Code and returns control, by means of the $SAVVR routine, to the caller.

The interaction of the $LCKPG routine with the task and page management routines is shown in Figure 8–15.

Figure 8–15 General Block Diagram of the $LCKPG Routine


Caller
Task
$CVLOK
$LCKPG
$SAVVR
Call $FNDPG to find page
$SAVVR
$FNDPG
Page in Core?
No
Set C bit; transfer to $SAVVR to restore R0 – R2 and exit to caller
Yes
Lock it; clear C bit; transfer to $SAVVR to restore R0 – R2 and exit to caller

EXAMPLE

The following source statements call the $LCKPG routine to lock a page in dynamic memory if the listhead contains more than one element:

```
TEMP1:  .WORD   0        ; TEMPORARY STORAGE FOR VIRTUAL MEMORY
LISTHD: .BLKW   1        ; LISTHEAD LOCATION
ER55:   .ASCIZ  <15>/ACNT --Work file - page lock /
        .EVEN

        MOV     LISTHD,R1         ; MOVE 1ST VIRTUAL ADDRESS
        CALL    $CVLOK            ; 1ST PAGE REAL ADDRESS IN REGISTER 0
        BCS     LCKERR            ; ERROR
        TST     (R0)              ; ONLY 1 ELEMENT?
        BNE     40$               ; NO, MORE THAN ONE
        CALL    $UNLPG            ; YES, ONLY ONE, UNLOCK IT

40$:    MOV     TEMP1,R1          ; SET UP VIRTUAL ADDRESS FOR $LCKPG
        CALL    $CVRL             ; SAVE REAL ADDRESS OF NEXT PAGE IN REGISTER 0
        CALL    $LCKPG            ; LOCK
        BCS     LCKERR            ; ERROR
   .
   .
   .
LCKERR: MOV     #ERR55,R0         ; GET ERROR MESSAGE
        BR      ERROR             ; ERROR ROUTINE
```

$UNLPG—Unlock page routine

The $UNLPG routine clears a lock byte in a memory-resident page to allow the page to be swapped from dynamic memory to the disk work file.

FORMAT

CALL $UNLPG

INPUT

virtual page address

In Register 1: the virtual address of the page you want to unlock

OUTPUT

Condition Code

C bit	=	Clear if the page was unlocked
C bit	=	Set if the page was not found

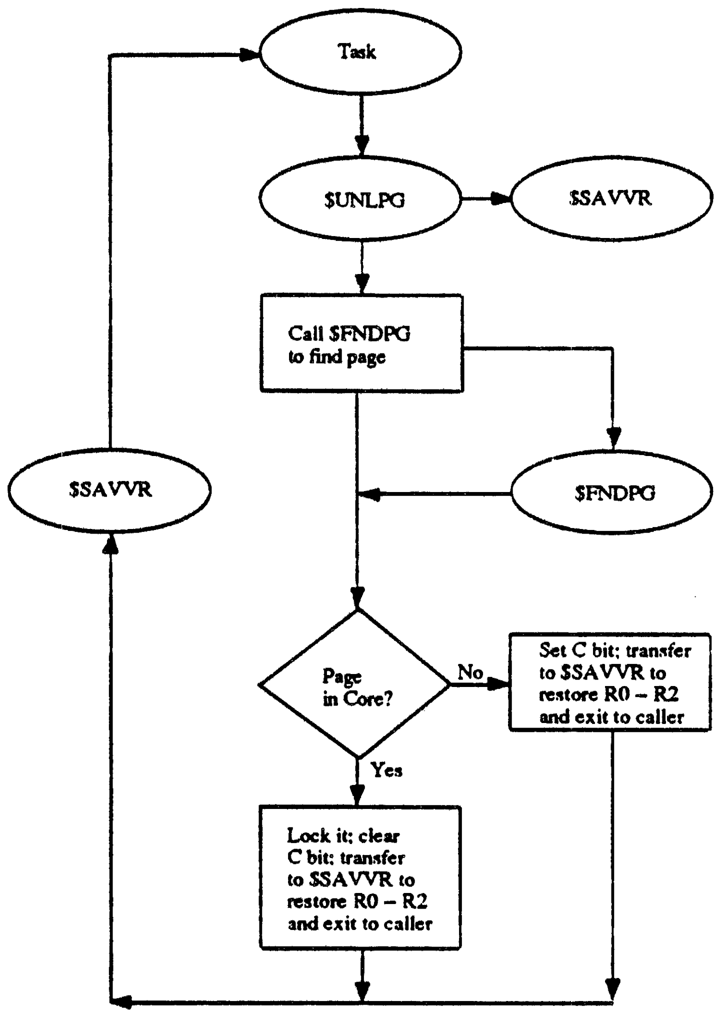
DESCRIPTION

$UNLPG calls the Find Page Routine ($FNDPG) to determine whether the memory page is resident. If so, the page lock byte and the C bit in the Condition Code are cleared and control is transferred to the $SAVVR routine to restore Registers 0 through 2 and return to the caller.

If the specified page is not in memory, the C bit in the Condition Code is set and control is returned, by means of the $SAVVR routine, to the caller.

The interaction of the $UNLPG routine with the task is shown in Figure 8–16.

Figure 8–16 General Block Diagram of the $UNLPG Routine


Task
$UNLPG
$SAVVR
Call $FNDPG to find page
$SAVVR
$FNDPG
Page in Core?
No
Set C bit; transfer to $SAVVR to restore R0 – R2 and exit to caller
Yes
Lock it; clear C bit; transfer to $SAVVR to restore R0 – R2 and exit to caller

EXAMPLE

The following source statements call the $UNLPG routine to allow pages to be swapped from real memory to virtual memory:

```
TEMP1:   .WORD    0        ; TEMPORARY STORAGE FOR VIRTUAL MEMORY
TEMP2:   .WORD    0        ; TEMPORARY STORAGE FOR VIRTUAL MEMORY
FREECT:  .BLKW    1        ; NUMBER OF AVAILABLE PAGE ENTRIES
LISTHD:  .BLKW    1        ; LISTHEAD LOCATION
ER56:    .ASCIZ   <15>/ACNT --Work file - page unlock /
         .EVEN

10$:     MOV      #LISTHD,TEMP2    ; GET FIRST REAL ADDRESS POINTER
         MOV      LISTHD,R1        ; MOVE FIRST VIRTUAL ADDRESS
         MOV      R1,TEMP1         ; SAVE IN SECOND VIRTUAL ADDRESS BUFFER
         CLR      FREECT           ; CLEAR NUMBER OF SWAPS PER PASS
         CALL     CVLOK            ; PUT REAL ADDRESS IN REGISTER 0
         BCS      LCKERR           ; ERROR, PAGE LOCK FAILED
         TST      (R0)             ; LINK = 0, ONLY ONE ELEMENT?
         BNE      20$              ; NO, MORE THAN ONE
         CALL     $UNLPG           ; YES, ONLY ONE, UNLOCK IT
         BCS      UNLERR           ; ERROR
            .
20$:        .
            .
```

9 Summary Procedures

The procedures for using the system library routines are summarized in the tables in this chapter. These summaries are presented as quick reference guides for users who are familiar with the detailed procedures and requirements for using individual routines, as described in the preceding chapters of this manual.

Table 9–1 Register Handling Routines Summary

Routine Name/ Mnemonic	Function	Call Statement
Save All Registers $SAVAL	Saves/restores R0—R5	CALL $SAVAL
Save Registers 3—5 $SAVRG	Saves/restores R3—R5	JSR R5,$SAVRG
Save Registers 0—2 $SAVVR	Saves/restores R0—R2	JSR R2,$SAVVR
Save Registers 1—5 .SAVR1	Saves/restores R1—R5	JSR R5,.SAVR1

Table 9–2 Arithmetic Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output

Integer Multiply $MUL	R0 = Multiplier
	R1 = Multiplicand
	CALL $MUL
		R0 = Product (high-order part)
		R1 = Product (low-order part)
		R2—R5 preserved

Integer Divide $DIV	R0 = Dividend
	R1 = Divisor
	CALL $DIV
		R0 = Quotient
		R1 = Remainder
		R2—R5 preserved

Double-Precision Multiply $DMUL	R0 = Multiplier
	Multiplicand:
	R2 = High-order part
	R3 = Low-order part
	CALL $DMUL
		R0 = Product (high-order part)
		R1 = Product (low-order part)
		R4—R5 preserved
		R2—R3 destroyed
		C = Clear

Double-Precision Divide $DDIV	R0 = Unsigned divisor
	Dividend:
	R1 = High-order part
	R2 = Low-order part
	CALL $DDIV
		R0 = Remainder
		R1 = Quotient (high-order part)
		R2 = quotient (low-order part)
		R3 preserved

Table 9–3 Input Data Conversion Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output

Decimal to Binary Double Word .DD2CT	R3 = Output address
	R4 = Number input characters
	R5 = Input string address
	CALL .DD2CT
		Successful:
		Converted number at output address:
		Word 1 = High-order part
		Word 2 = Low order part
		C = Clear
		Unsuccessful:
		C = Set
		All registers preserved

Octal to Binary Double Word .OD2CT	R3 = Output address
	R4 = Number input characters
	R5 = Input string address
	CALL .OD2CT
		Successful:
		Converted number at output address:
		Word 1 = High-order part
		Word 2 = Low-order part
		C = Clear
		Unsuccessful:
		C = Set
		All registers preserved

Decimal to Binary $CDTB	R0 = Address first input byte
	CALL $CDTB
		R0 = Address first byte of next string
		R1 = Converted number
		R2 = Terminating character
		R3—R5 preserved

Octal to Binary $COTB	R0 = Address first input byte
	CALL $COTB
		R0 = Address first byte of next string
		R1 = Converted number
		R2 = Terminating character
		R3—R5 preserved

ASCII to Radix-50 $CAT5	R0 = Address first input character
	R1 = 0 (period is terminating character)
	R1 = 1 (period is valid character)
	CALL $CAT5
		Successful:
		R0 = Address next input character
		R1 = Converted Radix-50 value
		R2 = Terminating character
		C = Clear
		Unsuccessful:
		R2 = Illegal character
		C = Set
		R3—R5 preserved

ASCII with Blanks to Radix-50 $CAT5B	R0 = Address first input character
	R1 = 0 (period is terminating character)
	R1 = 1 (period is valid character)
	CALL $CAT5B
		Successful:
		R0 = Address next input character
		R1 = Converted Radix-50 value
		R2 = Terminating character
		C = Clear
		Unsuccessful:
		R2 = Illegal character
		C = Set
		R3—R5 preserved

Table 9–4 Output Data Conversion Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output

Binary Date Conversion $CBDAT	R0 = Output address
	R1 = Binary date
	R2 = 0 (zero suppress)
	R2 = Nonzero (no zero suppress)
	CALL $CBDAT
		Converted date at output address
		R0 = Next available output address
		R3—R5 preserved
		R1—R2 destroyed

Convert Binary to Decimal Magnitude $CBDMG	R0 = Output address
	R1 = Binary number
	R2 = 0 (zero suppress)
	R2 = Nonzero (no zero suppress)
	CALL $CBDMG
		Converted number at output address
		R0 = Next available output address
		R3—R5 preserved
		R1—R2 destroyed

Convert Binary to Signed Decimal $CBDSG	R0 = Output address
	R1 = Binary number
	R2 = 0 (zero suppress)
	R2 = Nonzero (no zero suppress)
	CALL $CBDSG
		Converted number at output address
		R0 = Next available output address
		R3—R5 preserved
		R1—R2 destroyed

Convert Double-Precision Binary to Decimal $CDDMG	R0 = Output address
	R1 = Input address
	R2 = 0 (zero suppress)
	R2 = Nonzero (no zero suppress)
	CALL $CDDMG
		Successful:
		Converted number at output address
		Unsuccessful:
		String of ASCII asterisks at output address
		R0 = Next available output address
		R3—R5 preserved
		R1—R2 destroyed

Convert Binary to Octal Magnitude $CBOMG	R0 = Output address
	R1 = Binary number
	R2 = 0 (zero suppress)
	R2 = Nonzero (no zero suppress)
	CALL $CBOMG
		Converted number at output address
		R0 = Next available output address
		R3—R5 preserved
		R1—R2 destroyed

Convert Binary to Signed Octal $CBOSG	R0 = Output address
	R1 = Binary number
	R2 = 0 (zero suppress)
	R2 = Nonzero (no zero suppress)
	CALL $CBOSG
		Converted number at output address
		R0 = Next available output address
		R3—R5 preserved
		R1—R2 destroyed

Convert Binary Byte to Octal Magnitude $CBTMG	R0 = Output address
	R1 = Binary byte
	R2 = 0 (zero suppress)
	R2 = Nonzero (no zero suppress)
	CALL $CBTMG
		Converted byte at output address
		R0 = Next available output address
		R3—R5 preserved
		R1—R2 destroyed

Table 9–4 (Cont.) Output Data Conversion Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output
General Purpose Binary to ASCII $CBTA	R0 = Output address R1 = Binary value R2 = Conversion parameters: Bits 0—7: = Radix (2 to 16_{10}) Bit 8: = 0 = Unsigned value = 1 = Signed value Bit 9: = 0 = Zero suppress = 1 = No zero suppress Bit 10: = 1, replace leading zeros with blanks = 0, do not replace leading zeros with blanks Bits 11—15: = Field width (value 1—32) CALL $CBTA	Converted number at output address R0 = Next available output address R3—R5 preserved R1—R2 destroyed

Radix-50 to ASCII $C5TA	R0 = Output address
	R1 = Radix-50 word
	CALL $C5TA
		Converted number at output address
		R0 = Next available output address
		R3—R5 not used
		R1—R2 destroyed

Table 9–5 Output Formatting Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output

Uppercase Text $CVTUC	R0 = Input address
	R1 = Output address
	R2 = Number input bytes (cannot be zero)
	CALL $CVTUC
		Converted text at output address
		R3—R5 not used
		R2 destroyed
		R0—R1 left pointing to the character following the string

Date String Conversion $DAT	R0 = Output address
	R1 = Input address
	CALL $DAT
		Converted date string at output address
		R0 = Next available output address
		R1 = Address of next input word
		R3—R5 preserved
		R2 destroyed

Table 9–5 (Cont.) Output Formatting Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output
Time Conversion $TIM	R0 = Output address R1 = Input address R2 = Parameter count: = 0 or 1, hour (HH) = 2, hour:minute (HH:MM) = 3, hour:minute:second (HH:MM:SS) = 4 or 5, hour:minute:second. tenth of second (HH:MM:SS.S) CALL $TIM	Converted time string at output address R0 = Next available output address R1 = Address of next input word R3—R5 preserved R0—R1 updated R2 destroyed
Edit Message $EDMSG	Define ASCIZ input string directives in the form: %I %nI %VI where n = Optional decimal repeat count; V specifies an optional value to be used as a repeat count; and I = One of the following characters:	Converted/formatted data in output block R0 = Address of last zero byte in output block R1 = Number of bytes in output block R2 = Address of next argument in argument block R3—R5 preserved

Table 9–5 (Cont.) Output Formatting Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output

	A = ASCII string transfer
	B = Binary byte to octal conversion
	D = Binary to signed decimal conversion
	E = Extended ASCII string transfer
	F = Form control insertion
	I = ASCIZ address
	M = Binary to decimal magnitude conversion, zero suppression
	N = New line insertion
	O = Binary to signed octal conversion
	P = Binary to octal magnitude conversion, no zero suppression
	Q = Binary to octal magnitude conversion, zero suppression
	R = Radix-50 to ASCII conversion
	S = Space insertion
	T = Double-precision binary to decimal conversion
	U = Binary to double-precision decimal conversion, no zero suppression
	X = Filename conversion
	Y = Date conversion
	Z = Time conversion
	< = Define fixed-length byte field
	> = Locate field mark
	Set up argument and output block:
	R0 = Output address
	R1 = Input string address
	R2 = Argument block address
	CALL $EDMSG	

Table 9–6 Dynamic Memory Management Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output

Initialize Dynamic Memory $INIDM	Include FREEHD: .BLKW 2 in data section
	R0 = Free memory listhead address
	CALL $INIDM
		R0 = Task's first address
		R1 = Free pool first address
		R2 = Size memory pool
		R3—R5 not used

Request Core Block $RQCB	R0 = Free memory listhead address R1 = Byte size of block CALL $RQCB	Successful: R0 = Block memory address R1 = Actual size of block C = Clear Unsuccessful: C = Set R3—R5 preserved R2 destroyed
Release Core Block $RLCB	R0 = Free memory listhead address R1 = Byte size of block R2 = Block memory address	Released block R3—R5 preserved R0 unchanged R1—R2 destroyed

Table 9–7 Virtual Memory Management Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output

Initialize Virtual Memory $INIVM	Define $FRHD block with first address of free memory
	Define 4 global symbols:
	W$KLUN (work file LUN)
	W$KEXT (work file extension size)
	N$MPAG (fast page search page count)
	$WRKPT (address of FDB)
	R1 = Free memory highest address
	CALL $INIVM
		Successful:
		R0 = 0
		C = Clear
		Unsuccessful:
		R0 = –2, file not opened
		R0 = –1, file not marked
		C = Set
		R3— R5 preserved
		Original contents R0—R2 destroyed

Allocate Block $ALBLK	R1 = Byte size of requested block CALL $ALBLK	Successful: R0 = Block memory address Unsuccessful: User's $ERMSG routine is called R3—R5 preserved R0—R2 destroyed

Get Core $GTCOR	R1 = Byte size of requested block
	CALL $GTCOR
		Successful:
		R0 = Block memory address
		C = Clear
		Unsuccessful:
		C = Set
		R3—R5 preserved

Extend Task $EXTSK	R1 = Byte size of requested block
	CALL $EXTSK
		Successful:
		R1 = Actual extension size
		C = Clear
		Unsuccessful:
		C = Set
		R2—R5 preserved

Write Page $WRPAG	R2 = Memory address of page CALL $WRPAG	Successful: C = Clear Unsuccessful: User's $ERMSG routine is called R0—R2 preserved
Allocate Virtual Memory $ALVRT	R1 = Byte size of requested block CALL $ALVRT	Successful: R0 = Allocated block memory address R1 = Allocated block disk address Unsuccessful: User's $ERMSG routine is called R3—R5 preserved R2 destroyed

Table 9–7 (Cont.) Virtual Memory Management Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Output

Allocate Small Virtual Block $ALSVB	Define N$DLGH = = 512_{10}
	R1 = Size of requested page block:
	= 0, for large block allocation on first call to $ALSVB
	= A value less than or equal to 512_{10} bytes for small page allocation
	CALL $ALSVB
		R0 = Block memory address
		R1 = Block virtual address
		R3—R5 preserved
		R2 destroyed

Convert and Lock Page $CVLOK	R1 = Virtual address
	CALL $CVLOK
		Successful:
		R0 = Memory address
		R1 = Virtual address
		C = Clear
		Unsuccessful:
		C = Set
		R2—R5 preserved

Convert Virtual to Real Address $CVRL	R1 = Virtual address CALL $CVRL	R0 = Memory address R3—R5 preserved R1 unchanged R2 destroyed
Read Page $RDPAG	R0 = Page disk address CALL $RDPAG	Successful: C = Clear Unsuccessful: User's $ERMSG routine is called R0—R2 preserved

Find Page $FNDPG	R1 = Page virtual address
	CALL $FNDPG
		Page found:
		R0 = Block memory address
		C = Clear
		Page not found:
		C = Set

Write-Marked Page $WRMPG	R1 = Virtual address in page CALL $WRMPG	C = Clear, page write-marked C = Set, page not found R0—R2 preserved
Lock Page $LCKPG	R1 = Virtual address in page CALL $LCKPG	C = Clear, page locked C = Set, page not found R0—R2 preserved
Unlock Page $UNLPG	R1 = Virtual address in page CALL $UNLPG	C = Clear, page unlocked C = Set, page not found R0—R2 preserved

A System Reference Bibliography

This bibliography identifies manuals that contain descriptions of additional routines available to IAS system library users.

First level entries are manual titles. Second level entries are functional headings that indicate the types of services described in the respective manuals.

- *IAS Executive Facilities Reference Manual*
  - Task execution control directives
  - Informational directives
  - Event-associated directives
  - Trap-associated directives
  - I/O related directives
  - Task status control directives
- *IAS Device Handlers Reference Manual*
  - Laboratory and industrial I/O routines
- *IAS I/O Operations Reference Manual*
  - I/O preparation services
  - File processing services
  - File control routines
  - File structuring services
  - Command line processing services
  - Parsing services
  - Spooling services

B Universal Library Access

On most IAS systems, you can create a universal library to store related groups of files. The LBR utility creates the universal library file with a file type ULB. By means of the LBR utility, you can subsequently insert files as modules in the library.[1]

To access a module of a universal library, a program can call the $ULA routine, which establishes the necessary conditions for access (read only). The $ULA routine first calls an initializing routine, $ULAIN, to validate that the library file is in the correct format and to obtain the needed information from the library header. $ULA then calls a second routine, $ULAFD, to read the module header, to position libary file pointers to the beginning of the module, and to establish the necessary FDB locations for the File Control System (FCS).[2] Once the necessary FDB locations are established, the program can access the module as if it were a separate file. That is, the program can perform GET$ operations in move mode for each record in the module.

To call the $ULA routine, supply the following data:

- In Register 0, the address of the universal library FDB. The library file must already be open for read access.
- In Register 1, the address of a 42_8-word buffer. The first two words of the buffer must contain the name (in Radix-50 format) of the module to be accessed. $ULA will put a copy of the module header from the library into the remaining 100_8 (64_{10}) bytes. Initialize the FDRC$A arguments urba and urbs (FDB offsets F.URBD and F.URBD+2) in the FDB for the library file. The $ULA routine saves the arguments, uses the space for storing module header information, and restores the values before returning control to the calling program.

The $ULA routine produces the following data:

- Register 0 is unchanged.
- Register 1 is unchanged. The $ULA routine fills in the 40-word buffer with a copy of the header for the module accessed.
- The first seven words of the library file FDB contain the first seven words of the FDB of the module's associated input file (as if it were a separate input file).
- The offset F.EFBK+2 of the library file FDB contains the last block number of the module.
- The offset F.FFBY of the library file FDB contains the number of the next available byte past the end of the module.
- The offset F.ERR of the library file FDB has the standard interpretations except for the following special meanings:
  - The symbol IE.BHD means either "File not a universal library" or "Bad library header."
  - The symbol IE.NSF means "No such module."
- The C bit is set to indicate an error.

[1] See the description of the LBR utility in the *IAS Utilities Manual*, or see the description of the DCL command LIBRARY in the *IAS Command Language Manual*.

[2] See the *IAS I/O Operations Reference Manual* for information on FCS and the use of FDB locations.

To use the $ULA routine properly, use the following coding sequence:

```
        .
        .
        .
OPEN$   R0                   ; OPEN UNIVERSAL LIB FILE
        .
        .                    ; STORE FIRST SEVEN WORDS OF LIBRARY FDB
        .
CALL    $ULA
        .
        .
        .
GET$    R0                   ; ACCESS MODULE IN MOVE MODE ONLY
        .
        .                    ; RESTORE FIRST SEVEN WORDS OF LIBRARY FDB
        .
CLOSE$ R0 or invoke $ULA again
```

NOTE: Note that the program must open the library file for read-only access. (To change a module in the universal library, use the LBR utility.) The program must save the first seven words of the library file FDB before calling the $ULA routine for the first time. The $ULA routine modifies these words during processing, but their original values are necessary either to access another module or to ensure that the library file is closed properly. The program must restore the seven words after accessing a module and before accessing another module or before closing the library file.

Index

Reader's Comments

This form is for document comments only. Digital will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)________________

Name________________ Date________

Organization________________

Street________________

City____________ State______ Zip Code or Country______

Do Not Tear · Fold Here and Tape

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 33 MAYNARD MASS

POSTAGE WILL BE PAID BY ADDRESSEE

IAS Engineering/Documentation
Digital Equipment Corporation
5 Wentworth Drive GSF/L20
Hudson, NH 03051-4929

Do Not Tear · Fold Here