pdp11

IAS/RSX-11
System Library Routines Reference Manual

Order No. AA-5580A-TC

digital

IAS/RSX-11
System Library Routines
Reference Manual

Order No. AA-5580A-TC

RSX-11M V3.1
IAS V2.0
RSX-11D V6.2

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, December 1977

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10

CONTENTS

FIGURES

TABLES

PREFACE

0.1 MANUAL OBJECTIVES AND READER ASSUMPTIONS

The IAS/RSX-11 System Library Routines Reference Manual describes the usage and function of the system library routines that may be called from MACRO-11 assembly language programs. This manual is intended for use by experienced MACRO-11 assembly language programmers.

0.2 STRUCTURE OF THE DOCUMENT

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

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

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

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

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

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

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

Chapter 8 describes the usage 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/RSX-11 systems users.

0.3 ASSOCIATED DOCUMENTS

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

- IAS/RSX-11 MACRO-11 Reference Manual.
- The Task Builder Reference Manual for the appropriate system.
- The manuals referenced in Appendix A.

The reader should refer to the applicable documentation directory for descriptions of other documents associated with this manual.

CHAPTER 1

INTRODUCTION

The routines described in this manual were written to provide commonly needed capabilities for DIGITAL-supplied utilities. We are happy to 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, due to the potentially widespread effect it may have on our system utilities.

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

- Save and restore register content 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 outputs that are returned to the executing task, and the interaction between the library routines and with 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:

- Push register contents to the stack.
- Subsequently pop the contents back into 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 via registers in which outputs are returned and/or 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(s) and/or 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 two files:

- The system library file (SYSLIB.OLB), which contains the following:
  - the register handling routines, described in Chapter 2.
  - the arithmetic routines, described in Chapter 3.
  - the input and output data conversion routines, described in Chapter 4 and Chapter 5.
  - the output formatting routines, described in Chapter 6.
  - the dynamic memory allocation and release routines descriibed in Chapter 7.
- 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.

Summarized procedures for using the system library routines are presented, in tabular format, 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 of this manual.

Additional Executive and I/O routines available to IAS/RSX-11 systems users are described in other manuals. The System Reference Bibliography in Appendix A presents a cross-reference listing of these manual titles, and functional descriptions of types of services described in the respective manual.

CHAPTER 2

REGISTER HANDLING ROUTINES

There are four register handling routines in the system library:

- Save All Registers Routine ($SAVAL), which saves and subsequently restores registers 0 - 5, as described in Section 2.1.
- Save Registers 3 - 5 Routine ($SAVRG), which saves and subsequently restores registers 3 - 5, as described in Section 2.2.
- Save Registers 0 - 2 Routine ($SAVVR), which saves and subsequently restores registers 0 - 2, as described in Section 2.3.
- Save Registers 1 - 5 Routine ($SAVR1), which saves and subsequently restores registers 1 - 5, as described in Section 2.4.

The register handling routines function as co-routines to enable control swapping between themselves, a subroutine, and the original caller of the subroutine.

To illustrate the effect of using the register handling routines, assume the following: an original caller calls a subroutine. The subroutine calls a register handling co-routine. The co-routine pushes the contents of the specified registers to the stack, and issues a co-routine call back to the subroutine. The subroutine executes to completion, when a RETURN instruction is executed to swap control back to the co-routine. The co-routine pops the initial contents of the registers from the stack 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 throughout this document.

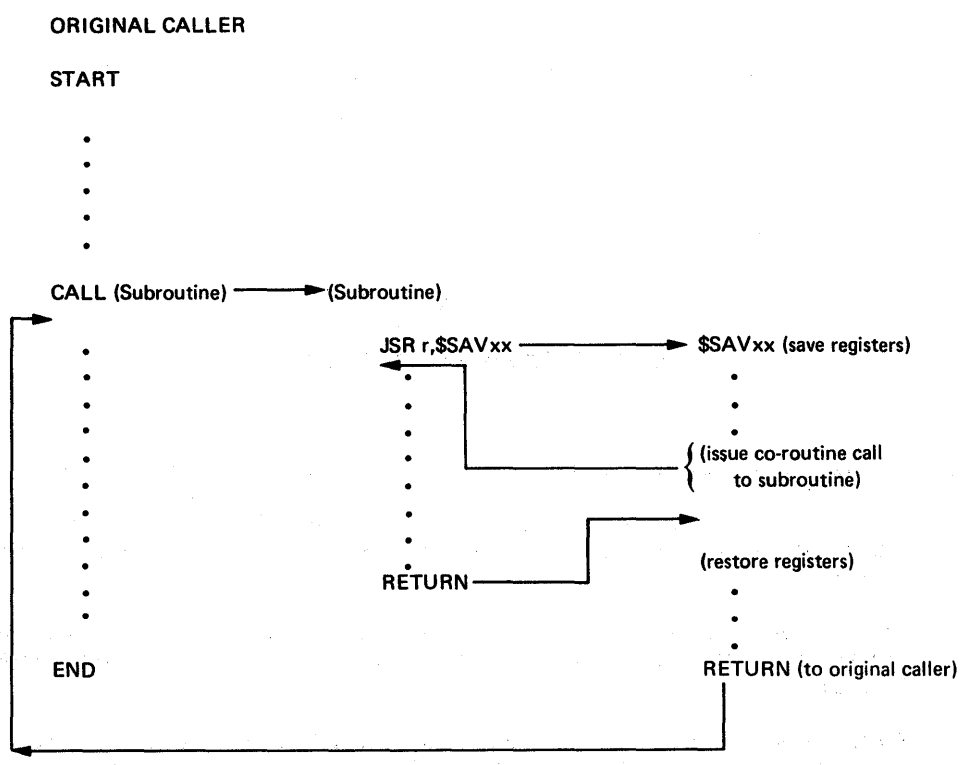

Figure 2-1 Control Swapping of the Register Handling Routines

2.1 SAVE ALL REGISTERS ROUTINE ($SAVAL)

The $SAVAL routine saves and subsequently restores registers 0 - 5 for a subroutine. The $SAVAL routine functions as a co-routine which swaps control between itself, a subroutine, and the original caller.

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

```
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 - 0 to the stack.

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

The $SAVAL routine issues a co-routine call, in the form CALL @(SP)+, to swap control back to the subroutine. The co-routine call replaces the subroutine's 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:

Return Address to Original Caller
Register 5
Register 4
Register 3
Register 2
Register 1
Register 0
Return Address to $SAVAL

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

2.2 SAVE REGISTERS 3 - 5 ROUTINE ($SAVRG)

The $SAVRG routine saves and subsequently restores registers 3 - 5 for a subroutine. The $SAVRG routine functions as a co-routine which 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.

The $SAVRG routine copies the original contents back into R5 and issues a co-routine call, in the form CALL @(SP)+, to swap control back to the subroutine. The co-routine 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, which swaps control back to the $SAVRG routine. The contents of registers 3 - 5 are restored (popped from the stack) and the $SAVRG routine RETURNs, via an 'RTS PC' instruction, to the original caller.

2.3 SAVE REGISTERS 0 - 2 ROUTINE ($SAVVR)

The $SAVVR routine saves and subsequently restores registers 0 - 2 for a subroutine. The $SAVVR routine functions as a co-routine which 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
```

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

On entry to the $SAVVR routine, the program stack contains the return address to the original caller and the contents of R2 of the original caller. The $SAVVR routine pushes the contents of registers 1 and 0 to the stack, then pushes the current contents of R2 (the return address to the subroutine) to the stack.

The $SAVVR routine copies the original contents back into R2 and issues a co-routine call, in the form CALL @(SP)+, to swap control back to the subroutine. The co-routine 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:

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 ('RTS PC') instruction is executed, which swaps control back to the $SAVVR routine. The contents of registers 0 - 2 are restored (popped from the stack) and the $SAVVR routine RETURNs, via an 'RTS PC' instruction, to the original caller.

2.4 SAVE REGISTERS 1 - 5 ROUTINE ($SAVR1)

The $SAVR1 routine saves and subsequently restores registers 1 - 5 for a subroutine. The $SAVR1 routine functions as a co-routine which 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.

On entry to the $SAVR1 routine, the program stack contains the return address to the original caller and the contents of R5 of the original caller. The $SAVR1 routine pushes the contents of registers 4, 3, 2, and 1, and the current contents of R5 (the return address to the subroutine) to the stack.

The $SAVR1 routine copies the original contents back into R5 and issues a co-routine call, in the form CALL @(SP)+, to swap control back to the subroutine. The co-routine call replaces the subroutine's 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:

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 ('RTS PC') instruction is executed, which swaps control back to the $SAVR1 routine. The contents of registers 1 - 5 are restored (popped from the stack) and the $SAVR1 routine RETURNs, via an 'RTS PC' instruction, to the original caller.

CHAPTER 3

ARITHMETIC ROUTINES

The system library contains four arithmetic routines that perform unsigned integer multiplication and division. This chapter describes the usage and function of these arithmetic routines.

3.1 INTEGER ARITHMETIC ROUTINES

The system library contains two routines which 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.

The usage of these routines is described and illustrated in the following paragraphs.

3.1.1 Integer Multiply Routine ($MUL)

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

To call the $MUL routine:

- Specify two input arguments:
  - in Register 0, the multiplier.
  - in Register 1, the multiplicand.
- Include the statement

  ```
  CALL $MUL
  ```

  in the source program.

Registers 2 - 5 of the calling task are preserved.

The output from the $MUL routine is:

- R0 = high order part of result.
- R1 = low order part of result.

The $MUL routine does not return any error indications to the caller.

3.1.2 Integer Divide Routine ($DIV)

The $DIV routine performs unsigned integer division.

To call the $DIV routine:

- Specify two input arguments:
  - in Register 0, the dividend.
  - in Register 1, the divisor.
- Include the statement

  CALL $DIV

  in the source program.

Registers 2 - 5 of the calling task are preserved.

The output from the $DIV routine is:

- R0 = quotient.
- R1 = the remainder.

The $DIV routine does not return any error indications to the caller.

3.1.3 Example

The following source statements call the $MUL routine to perform multiplication and store the results.

```
      .
      .
      .
MOV   #1200,R0      ;PUTS MULTIPLIER IN R0.
MOV   NDV,R1        ;PUTS THE MULTIPLICAND AT NDV IN R1.
CALL  $MUL          ;CALLS $MUL ROUTINE.
MOV   R0,WORK       ;STORES HIGH ORDER PART OF RESULT IN WORK.
MOV   R1,WORK+2     ;STORES LOW ORDER PART OF RESULT IN
                    ;WORK+2.
      .
      .
      .
(continue)
```

3.2 DOUBLE-PRECISION ARITHMETIC ROUTINES

There are two double-precision integer arithmetic routines in the system library:

- 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.

The usage of these routines is described and illustrated in the following paragraphs.

3.2.1 Double-precision Multiply Routine ($DMUL)

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

To call the $DMUL routine:

- Specify two input arguments:
  - in Register 0, the single-precision magnitude multiplier.
  - the double-precision magnitude multiplicand, where:
    - Register 2 contains the high order part.
    - Register 3 contains the low order part.
- Include the statement

  ```
  CALL $DMUL
  ```

  in the source program.

Registers 4 and 5 of the calling task are preserved. Registers 2 and 3 are destroyed on return to the calling task.

The outputs from the $DMUL routine are:

- The double-precision magnitude product, where:
  - R0 = high order part.
  - R1 = low order part.

The $DMUL routine does not return any error indications to the caller.

3.2.2 Double-precision Divide Routine ($DDIV)

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

To call the $DDIV routine:

- Specify two input arguments:
  - in Register 0, the unsigned divisor.
  - the double-precision dividend, where:
    - Register 1 contains the high order part.
    - Register 2 contains the low order part.

- Include the statement

  CALL $DDIV

  in the source program.

The content of Register 3 of the calling task is saved and restored. Registers 4 and 5 are not used.

The outputs from the $DDIV routine are:

- R0 = remainder.
- Quotient, where:
  - R1 = high order part of quotient.
  - R2 = low order part of quotient.

The $DDIV routine does not return any error indications to the caller.

3.2.3 Example

The following source statements call the $DDIV routine to perform division and store the results.

```
      .
      .
      .
MOV   #150,R0       ;PUTS DIVISOR IN R0.
MOV   DVD,R1        ;PUTS THE HIGH ORDER PART OF THE DIVID-
                    ;END, STORED IN DVD, IN R1.
MOV   DVD+2,R2      ;PUTS LOW ORDER PART OF DIVIDEND, STORED
                    ;IN DVD+2, IN R2.
CALL  $DDIV         ;CALLS $DDIV ROUTINE.
MOV   R1,QUOT       ;PUTS HIGH ORDER PART OF QUOTIENT IN QUOT.
MOV   R2,QUOT+2     ;PUTS LOW ORDER PART OF QUOTIENT IN QUOT+2.
MOV   R0,RMAIN      ;PUTS REMAINDER IN REMAIN.
      .
      .
      .
(continue)
```

CHAPTER 4

INPUT DATA CONVERSION ROUTINES

The system library input data conversion routines accept ASCII data as input and convert it to the specified numeric representation. There are three types of routines that 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, as described in Section 4.1 of this chapter.
- ASCII to binary conversion routines, which accept ASCII decimal or octal input numbers and convert them to single-word binary numbers, as described in Section 4.2.
- ASCII to Radix-50 conversion routines, which accept ASCII alphanumeric input characters and convert them to Radix-50 internal format, as described in Section 4.3.

4.1 ASCII TO BINARY DOUBLE-WORD CONVERSIONS

There are two system library routines that 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, described in Section 4.1.1.
- The Octal to Binary Double-word Routine (.OD2CT), which accepts ASCII octal numbers as input and converts them to double-word binary format, described in Section 4.1.2.

4.1.1 Decimal to Binary Double-word Routine (.DD2CT)

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

The routine accepts leading plus (+) or minus (-) signs, and a trailing decimal point. A preceding # symbol is acceptable, but will force octal conversion. A # symbol and a period in the same input string is invalid. Acceptable characters in the string are the numbers 0 - 9. 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 $+2^{31}-1$.

To call the .DD2CT routine:

- Supply three input arguments in the task's source code:
  - in Register 3, the address of the two-word field in which the converted number is to be stored.
  - in Register 4, the number of characters in the string to be converted.
  - in Register 5, the address of the character string to be converted.
- Include the statement

  CALL .DD2CT

  in the source program.

The .DD2CT routine saves and restores all of the calling task's registers. Outputs from the .DD2CT routine are:

- The converted number, where the high order 16 bits are stored in word 1 of the field specified in R3 input, and the low order 16 bits are stored in word 2 of the field.
- Condition Code:

  C bit = clear if conversion was successful.

  C bit = set if an illegal character was found and conversion was incomplete.

The user can determine, in the task, whether conversion was complete by testing the C bit in the Condition Code.

4.1.2 Octal to Binary Double-word Routine (.OD2CT)

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

The routine accepts leading plus (+) or minus (-) signs, and the numbers 0 - 7. A preceding # symbol is legal in the octal number string. A # symbol and a period in the same input string is invalid. A trailing decimal point will be accepted in the input string, but will cause the decimal, rather than octal, radix to be used. This condition exists because the .OD2CT routine is an entry point in the .DD2CT routine, which converts decimal number strings to binary double-word values (see Section 4.1.1).

Any other characters in the ASCII octal number string will cause the .OD2CT routine to terminate the conversion procedure.

The value range of an octal number to be converted is -2^{31} to $+2^{31}-1$.

To call the .OD2CT routine:

- Supply three input arguments in the task's source code:
  - in Register 3, the address of the two-word field in which the converted number is to be stored.
  - in Register 4, the number of characters in the string to be converted.
  - in Register 5, the address of the character string to be converted.
- Include the statement

  CALL .OD2CT

  in the source program.

The .OD2CT routine saves and restores all of the calling task's registers. Outputs from the .OD2CT routine are:

- The converted number, where the high order 16 bits are stored in word 1 of the field specified in R3 input, and the low order 16 bits are stored in word 2 of the field.
- Condition Code:

  C bit = clear if conversion was successful.

  C bit = set if an illegal character was found and conversion was incomplete.

The user can determine, in the task, whether conversion was complete by testing the C bit in the Condition Code.

4.1.3 Example

The following source statements illustrate a way of calling the .OD2CT routine and checking the results on return.

```
BOUT:   .BLKW  2
          .
          .
          .
        MOV    #BOUT,R3   ;PUTS THE ADDRESS OF THE 2-WORD OUTPUT
                          ;FIELD (BOUT) IN R3.
        MOV    #7,R4      ;PUTS THE NUMBER 7 (NUMBER OF INPUT
                          ;CHARACTERS) IN R4.
        MOV    #ICHR,R5   ;PUTS THE ADDRESS OF THE INPUT CHARACTER
                          ;STRING (ICHR) IN R5.
        CALL   .OD2CT     ;CALLS THE .OD2CT ROUTINE.
        BCS    100$       ;BRANCHES TO 100$ IF C BIT SET (INPUT STRING
                          ;CONVERSION WAS NOT SUCCESSFUL).
        (continue)        ;IF C BIT CLEAR CONVERSION WAS SUCCESSFUL
          .               ;AND THE PROGRAM CONTINUES.
          .
          .
          .
          .
          .

100$:   CALL ERR          ;CALL ERR ROUTINE TO OUTPUT MESSAGE.
```

4.2 ASCII TO BINARY CONVERSIONS

There are two system library routines that convert unsigned ASCII input numbers to single-word unsigned binary numbers:

- The Decimal to Binary Conversion Routine ($CDTB), which accepts ASCII decimal numbers as input and converts them to single-word binary format, described in Section 4.2.1.
- The Octal to Binary Conversion Routine ($COTB), which accepts ASCII octal numbers as input and converts them to single-word binary format, described in Section 4.2.2.

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

4.2.1 Decimal to Binary Conversion Routine ($CDTB)

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

Valid characters in the input decimal number are the characters 0 - 9. 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 may be any ASCII character except the numbers 0 - 9. Examples of terminating characters are: blank, tab character, alphabetic character, and a 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. No significant condition code setting is returned to the calling task.

To call the $CDTB routine:

- Input, in Register 0, the address of the first byte of the ASCII characters to be converted.
- Include the statement

  ```
  CALL $CDTB
  ```

  in the source program.

The $CDTB routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task.

The outputs returned from the $CDTB routine are:

- R0 = the address of the next byte in the input buffer.
- R1 = the converted number.
- R2 = the terminating character.

The user can determine, in the task, whether an input string was successfully converted by testing the content of R2. If the content is other than the expected terminating character, the conversion was incomplete, since some other invalid character was found in the input string.

Since the $CDTB routine returns the address of the next byte in the input buffer to the calling task, successive input string conversion may be effected by setting up a processing loop back to the CALL $CDTB statement, as shown in Section 4.2.3.

4.2.2 Octal to Binary Conversion Routine ($COTB)

The $COTB routine converts an ASCII octal number to binary. Valid characters in the number to be converted are 0 - 7.

The end of a string must be marked by a terminating character, which may be any ASCII character except the numbers 0 - 7. Examples of terminating characters are: blank, a tab character, an alphabetic character, and a special symbol. Leading blanks and tab characters are ignored.

The maximum value of an octal number that can be converted by the $COTB routine is 177777.

To call the $COTB routine:

- Input, in Register 0, the address of the first byte of the ASCII characters to be converted.
- Include the statement

  ```
  CALL $COTB
  ```

  in the source program.

The $COTB routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task.

The outputs returned from the $COTB routine are:

- R0 = the address of the next byte in the input buffer.
- R1 = the converted number.
- R2 = the terminating character.

The user can determine, in the task, whether an input string was successfully converted by testing the content of R2. If the content is other than the expected terminating character, the conversion was incomplete, since some other invalid character was found in the input string.

Successive input string conversion may be effected by setting up a processing loop as shown in the example for the $CDTB routine in Section 4.2.3.

4.2.3 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. The character strings stored in the input buffer (IBUF) are in the format

```
xxx(TAB)xxxx(TAB)xx(TAB)xxxxx(SPACE)
```

where the TAB character is used as the terminating character of each string, and the SPACE character is used as the terminating character of the input buffer. If converted successfully, the binary numbers are to be stored in the buffer, BNUM.

```
        MOV   #BNUM,R4    ;PUTS THE OUTPUT BUFFER ADDRESS IN R4.
        MOV   #IBUF,R0    ;PUTS INPUT BUFFER ADDRESS IN R0 (REQUIRED
                          ;INPUT ARGUMENT FOR $CDTB ROUTINE).
LOOP:   CALL  $CDTB       ;CALLS CONVERSION ROUTINE
        MOV   R1,(R4)+    ;PLACE CONVERTED NUMBER IN BNUM
        CMP   #11,R2      ;COMPARE ASCII TAB (HT) VALUE TO TERMINATING
                          ;CHARACTER RETURNED IN R2.
        BEQ   LOOP        ;IF EQUAL, STRING SUCCESSFULLY CONVERTED;
                          ;GO BACK THROUGH LOOP TO CONVERT NEXT INPUT
                          ;STRING POINTED TO BY R0.
        CMP   #40,R2      ;COMPARE SPACE VALUE (40) WITH TERMINATING
                          ;CHARACTER IN R2.
        BEQ   10$         ;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$:    (continued source program)
```

4.3 ASCII TO RADIX-50 CONVERSIONS

There are two system library routines that convert ASCII alphanumeric input characters to 16-bit Radix-50 values:

- The ASCII to Radix-50 Conversion Routine ($CAT5), which accepts input characters from the ASCII subset and converts them to Radix-50 format, described in Section 4.3.1.
- The ASCII With Blanks to Radix-50 Conversion Routine ($CAT5B), which accepts input characters from the ASCII subset and blank characters and converts them to Radix-50 format, described in Section 4.3.2.

The Integer Multiply Routine ($MUL) is called to perform the multiplication required for the conversion.

4.3.1 ASCII to Radix-50 Conversion Routine ($CAT5)

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

Valid characters in the ASCII string to be converted are:

- Alphabetic characters A - Z.
- Numeric characters 0 - 9.
- The characters: dollar sign ($) and period (.).

For complete conversion the string must contain three valid characters. Invalid characters will cause the $CAT5 routine to terminate conversion. When an invalid character is found in the input string, the converted value will represent the valid character(s) and trailing blank(s).

NOTE

A blank character (space) in the ASCII character string will cause the $CAT5 routine to terminate. If blanks are to be included as valid characters in the string, the $CAT5B routine should be called to perform the conversion (see Section 4.3.2).

To call the $CAT5 routine:

- Input , in Register 0, the address of the first character in the ASCII string to be converted.
- Input, in Register 1, the period disposition flag, as follows:

  R1 = 0 to specify that the period is a conversion terminator (terminating character).

  R1 = 1 to specify that the period is to be accepted as a valid character and used in the conversion to Radix-50.

- Include the statement

  CALL $CAT5

  in the source program.

The $CAT5 routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Outputs from the $CAT5 routine are:

- R0 = the address of the next character in the input string.
- R1 = the converted Radix-50 value (1 - 3 characters).
- R2 = the terminating character. (This is the last character in the string that was converted, or the invalid character that caused conversion termination.)
- Condition Code:

  C bit = set if fewer than three characters were converted.

  C bit = clear if conversion was complete.

The user can determine, in the task, whether conversion was complete by testing the C bit in the Condition Code or the content of Register 2. Since the address of the next character in the input string is returned in Register 0, successive input string conversion may be effected by resetting R1 and setting up a processing loop back to the CALL $CAT5 statement.

4.3.2 ASCII with Blanks to Radix-50 Conversion Routine ($CAT5B)

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

Valid characters in the ASCII string to be converted are:

- Alphabetic characters A - Z.
- Numeric characters 0 - 9.
- The characters: dollar sign ($), period (.), and blank (space).

For complete conversion, the string must contain three valid characters. Invalid characters will cause the $CAT5B routine to terminate conversion. When an invalid character is found in the input string, the converted value will represent the valid character(s) and trailing blank(s).

To call the $CAT5B routine:

- Input, in Register 0, the address of the first character in the ASCII string to be converted.
- Input, in Register 1, the period disposition flag, as follows:

  R1 = 0 to specify that the period is a conversion terminator (terminating character).

  R1 = 1 to specify that the period is to be accepted as a valid character and used in the conversion to the Radix-50 value.

- Include the statement

```
CALL $CAT5B
```

  in the source program.

The $CAT5B routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Outputs from the $CAT5B routine are:

- R0 = the address of the next character in the input buffer.
- R1 = the converted Radix-50 value.
- R2 = the terminating character. (This is the last character in the string that was converted, or the invalid character which caused conversion termination.)
- Condition Code:

  C bit = set if conversion was incomplete.

  C bit = clear if conversion was complete.

The user can determine, in the task, whether conversion was complete by testing the C bit in the Condition Code or the content of Register 2. Since the address of the next character in the input string is returned in Register 0, successive input string conversion may be effected by resetting R1 and setting up a processing loop back to the CALL $CAT5B statement.

4.3.3 Example

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

```
CNVRTR: JSR    R2,$SAVVR     ;CALLS $SAVVR ROUTINE TO SAVE R0 - R2.
        MOV    #ASDAT,R0     ;PUTS THE ADDRESS OF THE FIRST ASCII
                             ;CHARACTER IN R0.
        CLR    R1            ;SETS R1 TO ZERO TO SPECIFY THAT PERIOD
                             ;IS CONVERSION TERMINATOR.
        CALL   $CAT5         ;CALLS $CAT5 ROUTINE TO CONVERT ASCII
                             ;TO RADIX-50.
        BCC    2$            ;BRANCH TO 2$ IF C BIT IS CLEAR
                             ;(CONVERSION COMPLETE).
        JMP    INER          ;JUMPS TO INPUT ERROR ROUTINE (INER) IF
                             ;C BIT IS SET (CONVERSION INCOMPLETE).
2$:     MOV    R1,RAD5       ;STORES CONVERTED CHARACTER IN RAD5.
        RETURN               ;RETURNS TO MAIN PROGRAM, VIA $SAVVR,
                             ;WHEN CONVERSION COMPLETE.
```

CHAPTER 5

OUTPUT DATA CONVERSION ROUTINES

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

- Binary to decimal conversion routines, described in Section 5.1, to convert binary data to one of the following:
  - a 2-digit day date, in the range 01-31.
  - a 5-digit unsigned decimal magnitude number.
  - a 5-digit signed decimal number.
  - a decimal number up to 9 digits in length.
- Binary to octal conversion routines, described in Section 5.2, to convert binary numbers to one of the following octal numbers:
  - a 6-digit unsigned octal magnitude number.
  - a 6-digit signed octal number.
  - a 3-digit octal number.
- A general purpose binary conversion routine, to convert binary data to ASCII format. This routine may be called directly by the user to perform converion from binary to ASCII according to user-defined parameters, or it may be called indirectly be calling the binary to decimal or octal routines which pass predefined parameters to this routine, as described in Section 5.3.
- A Radix-50 to ASCII conversion routine, to convert a Radix-50 value to a 3-character ASCII string, as described in Section 5.4.

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.

5.1 BINARY TO DECIMAL CONVERSIONS

There are four system library routines that 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, as described in Section 5.1.1.

- Convert Binary to Decimal Magnitude Routine ($CBDMG), which converts an internally stored binary number to a 5-digit unsigned ASCII decimal magnitude value, as described in Section 5.1.2.

- Convert Binary to Signed Decimal Routine ($CBDSG), which converts an internally stored binary number to a 5-digit signed ASCII decimal number, as described in Section 5.1.3.

- Convert Double-precision Binary to Decimal Routine ($CDDMG), which converts a double-precision, unsigned binary number to an ASCII decimal number of 9 digits or less, as described in Section 5.1.4.

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

5.1.1 Binary Date Conversion Routine ($CBDAT)

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

The $CBDAT routine uses the following predefined conversion parameters:

```
radix = 10.
field width = 2.  characters
sign flag = UNSIGNED
leading zeroes flag = NOSUP (no suppression)
```

To call the $CBDAT routine:

- Supply three input arguments in the task's source code:

  - in Register 0, the starting address of the output area in which the converted two-byte date is to be stored.

  - in Register 1, the binary date to be converted.

  - in Register 2, the zero suppression indicator, where:

    R2 = 0 to specify suppression of leading zeroes in the converted date, which will be left-justified.

    R2 = nonzero to specify that leading zeroes are not to be suppressed.

- Include the statement

  ```
  CALL $CBDAT
  ```

  in the source program.

The predefined conversion parameters are automatically pushed to the stack on entry to the $CBDAT routine. If the user specifies, via R2 = 0, that leading zeroes are to be suppressed, the NOSUP parameter is reset. In any case, the $CBDAT routine passes the parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBTA routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Registers 1 and 2 are destroyed.

Outputs from the $CBDAT routine are:

- The converted day date (in the range 01-31) in the specified output area.
- R0 = the next available address in the output area (the pointer to the location following the last digit stored).

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

5.1.2 Convert Binary to Decimal Magnitude Routine ($CBDMG)

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

The $CBDMG routine uses the following predefined conversion parameters:

```
radix = 10.
field width = 5.  characters
sign flag = UNSIGNED
leading zeroes flag = NOSUP (no suppression)
```

To call the $CBDMG routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the starting address of the output area in which the converted 5-digit number is to be stored.
  - in Register 1, the unsigned binary number to be converted.
  - in Register 2, the zero suppression indicator, where:

    R2 = 0 to specify suppression of leading zeroes in the converted number. The output number will be left-justified.

    R2 = nonzero to specify that leading zeroes are not to be suppressed.
- Include the statement

  ```
  CALL $CBDMG
  ```

  in the source program.

The predefined conversion parameters are automatically pushed to the stack on entry to the $CBDMG routine. If the user specifies, via R2 = 0, that leading zeroes are to be suppressed, the NOSUP parameter is reset. In any case, the $CBDMG routine passes the parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBTA routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Registers 1 and 2 are destroyed.

Outputs from the $CBDMG routine are:

- The converted number, a maximum of five digits in length, in the specified output area.
- R0 = the next available address in the output area (the pointer to the location following the last digit stored).

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

5.1.3 Convert Binary to Signed Decimal Routine ($CBDSG)

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

The $CBDSG routine uses the following predefined conversion parameters:

```
radix = 10.
field width = 5.  characters
sign flag = SIGNED
leading zeroes flag = NOSUP (no suppression)
```

To call the $CBDSG routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the starting address of the output area in which the converted 5-digit number is to be stored.
  - in Register 1, the binary number to be converted.
  - in Register 2, the zero suppression indicator, where:

    R2 = 0 to specify suppression of leading zeros in the converted number. The output number will be left-justified.

    R2 = nonzero to specify that leading zeroes are not to be suppressed.
- Include the statement

  ```
  CALL $CBDSG
  ```

  in the source program.

The predefined conversion parameters are automatically pushed to the stack on entry to the $CBDSG routine. If the user specifies, via R2 = 0, that leading zeroes are to be suppressed, the NOSUP parameter is reset. In any case, the $CBDSG routine passes the parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBTA routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Registers 1 and 2 are destroyed.

Outputs from the $CBDSG routine are:

- The converted number, a maximum of five digits in length, in the specified output area.
- R0 = the next available address in the output area (the pointer to the location following the last digit stored).

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

5.1.4 Convert Double-precision Binary to Decimal Routine ($CDDMG)

The $CDDMG routine converts a double-precision, unsigned binary number to an ASCII decimal number of 9 digits or less. If the number contains more than 9 digits, the routine inserts a string of five ASCII asterisk symbols in the output area.

To call the $CDDMG routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the starting address of the output area.
  - in Register 1, the address of the two-word input area containing the double-precision number.
  - in Register 2, the zero suppression indicator, where:

    R2 = 0 to specify that leading zeroes are to be suppressed. The number will be left-justified.

    R2 = nonzero, to specify that leading zeroes are not to be suppressed.

    NOTE

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

- Include the statement

  ```
  CALL $CDDMG
  ```

  in the source program.

The $CDDMG routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Registers 1 and 2 are destroyed. The $DDIV routine is called to perform the double-precision division, and the General Purpose Binary to ASCII Conversion Routine ($CBTA) is called to perform the actual ASCII conversion.

Outputs from the $CDDMG routine are:

- R0 = the pointer to the next available address in the output storage area.
- The converted ASCII number, stored in the output area, or a string of ASCII asterisks if more than 9 digits resulted in the conversion attempt. If the number was successfully converted, the output area will contain from four to nine digits.

5.1.5 Examples

Examples of the use of output conversion routines are presented in this section.

EXAMPLE 1. The following are source statements that illustrate the steps required to call the $CBDAT routine.

```
      .
      .
      .
     MOV   #ASDAT,R0     ;PUTS THE ADDRESS OF OUTPUT AREA IN R0.
     MOV   BDAT,R1       ;PUTS THE BINARY DATE, AT BDAT, IN R1.
     CLR   R2            ;CLEARS R2 TO ZERO TO SPECIFY THAT LEADING
                         ;ZEROES ARE TO BE SUPPRESSED.
     CALL  $CBDAT        ;CALLS THE $CBDAT ROUTINE.
      .
      .
      .
     (continue)
```

EXAMPLE 2. The following are source statements to call the $CDDMG routine to convert a double-precision number to an ASCII decimal number and check the result.

```
       .
       .
       .
      MOV   #ASDN,R0     ;PUTS ADDRESS OF OUTPUT AREA IN R0.
      MOV   #DPWRD,R1    ;PUTS STARTING ADDRESS OF DOUBLE-
                         ;PRECISION INPUT WORD IN R1.
      MOV   #4.,R2       ;PUTS NONZERO IN R2 (SETS THE ZERO
                         ;INDICATOR FLAG TO 1) TO SPECIFY
                         ;THAT LEADING ZEROES 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$:  (continue)
```

5.2 BINARY TO OCTAL CONVERSION

There are three system library routines that convert internally formatted binary numbers to external ASCII octal format:

- Convert Binary to Octal Magnitude Routine ($CBOMG), which converts an internally stored binary number to a 6-digit unsigned ASCII octal magnitude number, as described in Section 5.2.1.
- Convert Binary to Signed Octal Routine ($CBOSG), which converts an internally stored binary number to a 6-digit signed ASCII octal number, as described in Section 5.2.2.
- Convert Binary Byte to Octal Magnitude Routine ($CBTMG), which converts an internally stored binary byte to a 3-digit ASCII octal number, as described in Section 5.2.3.

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

5.2.1 Convert Binary to Octal Magnitude Routine ($CBOMG)

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

The $CBOMG routine uses the following predefined conversion parameters:

```
radix = 8.
field width = 6. characters
sign flag = UNSIGNED
leading zeroes flag = NOSUP (no suppression)
```

To call the $CBOMG routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the starting address of the output area in which the converted 6-digit number is to be stored.
  - in Register 1, the binary number to be converted.
  - in Register 2, the zero suppression indicator where:

    R2 = 0 to specify suppression of leading zeroes in the converted number. The output number will be left-justified.

    R2 = nonzero to specify that leading zeroes are not to be suppressed.
- Include the statement

  ```
  CALL $CBOMG
  ```

  in the source program.

The predefined conversion parameters are automatically pushed to the stack on entry to the $CBOMG routine. If the user specifies, via R2 = 0, that leading zeroes are to be suppressed, the NOSUP parameter is reset. In any case, the $CBOMG routine passes the parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBTA routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Registers 1 and 2 are destroyed.

Outputs from the $CBOMG routine are:

- The converted number, a maximum of six digits in length, in the specified output area.
- R0 = the next available address in the output area (the pointer to the location following the last digit stored).

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

5.2.2 Convert Binary to Signed Octal Routine ($CBOSG)

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

The $CBOSG routine uses the following predefined conversion parameters:

```
radix = 8.
field width = 6.  characters
sign flag = SIGNED
leading zeroes flag = NOSUP (no suppression)
```

To call the $CBOSG routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the starting address of the output area in which the converted 6-digit number is to be stored.
  - in Register 1, the binary number to be converted.
  - in Register 2, the zero suppression indicator, where:

    R2 = 0 to specify suppression of leading zeroes in the converted number. The output number will be left-justified.

    R2 = nonzero to specify that leading zeroes are not to be suppressed.
- Include the statement

  ```
  CALL $CBOSG
  ```

  in the source program.

The predefined conversion parameters are automatically pushed to the stack on entry to the $CBOSG routine. If the user specifies, via R2 = 0, that leading zeroes are to be suppressed, the NOSUP parameter is reset. In any case, the $CBOSG routine passes the parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary number.

The $CBTA routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Regsiters 1 and 2 are destroyed.

Outputs from the $CBOSG routine are:

- The converted number, a maximum of six digits in length, in the specified output area.
- R0 = 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.

5.2.3 Convert Binary Byte to Octal Magnitude Routine ($CBTMG)

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

The $CBTMG routine uses the following predefined conversion parameters:

```
radix = 8.
field width = 3 characters
sign flag = UNSIGNED
leading zeroes flag = NOSUP (no suppression)
```

To call the $CBTMG routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the starting address of the output area in which the converted 3-digit number is to be stored.
  - in Register 1, the binary byte to be converted in the low order byte.
  - in Register 2, the zero suppression indicator, where:

    R2 = 0 to specify suppression of leading zeroes in the converted number. The output number will be left-justified.

    R2 = nonzero to specify that leading zeroes are not to be suppressed.

- Include the statement

  ```
  CALL $CBTMG
  ```

  in the source program.

The predefined conversion parameters are automatically pushed to the stack on entry to the $CBTMG routine. If the user specifies, via R2 = 0, that leading zeroes are to be suppressed, the NOSUP parameter is reset. In any case, the $CBTMG routine passes the parameters in Register 2 to the General Purpose Binary to ASCII Conversion Routine ($CBTA), which performs the actual conversion of the binary byte.

The $CBTA routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. Registers 1 and 2 are destroyed.

Outputs from the $CBTMG routine are:

- The converted number, a maximum of three digits in length, in the specified output area.
- R0 = the next available address in the output area (the pointer to the location following the last digit stored).
- R1 = low order byte is unchanged; high order byte is cleared by the $CBTMG routine.

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

5.2.4 Example

Following are source statements that illustrate the calling sequence of the binary to octal conversion routines.

```
      .
      .
      .
MOV   #OCOUT,R0     ;PUTS THE STARTING ADDRESS OF THE OUTPUT
                    ;AREA IN R0.
MOV   BNUM,R1       ;PUTS THE BINARY NUMBER TO BE CONVERTED
                    ;IN R1.
MOV   #7,R2         ;PUTS THE VALUE 7 IN R2 (SETS THE ZERO
                    ;INDICATOR FLAG TO 1) TO SPECIFY THAT
                    ;LEADING ZEROES ARE NOT TO BE SUPPRESSED.
CALL  $CBOMG        ;CALLS THE $CBOMG ROUTINE.
      .
      .
      .
(continue)
```

5.3 GENERAL PURPOSE BINARY TO ASCII CONVERSION ROUTINE ($CBTA)

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 described in Sections 5.1 and 5.2 of this chapter. The $CBTA routine converts the values according to predefined parameters which the calling routine passes as an input argument in Register 2.

The $CBTA routine may be called directly by the user to convert internally stored binary values to an external ASCII format according to user-defined conversion parameters, as described in the following paragraph.

To call the $CBTA routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the starting address of the output area in which the converted ASCII number is to be stored.
  - in Register 1, the binary value to be converted.
  - in Register 2, the conversion parameters, as follows:

Bits 0 - 7: (low byte) must contain the conversion radix (2.-10.).

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.

Bit 9: zero suppression flag = 0; or nonzero suppression flag = 1.

Bit 10: blank fill flag = 1 to specify replacement of leading zeroes with blanks only if nonzero suppression flag = 1.

blank fill flag = 0 to specify no replacement of leading zeroes if bit 9 = 1.

Note: When the zero suppression flag = 0, the blank fill flag is ignored.

Bits 11-15: must contain a numeric value from 1 - 32 specifying the field width.

- Include the statement

  CALL $CBTA

  in the source program.

The $CBTA routine calls the $SAVRG routine to save and restore registers 3 - 5 of the caller. Registers 1 and 2 are destroyed. The Integer Divide Routine ($DIV) is called to perform the required division.

The outputs from the $CBTA routine are:

- The converted number, from 1 to 32 digits in length, in the specified output area.
- R0 = the next available address in the output area (the pointer to the location following the last digit stored).

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

5.4 RADIX-50 TO ASCII CONVERSION ROUTINE ($C5TA)

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

To call the $C5TA routine:

- Supply two input arguments in the task's source code:
  - in Register 0, the address at which the first byte of the converted string is to be stored.
  - in Register 1, the Radix-50 value which is to be converted.

- Include the statement

  ```
  CALL $C5TA
  ```

  in the source program.

The $DIV routine is called to perform the required division.

The $C5TA routine does not use registers 3 - 5. Registers 1 and 2 are destroyed.

The outputs from the $C5TA routine are:

- The converted ASCII characters, stored in a maximum of three consecutive bytes in the specified output area.
- R0 = the address of the next byte after the last character stored.

The $C5TA routine does not return error conditions to the caller.

CHAPTER 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. There are four output formatting routines:

- The Upper Case Text Conversion Routine ($CVTUC), which converts lower case ASCII text to upper case, as described in Section 6.1.
- The Date String Conversion Routine ($DAT), which converts a three-word binary date to a nine-character ASCII output string, as described in Section 6.2.1.
- The Time Conversion Routine ($TIM), which converts the binary time to an ASCII output string, as described in Section 6.2.2.
- The Edit Message Routine ($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, as described in Section 6.3.

6.1 UPPER CASE TEXT CONVERSION ROUTINE ($CVTUC)

The $CVTUC routine converts lower case ASCII text to upper case. The routine performs a byte-by-byte transfer of the input ASCII character string, converting all lower case alphabetic characters to upper case, and transferring all upper case characters.

To call the $CVTUC routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the address of the text string to be converted.
  - in Register 1, the address of the output area for the upper case string.
  - in Register 2, the number of bytes in the string to be converted.
- Include the statement

```
CALL $CVTUC
```

  in the source program.

The $CVTUC routine does not use registers 3 - 5 of the calling task. The content of R2 (byte count) is destroyed on return to the calling task. Registers 0 and 1 are not altered.

The $CVTUC routine converts all ASCII alphabetic characters in the input string to upper case. Any other characters are moved from the input area to the output area in their sequential positions. The user may specify the input area address as the output area address (R0 = R1) when the $CVTUC routine is called. In this instance, the lower case alphabetic characters are converted to upper case in place in the input area. The original lower case content of the input area is, of course, destroyed.

6.2 DATE AND TIME FORMAT CONVERSION

These routines convert the binary date and time to formatted ASCII output strings. The date is converted and formatted for output as follows:

```
day-month-year.
```

The time is converted and formatted for output in one of the following forms:

```
hour
hour:minute
hour:minute:second
hour:minute:second:tick
```

These routines and the procedures for using them are described in detail in the following sections.

6.2.1 Date String Conversion Routine ($DAT)

The $DAT routine converts the three-word internal binary date to the standard nine-character ASCII output format, as follows:

```
dd-mmm-yy
```

where:

```
 dd = day (1 - 31)
mmm = month (first 3 letters)
 yy = year (last 2 digits)
```

To call the $DAT routine, the user must:

- Supply two input arguments in the program's source code:
  - in Register 0, the address of the output area in which the converted date is to be stored.
  - in Register 1, the address of the three-word input area in which the binary date is stored. The input area must contain the following:

```
word 1 = last two digits of year
word 2 = a number from 1 - 12 (month of year)
word 3 = a numeric value 01 - 31 (day of month)
```

- Include the statement

  CALL $DAT

  in the source program.

NOTE: The $DAT routine does not perform validity checking of the input.

The $DAT routine calls the $SAVRG routine to save and restore the content of registers 3 - 5 of the calling task. The $DAT routine uses Register 2 and possibly destroys its content. Hence, the calling task should save any critical value in Register 2 before calling the $DAT routine. Outputs from the $DAT routine are the following:

- The converted date string, stored in the specified output area.
- R0 = the address of the next available location in the output area;
- R1 = the next address (input R1 + 6) in the input area.

6.2.2 Time Conversion Routine ($TIM)

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

HH:MM:SS.S

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 - 23
WD2	Minute-of-Hour	MM	0 - 59
WD3	Second-of-Minute	SS	0 - 59
WD4	Tick-of-Second	.S	depends on clock frequency
WD5	Ticks-per-Second	.S	depends on clock frequency

To call the $TIM routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the address of the output area in which the converted time is to be stored.
  - in Register 1, the starting address of the input area in which the time values are stored.
  - 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).

- Include the statement

```
CALL $TIM
```

  in the source program.

The $TIM routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. The contents of registers 0 and 1 are updated and returned to the calling task. Register 2, containing the parameter count, is destroyed.

The outputs from the $TIM routine are the following:

- The converted time string, which is stored in the specified output area.
- R0 = the address of the next available location in the output area.
- R1 = the address of the next word in the input area.

The $TIM routine calls two system library routines:

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

6.2.3 Example

Assume a program contains the following:

The input area containing the three-word binary date and the five-word binary time:

```
BDBLK:   .WORD 77.    ;YEAR
         .WORD 11.    ;11TH MONTH (NOV)
         .WORD 01.    ;DAY
         .WORD 10.    ;HOUR
         .WORD 15.    ;MINUTES
         .WORD 35.    ;SECONDS
         .WORD xx
         .WORD x
```

The output area:

```
DTBLK:   .BLKB  20.
```

The source statements:

```
      .
      .
      .
MOV   #DTBLK,R0     ;PUTS ADDRESS OF OUTPUT AREA IN R0.
MOV   #BDBLK,R1     ;PUTS ADDRESS OF INPUT BINARY DATE AREA IN R1.
CALL  $DAT          ;CALLS THE $DAT ROUTINE.
MOV   #11,(R0)+     ;PUTS TAB AFTER DATE IN OUTPUT BUFFER.
                    ;R0 NOW CONTAINS NEXT ADDRESS IN DTBLK FROM
                    ;$DAT.
                    ;R1 NOW CONTAINS ADDRESS OF NEXT WORD (THE
                    ;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:

```
01-NOV-77      10:15:35
```

6.3 GENERALIZED FORMATTING

Generalized output formatting is provided by the Edit Message Routine ($EDMSG). The $EDMSG routine allows the user to request the conversion of internally stored data to ASCII decimal, octal, or alphanumeric characters and to control the layout of the converted characters to produce formatted output to be printed or displayed as meaningful, readable text. The user may request the conversion of strings of data and their formatting in columns. Varying ASCII data types can be incorporated in a given formatted line of the output. Spacing within lines and between lines can be controlled by the user.

The procedures for converting data and producing formatted output with the $EDMSG routine are described and illustrated in the following section.

6.3.1 Edit Message Routine ($EDMSG)

The $EDMSG routine edits 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. Input strings may contain ASCII text as well as editing directives. Any non-editing directive characters are simply copied into the output block. Any number of directives may appear in an input string. Input strings must be in ASCIZ format.

Editing directives are of two types: data conversion and format control. The directives must be of the form:

```
%l, %nl or %Vl
```

where:

- % is a delimiter which identifies an editing directive to the $EDMSG routine.
- n is 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.
- l is an alphabetic letter specifying one of 18 editing operations to be performed by the $EDMSG routine, as shown in Table 6-1.

See Section 6.3.2 for examples of editing directive use in input strings.

Table 6-1
$EDMSG Routine Editing Directives

Directive	Form	Operation
A (ASCII* 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 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 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 no suppression conversion)	%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.

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

KEY: ARGBLK = The argument block containing the binary data to be converted, or the addresses of ASCII and extended ASCII characters.

OUTBLK = The output block in which $EDMSG is to store output.

(continued on next page)

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

Directive	Form	Operation
E (extended ASCII*)	%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 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.
M (binary to decimal magnitude, 0 suppress conversion)	%M	Convert the binary value in the next word in ARGBLK to decimal magnitude with leading zeroes suppressed and store the result in OUTBLK.
	%nM	Convert the next n binary values in ARGBLK to decimal magnitude with leading zeroes 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 zeroes suppressed and store the results in OUTBLK; insert tab between numbers.

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

KEY: ARGBLK = The argument block containing the binary data to be converted, or the addresses of ASCII and extended ASCII characters.

OUTBLK = The output block in which $EDMSG is to store output.

(continued on next page)

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

Directive	Form	Operation
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 octal magnitude conversion)	%P	Convert the binary value in the next word in ARGBLK to octal magnitude and store the result in OUTBLK.
	%nP	Convert the next n binary values in ARGBLK to octal magnitude 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, and store the results in OUTBLK; insert tab between numbers.

KEY: ARGBLK = The argument block containing the binary data to be converted, or the addresses of ASCII and extended ASCII characters.

OUTBLK = The output block in which $EDMSG is to store output.

(continued on next page)

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

Directive	Form	Operation
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.
	%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 in ARGBLK to decimal and store result in OUTBLK.
	%nT	Convert the next n double-precision binary values 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 to decimal, and store the results in OUTBLK; insert tab between numbers.

KEY: ARGBLK = The argument block containing the binary data to be converted, or the addresses of ASCII and extended ASCII characters.

OUTBLK = The output block in which $EDMSG is to store output.

(continued on next page)

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

Directive	Form	Operation
U (binary to decimal magnitude, no 0 suppress conversion)	%U	Convert the binary value in ARGBLK to decimal magnitude with no leading zeroes suppressed and store result in OUTBLK.
	%nU	Convert the next n binary values in ARGBLK to decimal magnitude with no leading zeroes 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 zeroes suppressed and store results in OUTBLK; insert tab between numbers.
X (file name string conversion)	%X	Convert Radix-50 file name 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 file name strings in ARGBLK to ASCII string 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 file name 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. In this directive a repeat is acceptable, but will be ignored.

KEY: ARGBLK = The argument block containing the binary data to be converted, or the addresses of ASCII and extended ASCII characters.

OUTBLK = The output block in which $EDMSG is to store output.

(continued on next page)

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

Directive	Form	Operation
Z (binary time to ASCII 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 byte field)	%n<	Insert n ASCII spaces followed by a field mark (NULL) 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.

KEY: ARGBLK = The argument block containing the binary data to be converted, or the addresses of ASCII and extended ASCII characters.

OUTBLK = The output block in which $EDMSG is to store output.

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

- If ASCII or extended ASCII characters are to be moved to the output block, the argument block must contain the addresses of the ASCII characters.
- If binary byte to octal conversion is to be performed, the argument block must contain the addresses of the binary bytes.
- If binary values are to be converted, the argument block must contain the values.

- If file name string conversion is to be specified, the argument block must contain the following:

  ```
  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
  ```

- If the binary date is to be converted, the argument block must contain the following:

  ```
  Word 1 = year (last 2 digits)
  Word 2 = number (1 - 12) of month
  Word 3 = day of month (1 - 31)
  ```

  There is no validity checking of the date values. If erroneous date values are specified, output results will be unpredictable.

- If the binary time is to be converted, the argument block must contain the following:

  ```
  Word 1 = hour-of-day (0 - 23)
  Word 2 = minute-of-hour (0 - 59)
  Word 3 = second-of-minute (0 - 59)
  Word 4 = tick-of-second (depends on clock frequency)
  Word 5 = ticks-per-second (depends on clock frequency)
  ```

To call the $EDMSG routine:

- Supply three input arguments in the task's source code:
  - in Register 0, the starting address of the output block.
  - in Register 1, the address of the input string.
  - in Register 2, the starting address of the argument block.
- Include the statement

  ```
  CALL $EDMSG
  ```

  in the source program.

The $EDMSG routine calls the $SAVRG routine to save and restore registers 3 - 5 of the calling task. The $EDMSG routine calls the output data conversion routines described in Chapter 5 and in Section 6.2 of this chapter 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 non-directive or ("unknown" directive) characters are found, they are transmitted directly to the task's output block. When a % delimiter is found, the $EDMSG routine interprets the character(s) following the delimiter. If a data conversion directive is encountered, 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 % delimiter is not followed by a valid operator, or if multiple delimiters are found, the $EDMSG routine transmits the first and any subsequent delimiters not followed by a valid directive character to the output block.

Outputs from the $EDMSG routine are:

- The converted/formatted data in the output block.
- R0 = the address of the next available byte in the output block.
- R1 = the length of the output block (the number of bytes transferred to the output block).
- R2 = the address of the next argument in the argument block.

The user may call an appropriate output routine to output the converted/formatted data.

6.3.2 Examples

Examples of the use of the $EDMSG routine to convert and format output data are presented in this section.

EXAMPLE 1: Assume a program contains the following:

An input string (ISTRNG)

```
ISTRNG:  .ASCIZ  /%F%12S***TEXT***%3N%8S%VD%2N%12S***END****/
```

The argument block

```
ARGBLK: .WORD 3.
        .WORD 99.
        .WORD -37.
        .WORD 137.
```

The output block (OUTBLK)

```
OUTBLK:  .BLKB 100.
```

The source statements

```
       START:  MOV #OUTBLK,R0
               MOV #ISTRNG,R1
               MOV #ARGBLK,R2
               CALL $EDMSG
```

The following output will be produced.

```
           ***TEXT***

        99     -37     137

           ***END*****
```

The editing directives, shown above, and their effect are:

- %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 3 pairs of CR-LF characters in OUTBLK (generate two blank lines).
- %8S - insert 8 spaces in OUTBLK (indent the next line 8 spaces).
- %VD - use the first value (3) in ARGBLK as the repeat count and convert the next 3 binary values in ARGBLK to signed decimal; store each value, followed by a tab, in OUTBLK (output three signed decimal numbers in columnar format).
- %2N - insert 2 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****).

EXAMPLE 2: Assume a program contains the following:

An input string (INSTR)

```
INSTR:  .ASCIZ  /%F5S***B.  MABRY WORK REPORT FROM %Y TO %Y ***/
```

The argument block (IBLK)

```
IBLK:   .WORD 77.     ;YEAR
        .WORD 8.      ;8TH MONTH (AUG)
        .WORD 22.     ;DAY
        .WORD 77.     ;YEAR
        .WORD 9.      ;9TH MONTH (SEP)
        .WORD 16.     ;DAY
```

The output block (PRBLK)

```
PRBLK:  .BLKB 100.
```

The source statements

```
BEGIN:  MOV #PRBLK,R0
        MOV #INSTR,R1
        MOV #IBLK,R2
        CALL $EDMSG
        .
        .
        .
```

The following output will be produced:

```
***B.  MABRY WORK REPORT FROM 22-AUG-77 TO 16-SEP-77***
```

The editing directives in the example and their effect are:

%F - insert a form feed in PRBLK (start a new page).

%5S - insert 5 spaces in PRBLK and move ASCII string to PRBLK (indent the line 5 spaces and output the header ***B. MABRY 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-77 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-77*** in header line).

CHAPTER 7

DYNAMIC MEMORY MANAGEMENT ROUTINES

The dynamic memory management routines allow the user to manually manage 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, free dynamic memory is allocated 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 two-word listhead. Each free memory block contains a two-word control field, where:

- The first word contains the address of the next available block, or zero 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.

There are three routines that are called to perform dynamic memory management functions:

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

7.1 USAGE CONSIDERATIONS

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

- A two-word free memory listhead:

```
FREEHD:  .BLKW 2
```

- The appropriate call argument(s) and statement for the given routine, as described in Sections 7.2, 7.3, and 7.4 of this chapter.

At task build time, the user must specify the following

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

to include the memory management modules INIDM and EXTSK in the task.

7.2 INITIALIZE DYNAMIC MEMORY ROUTINE ($INIDM)

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.

To call the $INIDM routine:

- Specify, in Register 0, the address of the free memory listhead.
- Include the statement

```
CALL $INIDM
```

  in the source program.

The outputs from the $INIDM routine are:

- R0 = the first address in the task.
- R1 = the address following the task code (the first available address in the free dynamic memory).
- R2 = the size of the free dynamic memory.

For the RSX-11M system, the $INIDM routine deallocates any previous task memory extension made by issuing the EXTK$ executive directive or by calling the $EXTSK routine (see Chapter 8).

The $INIDM routine performs the following:

- Rounds the free dynamic memory base address to the next four-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 Register 0 - 1 and returns to the calling task.

Registers 3 - 5 are not used.

After the dynamic memory has been initialized, the user's task may call the system library routines: Request Core Block Routine ($RQCB) to allocate memory blocks in the dynamic memory; and subsequently call the Release Core Block Routine ($RLCB) to release the allocated blocks.

7.3 REQUEST CORE BLOCK ROUTINE ($RQCB)

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.

To use the $RQCB routine the user must initialize dynamic memory once (usually with the $INIDM routine) then:

- Supply two input arguments in the task's source code:
  - in Register 0, the address of the free memory listhead.
  - 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.)

- Include the statement

  ```
  CALL $RQCB
  ```

  in the source program.

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

The $RQCB routine returns the following output to the calling task:

- Successful allocation:
  - R0 = dynamic memory address of the allocated block.
  - R1 = the actual size of the allocated block (requested size rounded to next two-word boundary.
  - Condition Code C bit = clear.
- Unsuccessful allocation:
  - Condition Code C bit = set.

7.4 RELEASE CORE BLOCK ROUTINE ($RLCB)

The $RLCB system library routine releases a block of previously allocated dynamic memory to the free memory list, which is sequentially ordered by the memory addresses of the blocks themselves.

To call the $RLCB routine:

- Supply three input arguments:
  - in Register 0, the address of the free memory listhead.
  - in Register 1, the size (number of bytes) of the block to be released.
  - in Register 2, the memory address of the block to be released.
- Include the statement

  ```
  CALL $RLCB
  ```

  in the source program.

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

The output from the $RLCB routine is the released block. The $RLCB routine searches the free memory list until the proper address slot is found, then the released block is merged into the list. If possible, the memory block to be released is merged with adjacent blocks already in the free memory list.

CHAPTER 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 the user 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 the user may call the individual virtual memory management routines, greater efficiency is gained by using them as an automatic control system by calling only key routines, as follows:

- The Initialize Virtual Memory Routine ($INIVM), which initializes the task's dynamic memory and the disk work file, as described in Section 8.2.
- The virtual memory allocation routines: Allocate Virtual Memory Routine ($ALVRT) and Allocate Small Virtual Block Routine ($ALSVB), which manage the allocation of large and small page blocks to enable page swapping to and from dynamic memory, as described in Section 8.4.
- Four page management routines:
  - The Convert and Lock Page Routine ($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. (See Section 8.5.1.)
  - 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. (See Section 8.5.7.)

- The Convert Virtual to Real Address Routine ($CVRL), which converts a virtual address to a dynamic memory address. (See Section 8.5.2).

- The Write-marked Page Routine ($WRMPG), which sets the "written into" flag of memory pages. (See Section 8.5.5).

8.1 GENERAL USAGE CONSIDERATIONS

To call the virtual memory management routines, the user must provide the appropriate call arguments and statements in the source program, as described in Sections 8.2 through 8.5 of this chapter.

The user's task must contain an error handling routine, as described in Section 8.1.1.

At task build time, the user must specify the file and the virtual memory management modules required by the task, as described in Section 8.1.2.

8.1.1 User Error Handling Requirements

There are four virtual memory management routines that detect fatal error conditions for which a user-written error handling routine, entitled $ERMSG, is required. The routines are:

- Allocate Block Routine ($ALBLK) which is called by several other virtual memory management routines.

- Allocate Virtual Memory Routine ($ALVRT) which is called by the user or by the Allocate Small Virtual Block Routine ($ALSVB).

- Read Page Routine ($RDPAG) which is called by the Convert Virtual to Real Address Routine ($CVRL).

- Write Page Routine ($WRPAG) which is called by the Get Core Routine ($GTCOR).

In conjunction with the $ERMSG routine, the user must include definitions of three global error codes and one global severity code in the task. The symbolic error codes are:

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.)

The severity code is:

S$V2 (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 two input arguments:

```
Register 1 = low byte:  error code
             high byte:  severity code (always S$V2)

Register 2 = argument block address
```

and issues the call

```
CALL $ERMSG
```

If the user has not defined the error handling routine within the task, the undefined global symbol diagnostic message will be output at task build time.

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

The user's error handling routine should not attempt to return to the virtual memory management routine that detected the fatal error, since no meaningful output would result.

8.1.2 Task Building Requirements

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

Table 8-1
Content of the Virtual Memory Management Library File

Module Name		Name of Routine(s)	Routines Referenced
Statistical	Non-statistical		
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)
FNDPG	FNDPG	$FNDPG	(none)
GTCOS	GTCOR	$GTCOR	$EXTSK, $WRPAG
INIDM*	INIDM*	$INIDM	$EXTSK
INIVS	INIVM	$INIVM	$ALBLK, $GTCOR, $EXTSK, $WRPAG
MRKPG	MRKPG	$LCKPG $UNLPG $WRMPG	$FNDPG $FNDPG $FNDPG
RDPAS	RDPAG	$RDPAG $WRPAG	(none)
RQVCB	RQVCB	$RQVCB	(none)
VMUTL	VMUTL	$CVLOK	$CVRL, $LCKPG, $FNDPG, $ALBLK, $RDPAG
VMDAS	VMDAT	Global data storage module	

* The INIDM module is a dynamic memory management module (see Chapter 7) that normally is used in conjunction with the virtual memory management routines.

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

- The INIVS modules, which initializes three double-word fields: the total work file access field ($WRKAC); the work file read count field ($WRKRD); and 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 the user's 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, the user must 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. Following is a specification that identifies 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 non-statistical routines use the global data storage module, VMDAT. To use the non-statistical routines, the user must specify, at task build time, the virtual memory management routines library file, the names of all non-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. Following is a specification that identifies all modules of the non-statistical 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
```

8.2 VIRTUAL MEMORY INITIALIZATION ROUTINE ($INIVM)

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

To use the $INIVM routine:

- Define and initialize a two-word field named $FRHD. To define the field, include the code

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

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

- Define three global symbols in the source program:

  W$KLUN — Logical unit number (LUN) to be used for the work file. The user is also responsible for assigning 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 at least the number of pages specified, 512 words of dynamic memory will be set aside to speed up the searching of memory-resident pages.

- Specify, in Register 1, the highest address of the task's free dynamic memory.

- Include the statement

  ```
  CALL $INIVM
  ```

  in the source program.

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

The outputs from the $INIVM routine are:

- Initialization successful:

  Condition Code C bit = clear.

  and

  R0 = 0.

- Initialization failure:

  Condition Code C bit = set.

  and

  R0 = -2 to indicate work file open failure.
  R0 = -1 to indicate work file mark-for-deletion failure.

  Also, the FCS error code may be examined at offset F.ERR in the work file FDB. The address of the FDB is stored in the word $WRKPT.

The interaction of the $INIVM routine with the user's task and the Allocate Block Routine ($ALBLK) is shown in Figure 8-1 and described below.

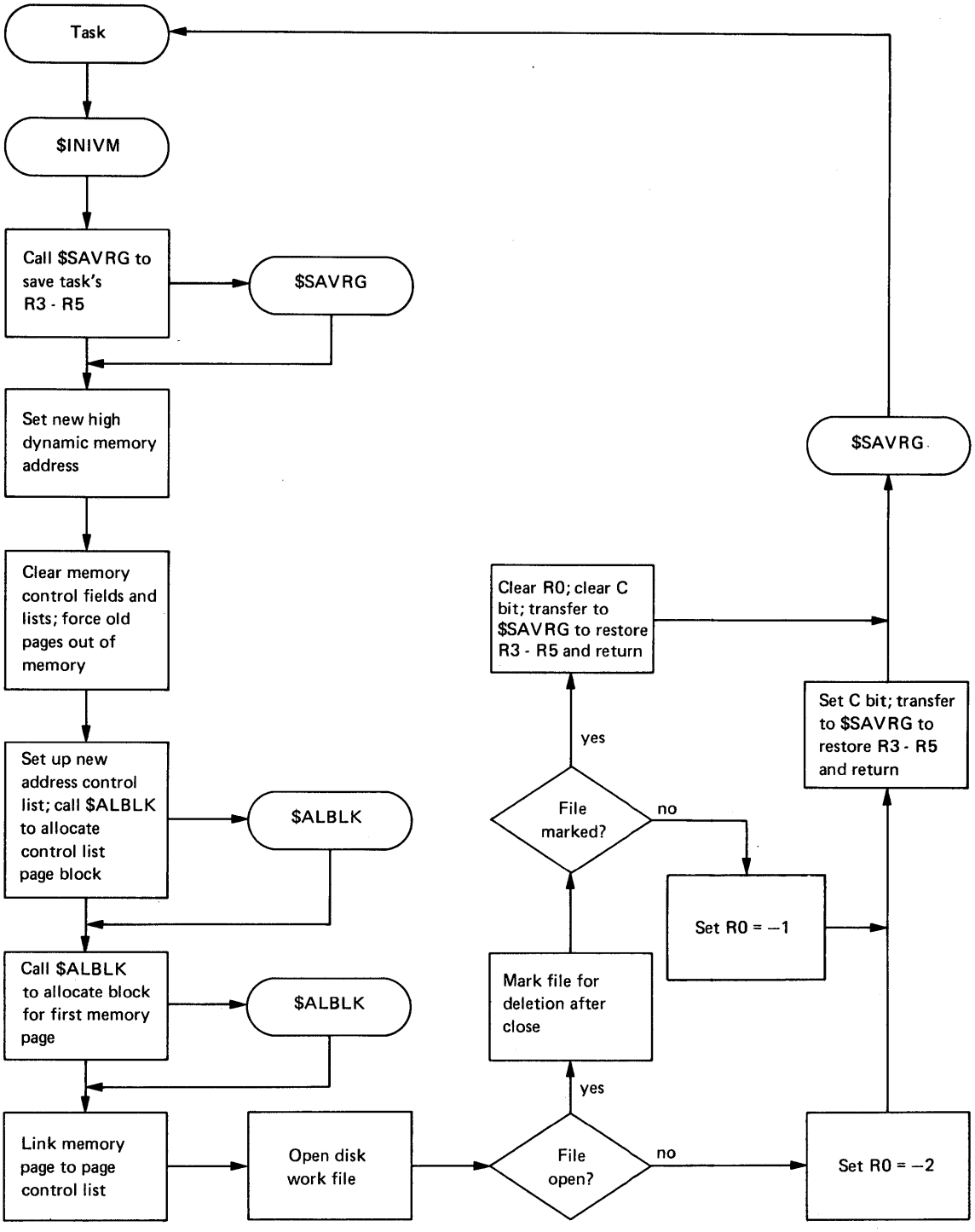

Figure 8-1 General Block Diagram of the $INIVM Routine

The $INIVM routine calls the $SAVRG routine to save and subsequently restore registers 3 - 5 of the calling task.

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 then calls the $ALBLK routine to allocate a page block for the first memory page for the calling task. The $INIVM routine links the first allocated page to the page control list.

The $INIVM routine initializes (opens) the user's disk work file. If the file is opened successfully, the $INIVM routine attempts to mark it for deletion. This ensures that the file will be deleted automatically when it is closed, or if the task terminates abnormally or exits. Note that the work file may be closed by the following operation: CLOSE$ $WRKPT. If the routine is successful, the C bit in the Condition Code is cleared, and control is transferred to the $SAVRG routine to restore registers R3 - R5 and return to the calling task.

If the work file is not opened successfully, the $INIVM routine sets the C bit in the Condition Code, sets R0 = -2, and transfers to the $SAVRG routine.

If the work file is not successfully marked for deletion, the $INIVM routine sets the C bit in the Condition Code, sets R0 = -1, and transfers to the $SAVRG routine.

The $SAVRG routine restores registers R3 - R5 and returns to the calling task. The original content of R0 - R2 is destroyed.

8.3 CORE ALLOCATION ROUTINES

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:

- 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 effect allocation of 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 Extend Task Routine ($EXTSK), which is called by the $GTCOR routine to extend the size of the task region to make enough memory available in the free dynamic memory to satisfy the allocation request.

- The Write Page Routine ($WRPAG), which is called by the $GTCOR routine to transfer memory pages to the user's 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.

The processing performed by the core allocation routines is described in the following sections.

8.3.1 Allocate Block Routine ($ALBLK)

The $ALBLK routine determines whether a block of memory storage can be allocated from the free dynamic memory. If so, the $ALBLK routine clears (zeroes) 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. In this case, the $ALBLK routine sets the error/severity code E$R4,S$V2 in Register 1, the address of the argument block $FRHD (free memory header) in Register 2, and calls the user's $ERMSG routine (see Section 8.1.1).

To use the $ALBLK routine:

- Input, in Register 1, the size (number of bytes less than or equal to 512(10)) of the memory storage block to be allocated.

- Include the statement

  ```
  CALL $ALBLK
  ```

  in the source program.

If allocation is successful, the output from the $ALBLK routine is

R0 = the dynamic memory address of the allocated, cleared, block

and control returns to the caller. If allocation is unsuccessful, the user's $ERMSG routine is called.

The interaction of the $ALBLK routine with the user's task and other virtual memory management routines is diagrammed in Figure 8-2 and described briefly below.

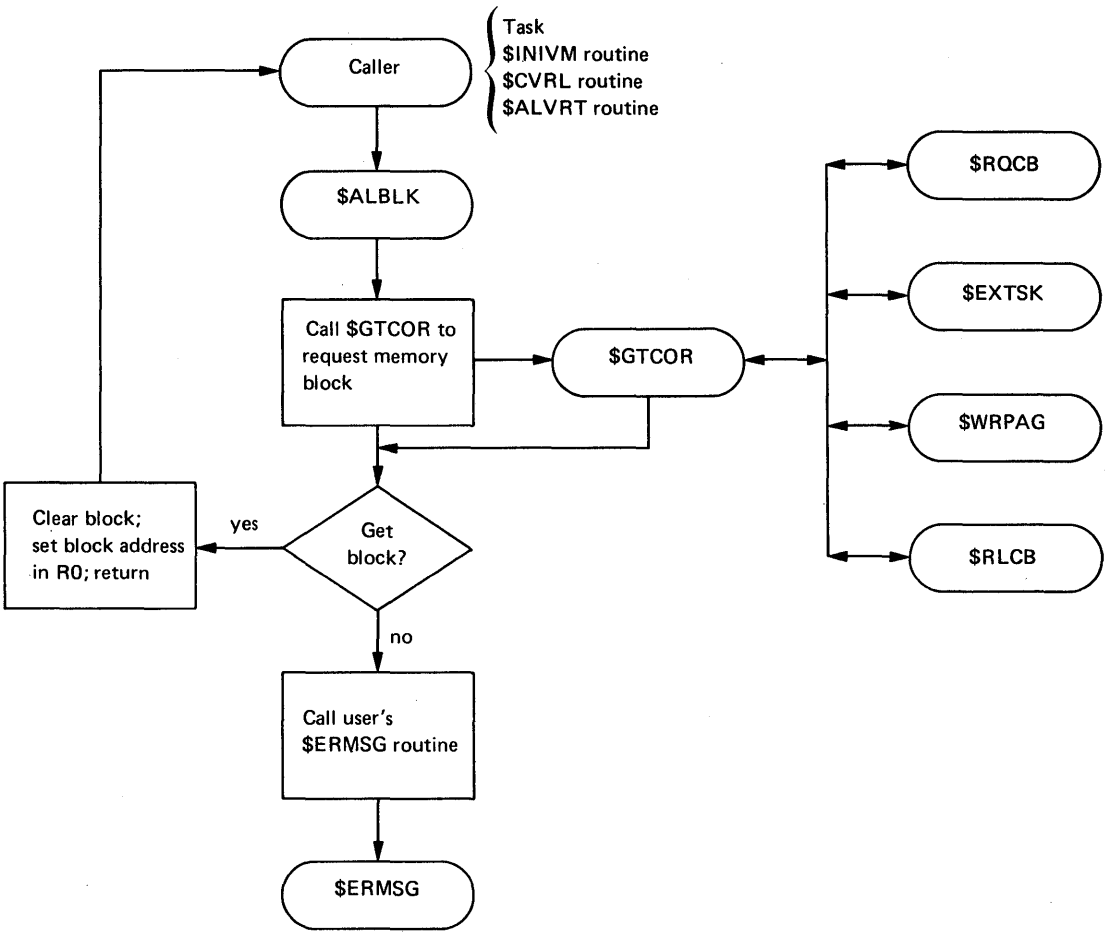

Figure 8-2 General Block Diagram of the $ALBLK Routine

In addition to the user's 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.

The $ALBLK routine always calls the Get Core Routine ($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.

8.3.2 Get Core Routine ($GTCOR)

The $GTCOR routine attempts to allocate requested dynamic memory blocks in three ways:

- Allocation from the currently available space in the free dynamic memory.
- Extension of the task region to increase the size of the free dynamic memory to accommodate the allocation request.
- Swapping unlocked page blocks from dynamic memory to disk to free previously allocated memory space for reallocation.

To call the $GTCOR routine:

- Input, in Register 1, the size (number of bytes less than or equal to 512(10)) of the dynamic memory block to be allocated.
- Include the statement

  CALL $GTCOR

  in the source program.

NOTE: If the statistical module (GTCOS) is to be specified at task build time, the source program must define and initialize the $FRSIZ field for use by the $GTCOR routine. (See Section 8.1.2).

The outputs from the $GTCOR routine are:

- R0 = 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.

The interaction of the $GTCOR routine with other system library and virtual memory management routines is diagrammed in Figure 8-3 and described briefly below.

The $GTCOR routine is always called by the Allocate Block Routine ($ALBLK). The $GTCOR routine calls the $SAVRG routine to save and subsequently restore registers 3 - 5 of the caller.

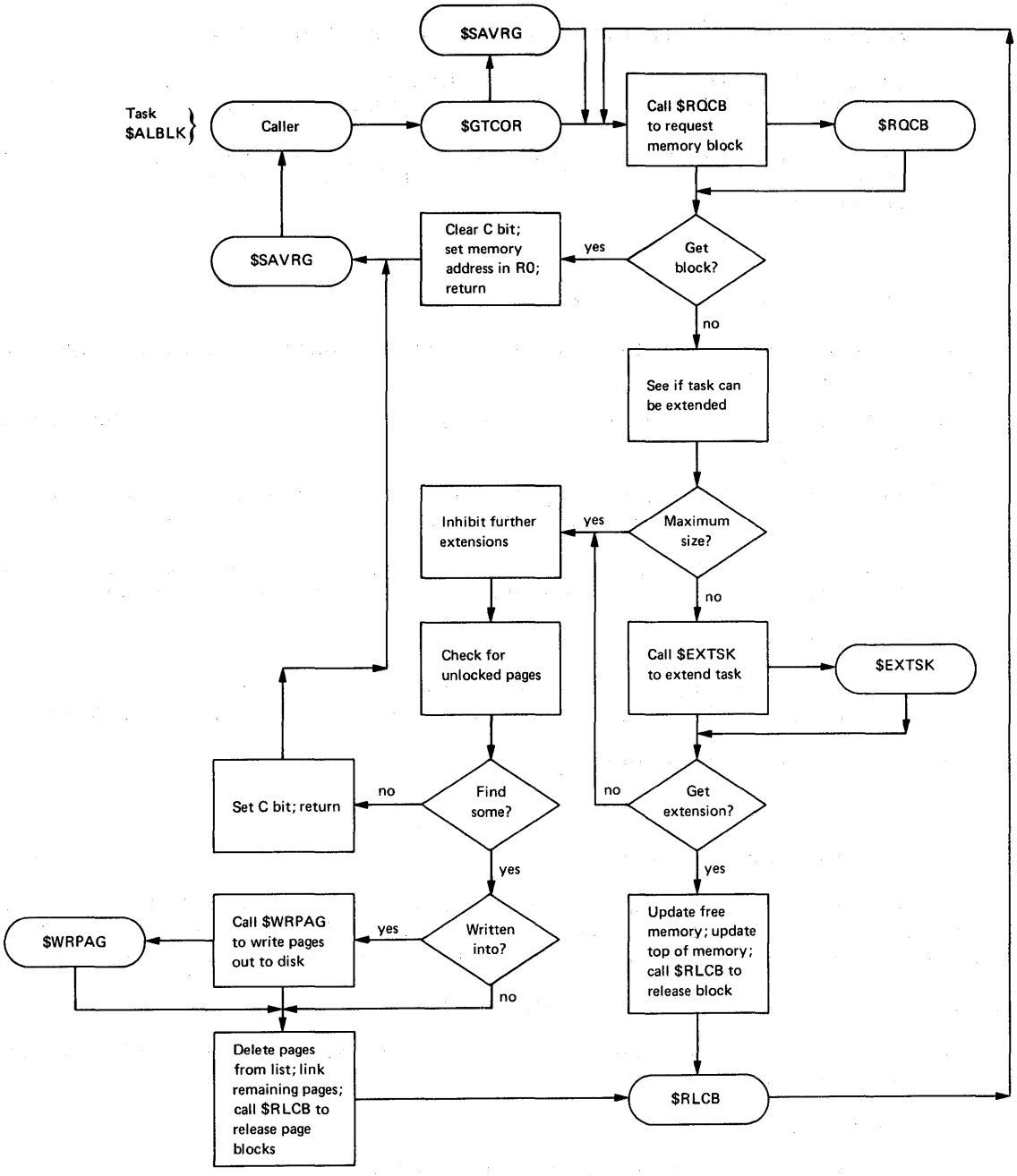

Figure 8-3 General Block Diagram of the $GTCOR Routine

The system library 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 system library 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 R0 to the caller. If the $GTCOR routine is not able to obtain sufficient memory for the requested block, the C bit in the Condition Code is set and control is returned to the caller.

8.3.3 **Extend Task Routine ($EXTSK)**

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

To call the $EXTSK routine:

- Input, in Register 1, the size (number of bytes less than or equal to 512(10)) of the memory storage block to be allocated.
- Include the statement

  ```
  CALL $EXTSK
  ```

  in the source program.

All registers of the caller are preserved except R1, which contains output from the $EXTSK routine, as follows:

- R1 = 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.

The interaction of the $EXTSK routine with the $GTCOR routine is shown in Figure 8-4 and described below.

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, the task region is extended and the total amount of the extension is returned, in Register 1, to the $GTCOR routine. If the task region cannot be extended, the $EXTSK routine sets the C bit in the Condition Code and returns to the $GTCOR routine.

The $EXTSK routine may be called directly by the user. The routine is called by the Initialize Dynamic Memory Routine ($INIDM), described in Chapter 7.

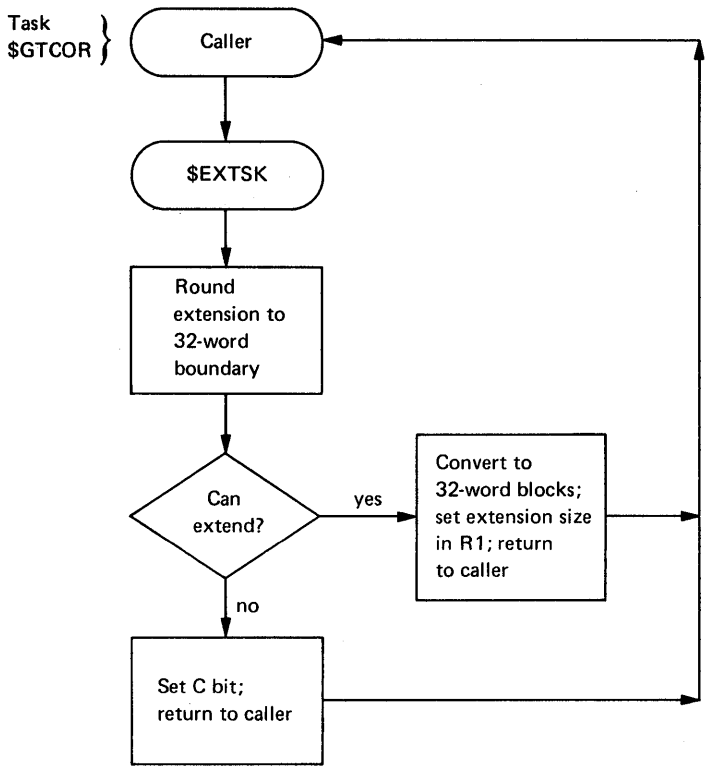

Figure 8-4 General Block Diagram of the $EXTSK Routine

8.3.4 Write Page Routine ($WRPAG)

The $WRPAG routine effects the transfer of a memory page to the disk work file.

To call the $WRPAG routine:

- Input, in Register 2, the dynamic memory address of the page to be transferred.
- Include the statement

  CALL $WRPAG

  in the source program.

The output from the $WRPAG routine is the transferred page, with successful transfer indicated by a cleared C bit in the Condition Code when control returns to the caller. If the page transfer is unsuccessful, the $WRPAG routine sets the error/severity code E$R73,S$V2 in Register 1, and calls the user's $ERMSG routine (see Section 8.1.1).

The interaction of the $WRPAG routine with the $GTCOR routine is shown in Figure 8-5 and described below.

The $WRPAG routine is called by the Get Core Routine ($GTCOR) when a resident memory page that has been written into is to be transferred to the user's disk work file.

The $WRPAG routine calls the $SAVVR routine to save and subsequently restore the caller's registers 0 - 2. The routine then:

- 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 the user's $ERMSG routine if a fatal work file I/O error prevented the page transfer.

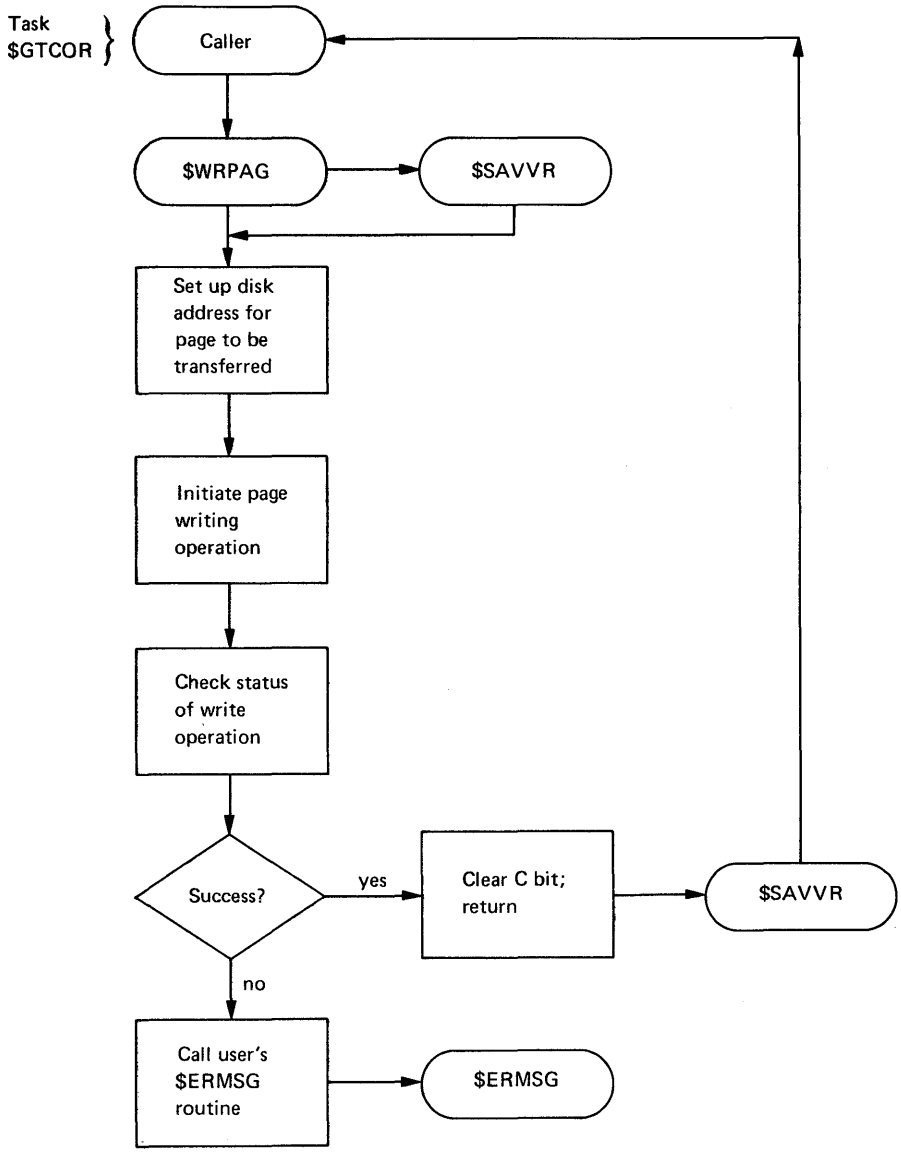

Figure 8-5 General Block Diagram of the $WRPAG Routine

8.4 VIRTUAL MEMORY ALLOCATION ROUTINES

The virtual memory allocation routines manage the allocation of disk and memory storage to enable page swapping from the free dynamic memory to the user's disk work file. There are three virtual memory allocation routines:

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

The processing performed by the virtual memory allocation routines is described in the following sections.

8.4.1 Allocate Virtual Memory Routine ($ALVRT)

The $ALVRT routine determines whether a page block of virtual storage can be allocated on the user's disk work file. If so, the $ALVRT routine ensures that an equal amount of memory storage is allocated, updates page control and address tables, and returns the disk and memory addresses of the allocated page blocks to the caller. If the $ALVRT routine cannot allocate the requested storage, the error/severity code E$R76,S$V2 is stored in Register 1 and the user's $ERMSG routine (see Section 8.1.1) is called.

To call the $ALVRT routine:

- Input, in Register 1, the size (number of bytes less than or equal to 512(10)) of the disk storage block to be allocated.
- Include the statement

```
CALL $ALVRT
```

  in the source program.

NOTE: The maximum size of a page block is 512(10) bytes.

If allocation is successful, the $ALVRT routine returns the outputs:

- R0 = memory address of allocated page block.
- R1 = disk address of allocated page block.

The $ALVRT routine calls the $SAVRG routine to save and subsequently restore registers 3 - 5 of the caller. Register 2 is destroyed.

The interaction of the $ALVRT routine with the user's task and other virtual memory management routines is shown in Figure 8-6 and described below.

In addition to the user's task, the $ALVRT routine is called by the Allocate Small Virtual Block Routine ($ALSVB).

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 the user's $ERMSG routine.

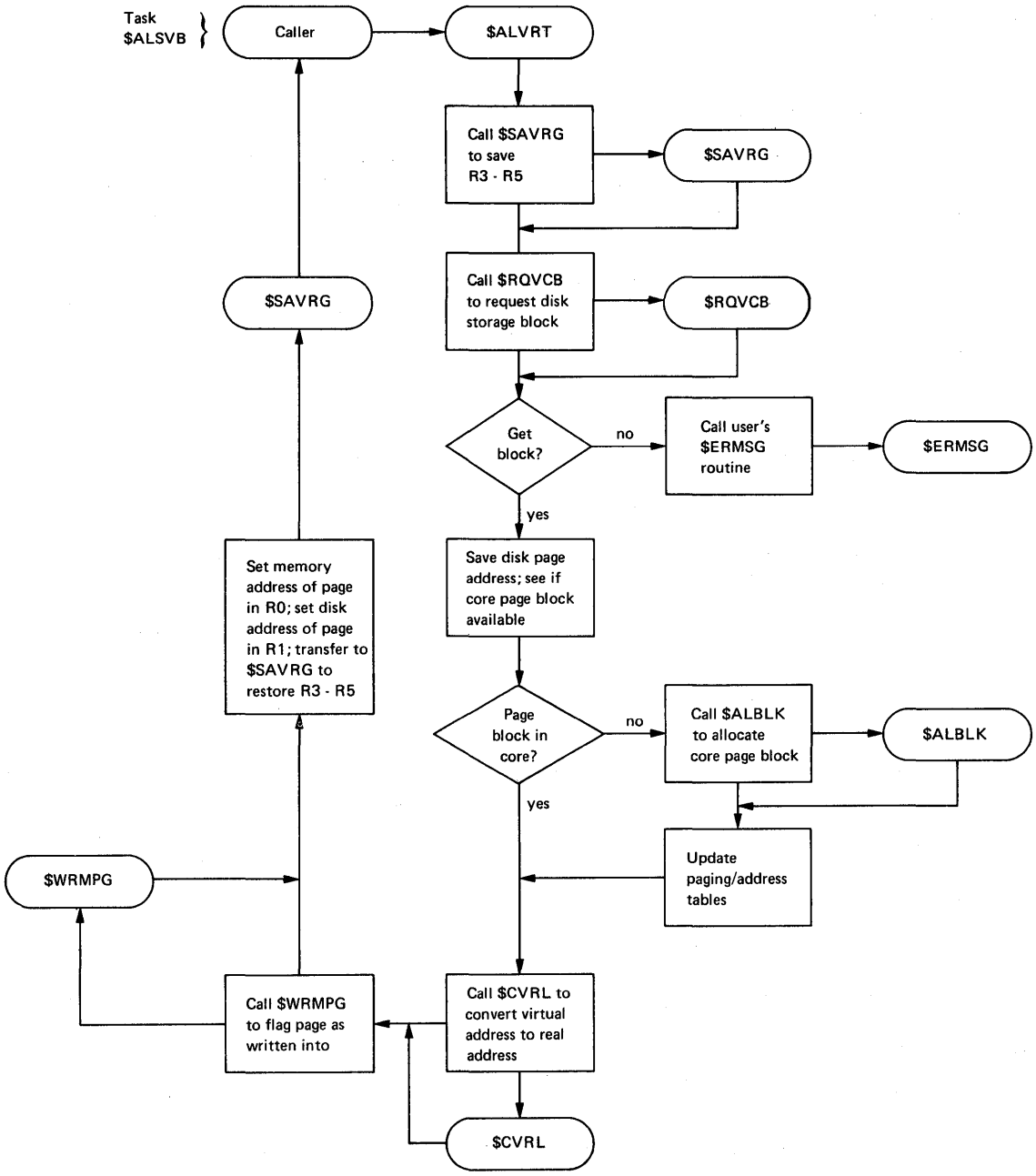

Figure 8-6 General Block Diagram of the $ALVRT Routine

If the disk storage can be allocated, 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.

8.4.2 Allocate Small Virtual Block Routine ($ALSVB)

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 sub-allocation 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.

To call the $ALSVB routine:

- Define the following in the source program:

  ```
  N$DLGH .WORD 512.
  ```

  Normally, this is the size of a large memory block. In any case, it must be less than or equal to 512(10).

- Specify, 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.

  NOTE: the maximum size of a page block is 512(10) bytes.

- Include the statement

  ```
  CALL $ALSVB
  ```

  in the source program.

The outputs from the $ALSVB routine are:

- R0 = the dynamic memory address of the allocated page block.
- R1 = the virtual address of the allocated block.

Registers 3 - 5 are preserved. Register 2 is destroyed.

The interaction of the $ALSVB routine with other virtual memory management routines is diagrammed in Figure 8-7, and described briefly below.

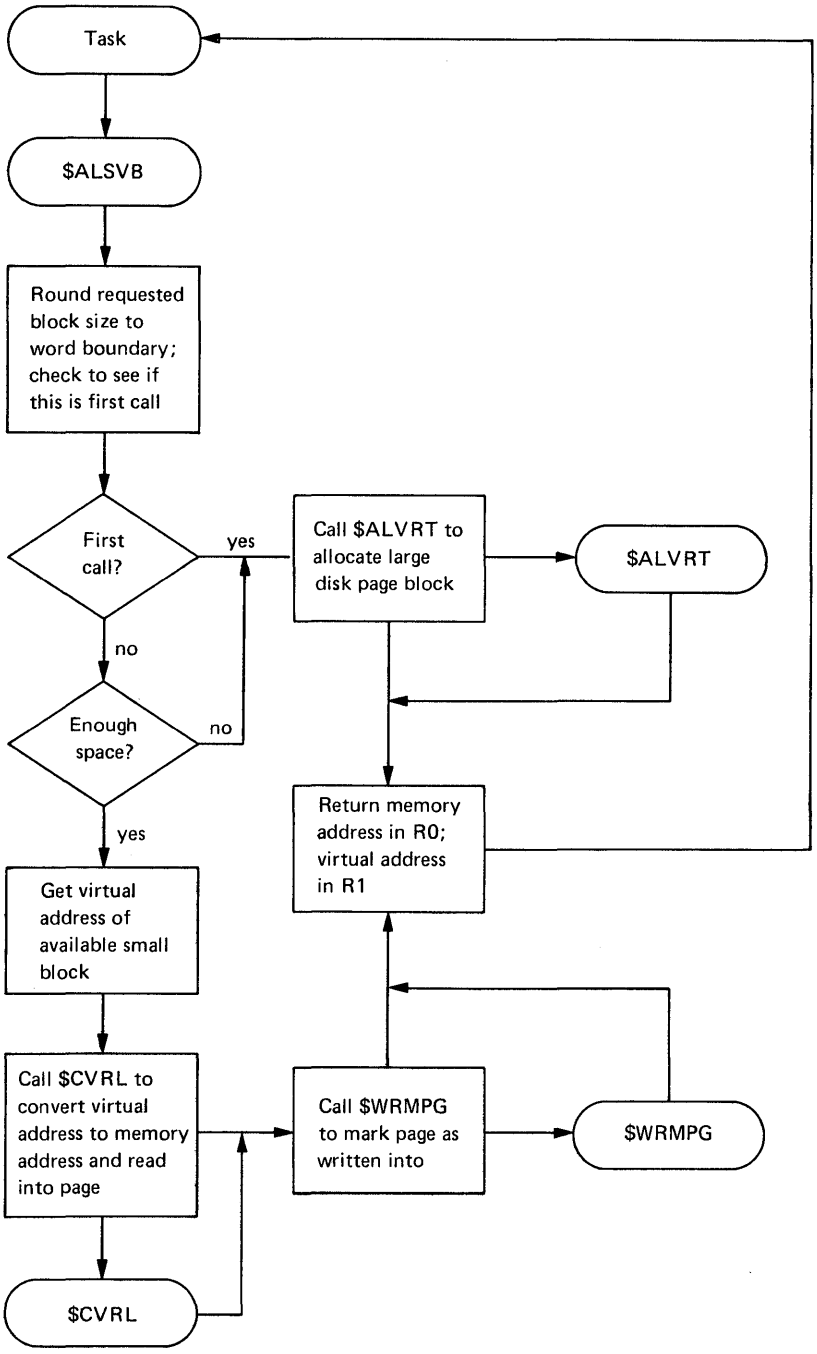

Figure 8-7 General Block Diagram of the $ALSVB Routine

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 perform the following:

- Locate the allocated large page, if it is memory resident. If 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 effect:

- Disk and dynamic memory allocation of the requested large page block.
- Virtual address conversion to a memory address.
- Transfer of the large block, if necessary, from disk to dynamic memory.
- Setting the "written into" flag of the allocated page block.

8.4.3 Request Virtual Core Block Routine ($RQVCB)

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

The interaction of the $RQVCB routine with the $ALVRT routine is diagrammed in Figure 8-8 and described briefly below.

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. Two conditions cause allocation failure:

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

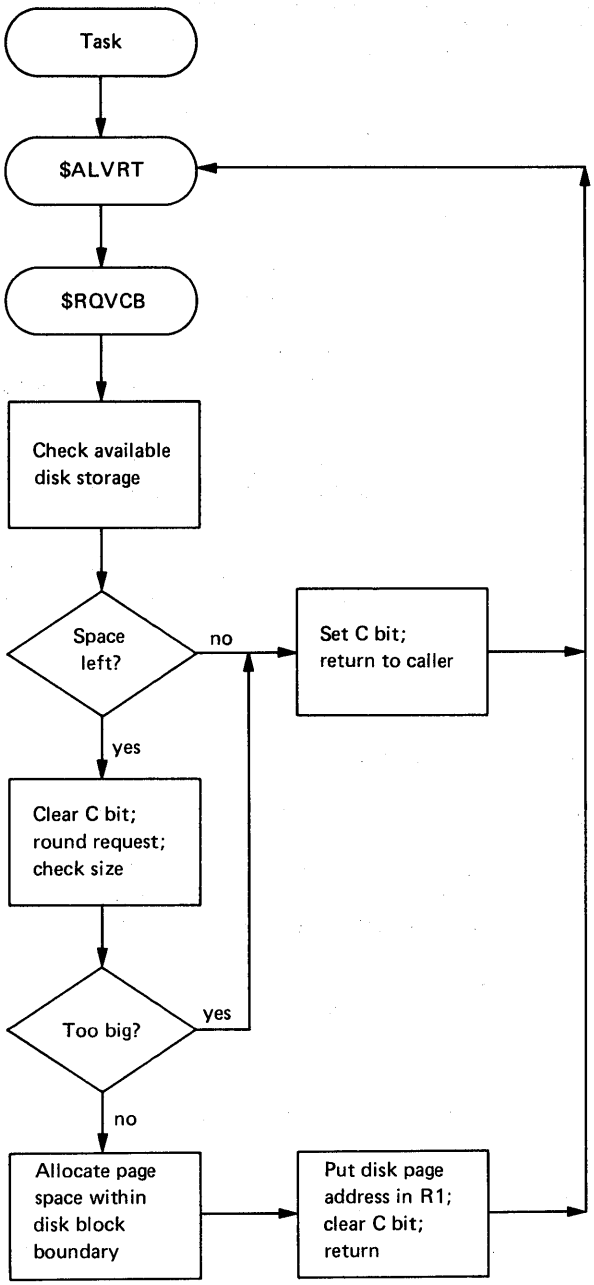

Figure 8-8 General Block Diagram of the $RQVCB Routine

8.5 PAGE MANAGEMENT ROUTINES

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

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

The processing performed by the page management routines is described in the following sections.

8.5.1 Convert and Lock Page Routine ($CVLOK)

The $CVLOK routine performs two functions:

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

To call the $CVLOK routine:

- Specify, in Register 1, the virtual address to be converted.
- Include the statement

  CALL $CVLOK

  in the source program.

Outputs from the $CVLOK routine are:

- R0 = converted memory address.
- R1 = virtual address.
- Condition Code:

  C bit = clear if the address was converted and the page locked.

  C bit = set if address conversion or page locking failed.

The contents of R2 are preserved. Registers 3 - 5 are preserved by the $CVRL routine.

The interaction of the $CVLOK routine with the calling task and other page management routines is shown in Figure 8-9.

The $CVLOK routine calls:

- The Convert Virtual to Real Address Routine ($CVRL) to convert the virtual address to a memory address.
- The Lock Page Routine ($LCKPG) to lock the page in memory.

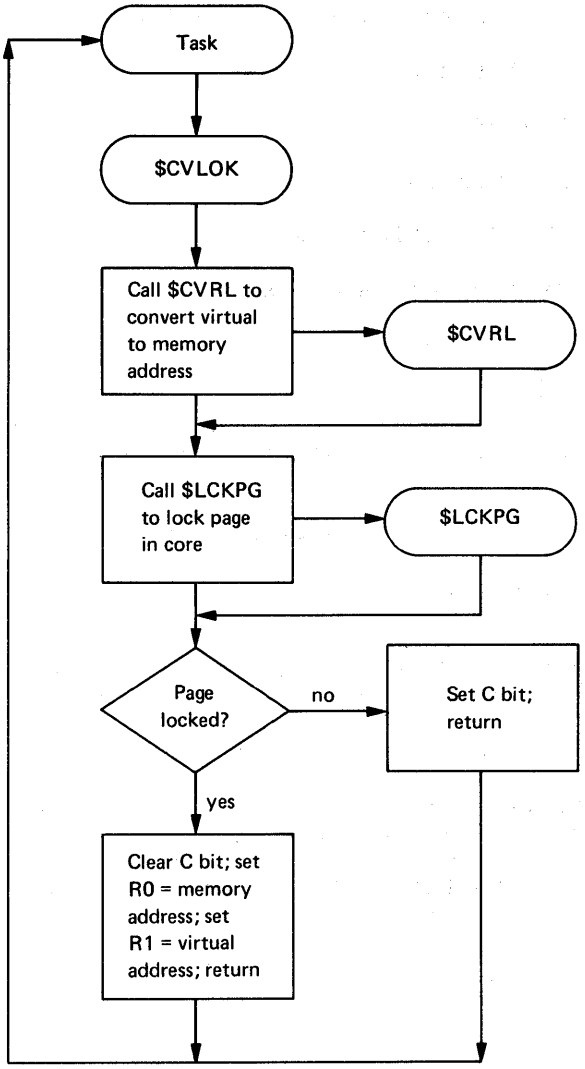

Figure 8-9 General Block Diagram of the $CVLOK Routine

8.5.2 Convert Virtual to Real Address Routine ($CVRL)

The $CVRL routine converts a virtual address to a dynamic memory address.

To call the $CVRL routine:

- Input, in Register 1, the virtual address.
- Include the statement

```
CALL $CVRL
```

  in the source program.

The output from the $CVRL routine is:

R0 = memory address.

The $SAVRG routine is called to save and restore registers 3 - 5. Register 1 is unchanged. Register 2 is destroyed.

The interaction of the $CVRL routine with the caller and other virtual memory management routines is shown in Figure 8-10 and described briefly below.

The $CVRL routine may be called by the user's task, and 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, the Allocate Block Routine ($ALBLK) is called 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 R0, and control is transferred to the $SAVRG routine, which restores registers 3 - 5 and returns to the caller.

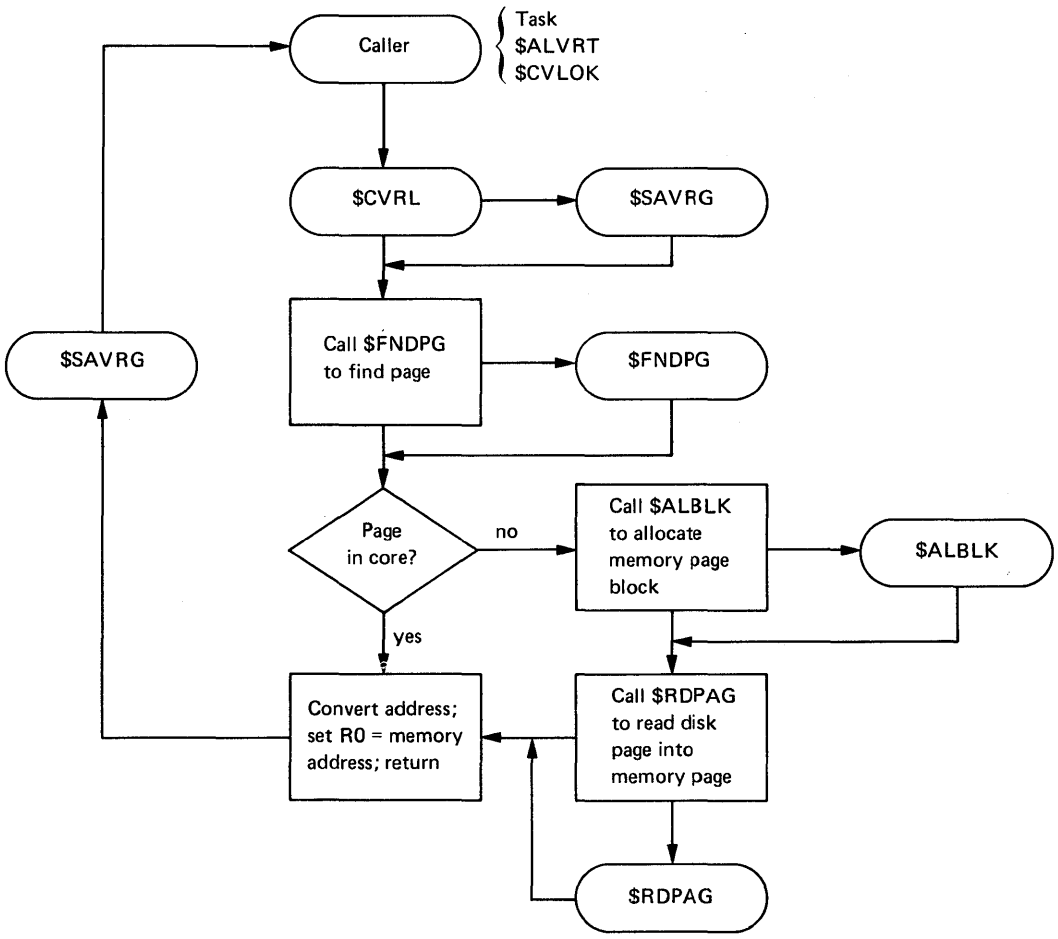

Figure 8-10 General Block Diagram of the $CVRL Routine

8.5.3 Read Page Routine ($RDPAG)

The $RDPAG routine effects the transfer of a disk page from the work file to the dynamic memory.

To call the $RDPAG routine:

- Input, in Register 0, the disk address of the page to be transferred.
- Include the statement

  ```
  CALL $RDPAG
  ```

  in the source program.

The output from the $RDPAG routine is the transferred page, indicated by a cleared C bit in the Condition Code when control returns to the caller. If the page transfer is unsuccessful, the $RDPAG routine sets the error/severity code E$R73,S$V2 in Register 1 and calls the user's $ERMSG routine (see Section 8.1.1).

The interaction of the $RDPAG routine with the task and the $CVRL routine is shown in Figure 8-11 and described below.

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 calls the $SAVVR routine to save and subsequently restore the caller's registers 0 - 2. The routine then:

- Sets up the address of the page to be transferred.
- Initiates the page reading operation.
- Checks the status of the read operation.
- Indicates a successful transfer (clears the C bit in the Condition Code) and returns control to the $SAVVR routine, or calls the user's $ERMSG routine if a fatal work file I/O error prevented the page transfer.

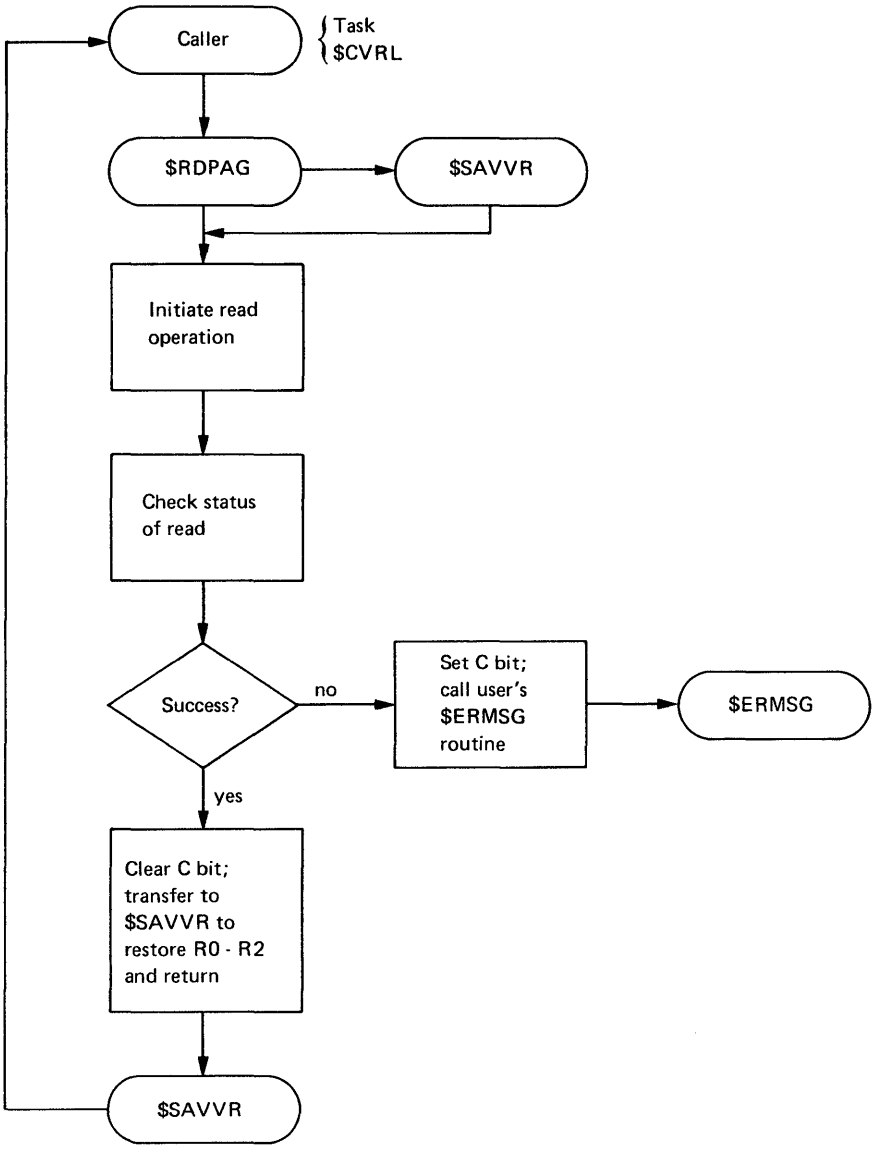

Figure 8-11 General Block Diagram of the $RDPAG Routine

8.5.4 Find Page Routine ($FNDPG)

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.

To call the $FNDPG routine:

- Specify, in Register 1, the virtual page address.
- Include the statement

  CALL $FNDPG

  in the source program.

The outputs from the $FNDPG routine are:

- R0 = the memory page block address in which the page is resident.
- Condition Code:

  C bit = cleared if page is resident.

  C bit = set if page was not found.

The content of Register 1 is not changed.

The interaction of the $FNDPG routine with the user's task and the page management routines is shown in Figure 8-12 and described below.

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 time-stamped, 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 C bit in the Condition Code is set, and control returns to the caller.

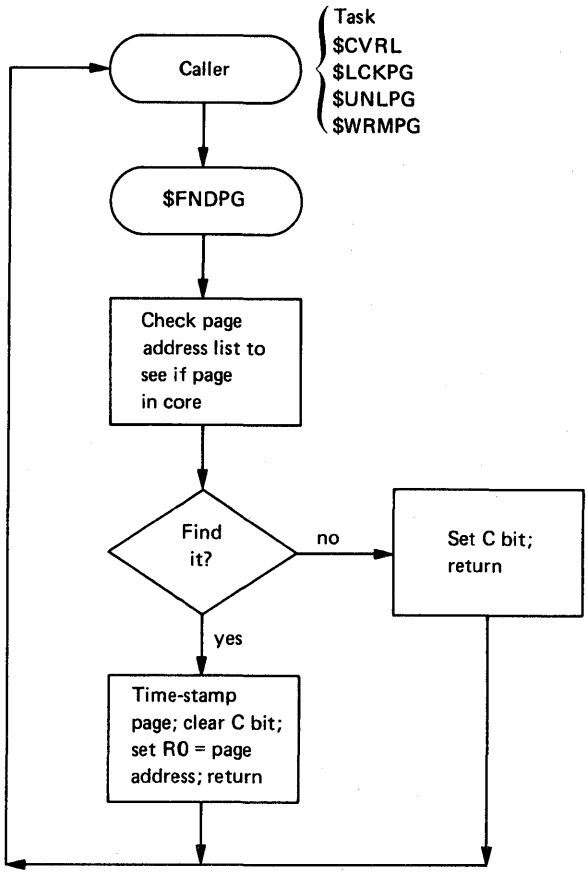

Figure 8-12 General Block Diagram of the $FNDPG Routine

8.5.5 Write-marked Page Routine ($WRMPG)

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

To call the $WRMPG routine:

- Specify, in Register 1, the virtual address in the page.
- Include the statement

  CALL $WRMPG

  in the source program.

The output from the $WRMPG routine is a Condition Code setting:

C bit = cleared to indicate that the page was write marked successfully.

C bit = set to indicate that the specified memory page was not resident in the task's free dynamic memory.

The interaction of the $WRMPG routine with the caller and virtual memory management routines is shown in Figure 8-13 and described below.

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.

The $WRMPG routine calls the $SAVVR routine to save and subsequently restore registers 0 - 2 of the caller.

The Find Page Routine ($FNDPG) is called 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 - 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 cleared, and control is transferred to the $SAVVR routine to restore R0 - R2 and return to the caller.

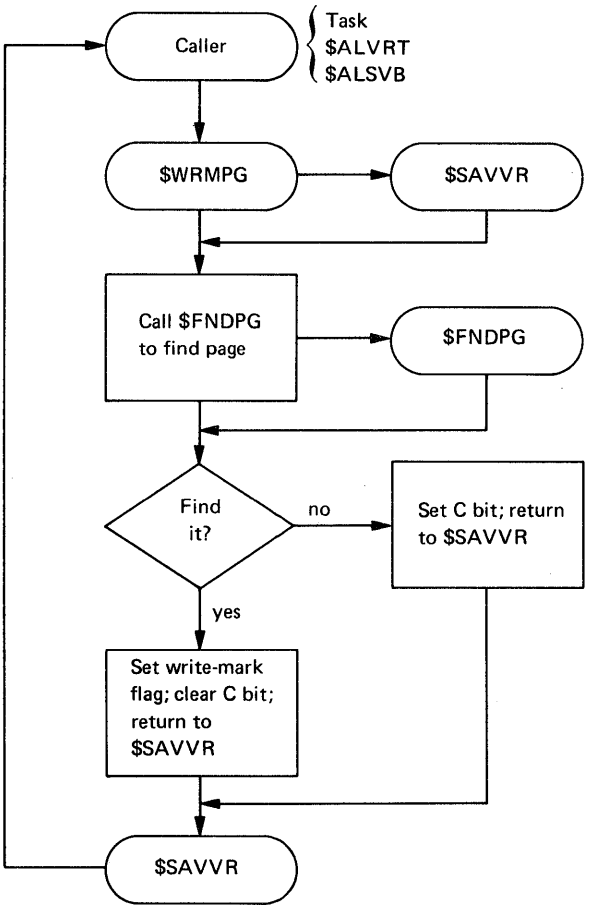

Figure 8-13 General Block Diagram of the $WRMPG Routine

8.5.6 Lock Page Routine ($LCKPG)

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.

To call the $LCKPG routine:

- Specify, in Register 1, a virtual address in the page to be locked in dynamic memory.
- Include the statement

  ```
  CALL $LCKPG
  ```

  in the source program.

The output from the $LCKPG routine is a Condition Code setting:

C bit = cleared if the page was locked in memory.

C bit = set if the page was not found.

The interaction of the $LCKPG routine with the task and page management routines is shown in Figure 8-14 and described briefly below.

The $LCKPG routine may be called by the user's task and by the Convert and Lock Page Routine ($CVLOK).

The $LCKPG routine calls the $SAVVR routine to save and subsequently restore the caller's registers 0 - 2.

The Find Page Routine ($FNDPG) is called 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 R0 - R2 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, via the $SAVVR routine, to the caller.

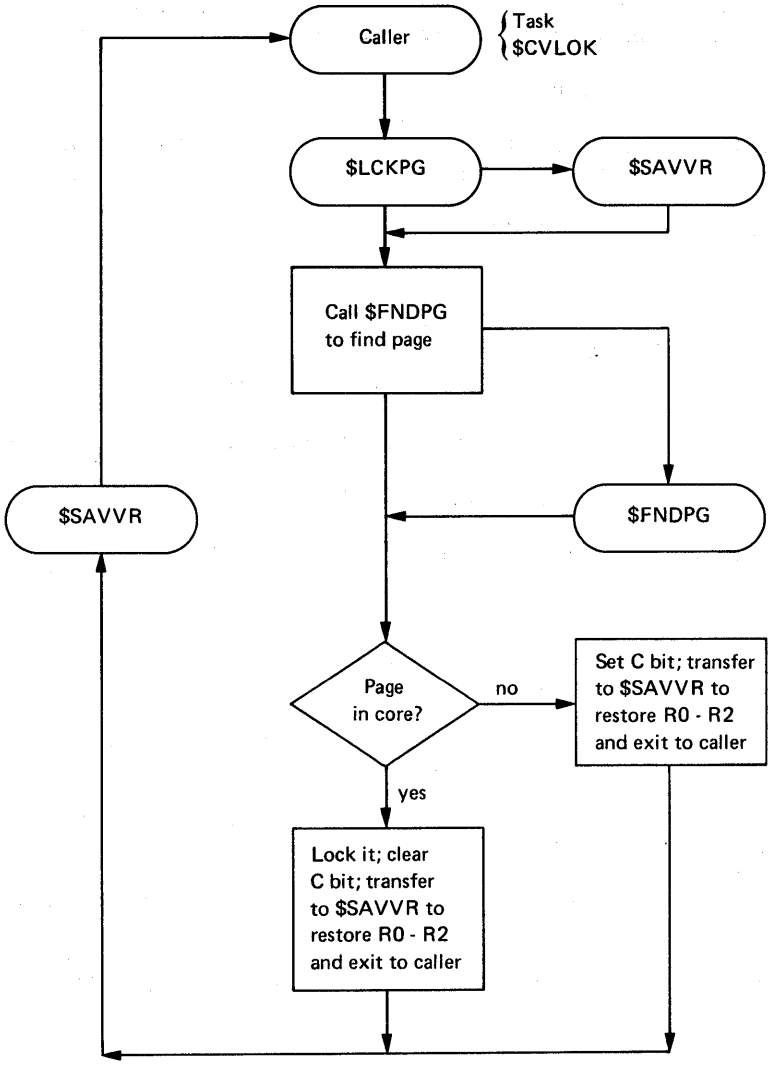

Figure 8-14 General Block Diagram of the $LCKPG Routine

8.5.7 Unlock Page Routine ($UNLPG)

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.

To call the $UNLPG routine:

- Specify, in Register 1, the virtual address in the page to be unlocked.
- Include the statement

  CALL $UNLPG

  in the source program.

The output from the $UNLPG routine is a Condition Code bit setting:

C bit = cleared if the page was unlocked.

C bit = set if the page was not found.

The interaction of the $UNLPG routine with the task is shown in Figure 8-15 and described briefly below.

The $UNLPG routine calls the $SAVVR routine to save and subsequently restore the caller's registers 0 - 2.

The Find Page Routine ($FNDPG) is called 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 R0 - R2 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, via the $SAVVR routine, to the caller.

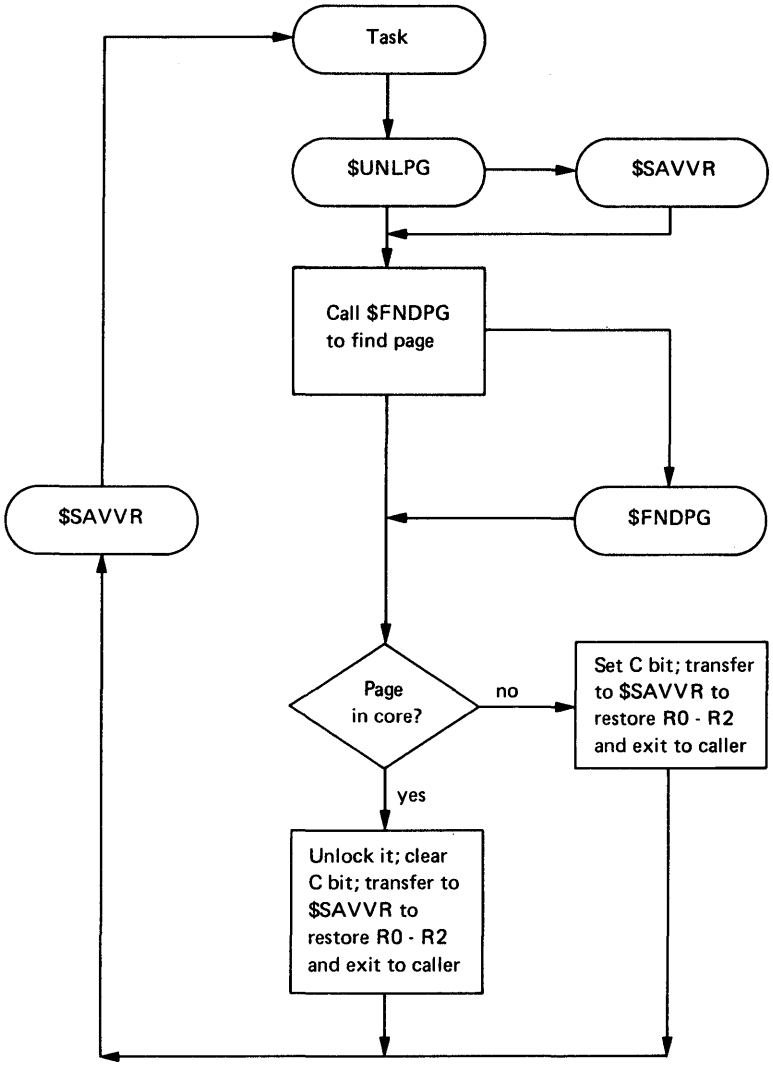

Figure 8-15 General Block Diagram of the $UNLPG Routine

CHAPTER 9

SUMMARY PROCEDURES

The procedures for using the system library routines are summarized in tabular format 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 preceding chapters of this manual.

Table 9-1
Register Handling Routines Summary

Routine Name/ Mnemonic	Function	Call Statement
Save All Registers $SAVAL	Saves/restores R0 - R5	JSR PC,$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	Outputs
Integer Multiply $MUL	R0 = multiplier R1 = multiplicand CALL $MUL	Product: R0 = high order part R1 = 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	Product: R0 = high order part R1 = low order part C = clear R4-R5 preserved R2-R3 destroyed
Double-precision Divide $DDIV	R0 = unsigned divisor Dividend: R1 = high order part R2 = low order part	R0 = remainder Quotient: R1 = high order part R2 = low order part R3 preserved
NOTE: The arithmetic routines accept unsigned inputs and produce unsigned results.		

Table 9-3
Input Data Conversion Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Outputs
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	Outputs
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

(continued on next page)

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

Routine Name/ Mnemonic	Input Arguments and Call Statement	Outputs
General Purpose Binary to ASCII $CBTA	R0 = output address R1 = binary value R2 = conversion parameters: bits 0-7: radix (2.-10.) bit 8: = 0 = unsigned value = 1 = signed value bit 9: = 0 = zero suppress = 1 = no zero suppress bit 10: = 1, replace leading zeroes with blanks. = 0, do not replace leading zeroes 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	Outputs
Upper Case Text $CVTUC	R0 = input address R1 = output address R2 = number input bytes CALL $CVTUC	Converted text at output address R3-R5 not used R2 destroyed R0-R1 not altered
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
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

(continued on next page)

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

Routine Name/ Mnemonic	Input Arguments and Call Statement	Outputs
Edit Message $EDMSG	Define ASCIZ input string with directives in the form: %l %nl %Vl where n = optional decimal repeat count; V specifies an optional value to be used as a repeat count; and l = one of the following: 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 M = Binary to decimal magnitude conversion, zero suppression N = New line insertion O = Binary to signed octal conversion P = Binary to octal magnitude conversion 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 = File name 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	Converted/formatted data in output block R0 = address of last byte in output block R1 = number of bytes in output block R2 = address of next argument in argument block R3-R5 preserved

Table 9-6
Dynamic Memory Management Routines Summary

Routine Name/ Mnemonic	Input Arguments and Call Statement	Outputs
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	Outputs
Initialize Virtual Memory $INIVM	Define $FRHD block with first address of free memory Define 3 global symbols: W$KLUN (work file LUN); W$KEXT (work file extension size); N$MPAG (fast page search page count) R0 = free memory highest address CALL $INIVM	Successful: C = clear and R0 = 0 Failure: C = set and R0 = -2, file not opened R0 = -1, file not marked R3-R5 preserved Original content 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 Failure: 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
Allocate Small Virtual Block $ALSVB	Define N$DLGH .WORD 512. 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

(continued on next page)

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

Routine Name/ Mnemonic	Input Arguments and Call Statement	Outputs
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

APPENDIX A

SYSTEM REFERENCE BIBLIOGRAPHY

This bibliography identifies manuals that contain descriptions of additional routines available to users of the IAS/RSX-11 system libraries.

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

IAS EXECUTIVE REFERENCE MANUAL (VOLUMES I AND II)

- Task Execution Control Directives
- Informational Directives
- Event-associated Directives
- Trap-associated Directives
- I/O Related Directives
- Task Status Control Directives

IAS/RSX-11D DEVICE HANDLERS REFERENCE MANUAL

- Laboratory and Industrial I/O Routines

IAS/RSX-11 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

RSX-11D EXECUTIVE REFERENCE MANUAL

- Task Execution Control Directives
- Informational Directives
- Event-associated Directives
- Trap-associated Directives
- I/O Related Directives
- Task Status Control Directives

SYSTEM REFERENCE BIBLIOGRAPHY

RSX-11M EXECUTIVE REFERENCE MANUAL

- Task Execution Control
- Task Status Control
- Event-associated Services
- Trap-associated Services
- I/O and Intertask Communication
- Memory Management Services

RSX-11M I/O DRIVERS REFERENCE MANUAL

- Laboratory and Industrial I/O Routines

READER'S COMMENTS

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

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

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

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

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
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754

digital

digital equipment corporation

Printed in U.S.A.