READER COMMENT SHEET

HP 1000 Computers

RTE-A • RTE-6/VM Relocatable Libraries Reference Manual

92077-90037 November 1993

We welcome your evaluation of this manual. Your comments and suggestions help us to improve our publications. Please explain your answers under comments, below, and use additional pages if necessary.

Is this manual technically accurate? ☐ YES ☐ NO

Are the concepts and wording easy to understand? ☐ YES ☐ NO

Is the format of this manual convenient in size, arrangement, and readability? ☐ YES ☐ NO

Comments:

This form requires no postage stamp if mailed in the U.S. For locations outside the U.S., your local HP representative will ensure that your comments are forwarded.

FROM: Date _____

Name: _____

Company: _____

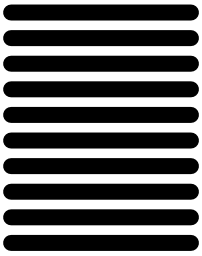
Address: _____

_____

_____

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1070 CUPERTINO, CA

POSTAGE WILL BE PAID BY ADDRESSEE

LEARNING PRODUCTS MANAGER
HEWLETT-PACKARD COMPANY
SOFTWARE TECHNOLOGY DIVISION
11000 N. WOLFE ROAD
CUPERTINO, CA 95014-9804

FOLD FOLD

Please tape closed

HEWLETT PACKARD

RTE-A / RTE-6/VM Relocatable Libraries

Reference Manual

Software Technology Division
11000 Wolfe Road
Cupertino, CA 95014-9804

Manual Part No. 92077-90037
E1193

Printed in U.S.A. November 1993
Sixth Edition

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARs 252.227.7013.

Copyright © 1983, 1985 - 1987, 1989, 1990, 1992, 1993 by Hewlett-Packard Company

Printing History

The Printing History below identifies the edition of this manual and any updates that are included. Periodically, update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this printing history page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past updates; however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the Manual Numbering File or the Computer User's Documentation Index. (The Manual Numbering File is included with your software. It consists of an "M" followed by a five digit product number.)

Edition 1	Jun	1983	
Edition 2	Dec	1983	Combine RTE-A and RTE-6/VM
Update 1	Jan	1985	
Reprint	Jan	1985	Update 1 incorporated
Update 2	Jan	1986	
Reprint	Jan	1986	Update 2 incorporated
Edition 3	Aug	1987	Rev. 5000 (Software Update 5.0)
Edition 4	Jan	1989	Rev. 5010 (Software Update 5.1)
Update 1	Jul	1990	Rev. 5020 (Software Update 5.2)
Edition 5	Dec	1992	Rev. 6000 (Software Update 6.0)
Edition 6	Nov	1993	Rev. 6100 (Software Update 6.1)

Preface

This manual is a programmer's guide to subroutines contained in the RTE-A or RTE-6/VM Operating System and describes the following libraries:

$MATH	Mathematics Library
$SYSLB	System Routines Library
%DECAR	Decimal String Arithmetic Routines Library
$VLB6B	Software Equivalents Library for VIS
$VLBA1	Vector Instruction Set Firmware Interface Library for RTE-A
$VLB6A	Vector Instruction Set Firmware Interface Library for RTE-6/VM

In addition, many of the utility and status subroutines are contained in the $FMP (RTE-A) and $FMP6 (RTE-6/VM) libraries.

Other collections of HP relocatable subroutines for more general use are grouped into other libraries distributed with the RTE Operating System. In addition, many RTE subsystems and languages, such as Pascal/1000 and Spooling, include subroutines that can be of general use. Refer to the appropriate subsystem and language manuals for more information.

Note All references to RTE pertain to both RTE-A and RTE-6/VM, except where specifically noted.

How This Manual is Organized

Chapter 1	Contains a functional listing of all RTE subroutines.
Chapter 2	Describes subroutine calling conventions.
Chapter 3	Provides an alphabetical grouping of the mathematical subroutines.
Chapter 4	Provides an alphabetical grouping of the double integer subroutines.
Chapter 5	Describes other system library routines useful on some subsystems, others maintained for backward compatibility, some general utilities, and CTU manipulation routines.
Chapter 6	Describes subroutines, available only with the HP 92078A product for RTE-A (the Virtual Code + or VC+ system extension package), that provide programmatic access to the multiuser handling system. This allows programs to set up and remove sessions, attach and detach from them, and convert between names, user numbers and session numbers.

Chapter 7	Describes utility and status subroutines.
Chapter 8	Describes the Vector Instruction Set (VIS).
Chapter 9	Describes how to use VIS.
Chapter 10	Describes the decimal string arithmetic subroutines.
Chapter 11	Describes the floating point conversion subroutines.
Chapter 12	Describes the HpCrt library routines.
Appendix A	Provides a functional grouping of library routines.

Table of Contents

Chapter 1
Functional Grouping of Library Routines

Chapter 2
Calling Conventions

.ENTR Call Sequence (Non-CDS) 2-1
Default Parameters 2-2
Alternate Returns 2-3
"Direct" Calls 2-3
PCAL Call Sequence (CDS) 2-4
The Stack 2-5
Character Strings and EMA Variables 2-5
Microcoded Routines (RPLs) 2-6
Fast FORTRAN Processor (FFP) 2-6
Routines Callable from FORTRAN 2-6
Routines Callable from PASCAL 2-7

Chapter 3
Mathematical Subroutines

Format of Routines 3-1
ABS 3-2
AIMAG 3-3
AINT 3-4
ALOG 3-5
ALOGT 3-6
AMAX0, MAX0, AMIN0, MIN0 3-7
AMAX1, MAX1, AMIN1, MIN1 3-8
AMOD 3-9
ATAN 3-10
ATAN2 3-11
CABS 3-12
CEXP 3-13
CLOG 3-14
CMPLX 3-15
CONJG 3-16
COS 3-17
CSNCS 3-18
CSQRT 3-19
DABS 3-20
DATAN 3-21
DATN2 3-22
DBLE 3-23
DCOS 3-24
DDINT 3-25
DEXP 3-26

DIM 3-27
DLOG 3-28
DLOGT 3-29
DMAX1, DMIN1 3-30
DMOD 3-31
DPOLY 3-32
DSIGN 3-34
DSIN 3-35
DSQRT 3-36
DTAN 3-37
DTANH 3-38
ENTIE 3-39
ENTIX 3-40
EXP 3-41
FLOAT 3-42
IABS 3-43
IAND 3-44
IDIM 3-45
IDINT 3-46
IFIX 3-47
INT 3-48
IOR 3-49
ISIGN 3-50
IXOR 3-51
MOD 3-52
REAL 3-53
SIGN 3-54
SIN 3-55
SNGL 3-56
SNGM 3-57
SPOLY 3-58
SQRT 3-59
TAN 3-60
TANH 3-61
.ABS 3-62
.ATAN 3-63
.ATN2 3-64
.BLE 3-65
.CADD 3-66
.CDBL 3-67
.CDIV 3-68
.CFER 3-69
.CHEB 3-70
.CINT 3-71
.CMPY 3-72
.CMRS 3-73
.COS 3-74
.CPM 3-75
.CSUB 3-76
.CTBL 3-77
.CTOI 3-78
.DCPX 3-79
.DFER 3-80

.DINT 3-81
.DTBL 3-82
.DTOD 3-83
.DTOI 3-84
.DTOR 3-85
.EXP 3-86
.FAD, .FSB 3-87
.FDV 3-88
.FLUN 3-89
.FMP 3-90
.FPWR 3-91
.ICPX 3-92
.IDBL 3-93
.IENT 3-94
.ITBL 3-95
.ITOI 3-96
.LOG 3-97
.LOG0 3-98
.MANT 3-99
.MAX1, .MIN1 3-100
.MOD 3-101
.MPY 3-102
.NGL 3-103
.PACK 3-104
.PWR2 3-105
.RTOD 3-106
.RTOI 3-107
.RTOR 3-108
.RTOT 3-109
.SIGN 3-110
.SIN 3-111
.SQRT 3-112
.TADD, .TSUB, .TMPY, .TDIV 3-113
.TAN 3-114
.TANH 3-115
.TCPX 3-116
.TDBL 3-117
.TENT 3-118
.TINT 3-119
.TPWR 3-120
.TTOI 3-121
.TTOR 3-122
.TTOT 3-123
.XADD, .XSUB 3-124
.XCOM 3-125
.XDIV 3-126
.XFER 3-127
.XMPY 3-128
.XPAK 3-129
.XPLY, XPOLY 3-130
.YINT 3-131
..CCM 3-132
..DCM 3-133

..DLC 3-134
..FCM 3-135
..TCM 3-136
#COS 3-137
#EXP 3-138
#LOG 3-139
#SIN 3-140
$EXP 3-141
$LOG 3-142
$LOGT 3-143
$SQRT 3-144
$TAN 3-145
%ABS 3-146
%AN 3-147
%AND 3-148
%ANH 3-149
%BS 3-150
%FIX 3-151
%IGN 3-152
%IN 3-153
%INT 3-154
%LOAT 3-155
%LOG 3-156
%LOGT 3-157
%NT 3-158
%OR 3-159
%OS 3-160
%OT 3-161
%QRT 3-162
%SIGN 3-163
%TAN 3-164
%XP 3-165
/ATLG 3-166
/COS 3-167
/CMRT 3-168
/EXP 3-169
/EXTH 3-170
/LOG 3-171
/LOG0 3-172
/SIN 3-173
/SQRT 3-174
/TAN 3-175
/TINT 3-176

Chapter 4
Double Integer Subroutines

Format of Routines 4-1
FLTDR 4-2
.DADS 4-3
.DCO 4-4
.DDE 4-5
.DDI, .DDIR 4-6
.DDS 4-7

.DIN 4-8
.DIS 4-9
.DMP 4-10
.DNG 4-11
.FIXD 4-12
.FLTD 4-13
.TFTD 4-14
.TFXD 4-15
.XFTD 4-16
.XFXD 4-17

Chapter 5
Utility Subroutines

Format of Routines 5-1
ABREG 5-2
ER0.E 5-3
ERRLU 5-4
ERR0 5-5
FTRAP, RTRAP 5-6
GETST 5-9
IGET, IXGET 5-10
INAMR 5-11
IND.E 5-14
ISSR 5-15
ISSW 5-16
MAGTP 5-17
NAMR 5-18
OVF 5-21
PAU.E 5-22
PNAME 5-23
PTAPE 5-24
RMPAR 5-25
RT_ER 5-26
TIMEI, TIMEO 5-27
.ENTC and .ENTN 5-28
.ENTP and .ENTR 5-29
.FMUI, .FMUO, .FMUP 5-32
.FMUR 5-34
.GOTO 5-35
.MAP 5-36
.OPSY 5-37
.PAUS 5-38
.PCAD 5-39
.TAPE 5-40
..MAP 5-41
$SETP 5-42
%SSW 5-43

Chapter 6
Subroutines for Multiuser Support

AccessLU, Check for LU Access 6-2
ATACH, Attach to Session 6-3
ATCRT, Attach a CRT (RTE-A Only) 6-4
Programmatic LOGON (RTE-A Only) 6-4
CLGOF, Call LOGOF (RTE-A Only) 6-5
CLGON, Call LOGON (RTE-A Only) 6-6
DTACH, Detach From Session 6-7
FromSySession, Check System Session Table Address (RTE-A Only) 6-8
GetAcctInfo, Access User and Group Accounting (RTE-A Only) 6-8
GetOwnerNum, Return Owner's ID 6-10
GetResetInfo, Access/Reset User Accounting (RTE-A Only) 6-10
GETSN, Get Session Number (RTE-A Only) 6-11
GPNAM, Return Group Name 6-11
GroupToId, Return Group ID 6-12
IdToGroup, Return Group Name 6-12
IdToOwner, Return User Name 6-13
LUSES, Return User Table Address 6-13
Member, Check if User is in Group (RTE-A Only) 6-13
OwnerToId, Return User ID and Group ID 6-14
ProgIsSuper, Check for Super Program 6-14
ResetAcctTotals, Resets User and Group Accounting Totals (RTE-A Only) 6-15
RTNSN, Return Session Number (RTE-A Only) 6-16
SessnToOwnerName, Return User Name 6-16
SetAcctLimits, Set User and Group Accounting Limits (RTE-A Only) 6-17
SuperUser, Check For/If Superuser 6-18
SYCON, Write Message to System Console 6-18
SystemProcess, Check For/If System Process
(RTE-A Only) 6-19
UserIsSuper, Check For/If Superuser 6-19
USNAM, Return User Name 6-19
USNUM, Return the Session Number 6-20
VFNAM, Verify User Name (RTE-A Only) 6-20

Chapter 7
Utility and Status Subroutines

AddressOf, Return Direct Address 7-1
Bit Map Manipulation Routines 7-2
    ChangeBits 7-2
    CheckBits 7-2
    FindBits 7-3
BlankString 7-4
BlockToDisc, Convert Block and Sector to Track and Sector 7-4
CaseFold, Convert Lowercase to Uppercase 7-5
CharFill 7-5
CharsMatch, Compare Characters in Arrays 7-6
ClearBuffer, Zero a Passed Buffer 7-6
CLCUC, Convert Lowercase to Uppercase 7-7
CMNDO Routines (RTE-A Only) 7-8
HpStartCmndo, Enable a CMNDO Slave Monitor 7-8

HpReadCmndo, Request CMNDO to Read from User's Terminal 7-9
HpStopCmndo, Terminate CMNDO Slave Monitor 7-10
Example Program Using CMNDO 7-10
CmndStackInit, Initialize Command Stack 7-12
CmndStackMarks, Check for Marked Lines 7-13
CmndStackPush, Add Line to Command Stack 7-13
CmndStackRestore, Restore Command Stack 7-14
CmndStackSaveP, CmndStackRstrP, Save and Restore Command Stack 7-15
CmndStackScreen, Do Stack Interactions with User 7-16
CmndStackStore, Store Command Stack Contents in a File 7-17
CmndStackUnmark, Remove Marks from Command Stack Lines 7-17
Command Stack Example Program 7-18
Concat, Concatenate Strings 7-20
ConcatSpace, Concatenate Strings with Embedded Blanks 7-20
DayTime, Seconds Since January 1, 1970 7-21
DecimalToDint, ASCII to Double Integer Conversion 7-21
DecimalToInt, ASCII to Single Integer Conversion 7-22
DintToDecimal, Double Integer to ASCII Conversion 7-22
DintToDecimalr, Double Integer to ASCII Conversion 7-23
DintToOctal, Double Integer to Octal Conversion 7-23
DintToOctalr, Double Integer to Octal Conversion 7-24
DiscToBlock 7-24
DiscSize, Tracks and Sectors Per Track 7-25
ElapsedTime 7-25
ETime 7-25
Fgetopt, Get a Runstring Option 7-26
GetFatherIdNum 7-28
GetRedirection, Extract I/O Redirection Commands 7-28
GetRteTime 7-29
HexToInt 7-29
HMSCtoRteTime 7-29
IdAddToName, Convert ID Segment Address to Program Name and LU Number 7-30
IdAddToNumber, Convert ID Segment Address to ID Segment Number 7-30
IDCLR 7-30
IdNumberToAdd, Convert ID Segment Number to ID Segment Address 7-31
IntString 7-31
IntToDecimal, Integer to ASCII Conversion 7-32
IntToDecimalr, Integer to ASCII Conversion 7-32
IntToHex 7-33
IntToHexR 7-33
IntToOctal, Integer to Octal Conversion 7-34
IntToOctalr, Integer to Octal Conversion 7-34
InvSeconds 7-35
LastMatch 7-35
LeapYear 7-35
LuLocked 7-36
MoveWords 7-36
MyIdAdd, Return ID Segment Address 7-36
NumericTime 7-37
OctalToDint, ASCII to Double Integer Conversion 7-37
OctalToInt, ASCII to Single Integer Conversion 7-38
ProgramPriority 7-38
ProgramTerminal 7-39

PutInCommas 7-39
ReadA990Clock (RTE-A Only) 7-40
ResetTimer 7-40
Rex (Regular Expression) Routines 7-41
RexBuildPattern 7-42
RexBuildSubst 7-42
RexExchange 7-43
RexMatch 7-44
RteDateToYrDoy 7-44
RteShellRead, Read from a Terminal and Enable Command Line Editing (RTE-A Only) 7-45
RteTimeToHMSC 7-46
SamInfo, Return SAM Size (RTE-A Only) 7-46
Seconds 7-47
SplitCommand, Parse String 7-47
SplitString, Parse String 7-48
StrDsc 7-49
StringCopy, Copy One String to Another 7-50
TIMEF 7-51
TimeNow 7-52
TrimLen, Remove Trailing Blanks 7-52
WhoLockedLu 7-53
WhoLockedRn (RTE-A Only) 7-53
WriteA990Clock (RTE-A Only) 7-54
YrDoyToMonDom 7-54
YrDoyToRteDate 7-55

Chapter 8
VIS Subroutines

The Vector Instruction Set (VIS) 8-1
Arrays in Memory 8-2
Index to VIS Routines 8-5
General Calling Sequence 8-6
Example 8-8
Vector Arithmetic Routines 8-9
Example 8-9
Scalar-Vector Arithmetic Routines 8-11
Example 8-12
Absolute Value Routine 8-13
Example 8-13
Sum Routines 8-14
Example 8-14
Example 8-15
Dot Product Routine 8-17
Example 8-17
Pivot Routine 8-18
Example 8-19
MAX/MIN Routines 8-20
Example 8-20
Comments 8-21
Case 1 (v1 is first array element and incr1 is not 1) 8-22
Case 2 (v1 is not first array element and incr1 = 1) 8-22
Case 3 (v1 is not first array element and incr1 is not 1) 8-23
Case 4 (multidimensional arrays − scanning rows) 8-23

Move Routines 8-24
Example 8-24
EMA (Extended Memory Area)/Non-EMA Move Routines 8-26
Comments 8-26
Example 8-26
Example 8-27

Chapter 9
Using VIS in Programs

Converting FORTRAN DO Loops 9-1
One Dimensional Array Examples 9-2
Two Dimensional Array Examples 9-4
Nested DO Loops Example 9-6
Combinations of Vector Instructions 9-7
Increment Parameters Other Than One 9-8
Zero Increment 9-9
Negative Increment 9-10
Useful Applications 9-11
Initialize a Square Matrix 9-11
Initialize an Array in a Certain Order 9-12
Statistical Examples 9-12
Matrix Transposition 9-13
Case 1 (Not in Place) 9-14
Case 2 (In Place) 9-14
Graphics Coordinate Transformation 9-15
Extended Memory Area (EMA) Considerations 9-16
EMA Call by Value and Call by Reference 9-17
Obtaining Efficiency with Multidimensional Arrays 9-19
Matrix Multiplication EMA Example 9-22
Example VIS Programs 9-24
Calculating Prime Numbers: Sieve of Eratosthenes 9-24
Solution of Linear Systems 9-27
Matrix Inversion 9-30
VIS Online Diagnostic 9-34
Required Hardware and Software 9-34
Test Sections 9-34
Self-Test Section 9-34
Non-Privileged Section 9-35
Privileged Section 9-35
Running the Diagnostic 9-36
FORTRAN Equivalents for VIS 9-38
Assembly Language Opcodes 9-45
A990, A900, and A700 9-45
F-Series 9-50
Firmware Interface Routines, .VSRP and .VDRP 9-55
Adding Your Own EMA Routines 9-59
Error Messages 9-62

Chapter 10
Decimal String Arithmetic Subroutines

Using the DCAR Routines 10-1
DCAR Data Formats 10-1
A2 Format 10-2
D2 Format 10-4
D1 Format 10-6
String Utilities Routines 10-7
JSCOM, Substring Character Compare 10-7
SFILL, Substring Fill 10-9
SGET, Substring Get 10-10
SMOVE, Substring Move 10-11
SPUT, Substring Put 10-13
SZONE, Substring Zone 10-14
String Arithmetic Routines 10-17
SADD, Substring Decimal Add 10-17
SDIV, Substring Decimal Division 10-19
SMPY, Substring Decimal Multiply 10-22
Short-String Routine 10-24
SSUB, Substring Subtract 10-26
Output Editing Routine, SEDIT 10-28
Alphanumeric Editing 10-28
X (Alphanumeric Replacement Holder) 10-28
Numeric Editing 10-28
Replacement 10-29
9 (Numeric Replacement Holder) 10-29
Z (Zero Suppression Replacement Holder) 10-29
* (Asterisk Replacement Holder) 10-29
$ (Dollar Sign Replacement Holder) 10-29
Sign Characters 10-29
Cr (Credit) 10-29
− (Minus) 10-29
Insertion Characters 10-29
Operation of SEDIT 10-30
Rules Governing Creation of Edit Mask 10-30
Errors 10-31
Internal Routines 10-31
SA2DE, Substring A2 Format to Decimal 10-31
SCARY, Substring D2 Decimal Carry 10-33
SDCAR, Substring D1 Decimal Carry 10-34
SDEA2, Substring Decimal to A2 Format 10-36
SD1D2, Substring Decimal D1 Format to Substring Decimal D2 Format 10-37
SD2D1, Substring Decimal D2 Format to Substring Decimal D1 Format 10-38
SSIGN, Substring Sign 10-40

Chapter 11
Floating Point Conversion Subroutines

DFCHI 11-1
FCHI 11-2
DFCIH 11-2
FCIH 11-3

Chapter 12
HpCrt Library Routines

A_Register, B_Register, A_B_Registers, ABREG 12-1
ClearBitMap 12-2
CompareBufs 12-3
CompareWords 12-3
CompressAsciiRLE 12-4
ExpandAsciiRLE 12-5
FakeSpStatus 12-6
FillBuffer 12-7
FirstCharacter 12-7
GetBitMap 12-8
GetByte 12-8
GetDibit 12-9
GetNibble 12-9
GetRunString 12-10
GetString 12-11
HpCrtCharMode 12-11
HpCrtCheckStraps 12-12
HpCrtCRC16_F, HpCrtCRC16_S 12-13
HpCrtGetCursor 12-14
HpCrtGetCursorXY 12-15
HpCrtGetfield_I 12-16
HpCrtGetfield_S 12-17
HpCrtGetLine_Pos 12-18
HpCrtGetMenuItem 12-19
HpCrtHardReset 12-19
HpCrtLineMode 12-20
HpCrtMenu 12-20
HpCrtNlsMenu 12-21
HpCrtNlsXMenu 12-21
HpCrtPageMode 12-22
HpCrtParityChk 12-22
HpCrtParityGen 12-23
HpCrtQTDPort7 12-23
HpCrtReadChar 12-24
HpCrtReadPage 12-25
HpCrtRestorePort 12-26
HpCrtSavePort 12-26
HpCrtSchedProg, HpCrtSchedProg_S 12-27
HpCrtScreenSize 12-27
HpCrtSendChar 12-28
HpCrtSSRCDriver, HpCrtSSRCDriver? 12-29
HpCrtStatus 12-30
HpCrtStripChar 12-31
HpCrtStripCntrls 12-31
HpCrtXMenu 12-32
HpCrtXReadChar 12-32
HpCrtXSendChar 12-33
HpLowerCaseName 12-33
HpRteA 12-34
HpRte6 12-34

HpZ, Mini-Formatter 12-35
How to Use the Mini-Formatter to Do Output 12-35
How to Use the Mini-Formatter to Do Input 12-35
Precautions 12-35
HpZAscii64 12-38
HpZAscii95 12-38
HpZAsciiHpEnh 12-39
HpZAsciiMne3 12-40
HpZAsciiMne4 12-41
HpZBackSpaceIbuf 12-41
HpZBinc 12-42
HpZBino 12-42
HpZDeco 12-42
HpZDecv 12-42
HpZDecc 12-43
HpZDefIBuf 12-43
HpZDefIString 12-44
HpZDefOBuf 12-44
HpZDicv 12-44
HpZDParse 12-45
HpZDumpBitMap 12-48
HpZDumpBuffer 12-49
HpZFieldDefine 12-50
HpZFmpWrite 12-51
HpZGetNextChar HpZPeekNextChar 12-51
HpZGetNextStrDsc 12-52
HpZGetNextToken 12-52
HpZGetNumD2 HpZGetNumO2 HpZGetNumB2
HpZGetNumD4 HpZGetNumO4 HpZGetNumB4 12-53
HpZGetNumStrDsc 12-54
HpZGetNumX 12-55
HpZGetRemStrDsc 12-56
HpZHexc 12-56
HpZHexi 12-57
HpZHexo 12-58
HpZIBufRemain 12-58
HpZIBufReset 12-58
HpZIBufUsed 12-58
HpZIBufUseStrDsc 12-59
HpZInsertAtFront 12-60
HpZmbt 12-60
HpZMesss 12-61
HpZMoveString 12-62
HpZmvc 12-62
HpZmvs 12-63
HpZmvs_Control 12-64
HpZmvs_Escape 12-65
HpZmvs_Large 12-65
HpZNlsMvs 12-66
HpZNlsSubset 12-66
HpZOBufReset 12-66
HpZOBufUsed 12-67
HpZOBufUseStrDsc 12-67

HpZOctc 12-68
HpZOctd 12-68
HpZOcto 12-69
HpZOctv 12-69
HpZPadToCount 12-70
HpZPadToPosition 12-70
HpZParse 12-71
HpZPlural 12-73
HpZPrintPort 12-74
HpZPushObuf and HpZPopObuf 12-75
HpZQandA 12-76
HpZReScan 12-76
HpZRomanNumeral 12-77
HpZsbt 12-78
HpZStripBlanks 12-78
HpZUdeco 12-78
HpZUdecv 12-79
HpZWriteExec14 12-79
HpZWriteLU 12-79
HpZWriteXLU 12-79
HpZWriteToString 12-80
HpZYesOrNo 12-80
MinStrDsc 12-82
PutBitMap 12-82
PutByte 12-83
PutDibit 12-83
PutNibble 12-84
SetBitMap 12-84
SetPriority 12-84
TestBitMap 12-85
Test_PutByte 12-85
Test_SetBitMap 12-86

List of Illustrations

Figure 8-1 One Dimensional Array in Memory 8-2
Figure 8-2 Two Dimensional Array in Memory 8-3
Figure 8-3 Three Dimensional Arrays in Memory 8-4
Figure 8-4 Accessing Row Elements 8-8
Figure 9-1 Troubleshooting Flowchart 9-37

Tables

Table 7-1 Expression Pattern Matching Summary 7-41
Table 7-2 Substitution Constructs 7-41
Table 10-1 Zoned Characters for Negative Strings 10-3
Table 10-2 Binary Representation of Decimal Digits 10-4
Table 10-3 Rightmost Digit for Negative Numbers 10-5
Table 10-4 SZONE Conversion 10-16
Table 12-1 Contents of the State Buffer after FakeSpStatus or HpCrtSavePort Call . 12-6

1

Functional Grouping of Library Routines

RTE-A is delivered with a collection of relocatable subroutines. These subroutines interface user programs with system services. This chapter contains a listing of those subroutines. The detailed description of each subroutine can be found in this manual, the *RTE-A Programmer's Reference Manual*, part number 92077-90007, or the *RTE-6/VM Programmer's Reference Manual*, part number 92084-90005. The functional listing given here indicates the page number on which the subroutine is documented. The subroutines documented in the *RTE-A Programmer's Reference Manual* are indicated in the listing by the mnemonic "*prog*", for example "*prog*-7-11" refers you to page 7-11 of the *RTE-A Programmer's Reference Manual*. Similarly, the subroutines documented in the *RTE-6/VM Programmer's Reference Manual* are indicated in the listing by the mnemonic "*prog6*".

The subroutines listed in this chapter are organized into the following functional groups:

- ASCII/Integer Conversion
- Bit Map Manipulation
- Buffer and String Manipulation
  - Character String Routines
  - HpCrt and HpZ Buffer Routines
  - Integer Buffer Routines
- Character Buffer Manipulation
- Command Stack
- Error Handling
- I/O
- Interprocess Communication
  - Class I/O
  - Parameter Passing
  - Programmatic Environment Variable Access
  - Signals
- Machine-level Access
- Math
  - Absolute Value Subroutines
  - Complex Number Arithmetic Subroutines
  - Double Integer Utilities
  - Exponents, Logs, and Roots
  - HP 1000/IEEE Floating Point Conversion Subroutines
  - Number Conversion Subroutines
  - Real Number Arithmetic Subroutines
  - Trigonometry Subroutines
  - VIS Subroutines
  - Miscellaneous Subroutines
- Multiuser
- Parsing Routines
- Privileged Operation
- Program Control
- Resource Management
- System Status
- Time Operations

ASCII/INTEGER CONVERSION SUBROUTINES

.FMUI	ASCII digit to internal numeric conversion	5-32
.FMUO	Numeric to ASCII conversion	5-32
.FMUP	Internal to normal format conversion	5-32
.FMUR	Rounding of digit string produced by .FMUO	5-34
CNUMD	Convert unsigned 16-bit integer to ASCII decimal	*prog*-7-11; *prog6*-5-70
CNUMO	Convert unsigned 16-bit integer to ASCII octal	*prog*-7-11; *prog6*-5-70
DecimalToDint	ASCII to double integer	7-21
DecimalToInt	ASCII to single integer	7-22
DintToDecimal	Double integer to ASCII	7-22
DintToDecimalr	Double integer to ASCII	7-23
DintToOctal	Double integer to octal	7-23
DintToOctalr	Double integer to octal	7-24
HexToInt	ASCII hexadecimal to single integer	7-29
HpZBinc	Convert a number to binary	12-42
HpZBino	Convert value to its binary ASCII representation	12-42
HpZDecc	Convert a number to ASCII numerals	12-43
HpZDeco	Convert an integer*2 number to ASCII decimal representation	12-42
HpZDecv	Convert an integer*2 number to ASCII decimal representation	12-42
HpZDicv	Convert double integer value to ASCII decimal representation	12-44
HpZDParse	Parse the next occurring token in the input buffer	12-45
HpZGetNumB2	Convert number in input buffer to integer*2 decimal or octal	12-53
HpZGetNumB4	Convert number in input buffer to integer*4 decimal or octal	12-53
HpZGetNumD2	Convert number in input buffer to integer*2 decimal	12-53
HpZGetNumD4	Convert number in input buffer to integer*4 decimal	12-53
HpZGetNumO2	Convert number in input buffer to integer*2 octal	12-53
HpZGetNumO4	Convert number in input buffer to integer*4 octal	12-53
HpZGetNumX	Convert digits to internal representation	12-55
HpZHexc	Convert a number to hexadecimal	12-56
HpZHexi	Parse hexadecimal ASCII integers	12-57
HpZHexo	Convert an integer*2 number to hexadecimal	12-58
HpZOctc	Convert a value to its octal ASCII representation	12-68
HpZOctd	Convert a double integer value to its octal ASCII representation	12-68
HpZOcto	Convert the passed value to its octal ASCII representation	12-69
HpZOctv	Convert the passed value to its octal ASCII representation	12-69
HpZParse	Parse routine for 16-character parameters	12-71
HpZRomanNumeral	Convert a value to its Roman numeral equivalent	12-77
HpZUdeco	Convert integer*2 number to unsigned decimal representation	12-78
HpZUdecv	Convert integer*2 number to unsigned decimal representation, suppressing leading zeros	12-79
INPRS	Inverse parse; convert buffer to original ASCII form	*prog*-7-8; *prog6*-5-74
IntString	Integer to ASCII	7-31
IntToDecimal	Integer to ASCII	7-32
IntToDecimalr	Integer to ASCII	7-32
IntToHex	Integer to ASCII hexadecimal	7-33
IntToHexR	Integer to ASCII hexadecimal with right justification	7-33
IntToOctal	Integer to octal	7-34
IntToOctalr	Integer to octal	7-34
KCVT	Convert positive integer to base 10; return last 2 ASCII digits	*prog*-7-11
OctalToDint	ASCII digit to internal numeric conversion	7-37
OctalToInt	ASCII to integer	7-38
PARSE	Parse ASCII input buffer	*prog*-7-7; *prog6*-5-77

BIT MAP MANIPULATION

ChangeBits	Change bits in a bit map	7-2
CheckBits	Check bits in a bit map	7-2
ClearBitMap	Clear specified bit in a bit map	12-2
FindBits	Find free bits	7-3
GetBitMap	Retrieve a bit from a bit map	12-8
HpZDumpBitMap	Display a bit map; useful for debugging	12-48
PutBitMap	Copy a bit to a bit map	12-82
SetBitMap	Set a bit in the buffer	12-84
TATMP	Map track assignment table into driver partition area in user map	*prog6*-5-63
Test_SetBitMap	Test if a bit is set in the buffer, then if it is not, set bit	12-86
TestBitMap	Test if a bit is set in the buffer	12-85

BUFFER AND STRING MANIPULATION

Character String Routines

BlankString	Determine blank character string	7-4
CaseFold	Convert a character string from lowercase to uppercase	7-5
CharFill	Fill string with characters	7-5
Concat	Concatenate strings	7-20
ConcatSpace	Concatenate strings with n spaces between strings	7-20
LastMatch	Return last occurrence of a character	7-35
MinStrDsc	Construct a string descriptor that describes a trimmed substring of the string that is passed to it	12-82
RexBuildPattern	Build pattern for use by RexMatch and RexExchange	7-42
RexBuildSubst	Build regular substitution string for use by RexExchange	7-42
RexExchange	Replace occurrences of pattern built by RexBuildPattern	7-43
RexMatch	Determine if string contains pattern built by RexBuildPattern	7-44
SplitCommand	Parse a string	7-47
SplitString	Parse a string	7-48
StrDsc	Construct a character string descriptor	7-49
StringCopy	Copy one string to another	7-50
TrimLen	Remove trailing blanks	7-52

HpCrt and HpZ Buffer Routines

HpCrtCRC16_F	Cyclic Redundancy Check	12-13
HpCrtCRC16_S	Cyclic Redundancy Check	12-13
HpCrtParityChk	Perform a parity check on a data buffer	12-22
HpCrtParityGen	Compute and set the parity bits in a data buffer	12-23
HpCrtStripChar	Delete characters from a buffer	12-31
HpCrtStripCntrls	Delete non-displayable characters from a string	12-31
HpZAscii64	Move characters from input buffer to output buffer	12-38
HpZAscii95	Move characters from input buffer to output buffer	12-38
HpZAsciiHpEnh	Move characters from input buffer to output buffer	12-39
HpZAsciiMne3	Translate characters in input buffer into output buffer	12-40
HpZAsciiMne4	Translate characters in input buffer into output buffer	12-41
HpZBackSpacelbuf	Back up the input buffer pointer	12-41
HpZBinc	Convert a number to binary	12-42
HpZBino	Convert a number to its binary ASCII representation	12-42
HpZDecc	Convert a number to ASCII numerals	12-43
HpZDeco	Convert an integer*2 number to ASCII decimal representation	12-42
HpZDecv	Convert an integer*2 number to ASCII decimal representation	12-42
HpZDeflBuf	Declare the attributes of the input buffer	12-43
HpZDeflString	Define a string as the input for the HpZ routines	12-44
HpZDefOBuf	Define the output buffer for the HpZ routines	12-44
HpZDicv	Convert double integer value to ASCII decimal representation	12-44
HpZDParse	Parse the next occurring token in the input buffer	12-45
HpZDumpBuffer	Dump a buffer in different formats; useful for debugging	12-49
HpZFieldDefine	Issue escape sequences to define a field in a block mode screen and optionally set display enhancements	12-50

HpZFmpWrite	Write current contents of output buffer to the file specified	12-51
HpZGetNextChar	Extract the next character from the input buffer	12-51
HpZGetNextStrDsc	Build a string descriptor for the next token in the input buffer	12-52
HpZGetNextToken	Copy the next token in the input buffer to the output string	12-52
HpZGetNumB2	Convert number in input buffer to integer*2 decimal or octal	12-53
HpZGetNumB4	Convert number in input buffer to integer*4 decimal or octal	12-53
HpZGetNumD2	Convert number in input buffer to integer*2 decimal	12-53
HpZGetNumD4	Convert number in input buffer to integer*4 decimal	12-53
HpZGetNumO2	Convert number in input buffer to integer*2 octal	12-53
HpZGetNumO4	Convert number in input buffer to integer*4 octal	12-53
HpZGetNumStrDsc	Return a string descriptor	12-54
HpZGetNumX	Convert digits to internal representation	12-55
HpZGetRemStrDsc	Return a string descriptor to the portion of the HpZ input buffer that has not yet been consumed by other HpZ calls	12-56
HpZHexc	Convert a number to hexadecimal	12-56
HpZHexi	Parse hexadecimal ASCII integers	12-57
HpZHexo	Convert an integer*2 number to hexadecimal	12-58
HpZIBufRemain	Return number of bytes remaining from current position to end of the input buffer	12-58
HpZIBufReset	Reset the current input position to the start of the input buffer	12-58
HpZIBufUsed	Return the current byte offset in the input buffer	12-58
HpZIBufUseStrDsc	Return a string descriptor for the portion of the input buffer that has already been passed over	12-59
HpZInsertAtFront	Insert data in front of the data currently in the buffer	12-60
HpZmbt	Copy bytes from an integer buffer to the output buffer	12-60
HpZMesss	Send a command to the operator interface section of the OS	12-61
HpZMoveString	Copy strings without FORTRAN limitations	12-62
HpZmvc	Copy characters from an integer buffer to the output buffer	12-62
HpZmvs	Copy a string to the current position in the output buffer	12-63
HpZmvs_Control	Move the string passed by the user to the output buffer	12-64
HpZmvs_Escape	Move the string passed by the user to the output buffer	12-65
HpZmvs_Large	Create large characters in a 3-by-3 character cell using line segments in the HP 264x alternate character set	12-65
HpZNlsMvs	Move an NLS string to the output buffer	12-66
HpZNlsSubset	Set up the linkage from NLS to HpZ routines	12-66
HpZOBufReset	Reset the current position to the start of the output buffer	12-66
HpZOBufUsed	Return the current byte offset in the output buffer	12-67
HpZOBufUseStrDsc	Return the current byte offset in the output buffer	12-67
HpZOctc	Convert a value to its octal ASCII representation	12-68
HpZOctd	Convert a double integer value to its octal ASCII representation	12-68
HpZOcto	Convert the passed value to its octal ASCII representation	12-69
HpZOctv	Convert the passed value to its octal ASCII representation	12-69
HpZPadToCount	Add the specified number of blanks to the output buffer	12-70
HpZPadToPosition	Add blanks to output buffer until desired position is reached	12-70
HpZParse	Parse routine for 16-character parameters	12-71
HpZPeekNextChar	Same as HpZGetNextChar but does not consume the character	12-51
HpZPlural	Conditionally make a string plural depending on count	12-73
HpZPopObuf	Inverse of the HpZPushObuf routine	12-75
HpZPushObuf	Declare a new output buffer for the HpZ routines	12-75
HpZReScan	Reset internal pointers used by HpZ routines	12-76
HpZRomanNumeral	Convert a value to its roman numeral equivalent	12-77
HpZsbt	Store the lower byte of the passed value into the output buffer	12-78
HpZStripBlanks	Adjust internal pointer to output buffer to “erase” trailing blanks	12-78
HpZUdeco	Convert integer*2 number to unsigned decimal representation	12-78
HpZUdecv	Convert integer*2 number to unsigned decimal representation, suppressing leading zeros	12-79
HpZWriteExec14	Perform an EXEC 14 call from the HpZ mini-formatter	12-79
HpZWriteLU	Write current contents of the output buffer to the LU specified	12-79
HpZWriteToString	Copy the contents of the output buffer to a string	12-80
HpZWriteXLU	Write current contents of the output buffer to the LU specified	12-79

Integer Buffer Routines

.CFER	Move four words from address x to address y (complex transfer)	3-69
.CPM	Compare two single integer arguments	3-75
.DFER	Move three words from one address to another (extended real transfer)	3-80
.XFER	Move three words from address x to address y (extended real transfer)	3-127
CharsMatch	Compare characters in arrays	7-6
CLCUC	Convert an integer buffer from lowercase to uppercase	7-7
ClearBuffer	Zero a passed buffer	7-6
CompareBufs	Compare two buffers and return offset	12-3
CompareWords	Compare two buffers for equality	12-3
CompressAsciiRLE	Move bytes from the input to the output buffer	12-4
CPUT	Put character in destination buffer set up by SETDB	*prog*-7-18
ExpandAsciiRLE	Process run length encoded ASCII data to expand it back to the original uncompressed contents	12-5
FillBuffer	Fill a buffer with null characters or a specified value	12-7
FirstCharacter	Return the first character of a buffer	12-7
GetByte	Retrieve a byte from a packed array of bytes	12-8
GetDibit	Retrieve a bit pair from a packed array	12-9
GetNibble	Retrieve 4 bits from a packed array	12-9
GetString	Copy a string or a constructed string descriptor	12-11
INAMR	Inverse parse of 10-word parameter buffer generated by NAMR	5-11
JSCOM	Compare substrings in two integer buffers	10-7
KHAR	Get next character from source buffer set up by SETSB	*prog*-7-18
MoveWords	Move words	7-36
NAMR	Read input buffer, produce 10-word parameter buffer	5-18
PutByte	Copy a byte to a packed array of bytes	12-83
PutDibit	Copy a bit pair to a packed array of bit pairs	12-83
PutInCommas	Prepare a string for parsing	7-39
PutNibble	Copy 4 bits to a packed array	12-84
SETDB	Set up character string destination buffer for KHAR, CPUT, ZPUT	*prog*-7-17
SETSB	Set up character string source buffer for KHAR, CPUT, ZPUT	*prog*-7-17
SFILL	Fill area in a substring array with a specified character	10-9
SGET	Get a specified character from a substring in an integer buffer	10-10
SMOVE	Move data from one integer buffer string to another	10-11
SPUT	Put a specified character in an integer buffer substring	10-13
StrDsc	Construct a character string descriptor	7-49
SZONE	Find the zone punch of a character	10-14
Test_PutByte	Copy a byte into an array with a test for zero	12-85
ZPUT	Store character string in destination buffer set up by SETDB	*prog*-7-18

COMMAND STACK

CmndStackInit	Initialize command stack	7-12
CmndStackMarks	Check for marked lines	7-13
CmndStackPush	Add line to command stack	7-13
CmndStackRestore	Restore command stack	7-14
CmndStackRstrP	Restore command stack	7-15
CmndStackSaveP	Save command stack	7-15
CmndStackScreen	Do stack interactions with user	7-16
CmndStackStore	Store command stack contents in a file	7-17
CmndStackUnmark	Remove marks from command stack lines	7-17
HpReadCmndo	Request CMNDO to read from user's terminal	7-9
HpStartCmndo	Enable a CMNDO slave monitor	7-8
HpStopCmndo	Terminate CMNDO slave monitor	7-10
RteShellRead	Read from terminal and enable command line editing	7-45

ERROR HANDLING

.PAUS	Halt program execution and print message	5-38
ER0.E	Specify the LU for printing library error messages	5-3
ERR0	Print four-character error code on list device	5-5
ERRLU	Change LU for printing library error messages	5-4
FTRAP	Traps FORTRAN runtime errors	5-6
IND.E	Select output LU for error messages	5-14
PAU.E	Select output LU for PAUSE messages	5-22
RT_ER	Formats and prints runtime errors	5-26
RTRAP	Traps FORTRAN runtime errors	5-6

I/O

.STIO	Configure driver for a select code currently in use	*prog6*-5-38
.TAPE	Rewind, backspace, or EOF operation on mag tape unit	5-40
AbortRq	Abort current request	*prog*-3-11
ABREG	Obtain contents of A- and B-Registers	5-2, 12-1
AccessLU	Check for LU access	6-2
BINRY	Transfer data to or from a disk device	*prog6*-5-39
BlockToDisc	Convert block to track and sector	7-4
CLRQ	Class management request	*prog*-4-8; *prog6*-5-3
DiscSize	Returns tracks and sectors per track	7-25
DiscToBlock	Convert track and sector to block	7-24
EQLU	Return LU of interrupting device that scheduled program	*prog*-7-10; *prog6*-5-41
EXEC 1	Read data from device	*prog*-3-3; *prog6*-2-19
EXEC 2	Write data to device	*prog*-3-3; *prog6*-2-19
EXEC 3	Perform I/O device control operation	*prog*-3-8; *prog6*-2-24
EXEC 4	Allocate contiguous disk tracks for use by a single program	*prog6*-2-85
EXEC 5	Release disk tracks previously allocated locally	*prog6*-2-87
EXEC 13	Get device status	*prog*-3-12; *prog6*-2-74
EXEC 15	Allocate contiguous disk tracks for use by multiple programs	*prog6*-2-85
EXEC 16	Release disk tracks previously allocated globally	*prog6*-2-87
EXEC 17	Class read request	*prog*-4-11
EXEC 18	Class write request	*prog*-4-11
EXEC 19	Class I/O device control request	*prog*-4-22; *prog6*-2-39
EXEC 20	Class write/read request	*prog*-4-11
EXEC 21	Class I/O Get	*prog*-4-18; *prog6*-2-41
FakeSpStatus	Return port status similar to a special status read	12-6
HpCrtCharMode	Sends the escape sequences to the terminal that place it in line mode, character mode with forms disabled	12-11
HpCrtCheckStraps	Check port and terminal for availability of screen mode operation	12-12
HpCrtGetCursor	Returns the coordinates of the cursor of an HP CRT	12-14
HpCrtGetCursorXY	Returns the coordinates of the cursor of an HP CRT	12-15
HpCrtGetfield_I	Retrieve the Nth field from an integer*2 buffer that contains the data read from an HP terminal in block page mode	12-16
HpCrtGetfield_S	Retrieve the Nth field from an integer*2 buffer that contains the data read from an HP terminal in block page mode	12-17
HpCrtGetLine_Pos	Return cursor position, contents of line, and delimiter	12-18
HpCrtGetMenuItem	Return a menu item from the screen	12-19
HpCrtHardReset	Perform a hard reset on an HP terminal	12-19
HpCrtLineMode	Send escape sequences to terminal for block line mode	12-20
HpCrtMenu	Used to print multiple character strings to an LU	12-20
HpCrtNlsMenu	Perform HpCrtMenu function from the NLS module	12-21
HpCrtNlsXMenu	Perform HpCrtXMenu function from the NLS module	12-21
HpCrtPageMode	Send escape sequence to terminal to place in block page mode	12-22
HpCrtQTDPort7	Return LU of port 7 when given LU of one of the other ports	12-23
HpCrtReadChar	Read directly from LU to character data type variable	12-24
HpCrtReadPage	Perform page mode write/read call	12-25
HpCrtRestorePort	Reset port to conditions in effect when HpCrtSavePort was called	12-26
HpCrtSavePort	Read current state of port driven	12-26
HpCrtSchedProg	Pass name of program to scheduled upon interrupt	12-27

HpCrtSchedProg_S	Pass name of program to scheduled upon interrupt	12-27
HpCrtScreenSize	Return width and height of an HP terminal screen	12-27
HpCrtSendChar	Call EXEC to print a FTN7X character variable or literal	12-28
HpCrtSSRCDriver	Determine if driver for LU will respond to a special status read	12-29
HpCrtSSRCDriver?	Determine if driver for LU will respond to a special status read	12-29
HpCrtStatus	Perform XLUEX write/read call to read status of an HP terminal	12-30
HpCrtXMenu	Print multiple character strings to an LU	12-32
HpCrtXReadChar	Input directly from an LU to a character data type variable	12-32
HpCrtXSendChar	Call EXEC to print a FTN7X character variable or literal	12-33
HpZDumpBuffer	Dump a buffer in different formats; useful for debugging	12-49
HpZPrintPort	Display port status using a special status read	12-74
HpZQandA	Ask a question and read reply	12-76
HpZWriteLU	Write current contents of the output buffer to the LU specified	12-79
HpZWriteXLU	Write current contents of the output buffer to the LU specified	12-79
HpZYesOrNo	Ask question to be answered with a yes or no reply	12-80
IFTTY	Determine if an LU is interactive	*prog*-7-12; *prog6*-5-42
LOGIT	Log message in error log file and display on terminal	*prog*-7-13
LOGLU	Get LU of invoking terminal	*prog*-7-9; *prog6*-5-43
LUTRU	Return true system LU associated with session LU	*prog*-7-9; *prog6*-5-44
MAGTP	Perform utility functions on magnetic tape unit	5-17
ProgramTerminal	Return program's terminal LU	7-39
PTAPE	Position magnetic tape	5-24
REIO	Buffered I/O	*prog*-3-10; *prog6*-5-45
RMPAR	Get extended status	5-25; *prog*-3-17; *prog6*-5-31
RteErrLogging	Determine if error logging is on or off	*prog*-7-13
RteShellRead	Read from terminal and enable command line editing	7-45
SYCON	Write message to system console	6-18; *prog*-3-7; *prog6*-5-79
TRMLU	Determine LU number of interrupting device	*prog6*-5-47
VMAIO	Perform large VMA or EMA data transfer	*prog*-9-21
XLUEX	Extended LU EXEC call	*prog*-3-10; *prog6*-5-45
XREIO	Extended LU REIO call	*prog*-3-11; *prog6*-5-45

INTERPROCESS COMMUNICATION

Class I/O

See "Class I/O" chapter in the *RTE-A Programmer's Reference Manual*

Parameter Passing

EXEC 14	Retrieve or pass string from or to calling program	*prog*-7-3; *prog6*-2-67
Fgetopt	Get a runstring option	7-26
GetRedirection	Extract I/O redirection commands	7-28
GetRunString	Retrieve runstring used to schedule program	12-10
GETST	Recover parameter string	5-9; *prog*-7-5; *prog6*-5-35
HpZWriteExec14	Perform an EXEC 14 call from the HpZ mini-formatter	12-79
PRTM	Pass 4 parameters back to parent program	*prog*-7-1; *prog6*-5-33
PRTN	Pass 5 parameters back to parent program	*prog*-7-1; *prog6*-5-33
RMPAR	Retrieve parameters passed to program	5-25; *prog*-7-2; *prog6*-5-31

Programmatic Environment Variable Access

EXEC 39	Programmatic environment variable access	*prog*-14-1

Signals

KillTimer	Cancel current timer	*prog*-13-27
QueryTimer	Return number of ticks remaining before timer signal is to be generated	*prog*-13-27
SetTimer	Establish a new timer or reset an existing timer	*prog*-13-27
SglAction	Return integer specifying action to take	*prog*-13-6
SglBlock	Return previous set of masked signals and block signals	*prog*-13-6

SglHandler	Set the signal handler address	*prog*-13-7
SglKill	Send a signal to a program	*prog*-13-7
SglLimit	Set the signal buffer limits	*prog*-13-8
SglLongJmp	Jump to the supplied environment	*prog*-13-9
SglPause	Wait for a signal to be delivered to the program	*prog*-13-9
SglSetJmp	Set an environment	*prog*-13-10
SglSetMask	Block signals and return previous set of masked signals	*prog*-13-10

MACHINE-LEVEL ACCESS

..MAP	Compute address of specified element of a 2- or 3-dimensional array	5-41
.DRCT	Resolves indirect address	*prog6*-5-72
.ENTC and .ENTN	Transfer true address of parameters from calling sequence into a subroutine; adjust return address to true return point	5-28
.ENTP and .ENTR	Transfer true address of parameters from calling sequence into a subroutine; adjust return address to true return point	5-29
.GOTO	Transfer control to the location indicated by a FORTRAN computed GOTO statement	5-35
.MAP	Return actual address of a particular element of a two-dimensional FORTRAN array	5-36
.MPY	Replace the subroutine call with the hardware instructions to multiply by integer i and j	3-102
.PCAD	Return true address of parameter passed to a subroutine	5-39
$SETP	Set up a list of pointers	5-42
%SSW	Set sign bit of A-Register according to bit n of switch register	5-43
ABREG	Obtain contents of A- and B-Registers	5-2, 12-1
AddressOf	Return direct address	7-1
COR.A, COR.B	Return address of first word of available memory	*prog6*-5-64
EXEC 25	Return status information about specified memory partition	*prog6*-2-72
IGET and IXGET	Read contents of a memory address	5-10
ISSR	Set S-Register to value n	5-15
ISSW	Set sign bit of A-Register according to bit n of switch register	5-16
OVF	Set sign bit of A-Register according to overflow bit	5-21
ReadA990Clock	Read the calendar clock of the A990	7-40
WriteA990Clock	Set the calendar clock on the A990	7-54

MATH

Absolute Value Subroutines

.ABS	Absolute value (double real)	3-62
%ABS	(Call-by-name) IABS	3-146
%BS	(Call-by-name) ABS	3-150
ABS	Absolute value (real)	3-2
CABS	Absolute value (complex)	3-12
DABS	Absolute value (extended real)	3-20
DIM	Positive difference (real)	3-27
IABS	Absolute value (integer)	3-43
IDIM	Positive difference (integer)	3-45

Complex Number Arithmetic Subroutines

..CCM	Complement (complex)	3-132
.CADD	Complex add	3-66
.CDBL	Extract the real part of a complex number in extended real form	3-67
.CDIV	Complex divide	3-68
.CMPY	Complex multiply	3-72
.CSUB	Complex subtract	3-76
AIMAG	Extract imaginary part of complex (real)	3-3
CMPLX	Combine real and imaginary complex	3-15
CONJG	Form conjugate of complex	3-16
REAL	Extract the real part of a complex	3-53

Decimal String Arithmetic Subroutines

JSCOM	Compare two substrings	10-7
SA2DE	Convert substring in A2 format to decimal	10-31
SADD	Perform a decimal add of two substrings	10-17
SCARY	Examine D2 decimal substring for carries	10-33
SD1D2	Convert substring in D1 format to D2	10-37
SD2D1	Convert substring in D2 format to D1	10-38
SDCAR	Examine D1 decimal substring for carries	10-34
SDEA2	Convert substring in decimal format to A2	10-36
SDIV	Perform a decimal division of two substrings	10-19
SEDIT	Edit data in substring array	10-28
SFILL	Fill area in a substring array with a specified character	10-9
SGET	Get a specified character from a substring	10-10
SMOVE	Move data from one string to another	10-11
SMPY	Perform a decimal multiply of two substrings	10-22
SPUT	Put a specified character in a substring	10-13
SSIGN	Find the sign of a number	10-40
SSUB	Perform a decimal subtraction of two substrings	10-26
SZONE	Find the zone punch of a character	10-14

Double Integer Utilities

.DADS	Double integer add and subtract	4-3
.DCO	Compare two double integers	4-4
.DDE	Decrement double integer	4-5
.DDI	Double integer divide	4-6
.DDIR	Double integer divide	4-6
.DDS	Double integer decrement and skip if zero	4-7
.DIN	Increment double integer	4-8
.DIS	Double integer increment and skip if zero	4-9
.DMP	Double integer multiply	4-10
.DNG	Negate double integer	4-11
.FIXD	Convert real to double integer	4-12
.FLTD	Convert double integer to real	4-13
.TFTD	Convert double integer to double real	4-14
.TFXD	Convert double real to double integer	4-15
.XFTD	Convert double integer to extended real	4-16
.XFXD	Convert extended real to double integer	4-17
FLTDR	Convert double-length record number to real	4-2

Exponents, Logs, and Roots

.CTOI	Raise complex to integer power	3-78
.DTOD	Raise extended real to extended real power	3-83
.DTOI	Raise extended real to integer power	3-84
.DTOR	Raise extended real to real power; extended real result	3-85
.EXP	Raise e to double real power	3-86
.FPWR	Raise real to integer power	3-91
.ITOI	Raise integer to integer power	3-96
.LOG	Natural log (double real)	3-97
.LOG0	Base 10 log (double real)	3-98
.PWR2	Multiply a real by 2 raised to integer power	3-105
.RTOD	Raise real to extended real power; extended real result	3-106
.RTOI	Raise real to integer power	3-107
.RTOR	Raise real to real power	3-108
.RTOT	Raise real to double real power	3-109
.SQRT	Square root (double real)	3-112
.TPWR	Raise double real to unsigned power	3-120
.TTOI	Raise double real to integer power	3-121
.TTOR	Raise double real to real power	3-122
.TTOT	Raise double real to double real power	3-123
$EXP	Raise e to extended real power; no error return	3-141

$LOG	Natural log (extended real); no error return	3-142
$LOGT	Base 10 log (extended real); no error return	3-143
$SQRT	Square root (extended real); no error return	3-144
%LOG	Natural log (real; call-by-name)	3-156
%LOGT	Base 10 log (real; call-by-name)	3-157
%QRT	Square root (real; call-by-name)	3-162
%XP	Raise e to real power; call-by-name	3-165
/EXP	.EXP with no error return	3-169
/EXTH	2**n*2**z (small double real z)	3-170
/LOG	.LOG with no error return	3-171
/LOG0	.LOG0 with no error return	3-172
/SQRT	.SQRT with no error return	3-174
ALOG	Natural log (real)	3-5
ALOGT	Base 10 log (real)	3-6
CEXP	Raise e to complex power	3-13
CLOG	Natural log (complex)	3-14
CSQRT	Complex square root (complex)	3-19
DEXP	Extended real e (extended real)	3-26
DLOG	Natural log (extended real)	3-28
DLOGT	Base 10 log (extended real)	3-29
DSQRT	Square root (extended real)	3-36
EXP	Raise e to real power	3-41
SQRT	Square root (real)	3-59

HP 1000/IEEE Floating Point Conversion Subroutines

DFCHI	HP 1000 double precision to IEEE	11-1
DFCIH	IEEE double precision to HP 1000	11-2
FCHI	HP 1000 single precision to IEEE	11-2
FCIH	IEEE single precision to HP 1000	11-3

Number Conversion Subroutines

.BLE	Convert real to double real	3-65
.CINT	Convert complex to integer	3-71
.CMRS	Reduce argument for SIN, COS, TAN, EXP	3-73
.CTBL	Convert complex real to double real	3-77
.DCPX	Convert extended real to complex	3-79
.DINT	Convert extended real to integer	3-81
.DTBL	Convert extended real to double real	3-82
.ICPX	Convert integer to complex	3-92
.IDBL	Convert integer to extended real	3-93
.IENT	Greatest integer no greater than given real x	3-94
.ITBL	Convert integer to double real	3-95
.NGL	Convert double real to real	3-103
.PACK	Convert signed mantissa of real into normalized real format	3-104
.TCPX	Convert double real to complex real	3-116
.TDBL	Convert double real to extended real without rounding	3-117
.TINT	Convert double real to integer	3-119
%FIX	Convert real to integer; call-by-name	3-151
%INT	Truncate real; call-by-name	3-154
%LOAT	Convert integer to real; call-by-name	3-155
%NT	Truncate real to integer; call-by-name	3-158
/CMRT	Range reduction for .SIN, .COS, .TAN, .EXP, and .TANH	3-168
/TINT	Convert double precision to integer	3-176
AINT	Truncate (real)	3-4
AMOD	x modulo y (real x and y)	3-9
DBLE	Convert real to extended real	3-23
DDINT	Truncate (extended real)	3-25
DMOD	x modulo y (extended real x and y)	3-31
ENTIE	Greatest integer not greater than given real	3-39
FLOAT	Convert integer to real	3-42
IDINT	Truncate extended real to integer	3-46

IFIX	Convert real to integer	3-47
INT	Truncate real to integer j	3-48
MOD	i modulo j (integer i and j)	3-52
SNGL	Convert extended real to real	3-56
SNGM	Convert extended real to real without rounding	3-57
SPOLY	Evaluate the quotient of two polynomials in single precision	3-58

Real Number Arithmetic Subroutines

..DCM	Complement (extended real)	3-133
..DLC	Load and complement (real)	3-134
..FCM	Complement (real)	3-135
..TCM	Negate (double real)	3-136
.DTBL	Convert extended real to double real	3-82
.FDV	Real divide	3-88
.FLUN	Unpack (real); place exponent in A-Register, lower mantissa in B-Register	3-89
.FMP	Real multiply	3-90
.MANT	Extract mantissa of real x	3-99
.MAX1 and .MIN1	Find the maximum (or minimum) of a list of double reals	3-100
.MOD	Double real remainder of real divide	3-101
.SIGN	Transfer sign of one double real to another	3-110
.TADD	Double real add	3-113
.TDBL	Convert double real to extended real with rounding	3-117
.TDIV	Double real divide	3-113
.TINT	Convert double real to integer	3-119
.TMPY	Double real multiply	3-113
.TSUB	Double real subtract	3-113
.XCOM	Complement extended real unpacked mantissa in place	3-125
.XDIV	Extended real divide	3-126
.XMPY	Extended real multiply	3-128
.XPAK	Normalize, round, and pack with the exponent an extended real mantissa	3-129
.YINT	Truncate fractional part of double real	3-131
%IGN	Transfer sign of real or integer to real	3-152
/ATLG	Compute (1−x)/(1+x) (double precision)	3-166
DSIGN	Transfer sign of one extended real to another	3-34
ENTIX	Greatest integer no greater than given extended real; result is extended real	3-40
SIGN	Transfer sign of real or integer to real	3-54

Trigonometry Subroutines

.ATAN	Arctangent (double real)	3-63
.ATN2	Arctangent double real quotient	3-64
.COS	Cosine (double real)	3-74
.SIN	Sine (double real)	3-111
.TAN	Tangent (double real)	3-114
.TANH	Hyperbolic tangent (double real)	3-115
$TAN	DTAN with no error return	3-145
%AN	Tangent (real); call-by-name	3-147
%ANH	Hyperbolic tangent (real); call-by-name	3-149
%IN	Sine (real); call-by-name	3-153
%OS	Cosine (real); call-by-name	3-160
%TAN	Arctangent (real); call-by-name	3-164
/COS	.COS with no error return	3-167
/SIN	.SIN with no error return	3-173
/TAN	.TAN with no error return	3-175
ATAN	Arctangent (real)	3-10
ATAN2	Arctangent (real)	3-11
COS	Cosine (real)	3-17
CSNCS	Complex sine or cosine (complex)	3-18
DATAN	Arctangent (extended real)	3-21
DATN2	Arctangent (extended real x, double real y)	3-22
DCOS	Cosine (extended real)	3-24
DSIN	Sine (extended real)	3-35

DTAN	Tangent (extended real)	3-37
DTANH	Hyperbolic tangent (real)	3-38
SIN	Sine (real)	3-55
TAN	Tangent (real)	3-60
TANH	Hyperbolic tangent (real)	3-61

VIS Subroutines

DVABS	Absolute value routine (double precision)	8-13
DVADD	Vector add (double precision)	8-9
DVDIV	Vector divide (double precision)	8-9
DVDOT	Vector dot product routine (double real)	8-17
DVMAB	Vector largest value (absolute) (double real)	8-20
DVMAX	Vector largest value (double real)	8-20
DVMIB	Vector smallest value (absolute) (double real)	8-20
DVMIN	Vector smallest value (double real)	8-20
DVMOV	Vector move routine (double real)	8-24
DVMPY	Vector multiply (double real)	8-9
DVNRM	Vector sum (absolute) routine (double real)	8-14
DVPIV	Vector pivot routine (double real)	8-18
DVSAD	Vector-scalar add (double real)	8-11
DVSDV	Vector-scalar divide (double real)	8-11
DVSMY	Vector-scalar multiply (double real)	8-11
DVSSB	Vector-scalar subtract (double real)	8-11
DVSUB	Vector subtract (double real)	8-9
DVSUM	Vector sum routine (double real)	8-14
DVSWP	Vector copy routine (double real)	8-24
DVWMV	Vector non-EMA to EMA move routine (EMA double real)	8-26
DWABS	Absolute value routine (EMA double real)	8-13
DWADD	Vector add (EMA double real)	8-9
DWDIV	Vector divide (EMA double real)	8-9
DWDOT	Vector dot product routine (EMA double real)	8-17
DWMAB	Vector largest value (absolute) (EMA double real)	8-20
DWMAX	Vector largest value (EMA double real)	8-20
DWMIB	Vector smallest value (absolute) (EMA double real)	8-20
DWMIN	Vector smallest value (EMA double real)	8-20
DWMOV	Vector move routine (EMA double real)	8-24
DWMPY	Vector multiply (EMA double real)	8-9
DWNRM	Vector sum (absolute) routine (EMA double real)	8-14
DWPIV	Vector pivot routine (EMA double real)	8-18
DWSAD	Vector-scalar add (EMA double real)	8-11
DWSDV	Vector-scalar divide (EMA double real)	8-11
DWSMY	Vector-scalar multiply (EMA double real)	8-11
DWSSB	Vector-scalar subtract (EMA double real)	8-11
DWSUB	Vector subtract (EMA double real)	8-9
DWSUM	Vector sum routine (EMA double real)	8-14
DWSWP	Vector copy routine (EMA double real)	8-24
DWVMV	Vector EMA to non-EMA move routine (double real)	8-26
VABS	Absolute value routine (single precision)	8-13
VADD	Vector add (single precision)	8-9
VDIV	Vector divide (single precision)	8-9
VDOT	Vector dot product routine (single precision)	8-17
VMAB	Vector largest value (absolute) (single precision)	8-20
VMAX	Vector largest value (single precision)	8-20
VMIB	Vector smallest value (absolute) (single precision)	8-20
VMIN	Vector smallest value (single precision)	8-20
VMOV	Vector move routine (single precision)	8-24
VMPY	Vector multiply (single precision)	8-9
VNRM	Vector sum (absolute) routine (single precision)	8-14
VPIV	Vector pivot routine (single precision)	8-18
VSAD	Vector-scalar add (single precision)	8-11
VSDV	Vector-scalar divide (single precision)	8-11

VSMY	Vector-scalar multiply (single precision)	8-11
VSSB	Vector-scalar subtract (single precision)	8-11
VSUB	Vector subtract (single precision)	8-9
VSUM	Vector sum routine (single precision)	8-14
VSWP	Vector copy routine (single precision)	8-24
VWMOV	Vector non-EMA to EMA move routine (single precision)	8-26
WABS	Absolute value routine (EMA single precision)	8-13
WADD	Vector add (EMA single precision)	8-9
WDIV	Vector divide (EMA single precision)	8-9
WDOT	Vector dot product routine (EMA single precision)	8-17
WMAB	Vector largest value (absolute) (EMA single precision)	8-20
WMAX	Vector largest value (EMA single precision)	8-20
WMIB	Vector smallest value (absolute) (EMA single precision)	8-20
WMIN	Vector smallest value (EMA single precision)	8-20
WMOV	Vector move routine (EMA single precision)	8-24
WMPY	Vector multiply (EMA single precision)	8-9
WNRM	Vector sum (absolute) routine (EMA single precision)	8-14
WPIV	Vector pivot routine (EMA single precision)	8-18
WSAD	Vector-scalar add (EMA single precision)	8-11
WSDV	Vector-scalar divide (EMA single precision)	8-11
WSMY	Vector-scalar multiply (EMA single precision)	8-11
WSSB	Vector-scalar subtract (EMA single precision)	8-11
WSUB	Vector subtract (EMA single precision)	8-9
WSUM	Vector sum routine (EMA single precision)	8-14
WSWP	Vector copy routine (EMA single precision)	8-24
WVMOV	Vector EMA to non-EMA copy routine (single precision)	8-26

Miscellaneous Math Subroutines

..TCM	Negate (double real)	3-136
.CFER	Move four words from address x to address y (complex transfer)	3-69
.CHEB	Evaluate Chebyshev series	3-70
.FLUN	Unpack (real); place exponent in A-Register, lower mantissa in B-Register	3-89
.MANT	Extract mantissa of real x	3-99
.TENT	Find the greatest integer i less than or equal to a double real	3-118
.XFER	Move three words from address x to address y (extended real transfer)	3-127
%AND	Logical product (two integers); call-by-name	3-148
%OR	Logical inclusive OR (two integers); call-by-name	3-159
%OT	Complement (integer); call-by-name	3-161
%SIGN	Transfer sign of real or integer z to integer i; call-by-name	3-163
DPOLY	Evaluate quotient of two polynomials (double precision)	3-32
IAND	Logical product (two integers)	3-44
IOR	Logical inclusive OR (two integers)	3-49
ISIGN	Transfer sign of real or integer z to integer i	3-50
IXOR	Exclusive OR (integer)	3-51
XPOLY and .XPLY	Evaluate extended real polynomial	3-130

MULTIUSER

AccessLU	Check for LU access	6-2
ATACH	Attach calling program to a session	6-3
ATCRT	Attach to CRT	6-4
CLGOF	Call LOGOF	6-5
CLGON	Call LOGON	6-6
DTACH	Detach from session	6-7
FromSySession	Check system session	6-8
GetAcctInfo	Access user and group accounting	6-8
GetOwnerNum	Return owner ID	6-10
GetResetInfo	Access and clear multiuser account	6-10
GETSN	Get session number	6-11
GPNAM	Return group name	6-11
GroupToId	Return group ID number given group name	6-12
GTERR	Return SCB error mnemonic from current Session Control Block	*prog6*-5-48
GTSCB	Return contents of the current Session Control Block	*prog6*-5-50
ICAPS	Return current session's capability level	*prog6*-5-51
IdToGroup	Return group name given group ID number	6-12
IdToOwner	Return user name	6-13
LUSES	Return user table address	6-13; *prog6*-5-52
Member	Determine if user is in group	6-13
OwnerToId	Return user ID	6-14
ProgIsSuper	Determine if program is a super program	6-14
PTERR	Update error mnemonic in current Session Control Block	*prog6*-5-53
ResetAcctTotals	Reset user and group accounting totals	6-15
RTNSN	Return session number	6-16
SESSN	Determine if calling program is in session	*prog6*-5-54
SessnToOwnerName	Return user name	6-16
SetAcctLimits	Set user and group accounting limits	6-17
SuperUser	Check for or if superuser	6-18
SYCON	Write a message to the system console	6-18; *prog6*-5-79
SystemProcess	Check for or if system process	6-19
UserIsSuper	Check for or if superuser	6-19
USNAM	Return user name	6-19
USNUM	Return session number	6-20
VFNAM	Verify user name	6-20

PARSING

HpZDParse	Parse the next occurring token in the input buffer	12-45
HpZParse	Parse routine for 16-character parameters	12-71
INAMR	Inverse parse of 10-word parameter buffer generated by NAMR	5-11
INPRS	Inverse parse; convert buffer to original ASCII form	*prog*-7-8; *prog6*-5-74
NAMR	Read input buffer, produce 10-word parameter buffer	5-18
PARSE	Parse ASCII input buffer	*prog*-7-7; *prog6*-5-77
SplitCommand	Parse a string	7-47
SplitString	Parse a string	7-48

PRIVILEGED OPERATION

$LIBR	Go privileged (highest level)	*prog*-12-3
$LIBX	Resume normal operation after calling $LIBR	*prog*-12-3
DispatchLock	Prevent all other user programs from executing	*prog*-12-2
DispatchUnlock	Remove lock set by DispatchLock	*prog*-12-2
GOPRV	Go privileged; disable normal memory protect mechanism	*prog*-12-1
UNPRV	Resume normal operation after calling GOPRV	*prog*-12-1

PROGRAM CONTROL

CHNGPR	Change program priority	*prog*-5-4
EXEC 6	Stop program execution	*prog*-5-5; *prog6*-2-50
EXEC 7	Suspend program execution	*prog*-5-8; *prog6*-2-53
EXEC 8	Load program overlay	*prog*-5-2; *prog6*-2-55
EXEC 9	Immediate program scheduling with wait	*prog*-5-8; *prog6*-2-57
EXEC 10	Immediate program scheduling without wait	*prog*-5-8; *prog6*-2-57
EXEC 22	Lock program into memory so it cannot be swapped	*prog*-5-13; *prog6*-2-70
EXEC 23	Queue program scheduling with wait	*prog*-5-8; *prog6*-2-57
EXEC 24	Queue program scheduling without wait	*prog*-5-8; *prog6*-2-57
EXEC 26	Return memory limits of the partition of calling program	*prog*-5-14; *prog6*-2-81
EXEC 29	Retrieve ID segment of specified program	*prog*-5-16
GetFatherIdNum	Return father ID segment number	7-28
HpLowerCaseName	Change the name of the program that calls it to lowercase	12-33
HpZMesss	Send a command to the operator interface section of the OS	12-61
IdAddToName	Convert ID segment address to program name and LU number	7-30
IdAddToNumber	Convert ID segment address to segment number	7-30
IDCLR	Deallocate ID segment	7-30
IDGET	Retrieve ID segment of specified program	*prog*-7-14; *prog6*-5-66
IDINFO	Return ID segment information	*prog*-7-15
IdNumberToAdd	Convert ID segment number to segment address	7-31
MESSS	Process base set commands	*prog*-7-12; *prog6*-5-75
MyIdAdd	Return segment address	7-36
PNAME	Return program name	*prog*-7-14; 5-23
ProgramPriority	Return program priority	7-38
SEGLD	Load program overlay; allows use of SEGRT and debug	*prog*-5-3; *prog6*-5-56
SEGRT	Return to main from overlay	*prog*-5-4; *prog6*-5-59
XQPRG	Load and execute a program	*prog6*-5-82

RESOURCE MANAGEMENT

LIMEM	Return starting location and size of memory area between end of program or stack area and end of program partition	*prog*-2-13; *prog6*-5-68
LuLocked	Report is passed LU is locked	7-36
LURQ	Give program exclusive access to an I/O device	*prog*-2-8; *prog6*-5-20
RNRQ	Allocate and manage resource numbers	*prog*-2-1; *prog6*-5-16
SetPriority	Set the priority of the currently executing program	12-84
WhoLockedLu	Return ID segment address of program that locked LU	7-53
WhoLockedRn	Return ID segment address of program that locked the specified resource number	7-53

SYSTEM STATUS

.OPSY	Determine which operating system is in control	5-37
CPUID	Get CPU identification	*prog*-7-9
HpRte6	Determine if calling program is running on RTE-6/VM	12-34
HpRteA	Determine if calling program is running on RTE-A	12-34
HpZMesss	Send a command to the operator interface section of the OS	12-61
IFBRK	Test break flag and clear if set	*prog*-7-11; *prog6*-5-67
MESSS	Process base set commands	*prog*-7-12; *prog6*-5-75
OPSY	Determine which operating system is in control	*prog6*-5-55
SamInfo	Return number of free words in SAM or XSAM	7-46

TIME OPERATIONS

DayTime	Return ASCII time string	7-21
ElapsedTime	Number of milliseconds since last time recorded by ResetTimer	7-25
ETime	Number of centiseconds since specified base time	7-25
EXEC 11	Retrieve current time	*prog*-6-1; *prog6*-2-72
EXEC 12	Schedule program at specified time interval	*prog*-6-2; *prog6*-2-63
FTIME	Return ASCII message giving time and date	*prog*-6-7; *prog6*-5-73
GetRteTime	Read the system clock in three-word format	7-29
HMSCtoRteTime	Convert Hr-Min-Sec-Centisec to RTE time format	7-29
InvSeconds	Perform conversion that is the inverse of the Seconds routine	7-35
KillTimer	Cancel current timer	*prog*-13-27
LeapYear	Test a given year to see if it is a leap year	7-35
NumericTime	Return numeric ASCII time string	7-37
QueryTimer	Return number of ticks remaining before timer signal is to be generated	*prog*-13-27
ResetTimer	Reset timer used by ElapsedTime routine	7-40
RteDateToYrDoy	Convert from RTE combined year/day format to year and day	7-44
RteTimeToHMSC	Convert centiseconds since midnight to Hr-Min-Sec-Centisec	7-46
Seconds	Convert a time buffer to seconds since January 1, 1970	7-47
SetTimer	Establish a new timer or reset an existing timer	*prog*-13-27
SETTM	Set system time	*prog*-6-7; *prog6*-5-61
TIMEF	Format time	7-51
TIMEI and TIMEO	Measure difference between time in and time out	5-27
TimeNow	Number of seconds since midnight January 1, 1970	7-52
TMVAL	Format system time into array of time parameters	*prog6*-5-80
YrDoyToMonDom	Convert year and day of year to day of month, month, and day of week	7-54
YrDoyToRteDate	Convert year and cardinal day to RTE format	7-55

2

Calling Conventions

This chapter discusses conventions to use when calling or writing subroutines that conform to calling sequences used by compilers and HP-supplied software. Use of these conventions is strongly recommended for any general purpose software.

.ENTR Call Sequence (Non-CDS)

The standard calling sequence for non-CDS programs uses a JSB instruction and a (usually microcoded) subroutine called .ENTR. This calling sequence is produced by FORTRAN and Pascal in non-CDS mode. Briefly, it uses a list of parameter addresses following the JSB and a word providing parameter count information. These are copied to the subroutine parameter pointer area by a .ENTR call:

```
Caller:                 JSB Sub
                        DEF Return
                        DEF parameter1[,I]
                        DEF parameter2[,I]
                        . . .
                        DEF parametern[,I]
                Return  EQU *

Subroutine:     ^Parm1  BSS 1
                ^Parm2  BSS 1
                        . . .
                ^Parmn  BSS 1
                Sub     BSS 1
                        JSB .ENTR
                        DEF ^Parm1

                    <Body of subroutine>

                        JMP Sub,I
```

This illustrates a call that passes n parameters to a subroutine that expects n parameters. The instructions must come in the order given; for example, the return point must immediately follow the last passed parameter, and the entry point and .ENTR call must immediately follow the last subroutine parameter pointer.

Important points about using .ENTR:

- Indirect addresses are allowed on the parameter addresses, but not on the “DEF return”. Indirects are resolved before storing the subroutine parameter pointers, so these are always direct addresses.
- If the number of parameters passed does not match the number expected, then the number of pointers transferred is equal to the smaller of the passed or expected number of pointers. Any remaining parameter pointers in the subroutine are left unchanged.
- The entry point is updated with the true return address, even if not all the parameters are transferred.
- The number of parameters passed or expected can be equal to zero parameters. (The DEF return is still required to indicate zero parameters expected. The DEF following the .ENTR points to the entry point in such case.)
- Use this statement for the DEF return to avoid the necessity of assigning new label names for return addresses:

  ```
  DEF  *+n+1
  ```

  where n is the number of parameters.
- The A, B, E and O-Registers should be considered indeterminate after the .ENTR call. This means you should not pass parameters indirectly through A or B. Thus, the following is not permitted:

  ```
  JSB   SUB
  DEF   *+2
  DEF   B,I
  ```

.ENTR uses call by reference, that is, it passes the addresses of parameters rather than the parameters themselves. This works well for FORTRAN or for Pascal VAR parameters. It simplifies handling parameters that are changed by the subroutine or are several words long (floating point, and so on). Because it is “pass by reference”, it is possible to change the values of your constants with a subroutine, so be careful. Note that Pascal implements non-VAR type parameters by using a .ENTR call followed by code to copy the value of the parameter to a local variable.

Default Parameters

Because .ENTR does not change the value of parameter pointers that are in excess of the number of parameters passed, it is possible to provide defaults for parameters not passed. This is done by initializing the pointers to point to default values. The defaults must be restored after each call in most cases; this is usually done just before returning. Note that default parameters are not available in some computers, so if you are concerned with program portability, do not make extensive use of them. It may be better to use certain key values, such as zeros, to indicate that the default value of a parameter is desired.

Alternate Returns

Some subroutines use alternate returns to show that the subroutine works. EXEC is the most notable example of this. It uses a technique called "P+1" return, meaning it returns one location past where you would normally expect it to. (It does this on no-abort EXEC calls when they don't abort.) It is technically possible to have P+2, P+3, and so on, returns, but they are rarely used. Compilers have difficulty with subroutines that use such P+1 returns. FORTRAN treats EXEC calls as a special case, and both FORTRAN and Pascal provide Alias mechanisms to generate the right code to call such routines. The code looks like this:

```
              JSB EXEC
              DEF Return
              DEF =B100027    ; no abort schedule
              DEF name
Return        JMP Aborted!    ; return here on error
              . . .           ; return here if it worked
```

FORTRAN multiple returns are handled by returning a value in the A-Register that indicates which of several returns to take; zero means normal return, 1 means the first alternate return, and so on.

"Direct" Calls

Some subroutines have a modified .ENTR calling sequence that does not use the DEF return. This calling sequence is usually called the direct calling sequence. It is not recommended for general use, but it does appear occasionally, especially when calling routines that are implemented in microcode on some processors. This method reduces the number of words needed to call a subroutine, but the disadvantage is that the number of parameters passed must match the exact number expected or the return address will be incorrect.

Compilers produce direct calling sequences for constructs that require them, such as intrinsics like .DAD. This calling sequence can also be requested with the Alias directive, but again it is not recommended. Some library routines like SIN combine direct calling sequences with alternate returns. The compiler handles such constructs only with difficulty; because they cannot be stepped over with the symbolic debugger and they reduce the number of ways the subroutines can be used, they should be avoided. If you must write a routine that has a direct calling sequence, you should know that the microcoded routine .ENTN (described in the appropriate hardware reference manual) can be used in place of .ENTR at the subroutine end to make this work. .ENTR will not do what you want, because there is no DEF return.

PCAL Call Sequence (CDS)

Programs that separate code and data (CDS programs in RTE-A) cannot use the .ENTR calling sequence, because the JSB instruction does not work with pure code. Instead, these programs use the PCAL calling sequence. PCAL uses a procedure call stack maintained in the program's data segment to hold the return address, parameters and other important information.

The standard PCAL interface, as used in assembly language, is shown below. It is similar to the .ENTR call in that addresses of the parameters are passed to the subroutine. It differs in that the called procedure does not do any special processing to retrieve the data or parameters or both; the parameter addresses just appear automatically on the stack. This example illustrates the simplest form of PCAL (described in the *Macro/1000 Reference Manua*l, part number 92059-90001), in which both the calling and called routines are written as CDS modules, and the address of the subroutine is known when the program is linked.

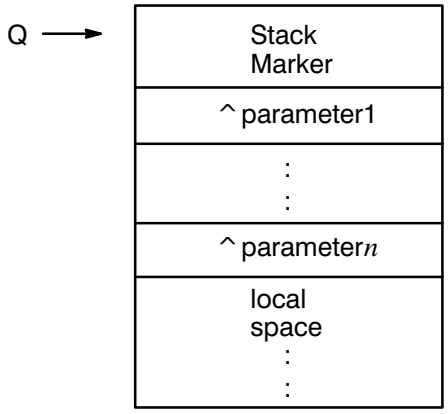
```
Caller:                 PCAL  Sub,n,0,0
                        DEF   parameter1[,I]
                        DEF   parameter2[,I]
                        . . .
                        DEF   parametern[,I]
                          <return here>

Subroutine:             RELOC  LOCAL
                  ^Parm1 BSS 1
                  ^Parm2 BSS 1
                        . . .
                  ^Parmn BSS 1
                        RELOC  CODE
                  Sub   DEC fs
                     <enter here>
                        . . .
                        EXIT
```

Sub is the entry point of the procedure to be called. N, the actual argument count, is in the range 0 to 255. The zeros following the parameter count specify the call sequence and PCAL-type; here they mean standard call sequence, standard PCAL type. The parameter DEFs always point to locations in the data segment; they can be base relative or indirect (or both). The address will be resolved to a direct, non-base relative address before being written on the stack.

Fs, the frame size, must include six words for the stack marker, n words for the formal arguments, and local space used by the procedure.

The Stack

After execution of a PCAL, the called routine has acess to a stack frame of the form:



Q contains the address of the first word of the (current) stack frame. The location of ^parameter*i* is Q + 5 + *i*; the stack marker uses Q + 0 through Q + 5. The first location for local space is Q + 5 + *n* + 1. Local space in the stack frame is not initialized by PCAL.

Character Strings and EMA Variables

The convention of passing a single word address to indicate a parameter works for most cases, but there are at least two cases that require a different technique: character strings and EMA variables.

Character string parameters are passed using a string descriptor that indicates where the string is located and how long it is. Various languages use different string descriptors; the kind that is used by FORTRAN is a two word descriptor, the first word containing the (fixed) length of the string in bytes, the second being the byte address of the string. Whenever a character string is passed as a parameter, the parameter address passed in the subroutine call (either .ENTR or PCAL) is a pointer to a descriptor. Note that the byte address can specify an odd byte, meaning the string can start in the middle of a word.

FORTRAN, Pascal, and MACRO can generate calls that refer to parameters in EMA. Because the reason for having EMA is to handle large data areas, it stands to reason that a 16-bit address (actually 15, plus an indirect bit) cannot be used as the address. Instead, EMA variables are passed with a single word pointer to a double word (32-bit) address, which really specifies where the data item is. Thus one level of indirect resolution is necessary when accessing the 32-bit address from the called routine.

Microcoded Routines (RPLs)

Many subroutines are microcoded on some (but not all) processors. These include such constructs as SIN and TAN, which are included with processors that have hardware floating point support. LINK handles these routines if they are defined when the program is loaded; the RTE generator handles them if they are defined when the system is generated. RPL files can also be specified at load time. Defining these files is usually done by including an RPL file in the system generation (although it is possible to specify the RPLs at load time); these subroutines are then included with all programs loaded for this system. Whenever a JSB to a microcoded routine occurs, LINK or the generator replaces the JSB instruction with the opcode for the desired instruction. The DEFs, and so on, remain as they are for the software equivalent of the subroutine.

Apart from the differences just described, the rules and recommendations for calling microcoded routines are the same as for calling software routines.

Fast FORTRAN Processor (FFP)

The HP 1000 E-Series computer is optionally equipped with a Fast FORTRAN Processor (FFP), while the F-Series computer includes the FFP as a standard feature. The FFP Firmware feature performs several frequently used FORTRAN operations including parameter passing, array address calculations, floating point conversion, packing, rounding, and normalization operations. The A-Series computer is not equipped with FFP; however, it supports a comparable group of instructions called the language instruction set.

See your computer reference manual to find the list of microcoded subroutines suitable for your computer.

Routines Callable from FORTRAN

Using FORTRAN, routines are callable as a function or subroutine; examples are ABS(x) and RMPAR(IBUF), respectively. Routines are callable under the same conditions as Pascal. Refer to the *FORTRAN-77 Reference Manual*, part number 92836-90001, for a discussion of compatibilities between the two languages.

Note Functions called from FORTRAN must have data types declared corresponding to the return value of the function (that is, Real, Integer, Character and so on). For example, the DecimalToDint function returns a double word integer, therefore DecimalToDint should be declared in a type declaration statement as a double word integer.

Routines Callable from PASCAL

Using Pascal, routines are callable if the calling sequence is either in the standard calling sequence format or one of the special calling sequences supported by the compiler. For additional information, refer to the *Pascal/1000 Reference Manual*, part number 92833-90005.

Routines that return results in the A-Register, or in the A- and B-Registers, are callable from Pascal as functions.

Routines that have names containing characters that are not allowed in Pascal identifiers, such as the leading dot in .RTOI, must have EXTERNAL declarations using the ALIAS compiler option. For example,

```
FUNCTION power
      $ALIAS '.RTOI',DIRECT,ERROREXIT$
      (X:REAL;I:INTEGER):REAL;
      EXTERNAL;
```

Unless stated otherwise, all parameters documented in this manual are not EMA parameters. In HEAP 2 programs, the $HEAPPARMS OFF$ compiler option must be given in the EXTERNAL declaration of routines that do not accept EMA parameters.

The non-CDS calling sequences supported by the Pascal compiler (where n(DEF) represents the DEF statement repeated n times) are:

1. Standard:

```
  JSB routine
  DEF *+n+1
n(DEF)
```

2. Direct:

```
  JSB routine
n(DEF)
```

3. Standard, ERROREXIT:

```
  JSB routine
  DEF *+n+1
n(DEF)
  JSB PAS.ERROREXIT
```

4. Direct, ERROREXIT:

```
  JSB routine
n(DEF)
  JSB PAS.ERROREXIT
```

The CDS calling sequences supported by the Pascal compiler are analogous to the non-CDS calling sequences, using the PCAL instruction rather than JSB.

Routines that have direct calling sequences and/or include an error return must have EXTERNAL declarations using the DIRECT and/or ERROREXIT compiler option(s). The EXTERNAL function declaration above is an example of a Direct, ERROREXIT calling sequence.

A routine is not callable from Pascal if it has any other calling sequence. For example, Pascal does not support passing parameters in registers.

3

Mathematical Subroutines

This chapter documents subroutines used for mathematical subroutines in programs produced by the FORTRAN, Pascal, and BASIC compilers. The subroutines can also be called from assembly language. Many of the subroutines are available as microcoded subroutines; refer to the specific processor reference manual for more information.

Format of Routines

The subroutines in this chapter are presented in the following format:

Name	The name of the subroutine.
Purpose	The use of the subroutine.
Entry Points	The entry points to the subroutine.
Assembly	The Macro/1000 assembly language calling sequence for each entry point. "A" and "B" indicate the A- and B-Registers.
FORTRAN	A statement on whether or not the subroutine is callable in FORTRAN-77.
Pascal	A statement on whether or not the subroutine is callable in Pascal.
Parameters	An explanation of the parameters' form and value.
Result	The type of result and the registers used (if any) where the result is returned.
Errors	A summary of the error conditions reported by the subroutine. Errors generated by external references are not described. Refer to the *FORTRAN 77 Reference Manual*, part number 92836-90001, for a more complete discussion of error messages.
External References	Other subroutines that are called by the subroutine.
Notes	Additional information for using the subroutine.

ABS

Purpose: Calculate the absolute value of a real x.

Entry Points: ABS

Assembly:
```
DLD x
JSB ABS
<Return> (result in A and B)
```

FORTRAN: Function: ABS (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: None

External References: ..FCM, .ZPRV

AIMAG

Purpose: Extract the imaginary part of a complex x.

Entry Points: AIMAG

Assembly:

```
JSB AIMAG
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Function: AIMAG (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	Complex number	Complex

Result: Real in A and B

Errors: None

External References: .ZPRV

AINT

Purpose: Truncate a real x.

Entry Points: AINT

Assembly:
```
DLD x
JSB AINT
<Return> (result in A and B)
```

FORTRAN: Function: AINT (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	real to truncate	Real

Result: Real in A and B

Errors: None

External References: .FAD, .ZPRV

ALOG

Purpose: Calculate the natural logarithm of a real x.

Entry Points: ALOG

Assembly:
```
DLD x
JSB ALOG
JSB ERR0 (error return)
<Return> (result in A and B)
```

FORTRAN: Function: ALOG (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: x < 0 ⟶ 02 UN

External References: .FLUN, FLOAT, .FAD, .FSB, .FDV, .FMP, .ZPRV

ALOGT

Purpose: Calculate the common logarithm (base 10) of a real x.

Entry Points: ALOGT ALOG0

Assembly:

```
DLD x
JSB ALOGT (or ALOG0)
JSB ERR0 (error return)
<Return> (result in A and B)
```

FORTRAN: Function: ALOGT (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: If $x \leq 0 \longrightarrow$ 02 UN

External References: ALOG, .FMP

AMAX0, MAX0, AMIN0, MIN0

Purpose: Calculate the maximum or minimum of a series of integer values.

Entry Points: AMAX0, MAX0, AMIN0, MIN0

Assembly:

```
JSB entry point
DEF *+n+1
DEF a
DEF b
:  :
DEF n
<Return> (result y in A or A and B)
```

FORTRAN: See Notes.

Pascal: Callable

Parameters:

Parameter	Description	Type
a	argument	Integer
b	argument	Integer
:	:	:
n	argument	Integer

Result: Real in A and B for AMAX0 and AMIN0
Integer in A for MAX0 and MIN0

Errors: If the number of parameters is less than 2, result = 0

External References: FLOAT

Notes: FORTRAN 7X functions:

AMAX0 (a,b,...,n),
MAX0 (a,b,...,n),
AMIN0 (a,b,...,n),
MIN0 (a,b,...,n)

AMAX1, MAX1, AMIN1, MIN1

Purpose: Calculate the maximum or minimum of a series of real values.

Entry Points: AMAX1, MAX1, AMIN1, MIN1

Assembly:

```
JSB Entry Point
DEF *+ n+1
DEF a
DEF b
:  :
DEF n
<Return> (result y in A or A and B)
```

FORTRAN: See Notes.

Pascal: Callable

Parameters:

Parameter	Description	Type
a	argument	Real
b	argument	Real
:	:	:
n	argument	Real

Result: Real in A and B for AMAX1 and AMIN1
Integer in A for MAX1 and MIN1

Errors: If the number of parameters is less than 2, result = 0

External References: IFIX, .FSB

Notes:

1. Callable as integer or real procedure, but only with a fixed number of parameters.

2. FORTRAN 7X functions:

   AMAX1 (a,b,...,n),
   MAX1 (a,b,...,n),
   AMIN (a,b,...,n),
   MIN1 (a,b,...,n).

AMOD

Purpose: Calculate the real remainder of x/y for a real x and y.

Entry Points: AMOD

Assembly:

```
JSB AMOD
DEF *+3
DEF x
DEF y
<Return> (result z in A and B)
```

FORTRAN: Function: AMOD (x,y)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	1st argument	Real
y	2nd argument	Real

Result: Real in A and B

Errors: If y = 0, then z = x

External References: .ENTP, .ZPRV, AINT, .FDV, .FMP, .FSB

ATAN

Purpose: Calculate the arctangent of a real x.

Entry Points: ATAN

Assembly:
```
DLD x
JSB ATAN
<Return> (result in A and B)
```

FORTRAN: Function: ATAN (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B (radians)

Errors: None

External References: .ZPRV, ..FCM, .FAD, .FSB, .FDV, .FMP

Notes: Result ranges from $-\pi/2$ to $\pi/2$.

ATAN2

Purpose: Calculate the real arctangent of the quotient of two reals.

Entry Points: ATAN2

Assembly:

```
JSB ATAN2
DEF *+3
DEF y
DEF x
<Return> (result in A and B)
```

FORTRAN: Function: ATAN2 (y,x)

Pascal: Callable

Parameters:

Parameter	Description	Type
y	dividend	Real
x	divisor	Real

Result: Real in A and B

Errors: None

External References: .ENTP, SIGN, ATAN, .ZRNT, .FDV, .FAD

CABS

Purpose: Calculate the real absolute value (modulus) of a complex x.

Entry Points: CABS

Assembly:

```
JSB CABS
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Function: CABS (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Complex

Result: Real in A and B

Errors: None

External References: ABS, .FSB, .FAD, .FDV, .FMP, .ENTP, SQRT, .ZRNT

CEXP

Purpose: Calculate the complex exponential of a complex x.

Entry Points: CEXP

Assembly:

```
JSB CEXP
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: CEXP (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Complex
y	result	Complex

Result: Complex

Errors: None

External References: .ENTP, EXP, .ZRNT, SIN, COS, .FMP

CLOG

Purpose: Calculate the complex natural logarithm of a complex x.

Entry Points: CLOG

Assembly:

```
JSB CLOG
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: CLOG (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Complex
y	result	Complex

Result: Complex

Errors: If x = 0 ⟶ 02 UN

External References: .ENTP, ALOG, .ZRNT, CABS, ATAN2

CMPLX

Purpose: Combine a real x and an imaginary y into a complex z.

Entry Points: CMPLX

Assembly:

```
JSB CMPLX
DEF *+4
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Function: CMPLX (x,y)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	real part	Real
y	imaginary part	Real
z	result	Complex

Result: Complex

Errors: None

External References: .ENTP, .ZPRV

CONJG

Purpose: Form the conjugate of a complex x.

Entry Points: CONJG

Assembly:

```
JSB CONJG
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: CONJG (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Complex
y	result	Complex

Result: Complex

Errors: None

External References: .ENTP, ..DLC, .ZPRV

COS

Purpose: Calculate the sine or cosine of a real x (radians).

Entry Points: COS

Assembly:
```
DLD x
JSB COS
Error return
<Return> (result in A and B)
```

FORTRAN: Function: COS (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: x outside $[-8192^{*}\pi, +8191.75^{*}\pi] \longrightarrow$ 050R

External References: .ZPRV, .CMRS, ..FCM, .FMP, .FAD

CSNCS

Purpose: Calculate the complex sine or cosine of a complex x.

Entry Points: CSIN, CCOS

Assembly:

```
JSB CSIN (or CCOS)
DEF *+3
DEF y (result)
DEF x
JSB error routine
<normal return>
```

FORTRAN: Function: CSIN (x) or CCOS (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Complex
y	result	Complex

Result: Complex

Errors: None

External References: .ENTR, SIN, COS, EXP, ..FCM

CSQRT

Purpose: Calculate the complex square root of a complex x.

Entry Points: CSQRT

Assembly:
```
JSB CSQRT
DEF +*3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: CSQRT (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Complex
y	result	Complex

Result: Complex

Errors: Overflow bit is set if result is out of range

External References: .ENTP, ..DLC, .CFER, SQRT, CABS, .ZRNT

DABS

Purpose: Calculate the absolute value of an extended real x.

Entry Points: DABS

Assembly:

```
JSB DABS
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DABS (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: None

External References: ..DCM, .DFER, .ENTP, .ZRNT

DATAN

Purpose: Calculate the extended real arctangent of an extended real x.

Entry Points: DATAN

Assembly:

```
JSB DATAN
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DATAN (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: None

External References: .ZRNT, .XADD, .XSUB, .XMPY, .XDIV, .ENTP, ..DCM, .FLUN, .DFER

DATN2

Purpose: Calculate the extended real arctangent of the quotient of two extended reals.

Entry Points: DATN, DATA2

Assembly:

```
JSB DATN2 (or DATA2)
DEF *+4
DEF z (result)
DEF y
DEF x
<Return>
```

FORTRAN: Function: DATN2 (y,x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	divisor	Extended Real
y	dividend	Extended Real
z	result	Extended Real

Result: Extended Real

Errors: None

External References: .ENTP, DSIGN, DATAN, .ZRNT, .XADD, .XDIV, .DFER

DBLE

Purpose: Convert a real x to an extended real y.

Entry Points: DBLE

Assembly:

```
JSB DBLE
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DBLE (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Real
y	result	Extended Real

Result: Extended Real

Errors: None

External References: .ZPRV

DCOS

Purpose: Calculate the extended real cosine of an extended real x (angle in radians).

Entry Points: DCOS

Assembly:

```
JSB DCOS
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DCOS (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument in radians	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: None

External References: .ENTP, DSIN, .ZRNT, .XADD

DDINT

Purpose: Truncate the fractional part of an extended real.

Entry Points: DDINT

Assembly:

```
JSB DDINT
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DDINT (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Integer

Result: Extended Real

Errors: None

External References: .XADD, .ENTP, .ZRNT, ENTIX

DEXP

Purpose: Calculate the extended real exponential of an extended real x.

Entry Points: DEXP

Assembly:

```
JSB DEXP
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DEXP (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: If $e^x > (1 - 2^{-39})\ 2^{127} \longrightarrow$ 1 OF

External References: .ENTP, .XADD, .XSUB, .XMPY, .XDIV, .DFER, .ZRNT, DDINT, SNGL, IFIX, .FLUN, .XPAK

DIM

Purpose: Calculate the positive difference between a real x and y.

Entry Points: DIM

Assembly:

```
JSB DIM
DEF *+3
DEF x
DEF y
<Return> (result in A and B)
```

FORTRAN: Function: DIM (x,y)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	first argument	Real
y	second argument	Real

Result: Real

Errors: None

External References: .FSB, .ZPRV

DLOG

Purpose: Calculate the extended real natural logarithm of an extended real x.

Entry Points: DLOG

Assembly:

```
JSB DLOG
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DLOG (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	first argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: If $x \leq 0$ 11 UN

External References: .ENTP, .XADD, .XSUB, .XMPY, .XDIV, .FSB, .FLUN, FLOAT, DBLE, .DFER, .ZRNT

DLOGT

Purpose: Calculate the extended real common logarithm of an extended real x.

Entry Points: DLOGT (DLOG0)

Assembly:

```
JSB DLOGT (DLOG0)
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DLOGT (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: If $x < 0 \longrightarrow$ 11 UN

External References: .ENTP, DLOG, .XMPY

DMAX1, DMIN1

Purpose: Calculate the maximum or minimum of a series of extended real values.

Entry Points: DMAX1, DMIN1

Assembly:

```
JSB DMAX1 (or DMIN1)
DEF *+n+2
DEF y (result)
DEF a
DEF b
:  :
DEF n
<Return>
```

FORTRAN: See Notes.

Pascal: Callable

Parameters:

Parameter	Description	Type
a	argument	Extended Real
b	argument	Extended Real
:	:	:
n	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: If n < 2, then y = 0.

External References: .XSUB, .DFER

Notes: FORTRAN 7X intrinsic functions:
DMAX1 (a,b,c,....)
DMIN1 (a,b,c,....)

DMOD

Purpose: Calculate the extended real remainder of two extended real values.

Entry Points: DMOD

Assembly:

```
JSB DMOD
DEF *+4
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Function: DMOD (x,y)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	first argument	Extended Real
y	second argument	Extended Real
z	result	Extended Real

Result: Extended Real

Errors: If y = 0, then z = x

External References: .ENTP, .XSUB, .XMPY, .XDIV, DDINT, .ZRNT

DPOLY

Purpose: Evaluate the quotient of two polynomials in double precision.

Entry Points: DPOLY, TRNL

Assembly:

Form 1

```
JSB DPOLY
DEF *+6
DEF z (result)
DEF x
DEF c
DEF m
DEF n
<Return>
```

or

Form 2

```
JSB DPOLY
OCT <flags>
DEF z (result)
DEF x
DEF c
DMF m
DEF n
<Return>
```

FORTRAN: CALL DPOLY (z,x,c,m,n)

Pascal: Callable

Parameters:

Parameter	Description	Type
z	result	Double Real
x	argument	Double Real
c	coefficient list	Address
m	order of numerator	Integer
n	order of denominator	Integer

Result: Double Real

Errors: None

External References: .ENTR, .CFER, .TADD, .TSUB, .TMPY, .TDIV, .4ZRO

The two polynomials are defined as follows:

$$P(x) = P_m x^m + P_{m-1} x^{m-1} + \ldots + P_1 x + P_0$$

$$Q(x) = x^n + Q_{n-1} x^{n-1} + \ldots + Q_1 x + Q_0$$

The coefficient list c is stored sequentially in memory as follows:

$$P_m, P_{m-1}, \ldots, P_1, P_0, Q_{n-1}, Q_{n-2}, \ldots, Q_1, Q_0$$

$Q_n = 1.0$ is implied but not stored.

If $n = 0$, no coefficients are provided for Q, and only P is evaluated.

The first form of the call evaluates the quotient $z = P(x)/Q(x)$.

FORTRAN uses the first form of the call.

Either form can be called from Assembler. If the second form is used, the format of the flags is as follows:

Bit 15 = F
Bit 14 = S
Bit 0 = T

The following equations can be evaluated by using the second form in Assembler and setting F, S, and T as follows;

F = 0 : $z = P(x)/Q(x)$
F = 1, S = 0, T = 0: $z = P(x^2)/Q(x^2)$
F = 1, S = 0, T = 1: $z = x * P(x^2)/Q(x^2)$
F = 1, S = 1, T = 0: $z = P(x^2) / (P(x^2) - Q(x^2))$ (n>0)
F = 1, S = 1, T = 1: $z = x * P(x^2) / (P(x^2) - Q(x^2))$ (n>0)

The case n = 0 and S = 1 is not allowed.

Any underflow or overflow that occurs invalidates the final result and will set the O-Register. The O-Register is cleared otherwise. Variable m must be at least one. The A, B, X, Y, and E-Registers are undefined after the exit from this routine.

DSIGN

Purpose: Transfer the sign of an extended real y to an extended real x.

Entry Points: DSIGN

Assembly:

```
JSB DSIGN
DEF *+4
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Function: DSIGN (x,y)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	first argument	Extended Real
y	second argument	Extended Real
z	result	Extended Real

Result: Extended Real

Errors: If y = 0, z = 0

External References: .DFER, .ENTP, ..DMC, .ZRNT

DSIN

Purpose: Calculate the extended real sine of an extended real x (angle in radians).

Entry Points: DSIN

Assembly:

```
JSB DSIN
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DSIN (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: None

External References: .ENTP, ..DCM, XPOLY, .DFER, .XSUB, ENTIX, .XADD, .XMPY, .XDIV, .ZRNT

DSQRT

Purpose: Calculate the extended real square root of an extended real x.

Entry Points: DSQRT

Assembly:

```
JSB DSQRT
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DSQRT (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: If x < 0 ⟶ 03 UN

External References: .ENTP, DBLE, SNGL, SQRT, .XDIV, .XADD, .ZRNT, .XMPY

DTAN

Purpose: Calculate tangent of an extended real x.

Entry Points: DTAN

Assembly:

```
JSB DTAN
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DTAN (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real (radians)
y	result	Extended Real

Result: Extended Real

Errors: x outside $[-8192\pi, +8191.75\pi]$ $\longrightarrow$ 0 OR

External References: .ENTR, .DFER, .TMPY, .TSUB, .TINT, .ITBL, .XADD, .XMPY, .XDIV, XPOLY

DTANH

Purpose: Calculate hyperbolic tangent of an extended real x.

Entry Points: DTANH

Assembly:

```
JSB DTANH
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DTANH (y,x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: None

External References: .ENTR, .DFER, .XFER, .FLUN, .PWRZ, DEXP, .XADD, .XMPY, .XDIV

ENTIE

Purpose:

1. Calculate the greatest integer not algebraically exceeding a real x (ENTIER).
2. Round a real x to the nearest integer; if half way between two integers, select the algebraically larger integer (.RND).

Entry Points: ENTIE, .RND

Assembly:

```
DLD x
JSB .RND (or ENTIE)
<Return> (result in A)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Two Integers: sign in A; Integer in B

Errors: See Notes.

External References: None

Notes:

```
If exponent > 15   Then (overflow detected)
 If (x > 0)   Then
   A = 077777B
 Else
   A = 100000B
 Endif
Else
   A = Result
Endif
```

ENTIX

Purpose: Calculate ENTIER of an extended real x.

Entry Points: .XENT, ENTIX

Assembly:

```
JSB .XENT (or ENTIX)
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Extended Real

Result: Extended Real

Errors: None

External References: .ENTP, .ZPRV

EXP

Purpose: Calculate e^x, where x is real.

Entry Points: EXP

Assembly:
```
DLD x
JSB EXP
JSB ERR0 (error)
<Return> (result in A and B)
```

FORTRAN: Function: EXP (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: $x^*\log_2 e \geq 127 \longrightarrow$ 07 OF

External References: .ZPRV, .CMRS, .PWRZ, .FMP, .FSB, .FAD, .FDV

FLOAT

Purpose: Convert integer i to a real x.

Entry Points: FLOAT

Assembly:
```
LDA i
JSB FLOAT
<Return> (result in A and B)
```

FORTRAN: Function: FLOAT (i)

Pascal: Not callable

Parameters:

Parameter	Description	Type
i	argument	Integer

Result: Real in A and B

Errors: None

External References: .PACK, .ZPRV

IABS

Purpose: Calculate absolute value of integer i.

Entry Points: IABS

Assembly:
```
LDA i
JSB IABS
<Return> (result in A)
```

FORTRAN: Function: IABS (i)

Pascal: Not callable

Parameters:

Parameter	Description	Type
i	argument	Integer

Result: Integer in A

Errors: See Notes.

External References: .ZPRV

Notes: If i is (−32768), the result is 32767 and the overflow bit is set.

IAND

Purpose: Take the logical product of integers i and j.

Entry Points: IAND

Assembly:
```
JSB IAND
DEF i
DEF j
<Return> (result in A)
```

FORTRAN: Function: IAND (i,j)

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
i	argument	Integer
j	argument	Integer

Result: Integer in A

Errors: None

External References: None

IDIM

Purpose: Calculate the positive difference between integers i and j.

Entry Points: IDIM

Assembly:

```
JSB IDIM
DEF *+3
DEF i
DEF j
<Return> (result in A)
```

FORTRAN: Function: IDIM (i,j)

Pascal: Callable

Parameters:

Parameter	Description	Type
i	argument	Integer
j	argument	Integer

Result: Integer in A

Errors: See Notes.

External References: .ZPRV

Notes: If IDIM(i,j) is out of range, the overflow bit is set and a value of 32767 returned.

IDINT

Purpose: Truncate an extended real to an integer.

Entry Points: IDINT

Assembly:

```
JSB IDINT
DEF *+2
DEF x
<Return> (result in A)
```

FORTRAN: Function: IDINT (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real

Result: Integer in A

Errors: If IDINT (x) is out of range, then result = 32767 and the overflow bit is set.

External References: IFIX, .ZPRV, SNGM

IFIX

Purpose: Convert a real x to an integer.

Entry Points: IFIX

Assembly:
```
DLD x
JSB IFIX
<Return> (result in A)
```

FORTRAN: Function: IFIX (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Integer in A (see notes)

Errors: None

External References: Nonfloating point library: .FLUN

Floating point library: .ZPRV

Notes:

1. Any fractional portion of the result is truncated. If the integer portion is greater than or equal to 2^{15}, the result is set to 32767.
2. The routine IFIX exists only in nonfloating point libraries.

INT

Purpose: Truncate a real x to an integer.

Entry Points: INT

Assembly:
```
DLD x
JSB INT
<Return> (result in A)
```

FORTRAN: Function: INT (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Integer in A

Errors: If INT (x) is out of range, the overflow bit is set. The result is set to 32767.

External References: IFIX

IOR

Purpose: Take logical inclusive OR of integers i and j.

Entry Points: IOR

Assembly:

```
JSB IOR
DEF i
DEF j
<Return> (result in A)
```

FORTRAN: Function: IOR (i,j)

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
i	argument	Integer
j	argument	Integer

Result: Integer in A

Errors: None

External References: None

ISIGN

Purpose: Calculate the sign of z times the absolute value of i, where z is real or integer and i is integer.

Entry Points: ISIGN

Assembly:
```
JSB ISIGN
DEF i
DEF z
<Return> (result in A)
```

FORTRAN: Function: ISIGN (i,z)

Pascal: Callable using $DIRECT$ directive

Parameters:

Parameter	Description	Type
i	argument	Integer
z	argument	Real or Integer

Result: Integer in A

Errors: None

External References: .ZPRV

IXOR

Purpose: Perform an integer exclusive OR.

Entry Points: IXOR

Assembly:

```
JSB IXOR
DEF *+3
DEF i
DEF j
<Return> (result in A)
```

FORTRAN: Function: IXOR (i,j)

Pascal: Callable

Parameters:

Parameter	Description	Type
i	argument	Integer
j	argument	Integer

Result: Integer in A

Errors: None

External References: None

MOD

Purpose: Calculate the integer remainder of i/j for integer i and j.

Entry Points: MOD

Assembly:

```
JSB MOD
DEF *+3
DEF i
DEF j
<Return> (result in A and B)
```

FORTRAN: Function: MOD (i,j)

Pascal: Callable

Parameters:

Parameter	Description	Type
i	argument	Integer
j	argument	Integer

Result: Integer in A

Errors: If j=0, then result = i

External References: .ZPRV

REAL

Purpose: Extract the real part of a complex x.

Entry Points: REAL

Assembly:

```
JSB REAL
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Function: REAL (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Complex

Result: Real in A and B

Errors: None

External References: .ZPRV

SIGN

Purpose: Calculate the sign of z times the absolute value of x, where z is real or integer and x is real.

Entry Points: SIGN

Assembly:

```
JSB SIGN
DEF x
DEF z
<Return> (result in A and B)
```

FORTRAN: Function: SIGN (x,z)

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Real
z	argument	Integer or Real

Result: Real in A and B

Errors: None

External References: ..FCM, .ZPRV

SIN

Purpose: Calculate the sine of a real x (radians).

Entry Points: SIN

Assembly:
```
DLD x
JSB SIN
Error return
<Return> (result in A and B)
```

FORTRAN: Function: SIN(x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: x outside $[-8192\pi, +8191.75\pi]$ ⟶ 050R

External References: .ZPRV, .CMRS, ..FCM, .FMP, .FAD

SNGL

Purpose: Convert an extended real x to a real y.

Entry Points: SNGL

Assembly:

```
JSB SNGL
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Function: SNGL (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real

Result: Real in A and B

Errors: None

External References: .ZPRV

SNGM

Purpose: Convert an extended real x to a real y without rounding.

Entry Points: SNGM

Assembly:
```
JSB SNGM
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Function: SNGM (x)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real

Result: Real in A and B

Errors: If y < ABS $((-1 + 2^{-23}) * 2^{-128})$, zero is returned.

External References: .ZPRV

Notes: Maximum error will be less than the least significant bit.

SPOLY

Purpose: Evaluate the quotient of two polynomials in single precision real.

Entry Points: SPOLY

Assembly:
```
JSB SPOLY
DEF *+5
DEF x
DEF c
DEF m
DEF n
<Return>
```

FORTRAN: Call SPOLY (x,c,m,n)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	Argument	Real
c	Coefficient	Address
m	Order of numerator	Integer
n	Order of denominator	Integer

Result: Real in A and B

Errors: None

External References: .ENTR, .FAD, .FDV, .FMP

The two polynomials are defined as follows:

$$P(x) = P_m x^m + P_{m-1} x^{m-1} + \ldots + P_1 x + P_0$$

$$Q(x) = x^n + Q_{n-1} x^{n-1} + \ldots + Q_1 x + Q_0$$

The coefficient list c is stored sequentially in memory as follows:

$$P_m, P_{m-1}, \ldots, P_1, P_0, Q_{n-1}, Q_{n-2}, \ldots, Q_1, Q_0$$

$Q_n = 1.0$ is implied but not stored.

If n = 0, no coefficients are provided for Q, and only P is evaluated. Otherwise, the call evaluates the quotient P(x)/Q(x).

Any underflow or overflow that occurs invalidates the final result. M must be at least one. The A, B, X, Y, and E-Registers are undefined after this routine. The O-Register is set if any underflow or overflow occurs; otherwise, it is cleared.

Notes: Use DPOLY for speed and accuracy (if it is in microcode). SPOLY is best used for completion.

SQRT

Purpose: Calculate the square root of a real x.

Entry Points: SQRT

Assembly:
```
DLD x
JSB SQRT
JSB ERR0 (error)
<Return> (result in A and B)
```

FORTRAN: Function: SQRT (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: x < 0 ⟶ 03 UN

External References: .ZPRV, .FLUN, .PWR2, .FMP, .FAD, .FDV

TAN

Purpose: Calculate the tangent of a real x (radians).

Entry Points: TAN

Assembly:
```
DLD x
JSB TAN
JSB ERR0 (error)
<Return> (result in A and B)
```

FORTRAN: Function: TAN (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument (radians)	Real

Result: Real in A and B

Errors: x outside $[-8192\pi, +8191.75\pi]$ $\longrightarrow$ 09 OR

External References: .ZPRV, .CMRS, .FMP, .FAD, .FDV

TANH

Purpose: Calculate the hyperbolic tangent of a real x.

Entry Points: TANH

Assembly:
```
DLD x
JSB TANH
<Return> (result in A and B)
```

FORTRAN: Function: TANH (x)

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: None

External References: .ZPRV, .EXP, .FAD, .FSB, .FDV, .FMP

.ABS

Purpose: Find the absolute value of a double real.

Entry Points: .ABS

Assembly:

```
JSB .ABS
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DABS (with Y compiler option)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: None

External References: .CFER, .TSUB, .4ZRO, .ENTR

.ATAN

Purpose: Calculate the inverse tangent of a double real x.

Entry Points: .ATAN

Assembly:

```
JSB .ATAN
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DATAN (with Y compiler option)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real (radians)

Errors: None

External References: .ENTR, CRER, TRNC, .TDIV, ..TCM, .FLUN, .TSUB, /ATCG

.ATN2

Purpose: Calculate the arctangent of the quotient x/y of two double real variables x and y.

Entry Points: .ATN2, .ATA2

Assembly:

```
JSB .ATN2
DEF *+4
DEF z (result)
DEF x
DEF y
<error return>
<normal return>
```

FORTRAN: Function: DATN2 or DATAN2 (with Y compiler option)

Pascal: Callable using $ERROREXIT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	argument	Double Real
z	result	Double Real

Result: Double Real (radians)

Errors: x = y = 0 ⟶ 15 UN

External References: .ATAN, .TADD, .TSUB, .TDIV, .ENTR, .4ZRO, .CFER

.BLE

Purpose: Convert real x to double real y.

Entry Points: .BLE

Assembly:

```
JSB .BLE
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DBLE (with Y option)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Real
y	result	Double Real

Result: Double Real

Errors: None

External References: .ENTR

.CADD

Purpose: Add complex x to complex y.

Entry Points: .CADD

Assembly:

```
JSB .CADD
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Complex
y	argument	Complex
z	result	Complex

Result: Complex

Errors: Overflow bit set if result out of range. See Notes.

External References: .ETNC, .ZRNT, .FAD

Notes: Example FORTRAN usage:

```
complex*8 Xcmplx, Ycmplx, Rcmplx

Rcmplx = Xcmplx + Ycmplx
IF (OVF( )) Then
 (overflow was set, so result was out of range)
Endif
```

.CDBL

Purpose: Extract the real part of a complex x and return it as an extended precision real y.

Entry Points: .CDBL

Assembly:

```
JSB .CDBL
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Complex
y	result	Extended Real

Result: Extended Real

Errors: None

External References: DBLE

.CDIV

Purpose: Divide complex x by complex y.

Entry Points: .CDIV

Assembly:

```
JSB .CDIV
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Complex
y	argument	Complex
z	result	Complex

Result: Complex

Errors: Overflow bit set if result out of range.

External References: .ZRNT, .ENTC

.CFER

Purpose: Move four words from address x to address y. Used to copy a complex x to complex y.

Entry Points: .CFER

Assembly:

```
JSB .CFER
DEF y
DEF x
<Return>
A = direct address of (x+4)
B = direct address of (y+4)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	source	Complex
y	destination	Complex

Result: Complex in y

Errors: None

External References: .ZPRV

.CHEB

Purpose: Evaluate the Chebyshev series at a real x for a particular table of coefficients c.

Entry Points: .CHEB

Assembly:
```
DLD x
JSB .CHEB
DEF c (table, note 1)
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Result: Real in A and B

Errors: None

External References: .ZRNT, .FAD, .FMP, .FSB

Notes: Table C consists of a series of real coefficients terminated by an integer zero.

.CINT

Purpose: Convert the real part of a complex x to an integer.

Entry Points: .CINT

Assembly:

```
JSB .CINT
DEF x
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Complex

Result: Integer in A

Errors: None

External References: IFIX

.CMPY

Purpose: Multiply a complex x by a complex y.

Entry Points: .CMPY

Assembly:

```
JSB .CMPY
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Complex
y	argument	Complex
z	result	Complex

Result: Complex

Errors: Overflow bit set if result out of range.

External References: .ZRNT, .ENTC

.CMRS

Purpose: Reduce the argument for SIN, COS, TAN, EXP.

Entry Points: .CMRS

Assembly:

```
DLD x
JSB .CMRS
DEF C
DEF N
<error return>
<normal return> (Real result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real
C	argument	Extended Real
N	result	Integer

Result: Real and Extended Precision

Errors: N outside the range $[-2^{15}, 2^{15}]$ gives error return.

External References: .ZPRV, .XMPY, .XSUB, SNGL, IFIX, FLOAT

.COS

Purpose: Calculate the cosine of double precision x (radians).

Entry Points: .COS

Assembly:

```
JSB .COS
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DCOS (with y option)

Pascal: Callable using $ERROREXIT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: x outside $[-2^{23}, 2^{23}]$ ⟶ 05 OR

External References: .ENTR, /CMRT, DPOLY, ..TCM

.CPM

Purpose: Compares two single integer arguments.

Entry Points: .CPM

Assembly:

```
JSB .CPM
DEF ARG1
DEF ARG2

<Return> if (ARG1 = ARG2)
<Return> if (ARG1 < ARG2)
<Return> if (ARG1 > ARG2)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
ARG1	argument	Integer
ARG2	argument	Integer

Result: None

Errors: None

Notes: Pass arguments or addresses of arguments via A- or B-Registers (address 0 or 1). This subroutine does not restrict the defined addresses of arguments 1 and 2.

.CSUB

Purpose: Subtract a complex y from a complex x.

Entry Points: .CSUB

Assembly:

```
JSB .CSUB
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Complex
y	argument	Complex
z	result	Complex

Result: Complex

Errors: Overflow bit set if result out of range.

External References: .ENTC, .ZRNT

.CTBL

Purpose: Convert the real part of a complex real to a double real.

Entry Points: .CTBL

Assembly:

```
JSB .CTBL
DEF y (result)
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Complex
y	result	Double Real

Result: Double Real

Errors: None

External References: .BLE

.CTOI

Purpose: Raise a complex x to an integer power i.

Entry Points: .CTOI

Assembly:

```
JSB .CTOI
DEF z (result)
DEF x
DEF i
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Complex
i	exponent	Integer
z	result	Complex

Result: Complex

Errors: $x = 0, i \leq 0 \longrightarrow$ 14 UN

External References: .CMPY, .CDIV, .CFER, .ENTC, .ZRNT

.DCPX

Purpose: Convert an extended real x to a complex y.

Entry Points: .DCPX

Assembly:

```
JSB .DCPX
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Complex

Result: Complex

Errors: None

External References: SNGL, CMPLX

.DFER

Purpose: Extended real transfer.

Entry Points: .DFER

Assembly:

```
JSB .DFER
DEF y
DEF x
<Return>
A = direct address of x+3
B = direct address of y+3
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	source	Extended Real
y	destination	Extended Real

Result: Extended Real

Errors: None

External References: .ZPRV

.DINT

Purpose: Convert an extended real x to an integer.

Entry Points: .DINT, .XFTS

Assembly:

```
JSB .DINT
DEF x
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Extended Real

Result: Integer in A

Errors: None

External References: SNGM, IFIX, .ZPRV

.DTBL

Purpose: Convert extended real to double real.

Entry Points: .DTBL

Assembly:

```
JSB .DTBL
DEF y(result)
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	result	Double Real

Result: Double Real

Errors: None

External References: .XFER

.DTOD

Purpose: Raise an extended real x to an extended real power y.

Entry Points: .DTOD

Assembly:

```
JSB .DTOD
DEF z (result)
DEF x
DEF y
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	exponent	Extended Real
z	result	Extended Real

Result: Extended Real

Errors: See Notes.

External References: DEXP, DLOG, .XMPY, .DFER, .ENTC, .ZRNT

Notes:

$x, y \leq 0 \longrightarrow$ (13 UN)

$x > (1 - 2^{-39})2^{127} \longrightarrow$ (10 OF)

.DTOI

Purpose: Raise an extended real x to an integer power i.

Entry Points: .DTOI

Assembly:

```
JSB .DTOI
DEF y (result)
DEF x
DEF i
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Extended Real
i	exponent	Integer
y	result	Extended Real

Result: Extended Real

Errors: If $x = 0, i \leq 0 \longrightarrow$ 12 UN

External References: .XMPY, .XDIV, .DFER, .ZRNT

.DTOR

Purpose: Raise an extended real x to a real power y.

Entry Points: .DTOR

Assembly:

```
JSB .DTOR
DEF z (result)
DEF x
DEF y
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	exponent	Real
z	result	Extended Real

Result: Extended Real

Errors: See Notes.

External References: .DTOD, DBLE

Notes:

$x, y \leq 0 \longrightarrow$ (13 UN)

$x > (1 - 2^{-39})2^{127} \longrightarrow$ (10 OF)

.EXP

Purpose: Calculate e^x where x is double real.

Entry Points: .EXP

Assembly:

```
JSB .EXP
DEF *+3
DEF z (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DEXP (x) (with Y option)

Pascal: Callable using $ERROREXIT directive

Parameters:

Parameter	Description	Type
x	exponent	Double Real
y	result	Double Real

Result: Double Real

Errors: x > 127*LN(2) gives error code 07 OF

External References: .ENTR, .CFER, .4ZRO, /CMRT, /EXTH

Notes: For x < −129*LN(2), a zero will be returned with no error indication.

.FAD, .FSB

Purpose: .FAD: Add real x to y. .FSB: Subtract real y from x.

Entry Points: .FAD, .FSB

Assembly:

```
DLD x                              DLD x
JSB .FAD          or               JSB .FSB
<Return> (result in A and B)       Return (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real

External References: .PACK, .ZPRV

.FDV

Purpose: Divide a real x by y.

Entry Points: .FDV

Assembly:

```
DLD x
JSB .FDV
DEF y
<Return> (quotient in A and B, O set if under/overflow)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	dividend	Real
y	divisor	Real

Result: Real in A and B

Errors: None

External References: .PACK, .ZPRV

.FLUN

Purpose: "Unpack" a real x; place exponent in A, lower part of mantissa in B.

Entry Points: .FLUN

Assembly:
```
DLD x
JSB .FLUN
<Return> exponent in A, lower mantissa in B
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Return exponent in A, lower mantissa in B

Errors: None

External References: .ZPRV

.FMP

Purpose: Multiply a real x by y.

Entry Points: .FMP

Assembly:

```
DLD y
JSB .FMP
DEF x
<Return> (product in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real
y	argument	Real

Result: Real in A and B

Errors: None

External References: .PACK, .ZPRV

.FPWR

Purpose: Calculate x^i for real x and unsigned integer i.

Entry Points: .FPWR

Assembly:
```
LDA i
JSB .FPWR
DEF x
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real
i	exponent	Unsigned Integer

Result: Real in A and B

Errors: None

External References: .FMP, FLOAT, .FLUN

Notes:

1. i must be in the range [2,32768]
2. If overflow occurs, the maximum positive number is returned with overflow set. Overflow is set if underflow occurs.
3. The X- and Y-Registers may be altered.

.ICPX

Purpose: Convert an integer i to a complex y.

Entry Points: .ICPX

Assembly:

```
LDA i
JSB .ICPX
DEF y
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
i	argument	Integer
y	result	Complex

Result: Complex

Errors: None

External References: FLOAT, CMPLX

.IDBL

Purpose: Convert an integer i to an extended real y.

Entry Points: .IDBL, .XFTS

Assembly:

```
LDA i
JSB .IDBL
DEF y
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
i	argument	Integer
y	result	Extended Real

Result: Extended Real

Errors: None

External References: FLOAT, DBLE

.IENT

Purpose: Calculate the greatest integer not algebraically exceeding a real x.

Entry Points: .IENT

Assembly:
```
DLD x
JSB .IENT
JSB error routine
<Return> (result in A)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Integer in A

Errors: Exponent (x) > 14, user must supply error routine

External References: IFIX, .FLUN, FLOAT, .ZPRV

.ITBL

Purpose: Convert an integer x to a double real y.

Entry Points: .ITBL, .TFTS

Assembly:

```
LDA x
JSB .ITBL
DEF y (result)
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Integer
y	result	Double Real

Result: Double Real

Errors: None

External References: .BLE, FLOAT

.ITOI

Purpose: Calculate i^j for integer i and j.

Entry Points: .ITOI

Assembly:

```
JSB .ITOI
DEF i
DEF j
JSB ERR0 (error return)
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
i	argument	Integer
j	exponent	Integer

Result: Integer in A

Errors: Overflow if result is greater than 32767

External References: .ZPRV

.LOG

Purpose: Calculate the natural logarithm of a double real x.

Entry Points: .LOG

Assembly:

```
JSB .LOG
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DLOG (with Y option)

Pascal: Callable using $ERROREXIT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: X < 0 ⟶ 02 UN

External References: .ENTR, .CFER, .FLUN, .TADD, .TMPY, TRNL, /ATLG, FLOAT

.LOG0

Purpose: Calculate the common (base 10) logarithm of a double real x.

Entry Points: .LOG0 (.LOGT)

Assembly:

```
JSB .LOG0 (.LOGT)
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DLOGT (or DLOG10) (with Y option)

Pascal: Callable using $ERROREXIT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: $x \leq 0$ gives error code 02 UN

External References: .LOG, .TMPY, .ENTR

.MANT

Purpose: Extract the mantissa of a real x.

Entry Points: .MANT

Assembly:
```
DLD x
JSB .MANT
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real mantissa in A and B

Errors: None

External References: .ZPRV

.MAX1, .MIN1

Purpose: Find the maximum (or minimum) of a list of double reals.

Entry Points: .MAX1, .MIN1

Assembly:

```
JSB .MAX1          or      JSB .MIN1
DEF *+N+2                  DEF *+N+2
DEF z (result)             DEF z (result)
DEF a                      DEF a
DEF b                      DEF b
 :  :                       :  :
DEF n                      DEF n
<Return>                   <Return>
```

FORTRAN: Functions: DMIN1 (with y option)
DMAX1 (with y option)
DMIN1 (with y option)

Pascal: Callable

Parameters:

Parameter	Description	Type
a,b,...,n	argument	Double Real
z	result	Double Real

Result: Double Real

Errors: None

External References: .CFER, .TSUB, .4ZRO

Notes:

1. If there is only one argument in the list, it is considered to be both the maximum and minimum of the list.
2. If the list is null, zero is returned.

.MOD

Purpose: Calculate the remainder of x/y, where x, y and result are double reals.

Entry Points: .MOD

Assembly:

```
JSB .MOD
DEF *+4
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Function: DMOD (x,y) (with y option)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	dividend	Double Real
y	divisor	Double Real
z	result	Double Real

Result: Double Real

Errors: If y = 0 then the result is zero.

External References: .CFER, .TSUB, .TMPY, .TDIV, .YINT, .ENTR, .4ZRO

Notes:

1. The function .MOD returns x if y=0, or x/y overflows or underflows.
2. If an overflow or underflow occurs elsewhere in the calculation, the result will be incorrect.
3. No attempt is made to recover precision lost in the subtract.

.MPY

Purpose: Replace the subroutine call with the hardware instruction to multiply integer i and j.

Entry Points: .MPY

Assembly:
```
LDA j
JSB .MPY
DEF i
<Return> (result in A and B) (See Notes)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
i	argument	Integer
j	argument	Integer

Result: Double Integer in A and B

Errors: None

External References: .MAC.

Notes:

1. B contains the most significant bits of product; A contains the least significant bits.
2. Because the subroutine call is replaced by the hardware instruction, the routine is called only once for each subroutine call.

.NGL

Purpose: Convert double real x to real.

Entry Points: .NGL

Assembly:

```
JSB .NGL
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Double Real

Result: Real in A and B

Errors: None

External References: SNGL, .CFER

Notes: The result is rounded unless this would cause overflow. If so, overflow is set and the result is truncated to the greatest positive number.

.PACK

Purpose: Convert the signed mantissa of a real x into normalized real format.

Entry Points: .PACK

Assembly:

```
DLD x
JSB .PACK
BSS 1 (exponent)
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real
BSS 1	exponent returned	Integer

Result: Real in A and B

Errors: None

External References: .ZPRV

.PWR2

Purpose: Multiply a number by 2 to an integer power ($x*2^n$).

Entry Points: .PWR2

Assembly:

```
DLD x
JSB .PWR2
DEF n
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real
n	exponent	Integer

Result: Real in A and B

Errors: None

External References: .ZPRV

.RTOD

Purpose: Raise a real x to a double real power y.

Entry Points: .RTOD

Assembly:

```
JSB .RTOD
DEF z (result)
DEF x
DEF y
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Real
y	exponent	Double Real
z	result	Double Real

Result: Double Real

Errors: See Notes.

External References: .DTOD, DBLE

Notes:

$x = 0, y \leq 0 \longrightarrow$ (13 UN)
$x < 0, y \leq 0 \longrightarrow$ (13 UN)
$x > (1-2^{-39})2^{127} \longrightarrow$ (10 OF)

.RTOI

Purpose: Calculate x^i for real x and integer i.

Entry Points: .RTOI

Assembly:

```
JSB .RTOI
DEF x
DEF i
JSB ERR0
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Real
i	exponent	Integer

Result: Real in A and B

Errors:

$x = 0, i \leq 0$ ⟶ 06 UN

$x^{|i|} > 2^{127}$ ⟶ (floating point overflow)

External References: .FPWR, .FDV

.RTOR

Purpose: Calculate x^y for real x and y.

Entry Points: .RTOR

Assembly:

```
JSB .RTOR
DEF x
DEF y
JSB ERR0
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Real
y	exponent	Real

Result: Real in A and B

Errors: $x < 0$ or ($x = 0$ and $y < 0$) ⟶ 04 UN

$|x^*ALOG(x)| \geq 124$ ⟶ 07 OF

On error return, the overflow bit is set.

External References: ALOG, EXP, .ZRNT, .FMP

.RTOT

Purpose: Calculate x^y, where x is a real and y is a double real.

Entry Points: .RTOT

Assembly:

```
JSB .RTOT
DEF z (result)
DEF x
DEF y
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Real
y	exponent	Double Real
z	result	Double Real

Result: Double Real

Errors: See Notes.

External References: .TTOT

Notes:

1. Underflow gives a zero result, with no error. Overflow returns the greatest positive number, sets overflow (cleared otherwise), and gives an error code of 07 OF.

2. If ($x < 0$) or ($x = 0$ and $y \leq 0$), there is an error code of 13 UN.

.SIGN

Purpose: Transfer the sign of a double real y to a double real x.

Entry Points: .SIGN

Assembly:

```
JSB .SIGN
DEF *+4
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	argument	Double Real
z	result	Double Real

Result: Double Real

Errors: None

External References: .CFER, .TSUB, .4ZRO, .ENTR

Notes:

1. Overflow is set or cleared depending on occurrence. Overflow only occurs if $y \geq 0$ and x is the maximum negative number.
2. .SIGN(x,0) = |x|

.SIN

Purpose: Calculate the sine of double precision x (radians).

Entry Points: .SIN

Assembly:

```
JSB SIN
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DSIN (x) (with y option)

Pascal: Callable using $ERROREXIT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: x outside $[-2^{23}, 2^{23}] \longrightarrow$ 05 OR

External References: .ENTR, /CMRT, DPOLY, ..TCM

.SQRT

Purpose: Calculate the square root of a double real x.

Entry Points: .SQRT

Assembly:

```
JSB .SQRT
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DSQRT (x) (with y option)

Pascal: Callable using $ERROREXIT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: x < 0 ⟶ error code 03 UN

External References: .ENTR, .CFER, .PWRZ, .TDJV, .XADD, .XDIV, .TADD, .SQRT

.TADD, .TSUB, .TMPY, .TDIV

Purpose: Double real arithmetic.

Entry Points: .TADD, .TSUB, .TMPY, .TDIV

Assembly:

```
JSB .TADD
DEF z (result z=x+y)
DEF x
DEF y
<Return>
```

or

```
JSB .TSUB
DEF z (result z=x-y)
DEF x
DEF y
<Return>
```

```
JSB .TMPY
DEF z (result z=x*y)
DEF x
DEF y
<Return>
```

or

```
JSB .TDIV
DEF z (result z=x/y)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	argument	Double Real
z	result	Double Real

Result: Double Real

Errors: None

External References: .FLUN, .XFER, .CFER, FLOAT

Notes: If underflow occurs, zero is returned with overflow set. If overflow or divide by zero occurs, the largest positive number is returned with overflow set. Otherwise, overflow is cleared.

.TAN

Purpose: Calculate the tangent of a double real x (radians).

Entry Points: .TAN

Assembly:

```
JSB .TAN
DEF *+3
DEF y (result)
DEF x
<error return>
<normal return>
```

FORTRAN: Function: DTAN (x) (with y option)

Pascal: Callable using $ERROREXIT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: x outside $[-2^{23}, 2^{23}]$ ⟶ OR

External References: .ENTR, /CMRT, TRNL, .TDIV

.TANH

Purpose: Calculate the hyperbolic tangent of a double real x.

Entry Points: .TANH

Assembly:

```
JSB .TANH
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DTANH (x) (with y option)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: None

External References: .ENTR, .CFER, .TADD, .TDIV, /CMRT, /EXTH, .4ZRO

.TCPX

Purpose: Convert a double real x to a complex real y. The second value is set to zero.

Entry Points: .TCPX

Assembly:
```
JSB .TCPX
DEF y (result)
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Complex Real

Result: Complex Real

Errors: None

External References: .NGL

Notes: The result is rounded unless this would cause overflow. If so, overflow is set and the result is truncated to the greatest positive number.

.TDBL

Purpose: Convert double real to extended real without rounding.

Entry Point: .TDBL

Assembly:

```
JSB .TDBL
DEF y(result)
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Extended Real

Result: Double Real

Errors: None

External References: .XPAK, .XFER, .FLUN

.TENT

Purpose: Find the greatest double real i of integer value less than or equal to a double real (floor x).

Entry Points: .TENT

Assembly:

```
JSB .TENT
DEF *+3
DEF i (result)
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Double Real
i	result	Double Real

Result: Double Real

Errors: None

External References: .FLUN, .ENTR, .CFER

Notes: Result is a double real value with no bits set after the binary point.

.TINT

Purpose: Convert a double real x to an integer.

Entry Points: .TINT, .TFXS

Assembly:
```
JSB .TINT
DEF x
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real

Result: Integer in A

Errors: None

External References: IFIX

Notes: If the argument is outside the range $[-2^{15}, 2^{15}]$, the result is $2^{15}-1$, and overflow is set. Overflow is cleared otherwise.

.TPWR

Purpose: Calculate x^i, where x is a double real and i is an unsigned integer.

Entry Points: .TPWR

Assembly:

```
LDA i
JSB .TPWR
DEF y (result)
DEF x
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
i	exponent	Unsigned Integer
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: None

External References: .TMPY, FLOAT, .FLUN, .CFER

Notes:

1. i must be in the range [2,32768].
2. If overflow occurs, the maximum positive number is returned with overflow set. Overflow is set if underflow occurs.
3. The X- and Y-Registers may be altered.

.TTOI

Purpose: Calculate x^i , where x is a double real and i is an integer.

Entry Points: .TTOI

Assembly:

```
JSB .TTOI
DEF y (result)
DEF x
DEF i
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
i	exponent	Integer
x	argument	Double Real
y	result	Double Real

Result: Double Real

Errors: $x = 0, i \leq 0 \longrightarrow$ 12 UN
$x^{|i|} \geq 2^{127} \longrightarrow$ (Floating point overflow)

External References: .TPWR, .TDIV, .CFER, .4ZRO

.TTOR

Purpose: Raise a double real x to a real power y.

Entry Points: .TTOR

Assembly:

```
JSB .TTOR
DEF z (result)
DEF x
DEF y
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	exponent	Real
z	result	Double Real

Result: Double Real

Errors: See .TTOT

External References: .TTOT

.TTOT

Purpose: Calculate x^y , where x and y are both double reals.

Entry Points: .TTOT

Assembly:

```
JSB .TTOT
DEF z (result)
DEF x
DEF y
<error return>
<normal return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT, $ERROREXIT directives

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	exponent	Double Real
z	result	Double Real

Result: Double Real

Errors: See Notes.

External References: .LOG, .EXP, .CFER, .TMPY, .4ZRO

Notes:

1. Underflow gives a zero result, with no error. Overflow returns no result and gives an error code of 07 OF.
2. If ($x < 0$) or ($x = 0$ and $y \leq 0$), there will be an error code of 13 UN.

.XADD, .XSUB

Purpose: Extended real addition and subtraction.

Entry Points: .XADD, .XSUB

Assembly:

```
JSB (.XADD or .XSUB)
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	argument	Extended Real
z	result	Extended Real

Result: Extended Real

External References: .XPAK, ADRES, .ZPRV

.XCOM

Purpose: Complement an extended real unpacked mantissa in place. Upon return, the A-Register = 1 if exponent should be adjusted; otherwise, A = 0.

Entry Points: .XCOM

Assembly:
```
JSB .XCOM
DEF x
ADA (exponent)
STA (exponent)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real

Result: Double Real

Errors: None

External References: .ZPRV

.XDIV

Purpose: Divide an extended real x by an extended real y.

Entry Points: .XDIV

Assembly:

```
JSB .XDIV
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	dividend	Extended Real
y	divisor	Extended Real
z	result	Extended Real

Result: Extended Real

Errors: None

External References: .ZRNT, .XPAK

.XFER

Purpose: Move three words from address x to address y. Used for extended real transfers.

Entry Points: XFER

Assembly:

```
LDA (address x)
LDB (address y)
JSB .XFER
<Return> (A = direct address x+3)
         (B = direct address y+3)
```

FORTRAN: Not callable

Pascal: Not callable

Result: Extended Real

Errors: None

External References: .DFER, .ZPRV

.XMPY

Purpose: Multiply an extended real x by an extended real y.

Entry Points: .XMPY

Assembly:

```
JSB .XMPY
DEF z (result)
DEF x
DEF y
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Extended Real
y	argument	Extended Real
z	result	Extended Real

Result: Extended Real

Errors: None

External References: .XPAK, .ZPRV

.XPAK

Purpose: Extended precision mantissa is normalized, rounded, and packed with exponent in place; result is an extended real.

Entry Points: .XPAK

Assembly:
```
LDA exponent
JSB .XPAK
DEF z (3-word mantissa)
<Return> (result in z)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
z	argument	Extended Real

Result: Extended Precision

Errors: None

External References: .ZPRV

If the result is outside the range:

$$[-2^{128}, 2^{127}(1-2^{-39})],$$

then the overflow bit is set and

$$z = 2^{127}(1-2^{-39}).$$

If the result is within the range:

$$[-2^{129}(1 + 2^{-22}), 2^{-129}],$$

then the overflow bit is set and $z = 0$.

.XPLY, XPOLY

Purpose: Evaluate extended real polynomial.

Entry Points: .XPLY, XPOLY

Assembly:

```
JSB .XPLY or XPOLY
DEF *+5
DEF y (result)
DEF n (degree + 1)
DEF x
DEF c  (first element of coefficient array)
<Return>
```

FORTRAN: Callable

Pascal: Callable

Parameters:

Parameter	Description	Type
y	result	Extended Real
n	degree of polynomial + 1	Integer
x	argument	Extended Real
c	coefficient list	Address

Result: Extended Real

Errors: If $n \leq 0$, $y = 0$

External References: .ZRNT, .ENTP, .XADD, .XMPY, .DFER

.YINT

Purpose: Truncate the fractional part of a double real.

Entry Points: .YINT

Assembly:

```
JSB .YINT
DEF *+3
DEF y (result)
DEF x
<Return>
```

FORTRAN: Function: DDINT (with y option)

Pascal: Callable

Parameters:

Parameter	Description	Type
x	argument	Double Real
y	result	Double Real

Result: Double Real (See Notes)

Errors: None

External References: .TENT, .TADD, .ENTR

Notes: Result is a double real value with no bits set after the binary point.

..CCM

Purpose: Complement a complex variable x in place.

Entry Points: ..CCM

Assembly:

```
JSB ..CCM
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Complex

Result: Complex

Errors: None

External References: ..DLC

..DCM

Purpose: Complement an extended real in place.

Entry Points: ..DCM

Assembly:
```
JSB ..DCM
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Extended Real

Result: Extended Real

Errors: See Notes.

External References: .ZRNT, .XSUB

Notes: If x is the smallest negative number (-2^{127}), then result is the largest positive number [$(1-2^{-23})*2^{127}$] and the overflow bit is set.

..DLC

Purpose: Load and complement a real x.

Entry Points: ..DLC

Assembly:

```
JSB ..DLC
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: None

External References: .ZPRV, .FSB

..FCM

Purpose: Complement a real x.

Entry Points: ..FCM

Assembly:

```
DLD x
JSB ..FCM
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Real in A and B

Errors: None

External References: .ZRPV, .FSB

..TCM

Purpose: Negate a double real.

Entry Points: ..TCM

Assembly:
```
JSB ..TCM
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable using $DIRECT directive

Parameters:

Parameter	Description	Type
x	argument	Double Real

Result: Double Real

Errors: None

External References: .TSUB, .4ZRO

Notes: This routine negates a double real number in place by subtracting it from zero. If the number is -2^{127}, the result is $(1-2^{-55})2^{127}$ and overflow is set. If the number is 2^{-129}, the result is zero and overflow is clear.

#COS

Purpose: Call-by-name entry to COS.

Entry Points: #COS

Assembly:

```
JSB #COS
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Complex

Errors: None

External References: ERR0, .ENTR, CCOS

#EXP

Purpose: Call-by-name entry to CEXP.

Entry Points: #EXP

Assembly:

```
JSB #EXP
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Complex

Errors: None

External References: ERR0, .ENTR, CEXP

#LOG

Purpose: Call-by-name entry to CLOG.

Entry Points: #LOG

Assembly:

```
JSB #LOG
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Complex

Errors: None

External References: ERR0, .ENTR, CLOG

#SIN

Purpose: Call-by-name entry to CSIN.

Entry Points: #SIN

Assembly:

```
JSB #SIN
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Complex

Errors: None

External References: ERR0, .ENTR, CSIN

$EXP

Purpose: Call-by-name entry to DEXP.

Entry Points: $EXP

Assembly:

```
JSB $EXP
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Extended Real

Errors: None

External References: ERR0, .ENTR, DEXP

$LOG

Purpose: Call-by-name entry to DLOG.

Entry Points: $LOG

Assembly:

```
JSB $LOG
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Extended Real

Errors: None

External References: ERR0, .ENTR, DLOG

$LOGT

Purpose: Call-by-name entry to DLOGT.

Entry Points: $LOGT, $LOG0

Assembly:

```
JSB $LOGT (or $LOG0)
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Extended Real

Errors: None

External References: DLOGT, .ENTR, ERR0

$SQRT

Purpose: Call-by-name entry to DSQRT.

Entry Points: $SQRT

Assembly:

```
JSB $SQRT
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Extended Real

Errors: None

External References: DSQRT, ERR0, .ENTR

$TAN

Purpose: DTAN with no error return.

Entry Points: $TAN

Assembly:

```
JSB $TAN
DEF *+3
DEF y
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Extended Real

Errors: x outside $[-8192\pi, +8192.75\pi]$ ⟶ 09 OR

External References: DTAN, .ENTR

%ABS

Purpose: Call-by-name entry to IABS (i).

Entry Points: %ABS

Assembly:

```
JSB %ABS
DEF *+2
DEF i
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer in A

Errors: None

External References: IABS

%AN

Purpose: Call-by-name entry to TAN (x).

Entry Points: %AN

Assembly:

```
JSB %AN
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: See TAN

External References: TAN, ERR0

%AND

Purpose: Call-by-name entry to calculate the logical AND (product) of the two integers i and j.

Entry Points: %AND

Assembly:

```
JSB %AND
DEF *+3
DEF i
DEF j
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer

Errors: None

External References: None

%ANH

Purpose: Call-by-name entry to TANH (x)

Entry Points: %ANH

Assembly:

```
JSB %ANH
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: None

External References: TANH

%BS

Purpose: Call-by-name entry to ABS (x).

Entry Points: %BS

Assembly:

```
JSB %BS
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: None

External References: ABS

%FIX

Purpose: Call-by-name entry to IFIX (x).

Entry Points: %FIX

Assembly:

```
JSB %FIX
DEF *+2
DEF x
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer in A

Errors: None

External References: IFIX

%IGN

Purpose: Call-by-name entry to SIGN (x,z)

Entry Points: %IGN

Assembly:

```
JSB %IGN
DEF *+3
DEF x
DEF z
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real

Errors: None

External References: SIGN

%IN

Purpose: Call-by-name entry to SIN (x).

Entry Points: %IN

Assembly:

```
JSB %IN
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: See SIN

External References: SIN, ERR0

%INT

Purpose: Call-by-name entry to AINT (x).

Entry Points: %INT

Assembly:

```
JSB %INT
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real

Errors: None

External References: AINT

%LOAT

Purpose: Call-by-name entry to FLOAT (i).

Entry Points: %LOAT

Assembly:

```
JSB %LOAT
DEF *+2
DEF I
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: None

External References: FLOAT

%LOG

Purpose: Call-by-name entry to ALOG (x).

Entry Points: %LOG

Assembly:

```
JSB %LOG
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: See ALOG

External References: ALOG, ERR0

%LOGT

Purpose: Call-by-name entry to ALOGT (x).

Entry Points: %LOGT, %LOG0

Assembly:

```
JSB %LOGT (or %LOG0)
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real

Errors: See ALOGT

External References: ALOGT, ERR0

%NT

Purpose: Call-by-name entry to INT (x).

Entry Points: %NT

Assembly:

```
JSB %NT
DEF *+2
DEF x (real)
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer

Errors: None

External References: INT

%OR

Purpose: Call-by-name entry to calculate the inclusive OR of two integers, i and j.

Entry Points: %OR

Assembly:

```
JSB %OR
DEF *+3
DEF i
DEF j
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer in A

Errors: None

External References: None

%OS

Purpose: Call-by-name entry to COS (x).

Entry Points: %OS

Assembly:

```
JSB %OS
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: See COS

External References: COS, ERR0

%OT

Purpose: Standard call-by-name subroutine for NOT function.

Entry Points: %OT

Assembly:

```
JSB %OT
DEF *+2
DEF i
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer in A

Errors: None

External References: None

%QRT

Purpose: Call-by-name entry to SQRT (x).

Entry Points: %QRT

Assembly:

```
JSB %QRT
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: See SQRT

External References: SQRT, ERR0

%SIGN

Purpose: Call-by-name entry to ISIGN (i,z).

Entry Points: %SIGN

Assembly:

```
JSB %SIGN
DEF *+3
DEF i
DEF z
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer in A

Errors: None

External References: ISIGN

%TAN

Purpose: Call-by-name entry to ATAN (x).

Entry Points: %TAN

Assembly:

```
JSB %TAN
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: See ATAN

External References: ATAN, ERR0

%XP

Purpose: Call-by-name entry to EXP (x).

Entry Points: %XP

Assembly:

```
JSB %XP
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Callable

Pascal: Callable

Result: Real in A and B

Errors: See EXP

External References: EXP, ERR0

/ATLG

Purpose: Compute (1−x)/(1+x) in double precision.

Entry Points: /ATLG

Assembly:
```
JSB /ATLG
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Double Real

Errors: None

External References: .TADD, .TSUB, .TDIV

Notes:

1. No error checking is performed.
2. The X- and Y-Registers may be changed.

/COS

Purpose: .COS with no error return

Entry Points: /COS

Assembly:

```
JSB /COS
DEF *+3
DEF <result>
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Double Real

Errors: None

External References: .COS, .ENTR

/CMRT

Purpose: Range reduction for .SIN, .COS, .TAN, .EXP, and .TANH.

Entry Points: /CMRT

Assembly:

```
LDA <flag>
JSB /CMRT
DEF <result>
DEF <constant>
DEF <argument>
<error return>
<normal return> (B-Register contains least significant
                 bits of n)
```

FORTRAN: Not callable

Pascal: Not callable

Result: Double Real

Errors:

External References: .CFER, .TADD, .TSUB, .TMPY, .PWR2, .YINT, .FLUN, IFIX, FLOAT, .FSB, .FAD

Notes:

1. This routine may alter the X- and Y-Registers.
2. This routine should be used by system programs only.

/EXP

Purpose: .EXP with no error return.

Entry Points: /EXP

Assembly:

```
JSB /EXP
DEF *+3
DEF <result>
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Double Real

Errors: None

External References: .EXP, .ENTR

/EXTH

Purpose: Compute $2^n * 2^{x/2}$ or, if n = −32768, then TANH (x).

Entry Points: /EXTH

Assembly:

```
LDA <n>
JSB /EXTH
DEF <result>
DEF <x>
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Result: Double Real

Errors: None

External References: .PWR2, .TADD, DPOLY

Notes: No error checking is performed. The final exponent will be in error by a multiple of 128 if overflow or underflow occurs.

/LOG

Purpose: .LOG with no error return.

Entry Points: /LOG

Assembly:

```
JSB /LOG
DEF *+3
DEF <result>
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Double Real

Errors: None

External References: .LOG, .ENTR

/LOG0

Purpose: .LOG0 with no error return.

Entry Points: /LOG0 or /LOGT

Assembly:

```
JSB /LOG0 or /LOGT
DEF *+3
DEF <result>>
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Double Real

Errors: None

External References: .LOG0, .ENTR

/SIN

Purpose: Calculate the sine of a double real x with no error return.

Entry Points: /SIN

Assembly:

```
JSB /SIN
DEF *+3
DEF <result>
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Double Real

Errors: None

External References: .SIN, .ENTR

/SQRT

Purpose: .SQRT with no error return.

Entry Points: /SQRT

Assembly:

```
JSB /SQRT
DEF *+3
DEF <result>
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Double Real

Errors: None

External References: .SQRT, .ENTR

/TAN

Purpose: .TAN with no error return.

Entry Points: /TAN

Assembly:

```
JSB /TAN
DEF *+3
DEF <result>
DEF x
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Double Real

Errors: None

External References: .TAN, .ENTR

/TINT

Purpose: Conversion of double precision to integer.

Entry Points: /TINT

Assembly:

```
JSB /TINT
DEF *+2
DEF <arguments>
<Return> (result in A)
```

FORTRAN: Callable as IDINT with y option

Pascal: Callable

Result: Integer in A

Errors: Overflow set if argument outside $[-2^{15}, 2^{15}]$

External References: .TFXS, .ENTR

4

Double Integer Subroutines

This chapter documents additional mathematical subroutines for programs produced by the FORTRAN, Pascal, and BASIC compilers. The subroutines can also be called from assembly language. Many of the subroutines are available as microcoded subroutines; refer to the specific processor reference manual for more information.

Double integer values contained in the A- and B-Registers have the most significant bits in the A-Register. Values stored in memory require two locations. The operand address in a double integer instruction points to the first memory location, which contains the most significant bits.

Format of Routines

The subroutines in this chapter are presented in the following format:

Name	The name of the subroutine.
Purpose	The use of the subroutine.
Entry Points	The entry points to the subroutine.
Assembly	The Macro/1000 assembly language calling sequence for each entry point. "A" and "B" indicate the A- and B-Registers.
FORTRAN	A statement on whether or not the subroutine is callable in FORTRAN-77.
Pascal	A statement on whether or not the subroutine is callable in Pascal.
Parameters	An explanation of the parameters' form and value.
Result	The type of result and the registers used (if any) where the result is returned.
Errors	A summary of the error conditions reported by the subroutine. Errors generated by external references are not described. Refer to the *FORTRAN 77 Reference Manual*, part number 92836-90001, for a more complete discussion of error messages.
External References	Other subroutines that are called by the subroutine.
Notes	Additional information for using the subroutine.

FLTDR

Purpose: Convert a double integer to real.

Entry Points: FLTDR

Assembly:

```
JSB FLTDR
DEF *+2
DEF x
<Return> (result in A and B)
```

FORTRAN: Function

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer

Result: Real in A and B

Errors: None

External References: .FLTD, .ENTR

Notes: Should not be used for numbers exceeding 2^{23} as the conversion may not be exact for such numbers.

.DADS

Purpose: Double integer add and subtract.

Entry Points: .DAD, .DSB, .DSBR

Assembly:

```
DLD x          DLD x          DLD x
JSB .DAD       JSB .DSB       JSB .DSBR
DEF y          DEF y          DEF y
```

← <Return> (results in A and B) →

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer
y	argument	Double Integer

Result: Double Integer in A and B

For .DSB, value equals x−y
For .DSBR, value equals y−x

Errors: None

External References: None

Notes: If overflow occurs, the least significant 32 bits are returned with overflow set. Overflow is cleared otherwise. E is never cleared, but is set if carry (.DAD) or borrow (.DSB and .DSBR) occurs.

.DSBR replaces the sequence:

```
DST temp                        JSB .DSBR
DLD x          with             DEF x
JSB .DSB
DEF temp
```

.DCO

Purpose: Compare two double integers.

Entry Points: .DCO

Assembly:

```
DLD x
JSB .DCO
DEF y
<Return> (if x=y)
<Return> (if x<y)
<Return> (if x>y)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer
y	argument	Double Integer

Result: None

Errors: None

External References: None

Notes: A, B, E, and O are left unchanged. The compare is correct even if X−Y is not representable in 32 bits.

.DDE

Purpose: Decrement the double integer in the A- and B-Registers.

Entry Points: .DDE

Assembly:
```
DLD x
JSB .DDE
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer

Result: Double Integer in A and B

Errors: None

External References: None

Notes:

1. If the largest negative number is decremented, the largest positive number is the result, with overflow set. Overflow is cleared otherwise.
2. E is preserved unless x = 0, in which case it is set.

.DDI, .DDIR

Purpose: Double integer divide.

Entry Points: .DDI, .DDIR

Assembly:

```
DLD x                          DLD x
JSB .DDI                       JSB .DDIR
DEF y                          DEF y
<Return> (result in A and B)   <Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer
y	argument	Double Integer

Result: Double Integer in A and B

For .DDI, value equals x/y
For .DDIR, value equals y/x

Errors: None

External References: FLOAT

Notes: If overflow or divide by zero occurs, the largest positive integer is returned with overflow set. Overflow is cleared otherwise. E is preserved.

.DDIR is used to replace the sequence:

```
DST temp        with        JSB .DDIR
DLD x                       DEF x
JSB .DDI
DEF temp
```

.DDS

Purpose: Double integer decrement and skip if zero.

Entry Points: .DDS

Assembly:

```
JSB .DDS
DEF x
<Return> (if x–1 not equal to 0)
<Return> (if x–1 equal to 0)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer

Result: Double Integer

Errors: None

External References: None

Notes: This routine decrements the double integer x. A, B, E, and O are left unchanged, except that A and B are changed if the effective address is zero.

.DIN

Purpose: Increment the double integer in the A- and B-Registers.

Entry Points: .DIN

Assembly:
```
DLD x
JSB .DIN
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer

Result: Double Integer in A and B

Errors: None

External References: None

Notes: If the largest positive number is incremented, the largest negative number is the result, with overflow set. Overflow is cleared otherwise. E is preserved unless x = −1, in which case E is set.

.DIS

Purpose: Double integer increment and skip if zero.

Entry Points: .DIS

Assembly:

```
JSB .DIS
DEF x
<Return> (if x+1 not equal to 0)
<Return> (if x+1 equal to 0)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer

Result: Double Integer

Errors: None

External References: None

Notes: This routine increments the double integer x by 1. A, B, E, and O are left unchanged, except that A and B are changed if the effective address is zero.

.DMP

Purpose: Double integer multiply.

Entry Points: .DMP

Assembly:

```
DLD X
JSB .DMP
DEF y
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer
y	argument	Double Integer

Result: Double Integer in A and B

Errors: None

External References: None

Notes: If overflow occurs, the largest positive integer is returned with overflow set. Overflow is cleared otherwise. E is preserved.

.DNG

Purpose: Negate double integer x.

Entry Points: .DNG

Assembly:
```
DLD x
JSB .DNG
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer

Result: Double Integer in A and B

Errors: None

External References: None

Notes: If overflow occurs, the argument is returned unchanged and overflow is set. Overflow is cleared otherwise. E is preserved unless X = 0, in which case E = 1.

.FIXD

Purpose: Convert real to double integer.

Entry Points: .FIXD

Assembly:
```
DLD x
JSB .FIXD
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Real

Result: Double Integer in A and B

Errors: None

External References: .FLUN

Notes:

1. If the argument is outside the range $[-2^{31}, 2^{31}]$ the result is $2^{31}-1$ and overflow is set. Overflow is cleared otherwise.
2. .FXDE is not a usable entry point. It is referenced by .XFXD and .TFXD.

.FLTD

Purpose: Convert double integer to real.

Entry Points: .FLTD

Assembly:
```
DLD x
JSB .FLTD
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer

Result: Real in A and B

Errors: None

External References: .XPAK

Notes: If the argument is outside the range $[-2^{23}, 2^{23}]$, the excess low order bits are truncated. Positive numbers may become smaller, negative numbers may become smaller in value (larger in absolute value).

.TFTD

Purpose: Convert a double integer to a double real.

Entry Points: .TFTD

Assembly:

```
DLD x
JSB .TFTD
DEF y (result)
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer
y	result	Double Real

Result: Double Real

Errors: None

External References: .XPAK

.TFXD

Purpose: Convert a double real to a double integer.

Entry Points: .TFXD

Assembly:
```
JSB .TFXD
DEF x
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Real

Result: Double Integer in A and B

Errors: None

External References: .FLUN, .CFER, .FIXD, .FXDE

Notes: If the argument is outside the range $[-2^{31}, 2^{31}]$ the result is $2^{31}-1$ and overflow is set. Overflow is cleared otherwise.

.XFTD

Purpose: Convert a double integer to an extended real.

Entry Points: .XFTD

Assembly:

```
DLD x
JSB .XFTD
DEF y (result)
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Double Integer
y	result	Extended Real

Result: Extended Real

Errors: None

External References: .XPAK

.XFXD

Purpose: Convert extended real to double integer.

Entry Points: .XFXD

Assembly:
```
JSB .XFXD
DEF x
<Return> (result in A and B)
```

FORTRAN: Not callable

Pascal: Not callable

Parameters:

Parameter	Description	Type
x	argument	Extended Real

Result: Double Integer in A and B

Errors: None

External References: .DTBL, .TFXD

Notes: If the argument is outside the range $[-2^{31}, 2^{31}]$, the result is $2^{31}-1$ and overflow is set. Overflow is cleared otherwise.

5

Utility Subroutines

This chapter describes subroutines that can be used by application programs, as well as some that can be used by programs produced by compilers. All subroutines can be called from assembly code; some can also be called from FORTRAN, Pascal, and BASIC.

Format of Routines

The subroutines in Chapters 3 through 5 are presented in the following format:

Name	The name of the subroutine.
Purpose	The use of the subroutine.
Entry Points	The entry points to the subroutine.
Assembly	The Macro/1000 assembly language calling sequence for each entry point. "A" and "B" indicate the A- and B-Registers.
FORTRAN	A statement on whether or not the subroutine is callable in FORTRAN-77.
Pascal	A statement on whether or not the subroutine is callable in Pascal.
Parameters	An explanation of the parameters' form and value.
Result	The type of result and the registers used (if any) where the result is returned.
Errors	A summary of the error conditions reported by the subroutine. Errors generated by external references are not described. Refer to the *FORTRAN 77 Reference Manual*, part number 92836-90001, for a more complete discussion of error messages.
External References	Other subroutines that are called by the subroutine.
Notes	Additional information for using the subroutine.

ABREG

Purpose: Obtain the contents of the A- and B-Registers from high level languages.

Entry Points: ABREG

Assembly:
```
JSB ABREG
DEF *+3
DEF IA
DEF IB
<Return>
```

FORTRAN:
```
CALL ABREG (IA,IB)
IA <-- AREG
IB <-- BREG
```

Pascal: Callable

Errors: None

External References: None

Note that the FORTRAN compiler (as of Revision 5010) recognizes ABREG and emits STA IA and STB IB instead. IA and IB must be direct addresses (they cannot be subscripted array elements).

ER0.E

Purpose: Specify the LU for printing library error messages. ER0.E is defaulted to 1. (See also ERRLU.)

Entry Points: ER0.E

Assembly:

```
EXT ER0.E
.
.
.
LDA LU       TO CHANGE THE LU, USE THIS CODE
STA ER0.E
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Result: None

Errors: None

External References: None

ERRLU

Purpose: Change the LU for printing library error messages.

Entry Points: ERRLU

Assembly:
```
JSB ERRLU
DEF *+2
DEF NEWLU
```

FORTRAN: Callable

Pascal: Callable

Result: None

Errors: None

External References: ER0.E

Notes: If ERRLU is called with a number < 0, the LU is not changed. If the LU is changed to 0, the system inhibits error printing. (Not recommended.)

ERR0

Purpose: Print a 4-character error code and a memory address on the logical unit ER0.E.

Entry Points: ERR0

Assembly:

```
LDA nn
LDB xx
JSB ERR0
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Result: Printed

Errors: None

External References: REIO, ER0.E, PNAME

FTRAP, RTRAP

Purpose: Traps FORTRAN runtime errors to user defined subroutines.

Entry Points: FTRAP, RTRAP

Assembly:

```
EXT TRAP
EXT FTRAP
JSB FTRAP
DEF *+2
DEF TRAP
<Return> A,B,=old TRAP value
```

FORTRAN: Callable

Pascal: External Trap

```
Call FTRAP(TRAP)
```

Result: For a runtime error control transfers to a user defined subroutine TRAP. Define TRAP as follows:

```
SUBROUTINE TRAP (ABREG,PREG)
INTEGER ABREG(2),PREG(2)
   :
   :   TRAP handling code
```

where:

ABREG	is a two-word error code that contains an ASCII string or number (see below).
PREG(1)	is the location of the error (P-Register).
PREG(2)	is the code segment number (-1 if error is found in data space).

The return values are from the stack. The sign bit on the PREG(1) indicates that the location is not in the current segment. The stack value for the code segment is from Q+3 and is in the high 8 bits of PREG(2).

Notes: You have two options when your program exits this subroutine. You can either return to let the detected error get a runtime error message or you can send a modified runtime error by storing it in 'ABREG'. (Errors are printed on LU ER0.E.)

The TRAP routine receives the following error codes:

Groups 1 and 3	ABREG contains the ASCII string (up to four characters) from the library subroutine error table in Appendix A of the *FORTRAN 77 Reference Manual*.
Group 2	ABREG(1) contains an unconverted number. Refer to the runtime errors table in Appendix A of the *FORTRAN 77 Reference Manual*.

where:

Group 1 = the library errors.

Group 2 = the I/O and string errors.

Group 3 = the EOF error.

Standard error handlers using ERR0 continue the program after Group 1 errors, setting A=B=0.

Group 2&3 errors normally terminate the program after the error is printed.

Some Group 2&3 errors can be trapped before they get to TRAP by using the ERR and EOF options of the FORTRAN READ/WRITE statements.

You can change the TRAP address as often as you desire. You may want to save it on entry to a subroutine, set a different address, and then restore the old address. (See RTRAP below.)

See RT_ER routine for more details on how to format and print a run-time error.

Errors: None

External References: ER0.T

Example:

```
External TRAP
INTEGER*4 FTRAP,RTRAP,intrap ! RTRAP is the restore
  :                          ! TRAP routine.
itrap=FTRAP (trap)           ! Old TRAP is saved in
  :                          ! itrap (a double integer)
CALL RTRAP (itrap)           ! Restore the old TRAP
Return
```

Notes: RTRAP also returns, as a double integer function, the old trap value which can be restored by a subsequent call to RTRAP.

To turn off traps and return to standard error handling, set the trap to 0 using RTRAP. Call RTRAP (0J).

EXIT calls are not trapped.

Caution: In CDS programs the TRAP routine, the routine that calls FTRAP, and the routine that declares TRAP an EXTERNAL must all be in the same space, either Code or Data.

GETST

Purpose: Recovers the parameter string from a programs command string storage area.

Entry Points: GETST

Assembly:

```
JSB GETST
DEF RTN
DEF IBUF
DEF ILEN
DEF ILOG
RTN
.
.
.
IBUF BSS  n
ILEN DEC  n
ILOG NOP
.
```

FORTRAN: Callable

Pascal: Callable (See appropriate Pascal user's manual for restrictions.)

Result: None

Errors: None

External References: EXEC, .ENTP, .ZPRV

See also: GetRunString (this manual); EXEC 14 (documented in the *RTE-A Programmer's Reference Manual*, part number 92077-90007*)*; RCPAR (documented in the *FORTRAN 77 Reference Manual*, part number 92836-90001*)*.

IGET, IXGET

Purpose: Allow programs to read the contents of a memory address. IXGET uses the system map.

Entry Points: IGET, IXGET

Assembly:

```
JSB IGET                    or          JSB IXGET
DEF *+2                                 DEF *+2
DEF IADRS                               DEF IADRS
<Return> (results in A)                 <Return> (results in A)
```

FORTRAN: Callable as a function

Pascal: Callable

Result: Contents of memory address

Errors: None

External References: None

Notes: On IXGET for RTE-A VC+, address refers to data segment not code segment.

INAMR

Purpose: Read a 10-word parameter buffer generated by the NAMR routine and produce an output buffer delimited by colons.

Entry Points: INAMR

Assembly:
```
JSB INAMR
DEF *+5
DEF IPBUF
DEF OUTBUF
DEF LENGTH
DEF ISTRC
<Return>
```

FORTRAN: Callable:
```
if (INAMR (IPBUF,OTBUF,LENGTH,ISTRC) .LT. 0) GOTO 10
```

Pascal: Callable

Parameters:

IPBUF is the ten-word input parameter buffer. The ten words are described as follows:

Word 1 = 0 if type = 0 (see below);
16-bit number if type = 1; and
chars 1 and 2 if type = 3.

Word 2 = 0 if type = 0 or 1;
chars 3 and 4 or trailing space(s) if type = 3.

Word 3 = 0 if type = 0 or 1;
chars 5 and 6 or trailing space(s) if type = 3.

Word 4 = Parameter type of all seven parameters in two-bit pairs. Note the difference between NAMR parameter types, and those for the system library routine PARSE.

0 = Null parameter.
1 = Integer numeric parameter.
3 = Left justified 6 ASCII character parameter.

Word 5 = First subparameter, is delimited by colons and has characteristics of word 1.

Word 6 = Second subparameter, is delimited by colons and has characteristics of word 1.

Word 7 = Third subparameter, is delimited by colons and has characteristics of word 1.

Word 8 = Fourth subparameter, is delimited by colons and has characteristics of word 1.

Word 9 = Fifth subparameter, is delimited by colons and has characteristics of word 1.

Word 10 = Sixth subparameter, is delimited by colons and has characteristics of word 1.

OUTBUF is the starting address of output buffer containing the namr to be passed.

LENGTH is the character length of OUTBUF.

ISTRC is the starting character number in OUTBUF. This parameter is updated for possible next call to INAMR and the start character in OUTBUF. An empty OUTBUF array should start with ISTART = 0.

Note ISTRC is modified by this routine; it must be passed as a variable (not a constant) from caller (FTN).

Result: INAMR = −1 Insufficient space in buffer.

INAMR = 0 If the character string has been emitted.

Errors: None

External References: .ENTR

Notes: The following table shows the results returned in OTBUF given the contents of IPBUF.

```
IPBUF W1   W2  W3      W4  W5   W6  W7  W8   W9    W10   OTBUF

  1  12345  0   0  00001B   0    0   0   0    0      0 = 12345,
  2   DO   UG      00037B  DB  -10   0   0    0      0 = DOUG:DB:-10,
  3    0    0   0  00000B   0    0   0   0    0      0 = ,
  4   GE   OR  GE  00017B   A    0   0   0    0      0 = GEORGE:A,
  5   &P   AR  SE  12517B  JB    0   4  -1 1775 -22738 = &PARSE:JB::
                                                        4:-1:1775:
                                                        -22738,
```

Sample Program

```
FTN,L
      PROGRAM testi
      DIMENSION ib2(18),ib1(36),ipbuf(100)
      CALL RMPAR(ib1)
      lu = ib1
      IF (lu.EQ.0) lu = 1
    1 WRITE (lu,100)
  100 FORMAT (”Input ASCII namr’s to parse ?”)
      READ (lu,101) ib1
  101 FORMAT (36A2)
      CALL ITLOG(len)
      IF (len.EQ.0)STOP 77
      ISTRC = 1
      NCHRS = 0
  200 IFLG1 = NAMR(ipbuf,ib1,len,istrc)
      IF (IFLG1.LT.0) WRITE (LU,206)
  206 FORMAT (”namr out of data”)
      IFLG2 = INAMR(ipbuf,ib2,36,nchrs)
      IF (iflg2.lt.0) WRITE (lu,207)
  207 FORMAT (”inamr out of buffer”)
      IF(iflg1.LT.0.or.iflg2.LT.0) GO TO 1
      CALL EXEC (2,lu,ib2,-nchrs)
      GO TO 200
      END
```

IND.E

Purpose: Used by .INDR and .INDA routines to select output LU for error messages. Default is 6; a 0 inhibits messages (not recommended).

Entry Points: IND.E

Assembly:

```
EXT IND.E
LDA LU
STA IND.E
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Result: None

Errors: None

External References: None

ISSR

Purpose: Set the CPU S-Register to the value n.

Entry Points: ISSR

Assembly:
```
JSB ISSR
DEF *+2
DEF n
<Return>
```

FORTRAN:
```
CALL ISSR(n)
```

Pascal: Callable

Result: None

Errors: None

External References: None

ISSW

Purpose: Set the sign bit (15) of A-Register equal to bit n of the S-Register.

Entry Points: ISSW

Assembly:
```
LDA n
JSB ISSW
<Return> (result in A)
```

FORTRAN: Function: ISSW(n)

Pascal: Not callable

Result: Integer in A

Errors: None

External References: None

MAGTP

Purpose: Performs utility functions on magnetic tape and other devices: checks status, performs rewind/standby, writes a gap, and issues a clear request.

Entry Points: IEOF, IERR, IEOT, ISOT, LOCAL,RWSTB

Assembly: The calling sequence and purpose of each entry point is:

```
JSB IEOF
DEF *+2
DEF unit
Return
```
Returns a negative value in A if an end-of-file was encountered during last tape operation on the logical unit specified.

```
JSB IERR
DEF *+2
DEF unit
Return
```
Returns a negative value in A if a parity or timing error was not cleared after three read attempts during the last operation on the specified unit (cannot occur if EOF occurs).

```
JSB IEOT
DEF *+2
DEF unit
Return
```
Returns a negative value in A if an end-of-tape was encountered during the last forward movement of the specified unit.

```
JSB ISOT
DEF *+2
DEF unit
Return
```
Returns a negative value in A if the start-of-tape marker is under the tape head of the specified unit.

```
JSB LOCAL
DEF *+2
DEF unit
Return
```
Returns a negative value in A if the specified unit is in local mode.

```
JSB RWSTB
DEF *+2
DEF unit
<Return>
```
Rewinds the specified logical unit and sets it to LOCAL.

FORTRAN: Callable as a subroutine

Pascal: Callable

Result: Not applicable

Errors: Returns on illegal call

External References: .ENTR, EXEC

NAMR

Purpose: Read an input buffer of any length, parse the buffer delimited by colons as in a file namr, and produce a parameter buffer of 10 words.

Entry Points: NAMR

Assembly:
```
JSB NAMR
DEF *+5
DEF IPBUF
DEF INBUF
DEF LENGTH
DEF ISTRC
<Return>
```

FORTRAN: Callable:
```
If (NAMR (IPBUF,INBUF,LENGTH,ISTRC) .LT. 0) GOTO 10
```

Pascal: Callable

Parameters: IPBUF is the 10-word destination parameter buffer. The ten words are described as follows:

Word 1 = 0 if type = 0 (see below);
16-bit number if type = 1; and
chars 1 and 2 if type = 3.

Word 2 = 0 if type = 0 or 1;
chars 3 and 4 or trailing space(s) if type = 3.

Word 3 = 0 if type = 0 or 1;
chars 5 and 6 or trailing space(s) if type = 3.

Word 4 = Parameter type of all seven parameters in two-bit pairs. Note the difference between INAMR parameter types and those for the system library routine PARSE.

0 = Null parameter.
1 = Integer numeric parameter.
3 = Left justified 6 ASCII-character parameter.

Bits for FNAME : P1 : P2 : P3 : P4 : P5 : P6,
0,1 2,3 4,5 6,7 8,9 10,11 12,13

Word 5 = First subparameter, is delimited by colons and has characteristics of word 1.

Word 6 = Second subparameter, is delimited by colons and has characteristics of word 1.

Word 7 = Third subparameter, is delimited by colons and has characteristics of word 1.

Word 8 = Fourth subparameter, is delimited by colons and has characteristics of word 1.

Word 9 = Fifth subparameter, is delimited by colons and has characteristics of word 1.

Word 10 = Sixth subparameter, is delimited by colons and has characteristics of word 1.

INBUF is the starting address of input buffer containing the namr to be parsed.

LENGTH is the number of characters in INBUF.

ISTRC is the starting character number in INBUF. This parameter is updated for possible next call to NAMR and the start character in INBUF.

Note ISTRC is modified by this routine; it must be passed as a variable (not a constant) from caller (FTN).

Result: NAMR = −1 If no characters are in INBUF.

NAMR = 0 If the character string has been parsed. (See Note.)

Errors: None

External References: .ENTR

Notes: Examples that can be parsed by successive calls to NAMR:

```
+12345, DOUG:DB:-12B:,,GEORGE:A,&PARSE:JB::4:-1:1775:123456B
```

where:

NAMR#	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10
1	12345	0	0	00001B	0	0	0	0	0	
2	DO	UG		00037B	DB	-10	0	0	0	0
3	0	0	0	00000B	0	0	0	0	0	0
4	GE	OR	GE	00017B	A	0	0	0	0	0
5	&P	AR	SE	12517B	JB	0	4	-1	1775	-22738

Sample Program

```
FTN7X,L
        PROGRAM testn
        DIMENSION ib(36), ipbuf(10)
        WRITE(1,'("Input ASCII namrs to parse?")')
        CALL EXEC(1,401B,ib,-72)
        CALL ABREG(a,len)
        iscr = 1
        DO i=1,10
           IF (namr(ipbuf,ib,len,iscr) .LT.  0) GOTO 999

           WRITE(1,220) iscr, ipbuf, ipbuf
        END DO
   220  FORMAT(" "/I3,10(X,I6)/" "3A2,7(X,O6))
   999  STOP
        END
```

See also HpZParse and HpZDParse.

OVF

Purpose: Return value of overflow bit in bit 15 of the A-Register and clear the overflow bit.

Entry Points: OVF

Assembly:
```
JSB OVF
DEF RTN
<Return> (result in A)
```

FORTRAN: Callable (see Notes)

Pascal: Callable

Result: Integer in A

Errors: None

External References: None

Notes:
```
Logical *2 OVF
IF (OVF( )) Then
  overflow was set
Endif
```

FORTRAN does not clear the overflow register, so before using this routine to test for overflow, it should be called prior to the code which may set overflow, unless that code will also clear it. (Refer to your CPU reference manual's description for the instructions involved.)

Note also that the single precision integer MPY instruction never sets overflow, (but returns a *4 integer); nor does the compiler check for overflow when it truncates *4 integers to *2 integers.

PAU.E

Purpose: Used by .PAUS and .STOP routines to select LU on which to output pause message. Default is LU 1; a 0 inhibits message (not recommended).

Entry Points: PAU.E

Assembly:

```
EXT PAU.E
LDA LU
STA PAU.E
<Return>
```

FORTRAN: Not callable

Pascal: Not callable

Result: None

Errors: None

External References: None

PNAME

Purpose: Copies the runtime name of the currently executing program from the program's ID segment to a three word array.

Entry Points: PNAME

Assembly:
```
JSB PNAME
DEF *+2
DEF IARAY
→ <Return>

IARAY    BSS 3
```

FORTRAN:
```
CALL PNAME (IARAY)
```

Pascal: Callable

Result: ASCII characters

Errors: None

External References: .ENTR, $OPSY

Notes: The sixth character is returned as an ASCII space.

Sample Program

```
    PROGRAM prnam
    DIMENSION iaray(3)
    CALL PNAME (iaray)
    WRITE (1,100) iaray
100 FORMAT (”  Program ”,3A2,”executing”)
    STOP
    END
```

PTAPE

Purpose: Position a magnetic tape unit by spacing forward or backward a number of files and/or records.

Entry Points: PTAPE

Assembly:

```
JSB PTAPE
DEF *+4
DEF logical unit
DEF file count (see notes)
DEF record count (see notes)
<Return>
```

For example:

```
  0  = Make no file movements.
 −1  = Backspace to the beginning of the current file.
  1  = Forward space to the beginning of the next file.
 −2  = Backspace to the beginning of the previous file.
```

Record count: Positive for forward, negative for backward.

The file count is executed first, then the record count EOF marks count as a record.

For example:

0,−1 = Move back one record.

−1,0 = Backspace to the first record of the current file.

See Notes.

FORTRAN:

```
CALL PTAPE (LU,file count,record count)
```

Pascal: Callable

Result: None

Errors: None

External References: EXEC, .ENTR

Notes: After using PTAPE, always check status with MAGTP.

RMPAR

Purpose: Move five parameters from the program's ID segment into a buffer within the program memory space. Used to retrieve up to five parameters passed to a program by the operating system (see Notes).

Entry Points: RMPAR

Assembly:

```
Suspend call or program entry point
JSB RMPAR
DEF *+2
DEF ARRAY
→ <Return>
ARRAY BSS 5
```

FORTRAN: Callable

Pascal: Callable

Result: Integer

Errors: None

External References: $OPSY

Notes:

1. The operating system inserts parameters into a program's ID segment as a result of:

   a. ON, GO, and other functions in RTE (refer to the appropriate RTE manual for other functions of this call).

   b. Program execution of an EXEC schedule call.

2. The RMPAR call must occur before any EXEC call or other subroutine call that calls EXEC.

3. Also used to retrieve parameters after a son program terminates, and to get the extended status following I/O calls. Refer to the *RTE-A* or *RTE-6/VM Programmer's Reference Manual.*

Example

```
FTN7X,L
      PROGRAM test
      DIMENSION ibuf (5)
      CALL RMPAR (ibuf)
             or
      PAUSE
      CALL RMPAR (ibuf)
```

See also EXEC 14 and PRTN in *Programmer's Reference Manual*, and GetRunString in this manual.

RT_ER

Purpose: To format and print a run-time error.

Entry Points: RT_ER

Assembly:

FORTRAN:

```
CALL RT_ER (ABREG,PREG)
```

or

```
CALL RT_ER (ABREG)
```

Use the second form to print a run-time error from the current location. See FTRAP for contents of ABREG and PREG.

Pascal: Callable

Result: Prints a 'RUN TIME' error message on 'ER0.E'. Refer to Appendix A of the *FORTRAN 77 Reference Manual* for the message format.

Errors: See FTRAP, RTRAP for details.

External References: CDS Version: .ENTR, ER0.T, !ERR0, CODE^DATA_ENTN

Non-CDS Version: .ENTR, ER0.V, ER0.C, PNAME, EXEC, ER0.E

TIMEI, TIMEO

Purpose: These two subroutines measure the accumulated differences between the time-in and timeout calls.

Entry Points: TM.IN, TM.OU, TIMEI, TIMEO

Assembly:

```
JSB TM.IN
DEF TMARY
<Return> (All registers preserved)
  :                          ! routine to be timed goes here
JSB TM.OU
DEF TMARY
<Return> (All registers preserved)
  :
TMARY DEC 0,0,0,0,0
```

FORTRAN:

```
INTEGER *4 TMARY(3)
  :
TMARY(1)=0J
  :
CALL TIMEI(TMARY)
  :                          ! routine to be timed goes here
CALL TIMEO(TMARY)
```

Pascal: Callable

Result: After each set of calls the first element of TMARY, a two word integer time array, advances by the number of 10's of milliseconds between the two calls.

Errors: None

External References: .ENTR, $TIME

Notes: You may call these sets of calls as often as you like, but you should call them in the order given. By specifying different 'TMARY' arrays, you can keep time on several events at once.

Assembly calls preserve all registers; FORTRAN calls do not.

See the ElapsedTime, ETime, ResetTimer, and TIMEF routines in Chapter 7 of this manual.

.ENTC and .ENTN

Purpose: Transfer the true address of parameters from a calling sequence into a subroutine and adjust return addresses to the true return point.

Entry Points: .ENTC, ENTN

Assembly:
```
.ENTN same as .ENTR
.ENTC same as .ENTP
```

FORTRAN: Callable

Pascal: Callable

Result: Address

Errors: None

External References: .ZPRV

Notes: This routine assumes the subroutine call is of the form:

```
JSB SUB
DEF p1 (first parameter)
.
.
.
DEF pm
```

The number of parameter addresses actually passed by the calling routine must agree with the number requested by the receiving routine.

See the .ENTR and .ENTP writeups for calling conventions.

.ENTP and .ENTR

Purpose: Transfer the true addresses of the parameters from a calling sequence into a subroutine; adjust return address to the true return point.

Entry Points: .ENTP, .ENTR

Assembly: For all utility routines:

```
      EXT  .ENTR

PARAM1    NOP
PARAM2    NOP
  .
  .
  .
PARAM     NOP

SUB       NOP
          JSB  .ENTR
          DEF  PARAM1
          <Return Pt>

          LDA  PARAM1     Get the address of a parameter

          LDB  @PARAM2    Get the value of a parameter
           .
           .
           .
          STA  @PARAM2    Pass back a value
           .
          JMP  @SUB       Exit the subroutine
```

For all privileged routines:

```
P1        NOP
P2        NOP
P3        NOP
  .
  .
  .
Pn        NOP

PSUB      NOP
          JSB $LIBR
          OCT 0
          JSB .ENTP
          DEF P1
```

Privileged code

```
          JSB $LIBX
          DEF PSUB
```

For all reentrant routines:

```
TDB       NOP
          ABS Q+N+3
RETURN    NOP
SV1       NOP
SV2       NOP
SV3       NOP
  .         .
  .         .
  .         .
SVq       NOP
P1        NOP
P2        NOP
P3        NOP
  .         .
  .         .
  .         .
Pn        NOP
RSUB      NOP
          JSB .ZRNT
          DEF EXIT
          JSB    .ENTP
          DEF P1
          STA RETURN
           .
           .
           .
EXIT      JMP @RETURN
          DEF TDB
          DEC 0
```

FORTRAN: Callable

Pascal: Callable

Result: Address

Errors: None

External References: .ZPRV

Notes:

1. The true parameter address is determined by eliminating all indirect references.

2. .ENTR and .ENTP assume the subroutine call is of the form:

```
JSB SUB
DEF *+m+1   (m = number of parameters)
DEF p1
.
.
DEF pm
```

   If m > n, then n parameters are passed. If n > m, then m parameters are passed, and any parameter addresses not passed remain as they were from the previous call.

3. "PARAM BSS n" must appear immediately before the subroutine entry point "SUB NOP". The entry point is set to the return address (DEF *+m+1). "JSB .ENTR" must be the first instruction after the subroutine entry point. "JSB .ENTP" must be the third instruction after the subroutine entry point.

.FMUI, .FMUO, .FMUP

Purpose: .FMUI contains three entry points corresponding to three conversion procedures in the FORTRAN formatter:

.FMUI	Convert an ASCII digit string to internal numeric form.
.FMUO	Convert a numeric value to ASCII.
.FMUP	Convert an unpacked internal format number (from .FMUI) to a normal format.

Entry Points: .FMUI, .FMUO, .FMUP

Assembly:

```
JSB .FMUI
DEF *+8
DEF <buffer> ASCII, one digit/word, FORTRAN R1 format
DEF <bufsiz> # of digits in <buffer> between 0 and 20,  inclusive.
DEF <sign>   0 = positive, 1 = negative.
DEF <exp>    Scale factor; power of ten.
DEF <result> Return value.
DEF <type>   Type of <result> (see below).
DEF <ovfl>   Returned from .FMUI, 1 if overflow or
             underflow; else 0.
<Return>

JSB .FMUO
DEF *+7
DEF <buffer> Returned from .FMUO
DEF <bufsiz> Number of digits to return
DEF <sign>   Returned from .FMUO
DEF <exp>    Returned from .FMUO
DEF <value>  Input value
DEF <type>   Type of value (see below)
<Return>

JSB .FMUP
DEF *+5
DEF <result>
DEF <type>
DEF <unpkd>  Input <result> from .FMUI
DEF <ovfl>   Returned from .FMUP, 1 if overflow or
             underflow else 0.
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: None

Errors: None

External References: .PACK, .ENTR, .MVW, IFIX

Method:

.FMUI The value in <buffer> is converted to binary with the digit in buffer (i) having weight 10** (<exp>−i). The result is negated if<sign> =1, and rounded to the specified type:

<TYPE>	TYPE
0	16-bit integer (1 word)
1	32-bit integer (2 words)
2	32-bit real (2 words)
3	48-bit real (3 words)
4	64-bit real (4 words)
5	Unpacked internal format (5 words)

.FMUO Reverse of .FMUI; that is, generates <buffer> <exp> and <sign> from <value> as described in .FMUI. The result should be rounded by calling .FMUR since there may be some round-off error by .FMUO (for example, 2.0 could convert to 1.99999).

.FMUP A type 5 buffer <unpkd> created by .FMUI is converted to a normal type buffer <result>. The type of <result> is specified by <type> and must be 0 to 4.

Notes: See also HpZ.

.FMUR

Purpose: Rounding of digit string produced by .FMUO.

Entry Points: .FMUR

Assembly:

```
JSB .FMUR
DEF *+5
DEF <buffer>  ASCII, one digit/word, FORTRAN R1 format
              (input and returned value)
DEF <bufsiz>  # of digits in <buffer> between 0 and 20, inclusive
DEF <rndsiz>  # of digits to round to
DEF <ovfl>    Returned from .FMUR, 1 if carry overflow occurs, else 0.
<Return>
```

FORTRAN: Callable

Pascal: Callable

Example:

A conversion to 10 digits would be as follows:

```
.FMUO (buffer,11,sign,exp,value,type)
.FMUR (buffer,11,10,ovfl)
exp=exp+ovfl
```

Result: None

Errors: None

External References: .ENTR

.GOTO

Purpose: Transfer control to the location indicated by a FORTRAN computed GOTO statement:

```
GOTO (k1, k2, ... , kn) j
```

Entry Points: .GOTO

Assembly:

```
JSB .GOTO
DEF *+n+2
DEF J
DEF k1
.
.
.
DEF kn
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Branch to address k_n

Errors: If $j < 1$ then k_1; if $j > n$ then k_n

External References: None

.MAP

Purpose: Return actual address of a particular element of a two dimensional FORTRAN array.

Entry Points: .MAP

Assembly:

```
JSB .MAP
DEF array
DEF first subscript
DEF second subscript
OCT first dimension
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer in A

Errors: None

External References: None

.OPSY

Purpose: Determines which operating system is in control. Included for compatibility with previous libraries.

Entry Points: .OPSY

Assembly:

```
JSB .OPSY
——► result in A
A =   -7 (RTE-MI)
A =  -15 (RTE-MII)
A =   -5 (RTE-MIII)
A =   -3 (RTE-II)
A =   -1 (RTE-III)
A =   -9 (RTE-IV,RTE-IVB)
A =  -17 (RTE-6/VM)
A =  -13 (RTE-4E)
A =  -29 (RTE-XL)
A =  -31 (RTE-L)
A =  -37 (RTE-A Pre-Rev 2440)
A =  -45 (RTE-A.1)
A =  -53 (RTE-A Rev 2440 through 4010)
A =  -61 (RTE-A Rev 5000 through 5270)
A = -125 (RTE-A Rev 6000 or later)
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: Integer in A

Errors: None

External References: $OPSY

Notes: This routine is equivalent to

```
EXT $OPSY
XLA $OPSY
```

The $OPSY value of an operating system identifies the operating system and major version of the operating system. Each operating system has a unique $OPSY value. The value of $OPSY changes only when major internal table structures are affected by the revision. (Another system entry point $DATC contains the revision code for any given release.) Programs that are system dependent should check the value of $OPSY before executing.

The range of values reserved for RTE-6/VM Operating Systems is −17 through −28. The range of values reserved for RTE-A Operating Systems is −33 through −128.

See also the HpRTEA and HpRTE6 functions.

.PAUS

Purpose: Print the following message on the console device:

```
name PAUSE xxxxx
```

where name is the calling program name and xxxxx is the specified integer i. Halt program execution and return to operating system.

Entry Points: .PAUS, .STOP

Assembly:
```
LDA i
JSB .PAUS (or .STOP)
<Return>
```
(See Notes)

FORTRAN: Not callable. Use FORTRAN intrinsic PAUSE.

Pascal: Not callable

Result: None

Errors: None

External References: EXEC, PAU.E, REIO, PNAME

Notes: When .PAUS is used, the program can be continued using GO.

.PCAD

Purpose: Return the true address of a parameter passed to a subroutine.

Entry Points: .PCAD

Assembly:
```
JSB .PCAD
DEF SUB,i
<Return> (result in A) (See Notes)
```

FORTRAN: Callable

Pascal: Callable

Result: Direct address A

Errors: None

External References: .ZPRV

Notes:

1. .PCAD has the same purpose as GETAD.
2. .PCAD is used by reentrant or privileged subroutines because they cannot use GETAD.

.TAPE

Purpose: Perform magnetic tape rewind, backspace or end-of-file operations on a specified logical unit.

Entry Points: .TAPE

Assembly:
```
LDA constant
JSB .TAPE
<Return>
```

FORTRAN: Callable

Pascal: Callable

Result: None

Errors: None

External References: EXEC

Notes: In FORTRAN, use utility statements or PTAPE and MGTAP.

..MAP

Purpose: Computes the address of a specified element of a 1, 2, or 3 dimension array; returns the address in the A-Register.

Entry Points: ..MAP

Assembly:

For 1 dimension:

```
CCA, <CLE>
LDB n (see below)
JSB ..MAP
DEF base address
DEF 1st subscript
<Return> (address in A)
```

For 2 dimensions:

```
CLA, <CLE>
LDB n (see below)
JSB ..MAP
DEF base address
DEF 1st subscript
DEF 2nd subscript
DEF length of 1st dimension
<Return> (address in A)
```

For 3 dimensions:

```
CLA, INA, <CLE>
LDB n (see below)
JSB ..MAP
DEF base address
DEF 1st subscript
DEF 2nd subscript
DEF 3rd subscript
DEF length of 1st dimension
DEF length of 2nd dimension
<Return> (address in A)
```

n = Number of words per element in the array (1, 2, 3 or 4).

E-Register = 1 if store to this element.
0 if read from this element.

FORTRAN: Not callable

Pascal: Not callable

Result: Integer

Errors: None

External References: None

$SETP

Purpose: Set up a list of pointers.

Entry Points: $SETP

Assembly:

```
LDA Start Pntr
LDB Array Addr
JSB $SETP
DEF COUNT
<Return>

ARRAYADDR DEF ARRAY
ARRAY     BSS n
```

FORTRAN: Callable

Pascal: Callable

Result: Integer

Errors: None

External References: .ZPRV

Notes:

1. This routine is available in microcode.
2. The sign bit of B is ignored.

%SSW

Purpose: Call-by-name entry to ISSW (x).

Entry Points: %SSW

Assembly:

```
JSB %SSW
DEF *+2
DEF n (integer)
<Return> (result in A)
```

FORTRAN: Callable

Pascal: Callable

Result: Integer in A

Errors: None

External References: ISSW

6

Subroutines for Multiuser Support

The subroutines in this chapter provide programmatic access to the system handling of multiuser sessions. They allow programs to set up and remove sessions, attach and detach from them, and convert between names, user numbers, and session numbers.

For RTE-A Operating Systems, these routines do not return useful information unless you have the HP 92078A product (the Virtual Code +, VC+, System Extension Package).

Note All subroutines listed in this chapter are compatible within both the RTE-A and RTE-6/VM Operating Systems unless otherwise specified. All functions must be declared correctly (that is, the type that they return).

The subroutines in this chapter are presented in the following format:

The name of the subroutine, a statement of the use of the subroutine, followed by the subroutine's syntax, a description of the parameters, and then returns, if any.

If a parameter is underlined in a subroutine call description, the value is a variable returned or modified by the system subroutine.

AccessLU, Check for LU Access

This logical function determines if the specified LU is accessible by the calling program.

LogicalVar = AccessLU(*lu*)

```
logical LogicalVar, AccessLU
integer lu
```

where:

LogicalVar is TRUE if the calling program can access the LU; otherwise, it is FALSE.

lu is the LU to be checked.

ATACH, Attach to Session

This integer function attaches the calling program to an existing session for RTE-6/VM and RTE-A. In addition, on RTE-A you can attach a program other than the calling program to an existing session.

error = ATACH(*SesNum*, *error*, [*ProgName*,[*CurrentSes*]])

For RTE-A Only

where:

SesNum is a one-word integer, the session number. If *SesNum* is 0, it attaches to the "system" session.

error is a one-word integer, with these meanings:

= 0 no error; successful attach.

= −1 if session number does not exist.

= −2 if specified program does not exist.

= −3 if current session number does not exist.

= −4 must be superuser for action requested.

= −5 program with same name already exists in session *SesNum*.

ProgName is a three-word integer array containing the name of the program to be attached to session *SesNum*. (For RTE-A only.)

CurrentSes is a one-word integer containing the session number that program *ProgName* is in. Defaults to caller's current session. (For RTE-A only.)

For RTE-A Only:

If no program is specified, ATACH is performed on the calling program.

If the ATACHed program is not a system utility, that program's terminal LU changes to coincide with a new session LU number. If the ATACHed program is a system utility, you can change your terminal LU number by following the ATACH call with an ATCRT call.

A program cannot be attached to the system session if any session, other than the current session, has a program of that name in it. This is because the system session is considered to be an extension of each user session.

Note Only a superuser or system process can manipulate programs in sessions other than the session that the calling program is in.

ATCRT, Attach a CRT (RTE-A Only)

This subroutine inserts a CRT LU into the $CON word (word 29 of the ID segment) of a program (generally a system process).

```
CALL ATCRT(crtlu)
```

where:

crtlu is an integer representing the CRT LU to insert into the $CON word (word 29) of the ID segment.

Programmatic LOGON (RTE-A Only)

To create a programmatic session, your program must call GETSN and CLGON. To end a programmatic session, your program must call CLGOF and RTNSN.

For example, DS logs on programmatically as follows:

1. DS calls the subroutine GETSN (get a session number). The session numbers are defined to start from one larger than the largest assigned LU in the system (that is, $LU + 1). Session numbers are recyclable and can exceed eight bits. On initializing, DS allocates all the numbers it needs by making multiple calls to GETSN.

2. DS calls CLGON with the specified session number, user name, and password for a programmatic logon.

3. LOGON gets the buffer and creates a user entry with the specified session number. LOGON can treat the number as a session number because the value is greater than that of the largest assigned LU. The session number is placed in word 12 of the user table (the same location as the terminal LU). LOGON sets the status equal to 3, for programmatic LOGON, and sets the counter that counts the number of programs to 0.

4. DS programs make ATACH calls by specifying the session number.

5. DS programs make DTACH calls to remove programs from the session.

6. DS calls the subroutine CLGOF to log off the session.

7. DS calls RTNSN to deallocate (multiple times) session numbers reserved by calls to GETSN.

CLGOF, Call LOGOF (RTE-A Only)

This integer function logs off a user.

error = CLGOF(*SesNum*,*option*,*error*)

where:

SesNum is a one-word integer, the session number of the session that the user wants to terminate.

option is a one-word integer, with the following meanings:

= 0 if no active programs, just log off; if there are active programs, do not log off, but set *error* = −1.

= 1 log off and kill all active programs.

= 2 go noninteractive, let active programs continue to run; if there are no active programs, log off.

error is a one-word integer, with these meanings:

= 0 no error.

= −1 there are active programs (if option is 0).

= −2 wrong option given or wrong session number.

= −5 program with same name already exists in session *SesNum*.

If Security/1000 is turned on, this routine will be subject to security checking. If the security check fails, the calling program will either receive a -1713 error (Security Violation) or be aborted with a Security Violation, depending on the security configuration set up by the System Manager.

CLGON, Call LOGON (RTE-A Only)

This integer function logs a user on.

error = CLGON(*buffer*,*length*,*SesNum*,*error*)

where:

buffer is an integer buffer in which the username/password is placed. Maximum buffer length is 34 characters.

length is a one-word integer, the length of *buffer* in characters.

SesNum is a one-word integer, a session number.

error is a one-word integer, with these meanings:

= 0 no error.

= −1 internal error, such as no class numbers, or logon not performed.

= −3 too many sessions active.

= −4 no such user.

= −5 bad or missing password.

= −6 file is not valid user file.

= −7 user configuration file already open.

= −9 an FMP error occurred during LOGON.

If Security/1000 is turned on, this routine will be subject to security checking. If the security check fails, the calling program will either receive a −1713 error (Security Violation) or be aborted with a Security Violation, depending on the security configuration set up by the System Manager.

DTACH, Detach From Session

This routine in RTE-A detaches a program from its current session and associates the program with the "system" session. It also changes the terminal LU to 1. If the calling program in RTE-6/VM is not a session program, this routine does nothing more than a return. If no program is specified, the DTACH is performed on the calling program.

```
CALL DTACH
```

Although the routine does not require a parameter, it will accept three. The following alternatives are also possible:

RTE-6/VM Only:

```
CALL DTACH()
CALL DTACH(dummy)
```

RTE-A Only:

```
CALL DTACH()
CALL DTACH(error)
CALL DTACH(error,ProgName,CurrentSes)
```

where:

error is a one word integer, with these meanings:

=	0	no error, successful DTACH.
=	−2	program specified does not exist.
=	−3	session specified does not exist.
=	−4	must be a superuser for action requested.
=	−5	program with same name already exists in SYSTEM session.

ProgName is a three-word integer array containing the name of the program to be DTACHed to the system session.

CurrentSes is a one-word integer containing the session number that program *ProgName* is in. It defaults to caller's current session.

A program cannot be DTACHed to the system session if a session, other than the current session, has a program of that name in it. This is because the system session is considered to be an extension of each user session.

Note Only a superuser or system process can manipulate programs in sessions other than the session that the calling program is in.

FromSySession, Check System Session Table Address (RTE-A Only)

This integer function determines if the given user table address is equal to the system session table address.

```
i = FromSySession(UserTabAddr)
```

where:

UserTabAddr is an integer containing the user table address.

Returns:

1	User table address is equal to system session address.
0	User table address not equal to system session address.

GetAcctInfo, Access User and Group Accounting (RTE-A Only)

This routine retrieves the multiuser accounting information stored in the group and user configuration files.

```
CALL GetAcctInfo(AcctName,AcctInfo,error)
```

where:

AcctName is a character string containing the name of the account whose information is to be retrieved. It can be in any of the following forms:

username	get unique user information from block one and user CPU and connect limits from the user.nogroup record
username.	get unique user information from block one of the user configuration file
username.groupname	get user.group information from the user configuration file
.groupname	get group information from block one of the group configuration file

---

Note No masks are allowed.

AcctInfo is an integer array in which the following accounting information is returned:

For “user.” :

Words 1-2	Total CPU usage for user in all groups that user is a member of (double integer in tens of msecs).
Words 3-4	Total connect time for user in all groups that user is a member of (double integer in seconds).
Words 5-6	User’s last logon time (double integer in seconds since Jan. 1, 1970).
Word 7	Group ID the user last logged on with.
Word 8	LU the user last logged on to.
Words 9-10	Last logoff time (double integer in seconds since Jan. 1, 1970).

For “user”:

Words 1-10	Same as above.
Words 11-12	CPU usage limit for the user (double integer in tens of msecs).
Words 13-14	Connect time limit for the user (double integer in seconds).

For “user.group” or “.group” information:

Words 1-2	Total CPU usage for the user in the group specified or the total for the entire group (double integer in tens of msecs).
Words 3-4	Total connect time for the user in the group specified or the total for the entire group (double integer in seconds).
Words 5-6	CPU usage limit for the user in the group specified or the limit for the entire group (double integer in tens of msecs).
Words 7-8	Connect time limit for the user in the group specified or the limit for the entire group (double integer in seconds).

error < 0, routine was unsuccessful
≥ 0, routine was successful

GetOwnerNum, Return Owner's ID

This integer function returns the user's identification number.

```
i = GetOwnerNum()
```

Returns:

nonzero	Caller's identification number.
2	Calling program is attached to the system session.

GetResetInfo, Access/Reset User Accounting (RTE-A Only)

This integer function accesses and clears (if directed so) the multiuser accounting information stored in the user configuration file.

```
CALL GetResetInfo(user,ResetFlag,AcctInfo,error)
```

where:

user is a character string containing the user's logon name.

ResetFlag is a one-word logical with these meanings:

= true, reset User's accounting information to 0.
= false, leave accounting information unaltered.

AcctInfo is an array of three double integers in which the following is returned:

AcctInfo(1) − last logoff time in seconds since Jan 1, 1970
AcctInfo(2) − cumulative connect time in total seconds
AcctInfo(3) − cumulative CPU usage in tens of milliseconds

error is a one-word integer with these meanings:

≥ 0, no error.
= −1, file is not a valid user file.
< −1, FMP error encountered.

Note

1. CPU usage can accumulate to a total of approximately .7 years; connect time to approximately 70 years.
2. This function only exists for backward compatibility and should NOT be used by any new applications. Its functionality has been replaced by GetAcctInfo and ResetAcctTotals. It returns the user's last logoff time, cumulative connect time and cumulative CPU usage from block one of the user configuration file. It clears the cumulative CPU usage and connect time totals in block one and the USER.NOGROUP record, block 2, in the user configuration file.

GETSN, Get Session Number (RTE-A Only)

This integer function gets a unique session number.

```
error = GETSN(SesNum)
```

where:

SesNum If 0, user wants to allocate a session number. The routine returns the session number and sets the return error code.

If nonzero, user wants to allocate that number as the session number. If available, routine returns that number and sets the return error code.

SesNum is a one-word integer that starts from the number of system LUs plus 1 and is unique. The final number may be greater than an 8-bit number.

error is a one-word integer, with these meanings:

= 0, no error.
= −1, cannot get a session number.
= −2, no more available.

GPNAM, Return Group Name

This subroutine returns, in ASCII, the group name associated with the calling program.

```
CALL GPNAM (name)
```

where:

name is a returned character string containing the group name (16 character maximum for RTE-A; 10 characters for RTE-6/VM).

GroupToId, Return Group ID

This integer function returns the group's ID number when given the group's name.

groupID = GroupToId (*name*)

where:

name is a character string containing the group name (16 characters maximum).

groupID is a one-word integer with the following returns:

>0	Group ID number.
−254	No such group.
any other negative number	FMP error when accessing the group configuration file.

IdToGroup, Return Group Name

This integer function returns the group name when given a group ID number.

error = IdToGroup (*id*,*name*)

where:

id is an integer representing the group's ID number.

name is a return character string containing the group's name (16 characters maximum).

Returns:

−254	No such group.
any other negative number	File error encountered when accessing multiuser file to determine group name.
0	No error.

IdToOwner, Return User Name

This integer function returns a user name when given a user ID number.

error = IdToOwner (*id*,*name*)

where:

id is an integer representing the user's ID number.

name is a return character string containing user's name (maximum 16 characters).

Returns:

−233	No such user.
any other negative number	File error encountered when accessing multiuser file to determine group name.
0	No error.

LUSES, Return User Table Address

This integer function returns the address of the user table entry associated with the session number.

i = LUSES(*SesNum*)

If i is equal to 0, it is an error; no such user associated with the specified session number. Otherwise i is equal to the address of the user table. i and *SesNum* are one-word integers.

Member, Check if User is in Group (RTE-A Only)

This integer function determines if the specified user is in the specified group.

i = Member(*username*,*groupname*)

where:

username is a character string containing the user's name (16 characters maximum).

groupname is a character string containing the group's name (16 characters maximum).

Returns:

−1	User is not a member of the group.
−2	File error in determining if the user is a member of the group (returned if the group does not exist).
>0	User is a member of the group.

OwnerToId, Return User ID and Group ID

This integer function returns the user's ID number when given the user's name or the user's ID and the group's ID when given the usergroup name.

userID = OwnerToId(*name*[,*groupID*])

where:

name is a character string containing the owner name. If the optional "*groupID*" parameter is not specified, it is a user name (16 characters maximum). Otherwise, it is a "user.group" name (31 characters maximum for both; "user" and "group" are each 16 characters maximum).

groupID is an optional one-word integer for the group ID number with the following returns:

>0	group ID number.
0	never attempted to get group ID due to previous error.
−254	no such group.
−255	user is not a member of the specified group.
any other negative number	FMP error in accessing the user configuration file.

userID is a one-word integer for the user ID number with the following returns:

>0	user ID number.
−233	no such user.
any other negative number	FMP error in accessing the user configuration file.

ProgIsSuper, Check for Super Program

This logical function determines if a program is a Super Program. A Super Program is defined as having a ProgCplv of 31. If Security/1000 is not turned on, this function will always return FALSE.

Progtype = ProgIsSuper(*IdSegAdr*)

```
Logical ProgType, ProgIsSuper
Integer IdSegAdr
```

where:

ProgType is TRUE if the program is a Super Program; otherwise, it is FALSE.

IdSegAdr is the ID segment address of the program whose type is to be determined. If *IdSegAdr* is 0, the calling program's type is determined.

ResetAcctTotals, Resets User and Group Accounting Totals (RTE-A Only)

This routine resets the CPU usage and connect time totals stored in the group and user configuration files to zero. The routine can be used to reset the cumulative CPU usage and/or connect time totals. Either CP or CO, but not both, can be specified as the optional parameter "*only*". If CP is specified, only the cumulative CPU usage total is reset. Likewise, if CO is specified only the cumulative connect time total is reset. If neither is specified, both totals are reset.

```
CALL ResetAcctTotals(AcctName,AcctInfo,error,only)
```

where:

AcctName is a character string containing the name of the account whose totals are to be reset. It can be in any of the following forms:

user.	Reset unique user total(s) in block one of the user configuration file.
user	Reset the unique user totals in block one of the user configuration file and those in the USER.NOGROUP record (this is for backward compatibility).
user.group	Reset the total in the user.group record in the user configuration file for the specified user.group.
user.@	Reset the total(s) in all the user.group records for user. (Note the cumulative totals for the user in block one are not affected).
@.group	Reset user.group total(s) in the user files for all members in group. (Note: The cumulative totals for the group in block one of the group file are not affected).
.group	Reset the group total(s) in block one of the configuration file for the group (user.group totals of members are not affected).
.@	Reset the group total(s) in block one of all group configuration files.

AcctInfo contains the values of the CPU usage and connect time totals before they were reset. If a mask is used in Acct_Name, Acct_Info contains the totals of the last user.group or group that was processed.

Words 1−2 CPU usage total (double integer in tens of msecs).

Words 3−4 connect time total (double integer in seconds).

error ≥ 0, routine was successful.
< 0, routine was unsuccessful.

only is an optional parameter string containing CO or CP.

If CO, only reset the connect time total.

If CP, only reset the CPU usage total.

RTNSN, Return Session Number (RTE-A Only)

This integer function returns a session number to the system.

error = RTNSN(*SesNum*)

where:

SesNum is a one-word integer where the session number to be returned is placed.

Returns:

0	No error.
−1	Not a session number.

SessnToOwnerName, Return User Name

This integer function returns the ASCII user name when given the session number.

error = SessnToOwnerName(*SesNum*,*name*)

where:

SesNum	is a one-word integer representing the session number.
name	is a return character string containing user's name (maximum 16 characters for RTE-A, 21 characters for RTE-6/VM).
error	is a one-word integer.

Returns:

0	No error.
−1	Cannot find name.

SetAcctLimits, Set User and Group Accounting Limits (RTE-A Only)

This routine sets the multiuser accounting limits in the group and user configuration files to the value specified. The CPU usage limit and/or the connect time limit can be set. CP:limit and CO:limit are used to specify that the CPU usage limit and/or the connect time limit, respectively, are to be set.

```
CALL SetAcctLimits(AcctName,AcctInfo,error,Parm1,Parm2)
```

where:

AcctName is a character string containing the name of the account whose accounting limits are to be set. It can be in the following forms:

user	Set USER.NOGROUP limit(s) in the user configuration file (for backward compatibility.)
user.group	Set the limit(s) in the user.group record in the user configuration file for the specified user.group.
user.@	Set the limit(s) in all the user.group records for user. (Note the cumulative limits for the user in block one of the user configuration file are not affected.)
@.group	Set the user.group limit(s) in the user files for all members in group. (Note: The limits in block one of the group configuration file are not affected.)
group	Set the group limit(s) in block one of the configuration file for group. (Note: User.group limit(s) of members are not affected.)
@	Set the group limits in block one of all group configuration files.

AcctInfo contains the values of the CPU and connect time limits before they were modified. If a mask is used in Acct_Name, Acct_Info contains the totals for the last user.group or group that was processed:

Words 1−2 CPU usage limit (double integer in tens of msecs).

Words 3−4 Connect time limit (double integer in seconds).

error

≥ 0,	routine was successful.
< 0,	routine was unsuccessful.

Parm1/*Parm2* at least one must be specified, and they cannot both be CPs or COs. They are character strings with the following format and meaning:

CP:limit	set CPU limit to double integer value “limit” (in tens of msecs).
CO:limit	set connect time limit to the double integer value “limit” (in seconds).

SuperUser, Check For/If Superuser

This integer function determines if the user associated with this session number is a “superuser.” For RTE-6/VM, this is the MANAGER.SYS account.

```
i = SuperUser(SesNum)
```

where:

SesNum is a one-word integer representing the session number.

Returns:

1	User is superuser.
0	User is not superuser.
−1	User not found with the given session number.

SYCON, Write Message to System Console

The SYCON subroutine writes a message to the system console (system LU 1).

```
CALL SYCON(ibuf,ilen)
```

where:

ibuf is a buffer that contains the message to be written.

ilen is the length of *ibuf*. A positive value indicates the number of words, and a negative value indicates the number of characters.

This routine bypasses the Session Switch Table (SST) and writes directly to system LU 1.

The Macro/1000 calling sequence is as follows:

```
      EXT SYCON
       .
       .
      JSB SYCON
      DEF RTN
      DEF IBUF
      DEF ILEN
RTN    .
       .
```

SystemProcess, Check For/If System Process (RTE-A Only)

This logical function determines whether the program in the ID segment is a system process.

```
i = SystemProcess(idseg)
```

where:

idseg is a one-word integer representing an ID segment address.

i is a logical variable (Boolean).

Returns:

true The program is a system process (less than 0).

false The program is not a system process (greater than 0).

UserIsSuper, Check For/If Superuser

If the user is not a superuser, this integer function returns zero; otherwise, it returns nonzero. The nonzero value can be used as a logical condition (by declaring this to be a logical function).

```
SuperState = UserIsSuper()
```

where:

SuperState is the user's state.

Returns:

nonzero User is superuser.

0 User is not superuser.

USNAM, Return User Name

This subroutine returns, in ASCII, the user's name associated with the calling program.

```
CALL USNAM(name)
```

where:

name is a returned character string containing the user name (maximum 16 characters for RTE-A, 21 characters for RTE-6/VM).

USNUM, Return the Session Number

This integer function gets the session number of the calling program.

```
i = USNUM()
```

where:

i is a one-word integer in which the session number is returned.

Returns:

nonzero	The session number.
0	Error: there is no user session number.

VFNAM, Verify User Name (RTE-A Only)

This integer function verifies the validity of a user name.

```
error = VFNAM(name,length)
```

where:

error is a one-word integer.

name is a character string containing the user name/password, up to 34 characters.

length is a one-word integer representing the length of *name* in characters.

Returns:

0	The name and password are valid.
−1	The name or password is invalid.

If Security/1000 is turned on, this routine is subject to security checking. If the security check fails, the calling program will either receive a -1713 error (Security Violation) or be aborted with a Security Violation, depending on the security configuration set up by the System Manager.

7

Utility and Status Subroutines

This chapter describes routines that obtain information from RTE-A and RTE-6/VM Operating Systems or do operations that are otherwise difficult from high level language and MACRO/1000 programs.

Note All subroutines listed in this chapter are compatible within both the RTE-A and RTE-6/VM Operating Systems unless otherwise specified. All functions must be declared correctly (that is, the type that they return).

The subroutines in this chapter are presented in the following format:

The name of the subroutine, a statement of the use of the subroutine, followed by the subroutine's syntax, a description of the parameters, and then returns, if any.

If a parameter is underlined in a subroutine call description, the value is a variable returned or modified by the system subroutine.

AddressOf, Return Direct Address

This function is like .DRCT, but with a name easily used from FORTRAN and .ENTR call sequences. It returns the direct address of the passed item.

directAdd = AddressOf(*item*)

where:

item is an array, variable, or constant whose address you want.

directAdd is a one-word integer variable in which the direct address of the passed item is returned. Indirect references are resolved.

Bit Map Manipulation Routines

These routines operate on bit maps represented by one or more contiguous words containing bits to set, clear or test. These bit maps can have more than 32K bits in them, so all size parameters are double integers. These routines were written for the space allocation routines in the file system, which convert from external units (blocks) to bits by means of the parameter blocks per bit. This can be set to one if you want to deal directly with bits.

See also the ClearBitMap, SetBitMap, TestBitMap, and Test_SetBitMap routines.

ChangeBits

This subroutine changes a number of bits in a passed bit map.

```
CALL ChangeBits(bitmap,where,nblocks,bperb,how)
```

where:

bitmap is an integer array of words containing the bit map.

where is a double-word integer variable that specifies a starting block number to change, with block zero the first bit, located in bit 15 of the first word of the bit map.

nblocks is a double-word integer variable that specifies how many blocks to change; this is rounded up to an integral number of bits.

bperb is a user supplied conversion factor, the number of blocks per bit (the caller is assumed to be working in blocks).

how is a one-word integer variable, the method by which to change the passed bits:

0	= clear bits,
anything else	= set bits

CheckBits

This function checks a number of bits in a passed bitmap.

```
state = CheckBits(bitmap,where,nblocks,bperb,how)
```

where:

bitmap is an integer array of words containing the bit map.

where is a double-word integer variable that specifies a starting block number to check, with block zero the first bit, located in bit 15 of the first word of the bit map.

nblocks is a double-word integer variable that specifies how many blocks to check, rounded up to an integral number of bits.

bperb is a one-word integer variable, a user-supplied conversion factor, the number of blocks per bit (the caller is assumed to be working in blocks).

how is a one-word integer variable that indicates what to check for:

0	= check for zeros
anything else	= check for ones.

state is the returned value.

Returns:

nonzero All of the *nblocks* are in the state indicated by *how*.

0 Any of the *nblocks* are in the opposite state.

FindBits

This function locates contiguous holes in a bit map. It returns as its function value the block address where the free bits can be found. This block address can then be passed to ChangeBits to allocate the bits.

freebits = `FindBits(`*bitmap*`,`*size*`,`*nblocks*`,`*bperb*`)`

where:

bitmap is an integer array of words containing a bit map.

size is a one-word integer variable, the number of words in *bitmap*.

nblocks is a double-word integer variable that specifies how many contiguous zero bits to locate.

bperb is a one-word integer variable, a user supplied conversion factor, the number of blocks per bit (the caller is assumed to be working in blocks).

Returns:

freebits Block address where the free bits can be found.

BlankString

This function determines if a character string consists entirely of blanks.

bool = BlankString(*string*)

where:

string is the character string to be examined.

bool is a logical variable that contains the returned value.

Returns:

.TRUE.(−1) if the entire string is blank.

.FALSE.(0) if the string contains any nonblank characters.

Note This function performs the same operation as the FTN7X statement

```
bool = string .EQ. ' '
```

except BlankString does it using fewer instructions than FTN7X generated code.

BlockToDisc, Convert Block and Sector to Track and Sector

This subroutine converts a block number and number of sectors per track into the corresponding track and sector. No overflow check is performed on the track number.

CALL BlockToDisc(*block*,*spert*,*track*,*sector*)

where:

block is a double-word integer block number to be converted into a corresponding track and sector.

spert is a one-word integer variable, the number of 64 word sectors per track on the LU of interest.

track is a one-word integer variable, the returned value for the track corresponding to the specified block. No overflow check is performed on *track*.

sector is a one-word integer variable, the returned value for the sector corresponding to the passed block.

CaseFold, Convert Lowercase to Uppercase

This subroutine converts lowercase to uppercase in character strings (converting a−z to A−Z). Conversion is done in place.

```
CALL CaseFold(string)
```

where:

string is the character string to convert to all uppercase characters.

See also CLCUC.

CharFill

This subroutine fills the specified string with the supplied character. Typically, it can be used to blank fill a string or pad a substring with blanks. The operation is quick and uses less code than the standard string assignment in FTN7X.

```
CALL CharFill(string,char)
```

where:

string is the character string to be modified.

char is a single character string variable that contains the user supplied fill character.

If the specified string has a length of zero, nothing is done.

Example:

```
CALL CharFill(string(5:),' ')  !  pad string with blanks
```

CharsMatch, Compare Characters in Arrays

This integer function compares the characters in two arrays for matches.

match = CharsMatch(*bfr1*,*bfr2*,*chars*)

where:

bfr1 is the first integer array of characters to compare; must start on a word boundary.

bfr2 is the second integer array of characters to compare; must start on a word boundary.

chars is a one-word integer variable, the number of characters to compare; can be even or odd.

match is a one-word integer variable in which the match state of the characters is returned.

Returns:

nonzero Characters match.

0 Characters do not match.

See also CompareWords.

ClearBuffer, Zero a Passed Buffer

This subroutine zeros out the passed buffer, using the specified length in words.

CALL ClearBuffer(*buffer*,*len*)

where:

buffer is the name of a buffer to clear. It is of any type except *char*.

len is a one-word integer variable, the length of the buffer in words.

See also FillBuffer.

CLCUC, Convert Lowercase to Uppercase

This subroutine operates on an integer array buffer, converting ‘a’ through ‘z’ to ‘A’ through ‘Z’. Conversion is done in place.

```
CALL CLCUC (buffer,len)
```

where:

buffer	is an integer array of characters.
len	is an integer length, in words if positive, in bytes if negative.

See also CaseFold.

CMNDO Routines (RTE-A Only)

The CMNDO monitor performs command editing and/or command stack handling for a user program. Using a monitor to perform these tasks requires an additional ID segment at runtime; however, using the monitor minimizes the code changes and the code growth that are required to add this functionality directly to a program. The $VISUAL command editing modes are described in the *RTE-A User's Manual*, part number 92077-90002.

Note that the functionality provided by the CMNDO monitor can also be added directly to a program with calls to the CmndStackInit, CmndStackRestore, CmndStackMarks, CmndStackScreen, CmndStackPush, CmndStackSave, and RteShellRead routines.

HpStartCmndo, Enable a CMNDO Slave Monitor

HpStartCmndo schedules a copy of the CMNDO program. Communication with the clone is done via Class I/O. One Class number is allocated to the calling program, and another is allocated and owned by the CMNDO clone. The Class numbers are passed back to the user in the *class* array. These Class numbers are required in any subsequent calls to HpReadCmndo or HpStopCmndo.

error = HpStartCmndo(*lu*,*class*[,*stackfile*,*stkerr*])

```
integer*2 error,lu,class(2),stkerr
character*(*) stackfile
```

where:

error returns the following:

0	CMNDO successfully started.
< 0	FMP error trying to RP CMNDO.
1	CLRQ error obtaining class numbers.
2	EXEC error scheduling CMNDO.

lu is the LU number of the user's terminal.

class is a two-word integer array; *class*(1) is returned as the Class number owned by the caller, *class*(2) is returned as the Class number owned by CMNDO.

The following two parameters are required only when restoring from an existing command stack file.

stackfile is a FORTRAN character string containing the name of a command stack file.

stkerr is an error returned from CmndStackRestore.

HpReadCmndo, Request CMNDO to Read from User's Terminal

HpReadCmndo signals CMNDO to issue a read to the terminal. The prompt is assumed to have already been issued by the calling program. The prompt is used by CMNDO only when a refresh command is entered by the user. When recalling "marked" lines from the command stack, CMNDO echoes the marked line to the terminal.

The CMNDO program uses the following environment variables. The default is taken when the environment variable cannot be obtained.

Variable	Default
$LINES	24
$COLUMNS	80
$VISUAL	no command editing
$KILLCHAR	DEL
$FRAME_SIZE	$LINES − 2

If the CMNDO clone is no longer available, HpReadCmndo returns −1 in the A- and B-Registers.

```
CALL HpReadCmndo(class,buffer,len,prompt,promptlen)
CALL ABREG(areg,breg)

integer*2 class(2),buffer(*),len,prompt(*),promptlen
```

where:

class is a two-word integer array returned from HpStartCmndo.

buffer is an integer array in which the data is returned.

len is a positive number of characters to read.

prompt is the prompt currently issued for the read.

promptlen is the number of characters in the prompt, not including the underscore (_) character.

Returns:

A-Register DVT word 6 (returned from the EXEC read).

B-Register positive number of characters read, or −1 if the Class Get fails. (If the Class Get fails, CMNDO terminated abnormally.)

HpStopCmndo, Terminate CMNDO Slave Monitor

HpStopCmndo terminates the CMNDO clone and optionally posts the command stack to a file. If this routine is not called, the clone is aborted by the system when the caller of HpStartCmndo terminates. In addition to terminating the CMNDO clone, both Class numbers are returned to the system.

```
CALL HpStopCmndo(class[,stackfile])

integer*2 class(2)
character*(*) stackfile
```

where:

class is a two-word integer array returned by HpStartCmndo.

stackfile is an optional FORTRAN character string containing the name of a command stack file.

If the *stackfile* parameter is passed and the character string is not blank, CMNDO will try to post the current command stack into the file named in *stackfile*.

Example Program Using CMNDO

```
      Program ReadIt
      implicit none

c     The command stack monitor, CMNDO, can be used programmatically to
c     easily add the RTE-style command stack and the $VISUAL command
c     line editing features to a program.  Communication with the
c     monitor is performed via Class I/O.  The Class numbers returned
c     by HpStartCmndo must be used in subsequent calls to HpReadCmndo
c     and HpStopCmndo.

      integer*2 stkerr, class(2), buffer(128), areg, len
      integer*2 prompt(4), plen
      character stackfile*64

      integer*2 HpStartCmndo
      integer*4 HpReadCmndo

c     Start up a copy of CMNDO, the command stack slave monitor.  CMNDO
c     will perform the command stack initialization and attempt to
c     restore the command stack from the named file.

      stackfile = './READIT.STK'
      if (HpStartCmndo(1, class, stackfile, stkerr).ne.0) stop
      prompt(1) = 2hPr
      prompt(2) = 2hom
      prompt(3) = 2hpt
      prompt(4) = 1h>
```

```
      plen = 8          ! length of the prompt in characters, 'Prompt> '

c     Loop until the user enters "EX".

      do while (.not.(buffer(1).eq.2hEX.and.len.eq.2))

c     Issue the prompt.

        call exec(2,2101b,prompt,-plen)

c     Tell the CMNDO monitor to issue a read.

c     The CMNDO monitor will perform the read with a call to
c     RteShellRead.  Depending on the user's $VISUAL environment
c     variable, command line editing may or may not be enabled.  (Note
c     that CMNDO can only retrieve exported variables.)  The standard
c     "RTE-style" command stack functions are always available
c     regardless of the state of the user's exported environment
c     variable block.

c     Note that the prompt is assumed to have already been issued.
c     The contents of the prompt are still required because some
c     of the command line editing commands refresh the current
c     input line.  (If no prompt is being used, the length of the
c     prompt, "plen",  should be set to 0.)

        call HpReadCmndo(class, buffer, 256, prompt, plen, 1)
        call abreg(areg, len)

c     A-Register - DVT 6
c     B-Register - transmission log (number of characters read),
c                  positive number of characters read, or
c                  -1 if an EXEC error.

        if (len.lt.0) stop

c     Process the buffer : ...

        call exec(2,1,buffer,-len)
      enddo

c     Tell the CMNDO monitor that we are finished.  Post the current
c     stack to a stack file if we successfully read a stack file in
c     HpStartCmndo.  The Class numbers are returned to the system
c     and CMNDO will terminate.

c     If a stack file is not being used and the calling program is
c     going to exit immediately, the call to HpStopCmndo is optional.
c     The system will abort CMNDO when the calling program terminates.

      if (stkerr.lt.0) stackfile = ' '
      call HpStopCmndo(class, stackfile)

      end
```

CmndStackInit, Initialize Command Stack

This routine is used to initialize the stack area before other command stack routines can be used. There is one word per line overhead and one additional word. A useful approximation is 64 lines of 30 characters per 1KW allocated.

```
CALL CmndStackInit(stack,size,MaxLine,TimeOutMask,ScreenWidth)

integer*2 stack(*),size,MaxLine,TimeOutMask,ScreenWidth
```

where:

stack	is the buffer to be used to keep the stack lines.
size	is the number of words reserved for the stack.
MaxLine	is the number of bytes allowed in the longest command; should be the same as the length of the input buffer used to read the commands.
TimeoutMask	is the timeout bit (described below).
ScreenWidth	is the width of a line on the CRT screen (described below).

The *TimeOutMask* is used as a mask to check the status returned by the terminal driver. In RTE-A, it is always 2 (bit 1). For RTE-6/VM, the Revision C compatible drivers returned bit 0 on timeout, and the Revision D compatible drivers use bit 1, as does RTE-A. Routine HpCrtSsrcDriver determines if the driver is Revision D compatible.

The *ScreenWidth* parameter must be set correctly; otherwise, the stack display may or may not work, but stack read back definitely will not work correctly. We recommend that you call HpCrtCheckStraps (described in Chapter 12 of this manual) to get the correct number at runtime.

Note that the command stack routines treat all terminals as 80 column terminals regardless of their configuration. Therefore, the command stack routines will leave the margin at 80 even if you're working with margins greater than 80 columns on a horizontally scrolling terminal.

CmndStackMarks, Check for Marked Lines

This routine is called when your program is ready to receive another input line to determine if a line marked for grouped execution is pending. If so, the line is returned in *ibuf* and the line is no longer marked for grouped execution.

linemarked = CmndStackMarks(*ibuf*,*bytelen*)

```
integer*2 ibuf(*),bytelen
logical*2 linemarked,CmndStackMarks
```

where:

linemarked is true if a line was marked for grouped execution, or false if no marked line is pending.

ibuf returns the marked line, if any.

bytelen returns the length of the marked line in bytes.

CmndStackPush, Add Line to Command Stack

This routine is called to add a line to the command stack. If necessary, the oldest line or lines in the stack are "forgotten" to make room for the new line.

CALL CmndStackPush(*ibuf*,*bytelen*[,*duplicatesOK*])

```
integer*2 ibuf(*),bytelen,duplicatesOK
```

where:

ibuf is the line to be pushed onto the stack.

bytelen is the length of the line in bytes.

duplicatesOK is an optional parameter that specifies if duplicate lines are allowed in the command stack. If *duplicatesOK* is not passed or equals 0, duplicate lines are not kept in the command stack. If it is nonzero, duplicate lines are allowed in the command stack.

CmndStackRestore, Restore Command Stack

This routine is used to read a stack file previously saved by CmndStackStore into the command stack. The stack must have been previously initialized with CmndStackInit.

lines = CmndStackRestore(*dcb*,*fmperr*,*stackfile*,*buffers*,*linebuf*,*buflen*)

```
integer*2 lines,CmndStackRestore, dcb(*),fmperr,buffers,linebuf(*),buflen
character*(*) stackfile
```

where:

lines returns the positive number of lines read from the file, or the negative FMP error code, if any.

dcb is the DCB used to read the stack file.

fmperr is the returned FMP error code.

stackfile is the stack file descriptor.

buffers is the number of buffers in the *dcb* parameter. This is the value that is passed to the FmpOpen call that specifies the number of buffers in the DCB. Refer to the FmpOpen call description in the *RTE-A Programmer's Reference Manual*, part number 92077-90007.

linebuf is the buffer for lines read from the stack file; should be as long as the longest command line possible.

buflen is the maximum length in bytes of *linebuf*.

CmndStackSaveP, CmndStackRstrP, Save and Restore Command Stack

These routines are used to save and restore the command stack in some way other than by using files (for example, in EMA). To save a command stack, you must call CmndStackSaveP to read the number of lines and word occupied. This data must be saved along with the stack, by whatever means the programmer wishes to use. To restore the saved stack, two alternatives exist:

1. Call CmndStackInit to set up an empty stack.
   Transfer the data from EMA to the stack buffer.
   Call CmndStackRstrP to set the pointers.

2. Transfer the data from EMA first.
   Save the first word of the buffer in a safe place.
   Call CmndStackInit.
   Restore the first word of the buffer.
   Call CmndStackRstrP to set the pointers.

```
CALL CmndStackSaveP(pointers)
integer*2 pointers(2)
```

where:

pointers is a two-word buffer to receive stack pointer information.

Word 1 is the number of lines in the stack.

Word 2 is the number of occupied words.

```
CALL CmndStackRstrP(pointers)
integer*2  pointers(2)
```

where:

pointers is a two-word buffer to pass back to stack routines.

Word 1 is the number of lines in the stack.

Word 2 is the number of occupied words.

The pointers must not be altered between the save and restore calls. CmndStackSaveP can be used at any time to display the stack statistics since it does not alter the internal pointers but merely copies them.

CmndStackScreen, Do Stack Interactions with User

This subroutine manages the command stack for client programs. When user input is accepted and delivered to this routine, CmndStackScreen determines whether it is a command stack command. If it is not, it is returned just as given. If it is a command stack command, the appropriate stack functions are performed and the resulting line is returned.

```
CALL CmndStackScreen(crt,ibuf,bytelen,framesize,label,oops)

integer*2   crt,ibuf(*),bytelen,framesize
character*(*) label,oops
```

where:

crt	is the LU of the user's terminal.
ibuf	passes in the user command that might be a stack command; passes out the line undisturbed, or a line just read from the stack.
bytelen	passes in the number of characters in the current line; passes out the same, or the number of characters on the line just read from the stack.
framesize	is the number of lines to display on the screen.
label	is the message to display before the stack is displayed.
oops	is the message to display when search fails.

Note that it is possible for the routine to return a zero length and a blank command buffer, if that is what you enter. Also, do not trust anything in the buffer past the returned byte length, since the buffer is used extensively as a work buffer for all communications with the user.

CmndStackStore, Store Command Stack Contents in a File

This routine copies the contents of the command stack to a file. If no directory is specified in the stackfile parameter, the session's home directory is used for the file. If a file with the given name already exists, it is overwritten; otherwise, the file is created.

```
error = CmndStackStore(dcb,fmperr,stackfile,buffers)

integer*2 fmperr,buffers,dcb(*)
logical   error
character*(*) stackfile
```

where:

dcb is a DCB used to write to the stack file.

fmperr returns an FMP error code.

stackfile is the stack file file descriptor.

buffers is the number of buffers in the *dcb* parameter. This is the value that is passed to the FmpOpen call that specifies the number of buffers in the DCB. Refer to the FmpOpen call description in the *RTE-A Programmer's Reference Manual*, part number 92077-90007.

error returns TRUE if an error occurs.

CmndStackUnmark, Remove Marks from Command Stack Lines

This procedure removes all marks from lines in the command stack in order to facilitate error recovery.

```
CALL CmndStackUnmark
```

Command Stack Example Program

```
$alias     HpRte6, direct

  implicit none

  integer*2   Ibufl
  parameter  (IbufL = 256)        ! maximum byte length of Ibuf

  integer*2
     > Ibuf (0:127)               ! up to 256 character commands
     >,ByteLen                    ! current length in bytes of command

  integer*2                       ! declare a stack buffer as big
     >,StackBuffer (0:4095)       ! as desired

  integer*2
     >,DCB (16+128)               ! file DCB buffer

  integer*2
     > Error,                     ! error code from HpCrtCheckStraps
     > Crt,                       ! terminal lu
     > Status,                    ! terminal status from read
     > TOMask,                    ! timeout mask for terminal reads
     > MemSize,                 ! #K bytes of display memory in terminal
     > ScreenWidth,               ! number of characters visible at one
                                  ! time on a CRT line
     > Framesize                  ! # lines in stack window

  logical*2
     > HpCrtCheckStraps,          ! the CRT and Port are OK for stacks
     > CmndStackMarks,            ! a marked line is available for use
     > CmndStackStore,            ! save stack in a file
     > CmndStackRestore,          ! read stack from a file
     > HpRte6,                    ! is this an RTE-6 system?
     > HpCrtSSRCDriver,           ! is terminal on a Rev.D mux?
     > IfBrk                      ! user has set the break flag

*  Begin example:

  CRT = 1                         ! use terminal lu 1
  Framesize = 22                  ! use 22 lines per frame
  TOMask = 2                      ! determine timeout mask

  if (HpRte6()) then
    if (.not. HpCrtSSRCDriver (Crt)) TOMask = 1
  endif

  if ( HpCrtCheckStraps (Crt,Error,MemSize,ScreenWidth) ) then
     continue
```

```
  else
     call HpCrtSendChar(Crt, 'Command Stack Unavailable')
  endif

   call CmndStackInit(StackBuffer,4096,256,TOMask,ScreenWidth)

  if ( CmndStackRestore(Dcb,Error,'StackFile',1,Ibuf,IbufL) ) then
                          ! Process the FMP error that was returned
  endif

10     If ( .NOT.CmndStackMarks(Ibuf,ByteLen) ) then
       call HpZQandA(Crt,'Next Command? ',       !  issue prompt
       >     Ibuf,IbufL,Status,ByteLen)          !  and read a command
       if ( iand(Status,TOMask).ne.0 ) go to 10  !  check for timeout

  endif

   Call CmndStackScreen(Crt,Ibuf,ByteLen,
       >               FrameSize,'Commands:',
       >               'No match, try again')

   call CmndStackPush(Ibuf,ByteLen)

*  Process command:

  If ( Ibuf(0).eq.2hEX ) then
    go to 9000
  else

    ! (Handle other commands...)

  endif

  If (IfBrk()) then          !  when break is set,
     call CmndStackUnmark    !  abandon all marks.

  endif

  go to 10

*  This is the program exit section.

9000  continue

  if ( CmndStackStore(Dcb,Error,'StackFile',1) ) then
    !  Process the FMP error that was returned
  endif

  end
```

Concat, Concatenate Strings

This routine concatenates two strings together after removing trailing blanks from the first string. The result is returned in the first string.

```
CALL Concat(string1,string2)

character*(*) string1,string2
```

where:

string1,
string2 are character strings of any length.

If the second string is too large to fit in the first string, the result will be truncated on the right.

Example:

```
string1 = 'abc     '
string2 = 'de      '
call  Concat(string1,string2)
string1 = 'abcde   '
```

ConcatSpace, Concatenate Strings with Embedded Blanks

This routine is similar to Concat, except that a caller specified number of blanks are inserted between the two strings.

```
CALL ConcatSpace(string1,string2,spaces)

character*(*) string1,string2
integer*2 spaces
```

where:

string1,
string2 are character strings of any length.

spaces is the number of blanks to insert after *string1* before concatenating *string2*.

Example:

```
string1 = 'abc     '
string2 = 'de      '
call  ConcatSpace(string1,string2,2)
string1 = 'abc de  '
```

DayTime, Seconds Since January 1, 1970

This subroutine returns an ASCII time string corresponding to the passed number of seconds since 12 AM January 1, 1970.

```
CALL DayTime(time,buffer)

integer*4 time
character*(*) buffer
```

where:

time is the double-word integer number of seconds since 12 AM, January 1, 1970.

buffer is the returned character string, with the entire string occupying 28 characters, but a longer or shorter buffer can be supplied, in which case the string is truncated or padded with blanks, respectively.

For example, 360000000 => Fri May 29, 1981 4:00:00 pm

DecimalToDint, ASCII to Double Integer Conversion

This function is an ASCII to double integer conversion that returns the double integer value of the characters contained in the specified string (one integer for the entire string). Blanks are ignored, except that an all blank string is an error.

```
dintValue = DecimalToDint(string,error)

integer*4 dintValue,DecimalToDint
character*(*) string
integer*2 error
```

where:

string is the character string to convert, whose characters must comprise a legal double integer, in the range −2147483648 to +2147483647; a plus or minus sign can be specified, although the octal qualifier "B" is not allowed.

error is a required one-word integer variable that returns zero to indicate no error, or nonzero to indicate an invalid character was encountered or overflow occurred (zero is returned as the function value if an error occurs).

dintValue is a double-word integer variable in which the integer value of the character string is returned.

Compare this function to DecimalToInt.

See also HpZ.

DecimalToInt, ASCII to Single Integer Conversion

This function is an ASCII to single integer conversion that returns the single integer value of the characters contained in the specified string (one integer for the entire string). Blanks are ignored, except that an all blank string is an error.

```
intValue = DecimalToInt(string,error)

integer*2 intValue,DecimalToInt,error
character*(*) string
```

where:

string is the character string to convert, whose characters must comprise a legal single integer in the range −32768 to +32767; a plus or minus sign (or no sign at all) can be specified, although the octal qualifier "B" is not allowed.

error is a required one-word integer variable that returns zero to indicate no error, or nonzero to indicate an invalid character was encountered or overflow occurred (zero is returned as the function value if an error occurs).

Note: Prior to Revision 5000, the *error* parameter was not required and the call "`intvalue = DecimalToInt(`*string*`)`" was valid. However, as of Revision 5000, the *error* parameter is required.

intvalue is a one-word integer variable in which the integer value of the character string is returned.

Compare this function to DecimalToDint.

See also HpZ.

DintToDecimal, Double Integer to ASCII Conversion

This function returns a character string representation of a double integer number. The string includes a leading minus sign if the number is negative. Leading zeros are suppressed, and the whole number is left justified in the string with trailing blanks. The character string should be at least 11 characters so that the largest double integer can be represented. A smaller string can be used; however, the rightmost digits will be truncated.

```
string = DintToDecimal(number)

character*(*) string,DintToDecimal
integer*4 number
```

where:

number is a double integer number to process.

string is a character string in which a character representation of the number is returned.

Compare this function to IntToDecimal.

See also HpZ.

DintToDecimalr, Double Integer to ASCII Conversion

This function returns a character string representation of a double integer number. The string includes a leading minus sign if the number is negative. Leading zeros are suppressed, and the whole number is right justified in the string with leading blanks. The character string should be at least 11 characters so that the largest double integer can be represented. A smaller string can be used; however, the leftmost digits will be truncated.

```
string = DintToDecimalr(number)

character*(*) string,DintToDecimalr
integer*4 number
```

where:

number is a double integer number to process.

string is a character string in which a character representation of the number is returned.

Compare this function to IntToDecimalr.

See also HpZ.

DintToOctal, Double Integer to Octal Conversion

This function returns a character string representation of a double integer number. The number is returned unsigned. Leading zeros are suppressed, and the whole number is left justified in the string with trailing blanks. The character string should be at least 11 characters so that the largest double integer can be represented. A smaller string can be used; however, the rightmost digits will be truncated.

```
string = DintToOctal(number)

character*(*) string,DintToOctal
integer*4 number
```

where:

number is a double integer number to process.

string is a character string in which a character representation of the number is returned.

Compare this function to IntToOctal.

See also HpZ.

DintToOctalr, Double Integer to Octal Conversion

This function returns a character string octal representation of a double integer number. The number is returned unsigned. Leading zeros are suppressed, and the whole number is right justified in the string with leading blanks. The character string should be at least 11 characters so that the largest double integer can be represented. A smaller string can be used; however, the leftmost digits will be truncated.

```
string = DintToOctalr(number)

character*(*) string,DintToOctalr
integer*4 number
```

where:

number is a double integer number to process.

string is a character string in which a character representation of the number is returned.

Compare this function to IntToOctalr.

See also HpZ.

DiscToBlock

This subroutine converts track, sector, and sectors per track information into a double integer block number.

```
CALL DiscToBlock(block,spert,track,sector)
```

where:

block is a double-word integer containing the returned block number. (The first block on a disk LU is block 0.)

spert is a one-word integer variable containing the number of 64 word sectors per track on the LU of interest.

track is a one-word integer variable containing the track number to be converted.

sector is a one-word integer variable containing the sector to be converted.

DiscSize, Tracks and Sectors Per Track

This subroutine returns information about the number of tracks and sectors per track on a disk LU. This subroutine's behavior is not defined for other than disk LUs.

```
CALL DiscSize(lu,ntracks,spert)
```

where:

lu is a one-word integer, a disk LU.

ntracks is a one-word integer, the number of tracks on the specified LU.

spert is a one-word integer, the number of 64 word sectors per track on the specified LU.

ElapsedTime

This double integer function returns the number of milliseconds that have passed since the last time recorded by ResetTimer (refer to that section). It does not attempt to correct for any overhead in the timing measurement. It does not clear the timer, so it can be used to get "splits" (intermediate timing points in an ongoing event) at various checkpoints. Elapsed time is only valid within 24 hours of calling ResetTimer.

```
milliseconds = ElapsedTime()
```

where:

milliseconds is a double-word integer variable in which the number of milliseconds that have passed since the last time recorded by ResetTimer is returned.

ElapsedTime must be declared as a double integer function.

ETime

This double integer function returns the number of centiseconds that have passed since the specified base time. It does not attempt to correct for any overhead in the timing measurement. The base time is updated each time the routine is called so that running time is kept. The elapsed time is only valid within a 24-hour period.

```
centiseconds = ETime(btime)
```

where:

centiseconds is a double-word integer variable in which the number of centiseconds that have passed since the time specified by *btime* is returned.

btime is a double-word integer variable which represents the time from which to determine the number of centiseconds that have elapsed. *btime* is updated to the current time each time ETime is called so that running time is kept. By using more than one *btime*, you can time as many processes as desired.

Fgetopt, Get a Runstring Option

Fgetopt processes single-character options from a runstring. Fgetopt assumes that options are immediately preceded by a hyphen (for example,−O) and that all options are specified at the beginning of a runstring. Fgetopt can optionally search for options that begin with a plus (+). As each option is processed, it is removed from the runstring. Fgetopt can also process options with arguments. The argument is assumed to immediately follow the option (for example, "−O *arg*" or "−O*arg*"). Fgetopt returns FALSE when there are no more options to process.

logical = Fgetopt(*runstring*,*opts*,*option*,*argument*[,*optflag*])

```
logical*2 logical,Fgetopt
character*(*) runstring, opts, option, argument
integer*2 optflag
```

where:

runstring is a FORTRAN character string containing the runstring to be processed. It is expected that the "RU,PROG" or "XQ,PROG" has already been removed (GETST format). Fgetopt modifies this string such that a call to SPLITSTRING returns the next option or parameter.

opts is a FORTRAN character string containing the list of recognized character options. If the character is followed by a colon, the option is expected to have an argument that may or may not be delimited. The argument is returned in the *argument* parameter. Option characters cannot be a question mark (?), colon (:), comma (,) or space ( ).

option is a FORTRAN character string returned as the next option letter in the runstring, or returned as '?' when an error is encountered. Possible causes are unknown options and non-existent arguments.

argument is a FORTRAN character string that is returned when an option requires an argument.

optflag is an optional single-word integer with bits 15 and 14 defined as follows:

bit 15 if equal to 1, allow error messages to be output to the user's terminal LU. (Default = enabled.)

bit 14 if equal to 1, enable options to be flagged with either a plus (+) or minus (−) character. Options flagged with '+' are returned in uppercase and options flagged with '−' are returned in lowercase. (Default = disabled.)

Example:

The following program segment demonstrates how Fgetopt can be used to process a runstring for a program that has 3 possible options, “U”, “F”, and “O”. The “F” and “O” options require an argument to be passed with the option.

```
          Integer   areg, len, runbuff(128)
          Character runstring*256, option*1, arg*64,
         +          firstparm*64, file_options*10
          Logical   Fgetopt, Uflag
          equivalence (runbuff, runstring)

c     get the runstring,

          call exec(14,1,runbuff,-256)
          call abreg(areg, len)

c     take out the 'RU' and the program name,

          call SplitString(runstring(1:len),arg,runstring)
          call SplitString(runstring,arg,runstring)

          do while (Fgetopt(runstring,'UF:O:',option,arg)
             if (option.eq.'U') then              ! -U
                Uflag = .true.

             else if (option.eq.'F') then         ! -F arg
                file = arg

             else if (option.eq.'O') then         ! -O arg
                file_options = arg

             else if (option.eq.'?') then         ! illegal option
                 write(1,'("Usage: prog [-u] [-f name] [-o opts] parm")')
                stop
             endif
           enddo

c       Fgetopt removes the options from the "runstring".
c       A call to "splitstring" will return the first parameter
c       after the options in the runstring.

           call splitstring(runstring,firstparm,runstring)
          .
          .
          .
```

GetFatherIdNum

This function returns the ID segment number of the calling program's 'father', that is, the program that scheduled the calling program. If the calling program has no father, the value returned is zero.

```
fatheridnum = GetFatherIdNum()

integer*2 fatheridnum,GetFatherIdNum
```

where:

fatheridnum is the returned ID segment number of the program that scheduled the calling program.

Returns:

nonzero Father's ID segment number.

0 Program has no father.

GetRedirection, Extract I/O Redirection Commands

GetRedirection scans a runstring for I/O redirection strings (for example, >*output*, >>*output*, <*input*). The redirection strings are removed from the runstring and returned to the caller. For runstrings that contain multiple redirection strings, all redirection strings are removed, but only the last one is returned.

The less than (<) or greater than (>) character must be preceded by a comma. The redirection prefix ('<', '>', or '>>') will not be included in the input or output file descriptors that are returned.

If the *input* parameter is not supplied, GetRedirection does not scan for and remove input redirection strings.

```
length = GetRedirection(runstring,output,append[,input])

integer*2 length, GetRedirection
character*(*) runstring, output, input
logical append
```

where:

length returns the new length of the runstring in bytes.

runstring is the program runstring (it will get modified by GetRedirection if it contains a redirection string).

output is returned as the output file descriptor.

append is set to TRUE if the output file is preceded by '>>'.

input is optional and returns the input file descriptor.

GetRteTime

This function returns the current system time represented as the number of centiseconds since midnight. It also returns the combined year and day of year in the RTE system internal format. This routine executes with less system overhead than the EXEC 11 call.

```
TimeNow = GetRteTime(RteDate)

integer*4 TimeNow,GetRteTime
integer*2 RteDate
```

where:

TimeNow is the current system time of day, as the number of centiseconds since midnight.

RteDate is the combined year/day in RTE system internal format.

See also YrDoyToRteDate.

HexToInt

This function converts an ASCII hexadecimal value to the corresponding single integer value.

```
intval = HexToInt(string,error)

integer*2 intval,HexToInt,error
character*(*) string
```

where:

intval is a one-word integer in which the value of the hexadecimal string is returned.

string is a character string (legal, unsigned, hexadecimal) to convert to an integer value.

error returns 0 for no error, or nonzero if the string is not a valid hexadecimal number.

Returns:

0 if *error* is true.

See also HpZHexi.

HMSCtoRteTime

This function performs the inverse of RteTimeToHMSC; it converts Hour-Minute-Second-Centisecond to RTE time format. The calling routine must ensure the validity of the time values, as no checking is done in the function.

```
RteTime = HMSCtoRteTime(Hour,Minute,Second,Centisecond)

integer*4 RteTime,HMSCtoRteTime
integer*2 Hour,Minute,Second,Centisecond
```

IdAddToName, Convert ID Segment Address to Program Name and LU Number

This subroutine converts an ID segment address to a program name and LU number. No error checking of the address is done, but *name* and *lu* will be invalid if *add* is invalid.

```
CALL IdAddToName(add,name,lu)

integer*2 add,name(3),lu
```

where:

add	is a one-word integer variable, the ID segment address.
name	is a three-word integer array, the name of the program at the ID segment address specified by *add*. *name* is undefined if the passed ID segment address is not a valid ID segment address.
lu	is a one-word integer variable, the LU number. *lu* is undefined if the passed ID segment address is not a valid ID segment address.

IdAddToNumber, Convert ID Segment Address to ID Segment Number

This function converts an ID segment address to an ID segment number.

```
idNumber = IdAddToNumber(add)

integer*2 idNumber,IdAddToNumber,add
```

where:

add	is a single integer representing the ID segment address.
idNumber	is a one-word integer variable that returns the ID segment number.

Returns:

nonzero	ID segment number corresponding to the address.
0	There is no such ID segment.

IDCLR

This subroutine causes the ID segment for the calling program to be deallocated when the program terminates.

```
CALL IDCLR()
```

The ID segment is marked such that if the program terminates without going either "Dormant Serially Reusable" or "Dormant Saving Resources" and is not in the time list, the ID segment is returned to the system for use by another process.

IdNumberToAdd, Convert ID Segment Number to ID Segment Address

This function converts an ID segment number to an ID segment address.

```
idAddress = IdNumberToAdd(number)

integer*2 idAddress,IdNumberToAdd,number
```

where:

number is an integer representing the ID segment number.

idAddress is an integer variable that returns the ID segment address.

Returns:

nonzero ID segment number corresponding to the address.

0 Passed number is not a legal ID segment number.

IntString

This subroutine converts an integer into printable form. If the number is printable ASCII, the two ASCII characters are returned; otherwise, the ASCII equivalent of the number is returned. The returned string can be used for printing.

INTSTRING was written to process FMGR security codes into strings. Because lowercase security codes do not go through the standard interactive interfaces (they are upshifted), lowercase must be converted as numeric. This is also true of leading digits, (:) and (_).

```
CALL IntString(num,string)

integer*2 num
character*(*) string
```

where:

num is the single integer to be converted.

string is the character string that contains the returned ASCII equivalent of *num*.

Examples:

```
CALL IntString(num,string)

where:  num    = 345
        string = 345

        num    = 40502B
        string = AB

        num    = 18505
        string = HI
```

IntToDecimal, Integer to ASCII Conversion

This function returns a character string representation of a number. The string includes a leading minus sign if the number is negative. Leading zeros are suppressed, and the whole number is left justified in the string with trailing blanks. The character string should be at least six characters so that the largest one-word integer can be represented. A smaller string can be used; however, the rightmost digits will be truncated.

```
string = IntToDecimal(number)

character*(*) string,IntToDecimal
integer*2 number
```

where:

number is a one-word integer number to process.

string is a character string in which is returned a character representation of the number.

Compare this function to DintToDecimal.

See also HpZ.

IntToDecimalr, Integer to ASCII Conversion

This function returns a character string representation of a number. The string includes a leading minus sign if the number is negative. Leading zeros are suppressed, and the whole number is right justified in the string with leading blanks. The character string should be at least six characters so that the largest one-word integer can be represented. A smaller string can be used; however, the leftmost digits will be truncated.

```
string = IntToDecimalr(number)

character*(*) string,IntToDecimalr
integer*2 number
```

where:

string is a character string in which a character representation of the number is returned.

number is a one-word integer number to process.

Compare this function to DintToDecimalr.

See also HpZ.

IntToHex

This function returns a character string of the hexadecimal representation of a number. Four hexadecimal digits are returned, unsigned, with leading zeros.

```
string = IntToHex(number)

character*(*) string,IntToHex
integer*2 number
```

where:

string is the returned character representation.

number is a one-word integer to convert.

See also HpZ.

IntToHexR

This function converts an integer to an ASCII hexadecimal string with right justification.

```
string = IntToHexR(number)

character*(*) string,IntToHexR
integer*2 number
```

where:

string is the returned character representation.

number is a one-word integer to convert.

See also HpZ.

IntToOctal, Integer to Octal Conversion

This function returns a character string octal representation of a number. The number is returned unsigned. Leading zeros are suppressed, and the whole number is left justified in the string with trailing blanks. The character string should be at least six characters so that the largest one-word integer can be represented. A smaller string can be used; however, the rightmost digits will be truncated.

```
string = IntToOctal(number)

character*(*) string,IntToOctal
integer*2 number
```

where:

string is a character string in which is returned a character representation of the number.

number is a one-word integer number to process.

Compare this function to DintToOctal.

See also HpZ.

IntToOctalr, Integer to Octal Conversion

This function returns a character string octal representation of a number. The number is returned unsigned. Leading zeros are suppressed, and the whole number is right justified in the string with leading blanks. The character string should be at least six characters so that the largest one-word integer can be represented. A smaller string can be used; however, the leftmost digits will be truncated.

```
string = IntToOctalr(number)

character*(*) string,IntToOctalr
integer*2 number
```

where:

string is a character string in which a character representation of the number is returned.

number is a one-word integer number to process.

Compare this function to DintToOctalr.

See also HpZ.

InvSeconds

This routine performs a conversion which is the inverse of the Seconds routine. This routine dissasembles the number of seconds since January 1, 1970 into conventional time values. The seconds since 1970 format is used extensively in the FMP file system for timestamps.

```
Call InvSeconds(SecsSince70,Year,Doy,Hour,Minute,Second)

integer*4 SecondsSince70
integer*2 Year,Doy,Hour,Minute,Second
```

where:

SecsSince70	is seconds since midnight.
Year	is the year A.D. (for example, 1993).
Doy	is the cardinal day of the year (1..366).
Hour	is the hour of the day in military time (0..23).
Minute	is the number of minutes into the current hour (0..59).
Second	is the number of seconds into the current minute (0..59).

See also Seconds.

LastMatch

This function returns the position in string of the last occurrence of *char*. It performs a backward search from the end of string until it finds *char*. If *char* is not found, a zero is returned.

```
index = LastMatch(string,char)
```

where:

string	is the character string to be examined.
char	is a single character string searched for.
index	is the position of *char* in the string that is returned, an integer value.

LeapYear

This routine tests a given year to see if it is a leap year. The algorithm follows the correct rules for 4, 100, and 400 divisibility.

```
flag = LeapYear(year)

logical*2 flag,LeapYear
integer*2 year
```

where:

flag	is a flag that will be TRUE if the year is a leap year.
year	is the year to be tested (for example, 1980 or 1985).

LuLocked

This function reports if the passed LU is locked. The value can be used as a logical condition.

lockedFlag = LuLocked(*lu*)

where:

lu is a one-word integer variable containing the LU number.

lockedFlag is a one-word integer variable that reports the locked condition of the LU.

Returns:

0 LU is not locked or is illegal.

nonzero LU is locked.

MoveWords

This subroutine moves a specified number of words from one location to another. It uses the MVW instruction, which moves the words one at a time starting with the first. Do not use this subroutine to move character strings to or from another location, as it will not work.

CALL MoveWords(*from*,*to*,*count*)

where:

from is the location, usually an integer array or array element, from which the specified number of words are to be moved.

to is the location, usually an integer array or array element, to which the specified number of words are to be moved.

count is the one-word integer number of words to move.

See also CompareWords.

MyIdAdd, Return ID Segment Address

This function returns the calling program's ID segment address.

idAddress = MyIdAdd()

where:

idAddress is a one-word integer variable in which the ID segment address is returned.

NumericTime

This subroutine returns a numeric ASCII time string corresponding to the passed number of seconds since 12 AM January 1, 1970.

```
CALL NumericTime(time,buffer)
```

where:

time is the double-word integer number of seconds since 12 AM January 1, 1970.

buffer is a character string, with the entire returned string occupying 13 characters, but a longer or shorter buffer can be supplied, in which case the string is truncated or padded with blanks, respectively.

For example, 360000000 => 810529.160000

Compare this subroutine to DayTime.

OctalToDint, ASCII to Double Integer Conversion

This function is an ASCII to double integer conversion that returns the double integer value of the characters contained in the specified string (one integer per string). Blanks are ignored, except that an all blank string is an error.

```
dintValue = OctalToDint(string,error)
```

where:

string is the character string to convert, whose characters must comprise a legal, unsigned octal double integer, in the range 0 to 37777777777B. The octal qualifier “B” is not allowed.

error is a required one-word integer variable that returns zero to indicate no error, or nonzero to indicate an invalid character was encountered or overflow occurred (zero is returned as the function value if an error occurs).

dintValue is a double-word integer variable in which the integer value of the character string is returned.

Compare this function to OctalToInt.

See also HpZ.

OctalToInt, ASCII to Single Integer Conversion

This function is an octal ASCII to single integer conversion that returns the single integer value of the characters contained in the specified string (one integer per string). Blanks are ignored, except that an all blank string is an error.

intValue = `OctalToInt(`*string*`,`*error*`)`

where:

string is the character string to convert, whose characters must comprise a legal, unsigned octal integer, in the range 0 to177777B. The octal qualifier "B" is not allowed.

error is a required one-word integer variable that returns zero to indicate no error, or nonzero to indicate an invalid character was encountered or overflow occurred (zero is returned as the function value if an error occurs).

intValue is a one-word integer variable in which the integer value of the character string is returned.

See also HpZ.

ProgramPriority

This function returns the priority of a program.

priority = `ProgramPriority(`*program*`)`

where:

program is a three-word integer array containing the name of the program whose priority or existence is being requested.

priority is a one-word integer variable in which the program's priority is returned.

Returns:

nonzero Priority of the program.

0 Program does not exist.

See also SetPriority.

ProgramTerminal

This function returns the LU of the terminal associated with the named program. Among other things, this is the LU at which that program's I/O to LU 1 will actually take place.

```
lu = ProgramTerminal(program)
```

where:

program is a three-word integer array containing the program name for which the terminal LU is being requested.

lu is a one-word integer variable in which the terminal LU associated with the program is returned.

Returns:

nonzero LU of the terminal associated with the named program.

PutInCommas

This subroutine prepares a string of parameters for parsing by routines expecting commas as separators.

```
CALL PutInCommas(string)
```

where:

string is the character string to process.

The subroutine separates parameters at blanks or commas, then rebuilds the string in place with single commas between parameters and blanks deleted; for example:

```
expr  w,x y,,z
```

becomes

```
expr,w,x,y,,z
```

ReadA990Clock (RTE-A Only)

This routine reads the calendar clock of the A990.

```
flag = ReadA990Clock(TimeArray,WorkBuf,error)

integer*2 TimeArray(8),WorkBuf(20),error
logical*2 flag,ReadA990Clock
```

where:

flag returns TRUE if an error is detected.

TimeArray is an array of integers that returns the time as follows:

word 1:	seconds past the minute (0..59)
word 2:	minutes past the hour (0..59)
word 3:	the hour of the day (0..23)
word 4:	the day of the week (0..6, Sunday = 0)
word 5:	the day of the month (1..28, 29, 30, 31)
word 6:	the month (1..12)
word 7:	the year (for example, 1991)
word 8:	RTE-Date word when the time was set

WorkBuf is the buffer used for communication with the A990 clock chip.

error is the error code if the function is .TRUE. :

−1	not an A990 CPU
−2	clock is not stable, delay 10 ms and try again
−3	battery is dead, time may not be correct
−4	clock has not been set

See also WriteA990Clock.

ResetTimer

This subroutine resets the timer used by ElapsedTime. It must be called first for ElapsedTime to give meaningful values, or any time thereafter to reset the timer.

```
CALL ResetTimer
```

Rex (Regular Expression) Routines

The Rex routines perform regular expression pattern matching and substitution on arbitrary character strings in very much the same way as EDIT/1000. Matching is briefly summarized in Table 7-1. Regular substitutions have the constructs shown in Table 7-2.

Table 7-1. Expression Pattern Matching Summary

Pattern	Match
a	Matches character “a”
.	Matches any character
@	Matches any character zero or more times (same as “.*”)
^x	Anchors the pattern to the beginning of line
x$	Anchors the pattern to the end of line
[ai−k]	Matches any of the characters “a”, “i”, “j”, “k”
[^ai−k]	Matches any characters but “a”, “i”, “j”, “k”
x*	Matches zero or more occurrences of pattern x
x+	Matches one or more occurrences of pattern x
x<5>	Matches 5 repetitions of pattern x
a:b	Matches a word boundary between patterns a and b
*	Matches the character “*”
{x}	Tags a subexpression for recall in substitutions

Table 7-2. Substitution Constructs

Substitution	Meaning
&	Recall the entire matched string
&1	Recall tagged field #1 in the matched field
>	Same as “&” but fold characters to uppercase
	(“>1” folds tagged field #1)
<	Same as “&” but fold characters to lowercase
<&>	Break the line into 2 lines at this point

RexBuildPattern

RexBuildPattern takes pattern string *string*, of which *stringlen* characters are significant, and builds a pattern for use by RexMatch and RexExchange into *pattern*, an array of *patternmax* words.

```
error = RexBuildPattern(string,pattern,stringlen,patternmax)

integer*2 RexBuildPattern,pattern,stringlen,patternmax
character string*(*)
```

where:

string	is the pattern string, which should already be folded to uppercase if case sensitivity is not desired.
stringlen	contains the significant characters.
pattern	is an array of *patternmax* words built into a pattern for use by RexMatch and RexExchange.
patternmax	should be about 200 words to handle most expressions (the repeat operator uses a lot of space).

Returns (error code):

−2	If the pattern is too complicated; does not fit in *pattern*.
−1	If an illegal regular expression syntax is found.
>0	If no problem occurs, *error* is the number of words of *pattern* generated.

RexBuildSubst

RexBuildSubst takes pattern string *string*, of which *stringlen* characters are significant, and builds a regular substitution string into *subpattern*, an array of *subpatternmax* words, for use by RexExchange. The *subpatternmax* parameter should include one word per character to be substituted and two words per recall operator (&, >, <). About 100 words handles most substitutions.

```
error = RexBuildSubst(string,stringlen,subpattern,subpatternmax)

integer*2 RexBuildSubst,stringlen,subpattern(*),subpatternmax
character string*(*)
```

Returns (error code):

−2	If pattern will not fit in *subpattern*.
−1	If an invalid regular expression syntax is found.
>0	If no problem occurs, *err* is the number of words of *subpattern* generated.

RexExchange

This routine replaces all occurrences of *pattern* (built by RexBuildPattern) in *foldedline*, of which *linelen* characters are significant, with substitution pattern *subpattern* (built by RexBuildSubst) into *newline*, the length of which is returned in *newlen*. *origline* is the unfolded version *foldedline*; if case sensitivity is desired, *foldedline* and *origline* should be the same string passed twice; otherwise, *foldedline* should be case folded prior to calling RexExchange.

error = RexExchange (*foldedline*,*linelen*,*origline*,*pattern*,*subpattern*,*newline*,*newlen*)

```
integer*2       RexExchange,linelen,pattern(*),subpattern(*),newlen
character*(*)   foldedline,origline,newline
```

Returns:

<0	Number of matches found, new line was truncated.
0	No matches.
>0	Number of matches found.

If no matches occur (returns=0), *newline* is unmodified. *newline* should replace *origline* only if exchanges are performed.

If RexExchange is used, the calling program must supply a routine named RexBreakLine that can be called as follows:

```
CALL RexBreakLine(string,length)

character string*(*)
integer*2 length
```

RexBreakLine is called whenever the break line operator <$> is encountered. *string* is the *newline* parameter to RexExchange and *length* is the number of valid characters in the broken line. This routine should process the new broken line the same as the normal RexExchange returned *newline* and return so that RexExchange can continue substituting on the current line. For example:

Suppose the match pattern is “abcdef”, the substitution pattern is “ab<$>cd<$>ef”, and RexExchange is called on line xxabcdefzz.

1. RexBreakLine is called with parameter “xxab”.
2. RexBreakLine is called with parameter “cd”.
3. RexExchange returns string “efzz”.

This is in keeping with EDIT/1000’s notion of the break line operator, creating separate lines of the broken lines and not inserting a carriage return.

RexMatch

This routine determines whether the supplied *string*, of which *stringlen* characters are significant, contains the *pattern* previously built by RexBuildPattern. The routine returns values of .true. or .false. accordingly. Note that *string* should already be folded to uppercase if case sensitivity is not desired.

```
match = RexMatch(string,stringlen,pattern)

logical RexMatch
integer*2 stringlen,pattern(*)
character string*(*)
```

RteDateToYrDoy

Convert from RTE's combined year/day format to Year and Cardinal Day

```
call RteDateToYrDoy(RteDate,Year,DayOfYear[,OS_Flag])

integer*2 RteDate,Year,DayOfYear
logical*2 OS_Flag
```

where:

RteDate	is the combined year/day in system format.
OS_Flag	indicates which date format to use. Because RTE-A and RTE-6/VM maintain the date in different formats, the *OS_Flag* parameter indicates which operating system format is in use, TRUE for RTE-6/VM or FALSE for RTE-A. The default for *OS_Flag* is FALSE.
Year	is the year A.D. (for example, 1989).
DayOfYear	is the day of the year (1..366).

This routine separates the combined year and day, as read from the system, into individual variables. RTE-A computes *RteDate* as (year−1976)*366+day. RTE-6/VM computes *RteDate* as the number of days since 1970.

See also GetRteTime and YrDoyToRteDate.

RteShellRead, Read from a Terminal and Enable Command Line Editing (RTE-A Only)

This routine reads from a terminal LU and enables the $VISUAL command line editing. (See the *RTE-A User's Manual*, part number 92077-90002, for more information about the $VISUAL command line editing modes.) This routine uses the $VISUAL, $LINES, $COLUMNS, and $KILLCHAR variables in the exported Environment Variable Block (EVB). This routine should be used in conjunction with the standard HP command stack library routines, CmndStackInit, CmndStackScreen, and CmndStackPush.

If $VISUAL is not found in the EVB, RteShellRead performs the read with an REIO call and command line editing is not available. $LINES defaults to 24, $COLUMNS defaults to 80, and $KILLCHAR defaults to the DEL character.

If one of the $VISUAL editing modes is enabled and the terminal LU supports FIFO mode, the LU is enabled for FIFO during the terminal I/O phase. The port is reset to its original state before RteShellRead returns. For optimal performance, the port should be configured to use FIFO mode with Xon/Xoff handshaking.

```
CALL RteShellRead(crt,bufr,bufln,len,status,prompt,plen,*alt_rtn)

integer*2 crt,bufr(*),bufln,len,status,prompt(*),plen
```

where:

crt is the LU of the user's terminal.

bufr is the integer array in which the data is returned.

bufln is the buffer length. A positive value indicates the number of words; a negative value indicates the number of characters in *bufr*.

len is returned as the transmission log, the positive number of words or characters (depending on *bufln*) transmitted in the *bufr* array.

status returns DVT word 6 of the terminal LU after the last read.

prompt is the integer array containing the prompt that is currently issued by the calling program. (Some of the editing commands require the prompt to be reissued.)

plen is the length of the prompt in characters. This length should not include the trailing underscore (_) character.

alt_rtn is the alternate return taken if the EXEC read is aborted. *alt_rtn* is a FORTRAN line number and must be preceded by an asterisk (*). See the *FORTRAN Reference Manual*, part number 92836-90001, for information on alternate returns in subroutines.

RteTimeToHMSC

This routine converts centiseconds since midnight to Hour-Minute-Second-Centisecond. Given the current system time, as read by GetRteTime, this routine converts it to conventional time values.

```
Call RteTimeToHMSC(RteTime,Hour,Minute,Second,Centisecond)

integer*4 RteTime
integer*2 Hour,Minute,Second,Centisecond
```

where:

RteTime	is the time of day, as returned by GetRteTime.
Hour	is the hour of the day, "military time" (0..23).
Minute	is the number of minutes into the current hour (0..59).
Second	is the number of seconds into the current minute (0..59).
Centisecond	is the number of centiseconds into the current second (0..99).

SamInfo, Return SAM Size (RTE-A Only)

This routine returns information about the number of free words in SAM or XSAM.

```
CALL SamInfo(whichsam,totalwords,freewords,maxfree)
```

where:

whichsam	is an integer that is 0 if SAM information is desired, or non zero for XSAM.
totalwords	is an integer that returns the total number of words of SAM or XSAM.
freewords	is an integer that returns the number of free SAM or XSAM words.
maxfree	is an integer that returns the number of words in the largest free block of SAM or XSAM.

Seconds

This function converts a time buffer and year into the number of seconds since 12 AM January 1, 1970.

totalSeconds = Seconds(*timebuff*,*year*)

where:

timebuff is the five-word integer time buffer, in the same format as that returned by a call to EXEC 11.

year is a one-word integer variable, the corresponding year.

totalSeconds is a double-word integer variable, the number of seconds since 12AM January 1, 1970 of *timebuff*.

SplitCommand, Parse String

This subroutine parses the string, stopping at the first semicolon or user defined character.

CALL SplitCommand(*string*,*part1*,*part2*,[*char*])

where:

string is the character string to process, with leading blanks stripped before the string is split to make the subroutine easier to use as a parameter separator.

part1 is a character string variable that holds the first part of the string (up to but not including the delimiter).

part2 is a character string variable that holds the second part of the string (after the delimiter, not including it).

char is an optional character string variable which contains an optional user supplied delimiter; if not supplied, the delimiter defaults to a semicolon (;).

Note that this routine follows the normal CI quoting rules when it encounters a backquote (') or backslash (\) in *string*. Therefore, it is possible for *char* (the delimiter) to be passed as a simple character and not be treated as a delimiter.

Normal string assignment rules apply, including blank fill. As an extension to normal string assignment rules, null strings are blank filled; *part2* can specify the same string as *string*. This permits a statement such as:

CALL SplitCommand(*buffer*,*item*,*buffer*)

which sets up *buffer* for another call after putting the current item in *item*.

SplitString, Parse String

This subroutine parses the string, stopping at the first comma or blank.

```
CALL SplitString(string,part1,part2[,commas-only])
```

where:

string	is the character string to process, with leading blanks stripped before the string is split to make the subroutine easier to use as a parameter separator.
part1	is a character string variable that holds the first part of the string (up to but not including the delimiter).
part2	is a character string variable that holds the second part of the string (after the delimiter, not including it).
commas-only	is an optional boolean variable that, if set to TRUE, directs SplitString to stop only at the first comma.

Normal string assignment rules apply, including blank fill. As an extension to normal string assignment rules, null strings are blank filled; *part2* can specify the same string as *string*. This permits a statement such as:

```
CALL SplitString(buffer,item,buffer)
```

which sets up *buffer* for another call after putting the current item in *item*.

Null characters are treated as non-blank ASCII characters, not blanks. Therefore, character strings with these routines should always be initialized to blanks before they are used. See the "Character String" section in Chapter 6 of the *RTE-6/VM CI User's Manual*, part number 92084-90036, or Chapter 8 of the *RTE-A Programmer's Reference Manual*, part number 92077-90007, for more information.

See also HpZ Miniformatter, HpZParse, and HpZDParse.

StrDsc

This routine constructs a character string descriptor for the *nchars* section of *buffer* starting with *startchar* (first character is one, not zero). This character string descriptor can then be passed to routines that expect a character string descriptor, with the effect that the character string data will be taken from or put into the designated section of *buffer*.

Refer to the *RTE Programmer's Reference Manual* for information about using this routine in Pascal to call FMP routines.

stringDes `= StrDsc(`*buffer*`,`*startchar*`,`*nchars*`)`

where:

buffer is an integer array containing characters.

startchar is a one-word integer variable, the first character to be included in the string. (The first character in an array is character number 1.)

nchars is a one-word integer variable, the length of the string, in characters.

stringDes is a double-word integer variable in which a character string descriptor is returned.

Confusion can exist about whether a subroutine requires a character descriptor (such as one returned by StrDsc) or a simple integer array. Confusion can also arise because the string descriptor references an integer buffer where the string data really resides. For these reasons, use this function with care.

For example, a valid use of this call is:

```
err = fmppurge(StrDsc(8hEXPR::XX,1,8))
```

This is equivalent to the FORTRAN call using strings:

```
err = fmppurge('EXPR::XX')
```

The buffer, string character and number of characters do not have to be constants, although they are in this example. Note that FORTRAN does not treat the descriptor as a character string in the module where the descriptor is created; it must be passed to another module to get this effect.

An incorrect use is:

```
sd = StrDsc(buf,1,80)
IF (sd .EQ. 'STOP') GOTO 10
```

because FORTRAN does not recognize "sd" as a character string.

Note also that the string descriptor returned works for FORTRAN and FMP, but other languages do not interpret it as a string descriptor.

See also MinStrDsc, HpZMoveString, and GetString.

StringCopy, Copy One String to Another

This routine copies one string into another. It is equivalent to the FTN7X statement "*string2 = string1*". This routine is used when the caller has two FTN7X string descriptors, but they are not declared as such within the program, so a normal string assignment cannot be used. For instance, routine A may pass a string to routine B, but routine B declares the passed argument as INTEGER*4 rather than as character string (the programmer may choose to do this to reduce overhead or because the string is an optional parameter). Routine B cannot do normal assignments with these variables because FTN7X does not recognize them as string descriptors. This routine may be used to copy one of these strings into the other.

```
CALL StringCopy(string1,string2)
```

where:

string1,
string2 are character string descriptors, *string1* is copied into *string2*.

Example:

```
Character string1*10, string2*10 . . .
CALL Sub(string1,string2)

Subroutine Sub(string1,string2)
Integer*4 string1, string2
 . . .
CALL StringCopy(string1,string2)
```

See also HpZMoveString.

TIMEF

This subroutine formats time, a positive double-word integer, into FORTRAN compatible character strings.

```
CALL TIMEF(etime,ttime,hrs,mins,secs,percents)
```

where:

etime is a double-word integer for elapsed time (10s of ms.).

ttime is a double-word integer (10s of ms.).

hrs is a four-character string (up to 5760 hours).

mins is a two-character string (up to 59 minutes).

secs is a five-character string (up to 59.99 seconds).

percents is a four-character string which is the percent of *ttime* that *etime* represents.

Example:

```
SUBROUTINE  PRINT(PREAMBLE,SUB,TOTAL)
CHARACTER *(*) PREAMBLE
CHARACTER *2 MIN
CHARACTER *4 HR,PE
CHARACTER *5 SEC
CHARACTER *70 OUT
INTEGER OUTI(35)
EQUIVALENCE (OUTI,OUT)
INTEGER *4 SUB,TOTAL
.
.
.
CALL  TIMEF(SUB,TOTAL,HR,MIN,SEC,PE)
OUT=PREAMBLE//HR//' HRS '//MIN//' MIN '//SEC//' SEC'//PE X//'% OF
TOTAL'
CALL  EXEC(2,1,OUTI,-70)
RETURN
END
```

PREAMBLE is the string that you have passed to the PRINT subroutine.

Result: RELOCATABLE FILE READ TIME = 0 HRS. 1 MIN. 0.04 SEC. 19.2% OF TOTAL

See the ElapsedTime or ResetTimer subroutines in Chapter 7 and the TIMEO or TIMEI subroutines in Chapter 5 of this manual.

TimeNow

This routine returns the number of seconds since 12 AM January 1, 1970 corresponding to the current system time.

seconds = TimeNow()

where:

seconds is a double-word integer variable, the number of seconds since 12 AM January 1, 1970.

Note No correction is made for time zones or Daylight Savings Time.

TrimLen, Remove Trailing Blanks

This function returns the length of a character string, not including trailing blanks.

length = TrimLen(*string*)

where:

string is the character string whose length is to be determined.

length is a one-word integer variable in which the length of the passed string after trimming off trailing blanks is returned; this length can be zero, so be careful about using this length for substrings.

Null characters are treated as non-blank ASCII characters, not blanks. Therefore, character strings with these routines should always be initialized to blanks before they are used. See the "Character String" section in Chapter 6 of the *RTE-6/VM CI User's Manual* or Chapter 8 of the *RTE-A Programmer's Reference Manual* for more information.

WhoLockedLu

This function returns the ID segment address of the program that locked the passed LU. Use IdAddToName to get the program name, if desired.

idAddress = WhoLockedLu(*lu*)

where:

lu is the LU number.

idAddress is a one-word integer variable in which the ID segment address of the program that locked the LU is returned.

Returns:

nonzero ID segment address of the program that locked the passed LU.

0 LU is not locked, or is not a legal LU.

WhoLockedRn (RTE-A Only)

This function returns the ID segment address of the program that locked the passed resource number. Use IdAddToName to get the program name, if desired.

idAddress = WhoLockedRn(*rn*)

where:

rn is the resource number.

idAddress is an integer variable in which the ID segment address of the program that locked the resource number is returned.

Returns:

nonzero ID segment of the program that locked the passed resource number.

0 Resource number is not locked locally, or it is not a legal resource number.

Refer to the *RTE-A Programmer's Reference Manual*, part number 92077-90007, for more information about resource numbers.

WriteA990Clock (RTE-A Only)

This routine sets the calendar clock on the A990 CPU.

```
if ( WriteA990Clock(TimeArray,WorkBuf,dse) ) then not A990 endif

integer*2 TimeArray(7),WorkBuf(20),dse
logical*2 WriteA990Clock
```

where:

TimeArray is an array that contains the time to set:

word 1: seconds past the minute (0..59)
word 2: minutes past the hour (0..59)
word 3: the hour of the day (0..23)
word 4: the day of the week (0..6, Sunday = 0)
word 5: the day of the month (1..28, 29, 30, 31)
word 6: the month (1..12)
word 7: the year (for example, 1991)
word 8: RTE-Date word when the time was set

WorkBuf is the buffer used for communication with the A990 clock chip.

dse indicates daylight savings time: 0 to disable and 1 to enable.

See also ReadA990Clock.

YrDoyToMonDom

This routine converts year and day of the year to day of the month, month, and day of the week.

```
call YrDoyToMonDom(Year,DayOfYr,Month,DayOfMonth,Weekday)

integer*2 Year,DayOfYr,Month,DayOfMonth,Weekday
```

where:

Year is the year A.D. (for example, 1993).

DayOfYr is the day of the year (1..366).

Month is the corresponding month (1..12).

DayOfMonth is the corresponding day (1..31).

Weekday is the day of the week (0..6, 0 = Sunday).

Given the year and the cardinal day of the year, this routine calculates the month, day of the month, and day of the week. The algorithm used does account for leap years, including century leap years.

YrDoyToRteDate

This routine converts a year and cardinal day to the format used by RTE internally; it performs the inverse of RteDateToYrDoy. The calling routine must ensure the validity of the time values, as no checking is done in the function.

RteDate = YrDoyToRteDate(*Year*,*DayOfYear*[,*OS_Flag*])

```
integer*4 RteDate,YrDoyToRteDate
integer*2 Year,DayOfYear
logical*2 OS_Flag
```

where:

RteDate	is the combined year/day in the RTE system internal format (see below).
Year	is the year A.D. (1976..2155 for RTE-A; 1970..2149 for RTE-6/VM).
DayOfYear	is the day of the year (1..366).
OS_Flag	indicates which operating system algorithm to use; TRUE indicates RTE-6/VM and FALSE indicates RTE-A. If not specified, the algorithm RTE-A is used.

The *RteDate* parameter is the data word kept by the operating system in location $TIME+2. The format of the word depends on which operating system is in use:

For RTE-A, it is the approximate number of days since January 1, 1976, calculated as follows:

$$RteDate = (Year-1976)*366 + DayOfYear - 1$$

For RTE-6/VM, it is the actual number of days since January 1, 1970, calculated as follows:

$$RteDate = (Year-1970)*365 + (Year-1969)/4 + DayOfYear + 1$$

See also GetRteTime and RteDateToYrDoy.

8

VIS Subroutines

The HP 1000 Vector Instruction Set (VIS) is a group of arithmetic routines that operates on arrays of floating point numbers. There are 80 routines that can be called from FORTRAN, half of which involve the Extended Memory Area (EMA) of the RTE Operating System. EMA is an area for arrays that can extend beyond the program's logical address space. EMA is discussed in detail in the *RTE-A Programmer's Reference Manual*, part number 92077-90007. Arrays can also be in the Virtual Memory Area (VMA). This data resides on disk and pages of data are swapped into memory as needed. All references to EMA in this manual also pertain to VMA.

Note The VIS subroutines are included as part of the product for RTE-A only. These subroutines are a separate product for RTE-6/VM and are included here for the use of the RTE-A programmer and the convenience of the RTE-6/VM user.

In this chapter, as in the rest of this manual, single precision real is the same as Real *4 and double precision real is the same as Real *8. The VIS routines do not support extended precision (Real *6) operations.

The Vector Instruction Set (VIS)

The eight VIS routines include:

- 20 single precision, non-EMA routines.
- 20 double precision, non-EMA routines.
- 20 single precision, EMA routines.
- 20 double precision, EMA routines.

VIS provides the following operations:

- The sum, difference, product or quotient of corresponding elements of two arrays.
- The sum, difference, product or quotient of a scalar and an array.
- The dot product of two arrays.
- The absolute value of an array, element by element.
- The sum of the elements of an array or the sum of their absolute values.
- The sum of an array and the product of an array, and the product of a scalar and another array; this is known as a pivot operation.
- The identification, by index, of the maximum or minimum value of an array by actual value or absolute value.
- Copying of an array into another and swapping two arrays.
- Copying arrays to and from EMA (Extended Memory Area).

Note The overflow bit will always be cleared on a normal VIS termination, regardless of whether an overflow occurred during some intermediate calculation.

In this chapter, the term "VIS" refers to the Vector Instruction Set. The terms "array," "vector," and "matrix" are used interchangeably.

Arrays in Memory

The increment parameter in the calling sequence of VIS routines specifies the next array element to be processed. Therefore, it is important to understand how arrays in FORTRAN are arranged in memory, especially two and three dimensional arrays.

In a one dimensional array, the elements are physically contiguous. Figure 8-1 shows a one dimensional array in memory. Array a is a single precision array with 10 elements. Each element occupies two words in memory. If Array a were double precision, each element would occupy four words.

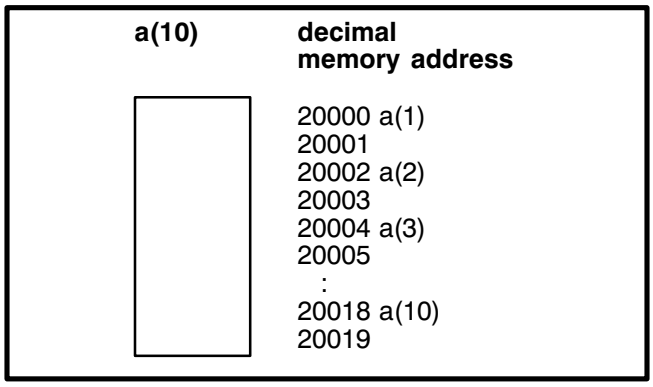


Figure 8-1. One Dimensional Array in Memory

In FORTRAN, two and three dimensional arrays, elements in a row are not physically contiguous. Elements are stored in column major order. Figures 8-2 and 8-3 show two and three dimensional arrays in memory. Array b is a 3x5 single precision array with 15 elements and Array c is a 3x5x2 array with 30 elements. Notice how the columns are stored contiguously in memory. In all arrays, an increment of 1 means to access the next contiguous array element. In two and three dimensional arrays, the spacing between elements in a row is determined by the number of rows. In Figures 8-2 and 8-3, an increment of 3 would allow every third element of each row element to be accessed. Therefore, starting at b(1,1), the next element would be b(1,2) and starting at c(1,1,1), the next element would be c(1,2,1). The increment is further described in "The Vector Instruction Set, VIS," that follows and appears in Chapter 9.

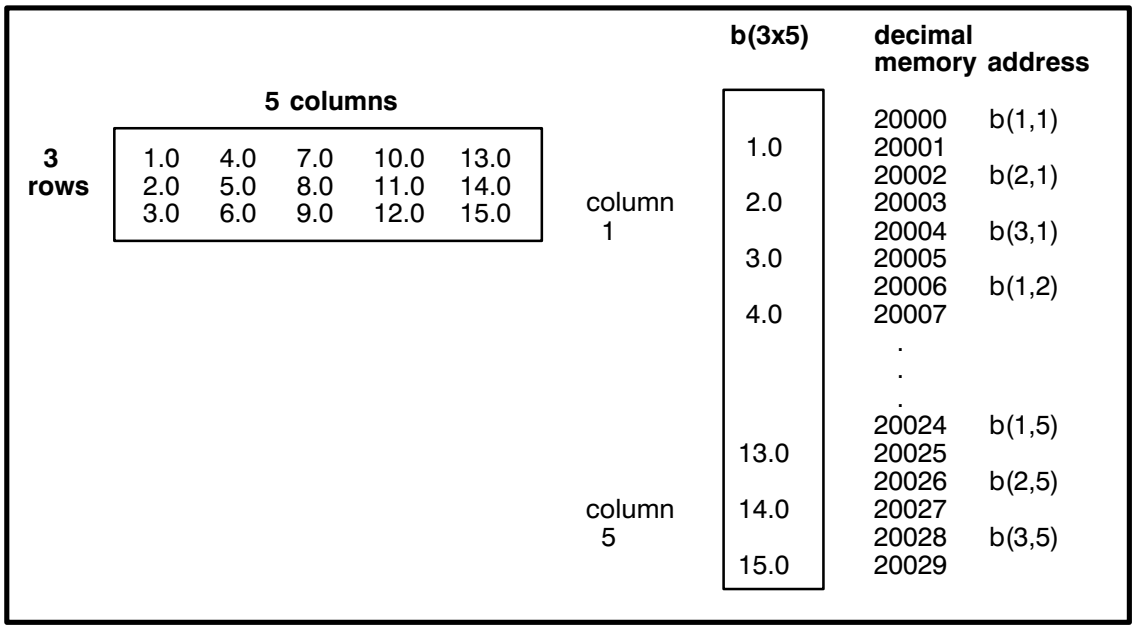


Figure 8-2. Two Dimensional Array in Memory

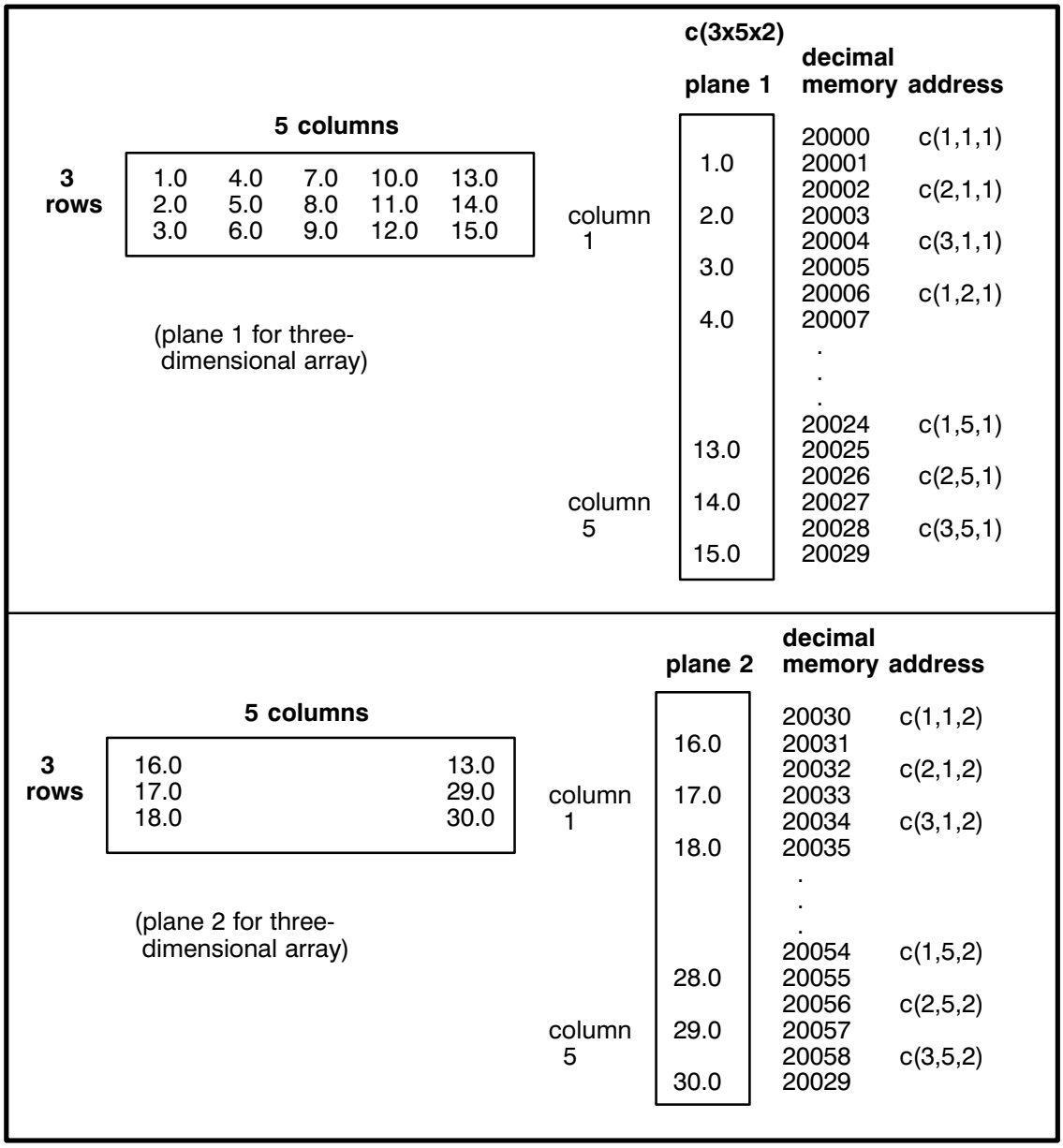


Figure 8-3. Three Dimensional Arrays in Memory

Index to VIS Routines

INDEX TO VIS ROUTINES

Fortran Calling Sequences	Operations	Page
CALL VABS (v1, incr1, v2, incr2, #elements)	v2 ← abs(v1)	13
CALL VADD (v1, incr1, v2, incr2, v3, incr3, #elements)	v3 ← v1 + v2	7
CALL VDIV (v1, incr1, v2,incr2, v3, incr3, #elements)	v3 ← v1 / v2	7
CALL VDOT (scalar, v1, incr1, v2, incr2, #elements)	scalar ← sum [v1 * v2	19
CALL VMAB (scalar, v1, incr1, #elements)	scalar = index of largest absolute value in v1	23
CALL VMAX (scalar, v1, incr1, #elements)	scalar = index of largest value in v1	23
CALL VMIB (scalar, v1, incr1, #elements)	scalar = index of smallest value in v1	23
CALL VMIN (scalar, v1, incr1, #elements)	scalar = index of smallest value in v1	23
CALL VMOV (v1, incr1, v2, incr2, #elements)	v2 ← v1	29
CALL VMPY (v1, incr1, v2, incr2, v3, incr3 #elements)	v3 ← v1 * v2	7
CALL VNRM (scalar, v1, incr1, #elements)	scalar ← sum [abs(v1)]	15
CALL VPIV (scalar, v1, incr1, v2, incr2, v3, incr3, #elements)	v3 ← (scalar * v1) + v2	21
CALL VSAD (scalar, v1, incr1, v2, incr2, #elements)	v2 ← scalar + v1	11
CALL VSDV (scalar, v1, incr1, v2, incr2, #elements)	v2 ← scalar / v1	11
CALL VSMY (scalar, v1, incr1, v2, incr2, #elements)	v2 ← scalar * v1	11
CALL VSSB (scalar, v1, incr1, v2, incr2, #elements)	v2 ← scalar − v1	11
CALL VSUB (v1, incr1, v2, incr2, v3, incr3, #elements)	v3 ← v1 − v2	7
CALL VSUM (scalar, v1, incr1, #elements)	scalar ← sum [v1]	15
CALL VSWP (v1, incr1, v2, incr2, #elements)	v1 ↔ v2	29
CALL VWMOV (v1, incr1, v2, incr2, #elements)	non-EMA v1 → EMA v2	31
CALL WVMOV (v1, incr1, v2, incr2, #elements)	EMA v1 → non-EMA v2	31

General Calling Sequence

The rest of this chapter describes each group of VIS routines. Examples given are all for single precision arrays and variables. The previous index of all the single precision VIS routines can be used as a quick reference guide.

Conventions used for the calling sequence description are:

Notation	Description
[ ]	Items within brackets are optional.
{ }	For items stacked within braces, choose one.
lowercase parameters	Lowercase parameters are to be replaced by user supplied variables.

VIS routines are callable from FORTRAN or assembly language. The general form of the FORTRAN calling sequence is:

```
CALL {subr} ([scalar,]v1,incr1,[v2,incr2,[v3,incr3,]]NumElements)
```

where:

subr — is the actual VIS routine name. The initial letters of the names indicate the type of the routine. The conventions are:

V − Single precision, non-EMA routines.
DV − Double precision, non-EMA routines.

DW − Double precision, EMA routines.
W − Single precision, EMA routines.

Default numeric type must be four-word double precision.

scalar — is the operand or result. Precision must match the associated arrays. For the MAX/MIN routines, not every VIS routine requires a scalar parameter. If the scalar is an operand, it is not modified.

v1, v2, v3 — are the starting array element(s), defining where to start processing the arrays. Arrays can be of one or more dimensions. All arrays must be of the same precision and must all be either in EMA or non-EMA (except for the EMA/non-EMA move routines). The same arrays can be specified for both operands and results.

incr1, incr2 — are integer value(s) indicating the next element(s) in the array(s) to be processed. If 0, only the first is processed.
$-32767 \leq \text{incr\#} \leq +32767$

(The increment parameter is discussed later.)

NumElements — is the integer value indicating the number of elements to process. If less than or equal to 0, no operation occurs, but the calling sequence must be valid.

NumElements $\leq +32767$

The array parameters (v1, v2, and v3) must all be either in EMA or non-EMA and must be of the same precision. The appropriate VIS routine should be used accordingly. All the other parameters (scalar, increment, and NumElements) should never be EMA "call by reference" variables. However, they can be "call by value" variables, except when used as a result parameter. Refer to "Extended Memory Area (EMA) Considerations" in Chapter 9 and to the *FORTRAN 77 Reference Manual*, part number 92836-90001, for more detailed information on EMA call by reference and call by value.

Note

If a starting array element (v1, v2, or v3) is the first element of the array, it can be specified with or without the unity subscripts. For example, a is the same as a(1,1) or b is the same as b(1).

The FORTRAN compiler does not verify that the array subscript values fall within declared DIMENSION bounds. Unpredictable results occur if the array subscripts are negative or greater than the declared array size.

When calling VIS firmware routines from CDS code, local arrays passed to the VIS routines cannot exceed the stack frame address space of 1018 words. The other six words are used for stack frame control variables. You must get these arrays out of the limited address space of the stack frame. In Pascal, this means the arrays should be in global space, while in FORTRAN, either labeled common or the SAVE statement will work. Also in FORTRAN, if the local array is completely above the 1K address limits of the stack frame, the language will compute and pass the data relative address to the VIS instructions, and they will work correctly. To do this, equivalence the data arrays in such a way that they are preceded by 1024 words. For example:

```
integer fill(1024)
real data(2000)
equivalence (fill(1024, data(1)))
```

The fill array may be used for other functions as long as it is not passed into a VIS routine.

The increment parameters, incr1, incr2, and incr3, indicate the next array elements to be processed. An increment of 1 accesses every element, an increment of 2 accesses every other element, and so on. For FORTRAN arrays, elements in a column are physically contiguous in memory. Therefore, an increment of 1 is used to access each column element. For arrays greater than one dimension, row elements are not physically contiguous. The increment must be greater than 1 to access row elements. For example, a three-by-five matrix has three rows and each row element is three elements apart. The increment would be 3. Figure 8-4 shows how row elements are accessed. Refer to "Introduction" above for a description of how arrays are arranged in memory.

Example

This example shows the use of the increment parameter to calculate the sum of the elements in row 1.

```
CALL VSUM (sum, matrix(1,1), 3, 5)
```

where:

scalar	=	sum, which will be the sum of row 1.
v1	=	matrix(1,1), which is the starting element.
incr1	=	3, to access row elements that are spaced three elements apart in memory.
NumElements	=	5, to access the five elements in one row.

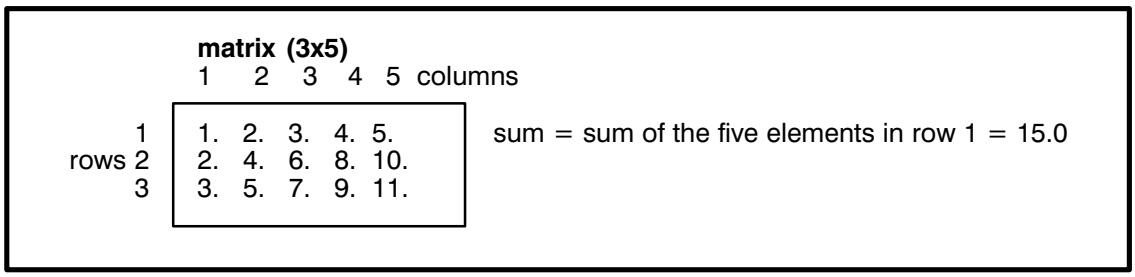


Figure 8-4. Accessing Row Elements

If a v1, v2, or v3 parameter is the very first element of an array and is specified with a negative increment, the VIS processes elements backward in memory and out of the array bounds. Therefore, with negative increments you must be careful to specify the starting array elements well within the bounds of the arrays.

The following descriptions of VIS routines give brief examples to explain the routines themselves and show incr1, incr2, and incr3 in simple uses. Refer to Chapter 9, "Using VIS in Your Programs," to see examples of increments other than 1.

Vector Arithmetic Routines

The vector arithmetic routines add, subtract, multiply, or divide the specified elements of two arrays and place results into elements of a third array.

```
CALL { VADD }
     { VSUB }  (v1,incr1,v2,incr2,v3,incr3,NumElements)
     { VMPY }
     { VDIV }
```

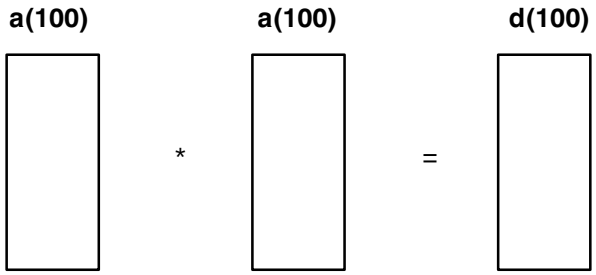
Operation	Single Precision	Double Precision	EMA Single Precision	EMA Double Precision
v3 ← v1 + v2	VADD	DVADD	WADD	DWADD
v3 ← v1 − v2	VSUB	DVSUB	WSUB	DWSUB
v3 ← v1 * v2	VMPY	DVMPY	WMPY	DWMPY
v3 ← v1 / v2	VDIV	DVDIV	WDIV	DWDIV

Example

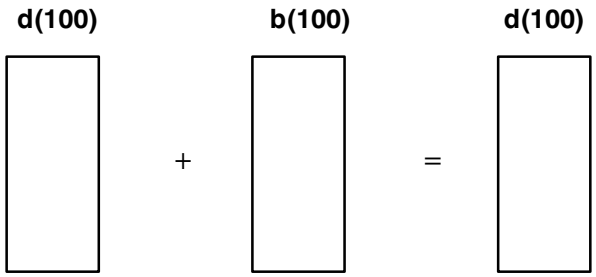
Problem: Calculate d = (a*a+b)/c for all elements in arrays a, b, c, and d.

Arrays a, b, c, and d are all one dimensional arrays with 100 elements each. Use array d to store intermediate and final results.

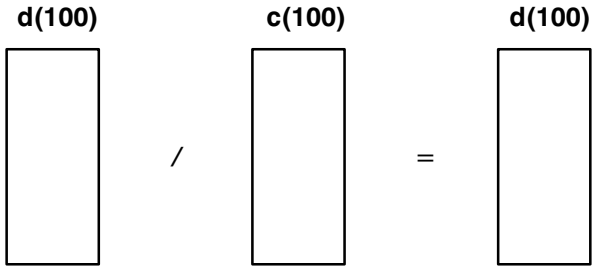
For d = a*a, every element of array a is multiplied by itself and the results put into array d.



For d = d+b, every element of array d is added to the corresponding element in array b and the results put into array d.



For d = d/c, every element of array d is divided by the corresponding element in array c and the results put into array d.




```
C      Calculate d = (a*a+b)/c for all elements.
       REAL*4 a, b, c, d
       DIMENSION a(100), b(100), c(100), d(100)
C
C      FORTRAN DO loop without VIS:
       DO 10 i = 1,100
            d(i) = a(i) * a(i)
            d(i) = d(i) + b(i)
            d(i) = d(i) / c(i)
10     CONTINUE
C      With VIS:
C      d = a*a Multiply each element in a by itself
       CALL VMPY (a,1,a,1,d,1,100)
C      d = d+b  Add each element of b with corresponding element in d

       CALL VADD (d,1,b,1,d,1,100)
C      d = d/c  Divide each element of d with
C               corresponding element in c
       CALL VDIV (d,1,c,1,d,1,100)
       END
```

Arrays a, b, and c are not modified; d always contains the results. Each operation involves the corresponding elements among the arrays. Therefore, results from computations with a(1), b(1), and c(1) are put into d(1), results from a(2), b(2), and c(2) are put into d(2), and so on. Incr1, incr2, and incr3 are all 1, and NumElements is 100 to obtain all elements in all arrays.

Scalar-Vector Arithmetic Routines

The scalar-vector arithmetic routines add, subtract, multiply, or divide a constant with specified elements of an array and place results into elements of a second array.

```
CALL { VSAD }
     { VSSB } (scalar,v1,incr1,v2,incr2,NumElements)
     { VSMY }
     { VSDV }
```

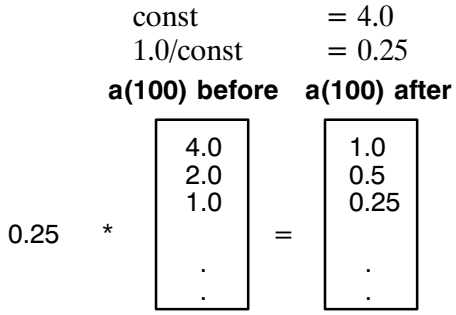
Operation	Single Precision	Double Precision	EMA Single Precision	EMA Double Precision
v2 ← scalar + v1	VSAD	DVSAD	WSAD	DWSAD
v2 ← scalar − v1	VSSB	DVSSB	WSSB	DWSSB
v2 ← scalar * v1	VSMY	DVSMY	WSMY	DWSMY
v2 ← scalar / v1	VSDV	DVSDV	WSDV	DWSDV

Note that the operations are in scalar-vector rather than vector-scalar order. Scalar-vector order means that the scalar is the first operand in the arithmetic operation. This affects the operations of subtraction and division. For example, to perform vector-scalar arithmetic, call VSAD with a negative scalar (v1 − scalar); similarly, call VSMY with the reciprocal (v1 / scalar).

Example

Problem: Compute a = a/const for all elements a.

Array a is a one-dimensional array with 100 elements. Each element in a is divided by const by actually multiplying with the reciprocal of const.




```
C      Compute a(i) = a(i)/const
       REAL*4 a
       DIMENSION a(100)
       const = 4.0
C
C      FORTRAN DO loop without VIS:
       DO 10 i = 1,100
10       a(i) = a(i)/const
C
C      With VIS:
C      Divide and replace each element of a
       CALL VSMY (1.0/const,a,1,a,1,100)
       END
```

incr1 and incr2 are both 1 and NumElements is 100 to access all elements in a.

Absolute Value Routine

The absolute value routine places the absolute values of specified array elements into a second array.

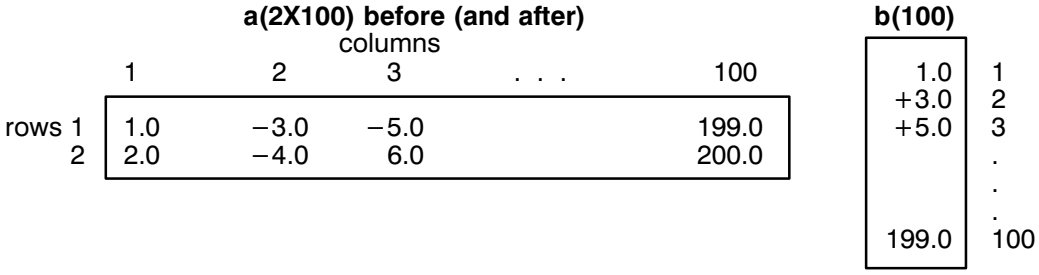
```
CALL VABS (v1,incr1,v2,incr2,NumElements)
```

Operation	Single Precision	Double Precision	EMA Single Precision	EMA Double Precision
v2 ← abs (v1)	VABS	DVABS	WABS	DWABS

Example

Problem: Find absolute values of row 1 of a and place into b.

Array a is 2 x 100 matrix and b is a one dimensional array of 100 elements. Take the absolute value of a's row 1 and place them into corresponding elements of array b.




```
C     Put absolute values of row a of a into b.
      REAL*4 a, b
      DIMENSION a(2,100), b(100)
C
C     FORTRAN DO loop without VIS:
      DO 10 j = 1,100
10      b(j) = abs(a(1,j))
C
C     With VIS:
      CALL VABS (a,2,b,1,100)
      END
```

Notice that incr1 is 2, which is the spacing between row elements for array a. The NumElements parameter is 100, which is the row length for a and the size of b. Array a is not modified.

Sum Routines

The sum routines calculate the sum or the sum of the absolute values of specified array elements. The result is stored into a scalar variable. VSUM calculates the sum, and VNRM is defined to be the sum of the absolute values.

```
CALL  { VSUM }   (scalar,v1,incr1,NumElements)
      { VNRM }
```

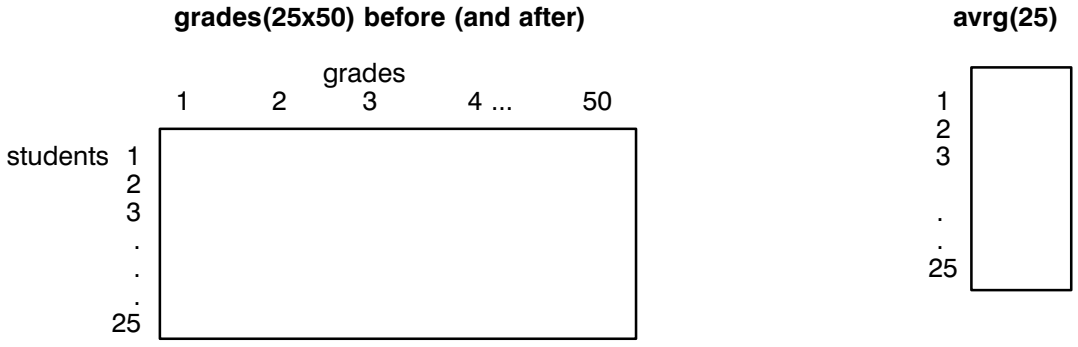
Operation	Single Precision	Double Precision	EMA Single Precision	EMA Double Precision
scalar ← sum {v1}	VSUM	DVSUM	WSUM	DWSUM
scalar ← sum {abs(v1)}	VNRM	DVNRM	WNRM	DWNRM

Note VNRM computes the "L1" norm.

Example

Problem: Compute the average of student grades.

Two arrays are needed: grades is a 25 x 50 matrix of students and grades, and avrg is a one-dimensional array of the average grades.



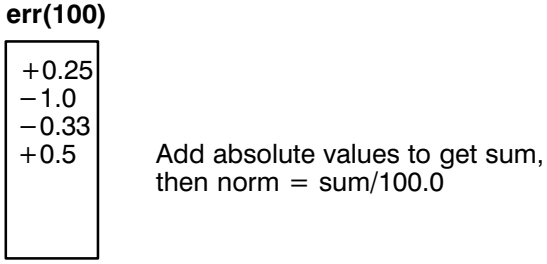
```
C     Compute the average.
C     grades = student grades
      REAL*4 grades, avrg
      DIMENSION grades(25,50), avrg(25)
C
C     FORTRAN DO loop without VIS:
      DO 10 istud = 1,25
              avrg(istud) = 0.0
              DO 20 j = 1,50
20                    avrg(istud) = avrg(istud) + grades(istud,j)
              avrg(istud) = avrg(istud)/50.0
10    CONTINUE
C
C     With VIS:
      DO 10 istud = 1,25
              CALL VSUM (avrg(istud),grades(istud,1),25,50)
              avrg(istud) = avrg(istud)/50.0
10    CONTINUE
      END
```

The average of each row is calculated for each student. The incr1 parameter is 25 to obtain each row element in grades, because row elements are 25 elements apart. The NumElements is 50 to process one row at a time. Elements in array grades are not modified.

Example

Problem: Compute the average of error values.

An array of error values, err, contains positive and negative values. Therefore, the VNRM routine should be used so that the absolute values are added. Otherwise, the positive and negative values would cancel each other out when added.



```
C      Calculate the average of errors.
C      error contains + or – errors.
       REAL*4 err
       DIMENSION err(100)
       REAL norm
C
C      FORTRAN DO loop without VIS:
       sum = 0.0
       DO 10 i = 1,100
10        sum = sum ABS(err(i)
C
C      With VIS:
       CALL VNRM (sum,err,1,100)
       norm = sum/100.0
       END
```

The incr1 parameter is 1 and NumElements is 100 to obtain all elements in err; sum contains the sum of the absolute values; err is not modified.

Dot Product Routine

The dot product routine performs the dot product operation between two arrays. Elements of two arrays are multiplied and the products are added together to give a scalar result. The operands are not modified.

```
CALL  VDOT  (scalar,v1,incr1,v2,incr2,NumElements)
```

Operation		Single Precision	Double Precision	EMA Single Precision	EMA Double Precision
scalar ←	sum {v1*v2}	VDOT	DVDOT	WDOT	DWDOT

Note VDOT of a vector with itself computes the square of the "L2" norm or the Euclidean norm.

Example

Problem: Compute the dot product of an array with itself.

Let a be a (5,25) matrix, and dot a scalar where the dot product is stored. First the dot product is computed for row 1 of a with itself by multiplying each element in row 1 by itself. These products are added together and the sum stored in dot. The equation for dot is:

dot = [a(1,1)*a(1,1)] + [a(1,2)*a(1,2)] + ... + [a(1,25)*a(1,25)]

a(5x25) before (and after)

		columns				
		1	2	3	. . .	25
rows	1	4.0	4.0	−3.0		3.0
	2	2.0	3.0	−1.0		5.0
	3					
	4					
	5					

```
C      Compute the dot product of row 1 with itself.
C      dot = dot product
       REAL*4 a
       DIMENSION a(5,25)
C
C      FORTRAN DO loop without VIS:
       i = 1
       dot = 0.0
       DO 10 j = 1,25
10        dot = dot + a(i,j) * a(i,j)
C
C      With VIS:
       CALL VDOT (dot,a,5,a,5,25)
       END
```

The NumElements parameter is 25, which is the size of one row; incr1 and incr2 parameters are 5, which is the spacing between elements in a row. This allows access to all of row 1. Elements in array a are not modified.

Pivot Routine

The pivot routine performs the pivot operation, in which specified array elements are multiplied by a constant and then added to elements of the second array. The corresponding results are placed into the third array. If operands only, the first and second arrays are not modified.

```
CALL VPIV (scalar,v1,incr1,v2,incr2,v3,incr3,NumElements)
```

Operation		Single Precision	Double Precision	EMA Single Precision	EMA Double Precision
v3 ←	(scalar*v1)+v2	VPIV	DVPIV	WPIV	DWPIV

Note Although it has other applications, VPIV was designed for row manipulation in matrix reduction.

Example

Problem: Perform pivot operation on two rows.

Let a be a (3,50) matrix and s a scalar. The pivot is performed with rows 1 and 2. The scalar, s, is a negative value multiplied with each element in row 1. These products are added to each element in row 2, and these results are placed back into row 2.

$s = -(a(2,1)/a(1,1)) = -2.0/4.0 = -0.5$

a(3x50) before

	columns	1	2	3	. . .	50
rows	1	4.0	4.0	−3.0		3.0
	2	2.0	3.0	−1.0		5.0
	3					

multiply each row element by −0.5 and add to corresponding elements in row 2.

a(3x50) before

	columns	1	2	3	. . .	50
rows	1	4.0	4.0	−3.0		3.0
	2	0.0	1.0	0.5		3.5
	3					

row 1 remains unmodified, row 2 is replaced, and a(2,1) = 0.0

```
C      Calculate a(2,j) = s * a(1,j) + a(2,j)
       REAL*4 a
       DIMENSION a(3,50)
C
C      s is the multiplier for the first row
       s = -(a(2,1)/a(1,1))
C      Perform pivot to replace row 2
C
C      FORTRAN DO loop without VIS:
       DO 10 j = 1,50
10       a(2,j) = s * a(1,j) + a(2,j)
C
C      With VIS:
       CALL VPIV (s, a(1,1), 3, a(2,1), 3, a(2,1), 3, 50)

       END
```

The incr1, incr2, and incr3 parameters are all 3, and the NumElements parameter is 50, which allows access to one row at a time. Row 1 is not modified and row 2 is replaced.

MAX/MIN Routines

The MAX/MIN routines solve for the index of the largest/smallest value in an array. Routines VMAX and VMIN give indexes for the largest and smallest values. Routines VMAB and VMIB give indexes for the largest and smallest absolute values.

```
CALL { VMAX } (scalar,v1,incr1,NumElements)
     { VMIN }
     { VMAB }
     { VMIB }
```

Operation	Single Precision	Double Precision	EMA Single Precision	EMA Double Precision
scalar = index of largest element in v1	VMAX	DVMAX	WMAX	DWMAX
scalar = index of smallest element in v1	VMIN	DVMIN	WMIN	DWMIN
scalar = index of largest absolute value in v1	VMAB	DVMAB	WMAB	DWMAB
scalar = index of smallest absolute value in v1	VMIB	DVMIB	WMIB	DWMIB

Example

Problem: Find the largest and smallest errors.

Let err be an array with 10 error values. Errors can be positive or negative. Search for the largest and smallest values.

err(10)

+0.25
−1.0
−0.3
+0.5
+0.3
−0.3
−0.75
+0.4
+0.1
−0.2

err(2) = smallest error

err(4) = largest error

```
C      Find largest and smallest errors.
C      errmax = largest error
C      errmin = smallest error
       REAL*4 err
       DIMENSION err(10)
C
C      FORTRAN DO loop without VIS:
       errmax = err(1)
       errmin = err(1)
       DO 10 i = 1,10
            IF (errmax .LE.  err(i)) errmax = err(i)
            IF (errmin .GE.  err(i)) errmin = err(i)
10     CONTINUE
C
C      With VIS:
       CALL vmax(imax, err, 1, 10)
       errmax = err(imax)
C
       CALL vmin(imin, err, 1, 10)
       errmin = err(imin)
       END
```

Comments

The indices returned are based only on the elements that are examined, as requested by the starting array element, the increment, and number of elements. Therefore, there are special cases concerning the MAX/MIN routines:

1. v1 is the first array element and incr1 is not 1.
2. v1 is not the first array element and incr1 = 1.
3. v1 is not the first array element and incr1 is not 1.
4. multidimensional arrays − scanning rows.

Case 1 (v1 is first array element and incr1 is not 1)

When incr1 is greater than 1, the actual index of the array element is:

```
index = 1 + incr1 * (ipos1 - 1)
```

where:

ipos1 is the scalar index returned from calling a MAX/MIN routine.

Example: Call VMAX (ipos1, err(1), 2, 5)

In array err above, the largest element is actually 0.5 in err(4). However, with incr1 = 2, only every other element is examined. Therefore the instruction returns ipos1 = 5, putting these values into the equation:

incr1 = 2
ipos1 = 5 returned from vmax
index = 1 + 2 * (5 − 1) = 9

err(9) = 0.4 largest value of every other element

Case 2 (v1 is not first array element and incr1 = 1)

When v1 is not exactly the first array element, the index returned is relative to that specified v1. To calculate the exact position (index) of the maximum/minimum variable, the equation is:

```
index = ipos2 + istart - 1
```

where:

ipos2 is the scalar index returned.

istart is the array subscript of v1.

Example: Call VMAX (ipos2, err(6), 1, 5)

Starting with err(6) and scanning the next five elements, the largest value is in err(9). However, ipos2 = 4 because err(9) is the fourth element from err(6). To find the actual scalar index, the calculations are:

istart = 6
ipos2 = 4, returned from VMAX
index = 4 + 6 − 1 = 9
err(9) = 0.4, largest value of last 5 elements

Case 3 (v1 is not first array element and incr1 is not 1)

Combine the two equations above to calculate the exact position (index) of the maximum/minimum variable.

ipos2 = 1 + incr1 * (ipos1 − 1)
index = ipos2 + istart − 1

Starting with the second element, scan every other element for the largest value. ipos1 = 2 because the largest value is the next element from err(2). To find the actual index, the calculations are:

ipos2 = 4, returned from VMAX
incr1 = 2
ipos2 = 1 + [2 * (2 −1)] = 3
istart = 2
index = ipos2 + istart − 1
index = 3 + 2 − 1 = 4
err(4) = 0.5, largest value of every other element
starting with err(2)

Case 4 (multidimensional arrays − scanning rows)

With two and three dimensional arrays, any number of columns can be scanned. However, only one row at a time can be scanned per VIS call. Starting with the first element of a row, the index returned is the column number.

Example: Call VMAX (index, err(3,1), 4, 5)

index = 5, column number of largest value in row 3
v1 = err(3,1), starting element in row 3
incr1 = 4, number of rows in err
NumElements = 5, number of elements in a row
err(3,index) = 13.0, maximum value in row 3

err(4x5)

5 columns

4 rows

10.0	20.0	30.0	40.0	50.0
5.0	6.0	7.0	99.0	8.0
9.0	10.0	11.0	12.0	13.0
16.0	15.0	14.0	13.0	95.0

err(3,5) = maximum value in row 3

Move Routines

The move routines copy or exchange elements between two arrays. Routine VMOV copies an array into another array. VSWP exchanges specified elements between two arrays.

```
CALL { VMOV } (v1,incr1,v2,incr2,NumElements)
     { VSWP }
```

Operation			Single Precision	Double Precision	EMA Single Precision	EMA Double Precision
v2	←	v1	VMOV	DVMOV	WMOV	DWMOV
v1	↔	v2	VSWP	DVSWP	WSWP	DWSWP

Example

Problem: Exchange elements between one row and another array.

Let a be a 5x25 matrix and b an array of 25 elements. Exchange row 1 of a with every element in b.

a(5x25) before

	columns 1	2	3	. . .	25
1	4.0	4.0	−3.0		3.0
rows 2					
3					
4					
5					

b(25) before

2.0
3.0
−1.0
5.0

array(5x25) before

	columns 1	2	3	. . .	25
1	2.0	3.0	−1.0		5.0
rows 2					
3					
4					
5					

b(25) before

4.0
4.0
−3.0
3.0

```
C      Exchange row 1 of a with every element in b.
       REAL*4 a, b
       DIMENSION a(5,25), b(25)
C
C      FORTRAN DO loop without VIS:
       DO 10 j = 1,25
            temp = a(1,j)
            a(1,j) = b(j)
            b(j)   = temp
10     CONTINUE
C
C      With VIS:
       CALL VSWP (a(1,1), 5, b, 1, 25)
C
       END
```

Incr1 is 5 to access elements in row 1. NumElements is 25, which is the size of one row of a or the length of b. Incr2 is 1, to obtain each element in b.

EMA (Extended Memory Area)/ Non-EMA Move Routines

The EMA/non-EMA move routines allow arrays to be copied to and from EMA and non-EMA areas. With all other VIS routines, arrays must be entirely in EMA or non-EMA. WVMOV copies an EMA array into a non-EMA array. VWMOV copies a non-EMA array into an EMA array.

```
CALL  { VWMOV }  (v1,incr1,v2,incr2,NumElements)
      { WVMOV }
```

Operation	Single Precision	Double Precision
v2 in EMA ← v1 in non-EMA	VWMOV	DVWMV
v2 in non-EMA ← v1 in EMA	WVMOV	DWVMV

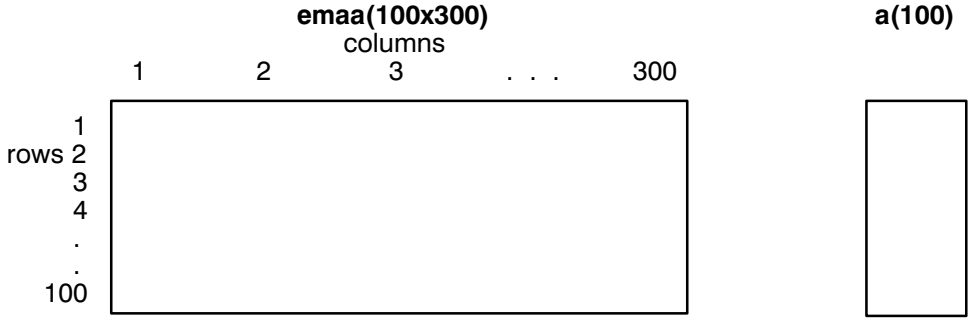
Comments

Only v1 or v2 can be in EMA. All the other parameters, incr1, incr2, and NumElements, cannot be in EMA. Further discussion on using EMA is in the section "Extended Memory Area (EMA) Considerations," in Chapter 9.

Example

Problem: Move part of an EMA array into non-EMA area.

Let emaa be a 100x300 matrix in EMA, and a be a one-dimensional array of 100 elements in non-EMA. Move one column from emaa into a.



```
$EMA(xyz,0)
      PROGRAM example
      COMMON /xyz/emaa(100,300)
      REAL*4 a
      DIMENSION a(100)
C     Move column 1 from emaa into a.
C
C     FORTRAN DO loop without VIS:
      DO 10 i = 1,100
10      a(j) = emaa(1,j)
C
C     With VIS:
      CALL WVMOV (emaa(1,1), 1, a, 1, 100)
      END
```

incr1 and incr2 are 1 and NumElements is 100 to copy column 1 of emaa into array a. Column 1 is not modified.

Example

Problem: Initialize an EMA array.

Initialize matrix emaa to all zeros.

```
C
C     FORTRAN DO loop without VIS:
C     DO 10 i = 1,100
C        DO 10 j = 1,300
C 10      emaa(i,j) = 0.0
C
C     With VIS:
      CALL VWMOV (0.0, 0, emaa, 1, 100*300)
      END
```

Incr1 is 0 to cause the value 0.0 to be repeatedly accessed and put into emaa.

9

Using VIS in Programs

This chapter describes using VIS subroutines in your programs, with examples from FORTRAN.

Converting FORTRAN DO Loops

VIS can replace FORTRAN DO loops performing repetitive operations on arrays. Chapter 8 already provided some simple examples of VIS routines.

Steps involved in converting to VIS are:

1. Write your program as you would normally with FORTRAN DO loops. Determine that the program is operating correctly.
2. Identify the DO loop(s) that need to be sped up. Find the VIS routine that is functionally equivalent.
3. Comment the program code that includes the FORTRAN DO loops to annotate calls to VIS routines.
4. Insert call(s) to VIS using the following criteria:
   a. The starting array elements (v1, v2, v3) are the same as the first values of the DO loop.
   b. Calculate the increments (incr1, incr2, incr3).
   c. Calculate the number of elements (NumElements).

One Dimensional Array Examples

The following examples show how FORTRAN DO loops using one dimensional arrays are converted to VIS.

Example: Adding two vectors in double precision.

```
C     FORTRAN DO loop:
C     Add two vectors in double precision
C
      DOUBLE PRECISION a(100), b(100), d(100)
C
      DO 10 i = 1,100
10      d(i) = a(i) + b(i)
C
C     VIS equivalent:
      CALL DVADD (a,1,b,1,d,1,100)
      END
```

The parameters for DVADD are:

```
v1 = a  incr1 = 1  NumElements = (100-1+1)/1 = 100
v2 = b  incr2 = 1
v3 = d  incr3 = 1
```

In general, for one dimensional arrays and one FORTRAN DO loop, such as

```
DO 10 i = istart, iend, incr
  .
  .
  .
10 CONTINUE
```

the NumElements parameter as shown above is calculated to be:

```
NumElements = (iend - istart + incr)/incr
```

where:

/ indicates an integer divide.

If incr is not specified, FORTRAN defaults to 1.

Example: Sum of every other element in an array.

```
C     FORTRAN DO loop:
C     Find the sum of every other element in array a
      DIMENSION array (100)
C
      sum = 0.0  ──────────►
      DO 10 i = 1,100,2
10      sum = sum + a(i)
C
C     VIS equivalent:
      CALL VSUM (sum,a,2,50)
      END
```

Note that VSUM automatically initializes sum to 0.0 so that this statement is not *required*.

The parameters for VSUM are:

```
scalar = sum
v1     = a
incr1  = 2
NumElements = (100–1+2)/2 = 50
```

Example: Find largest value in an array.

```
C     FORTRAN DO loop:
C     Find largest value in array a
      REAL mxm
      DIMENSION a (100)
C
      mxm = a(1)
      imxm = 1
      DO 10 i = 1,100
          IF (a(i) .LE.  mxm)  GOTO 10
          mxm = a(i)
          imxm = i
10    CONTINUE
C
C     VIS equivalent:
      CALL VMAX (imxm,a,1,100)
      mxm = a(imxm)
      END
```

The parameters for VMAX are:

```
scalar    = imxm
v1        = a
incr1     = 1
NumElements = (100–1+1)/1 = 100
```

Two Dimensional Array Examples

The following examples show how FORTRAN DO loops using two dimensional arrays are converted to VIS.

Example: Adding two matrixes in double precision.

```
C     FORTRAN DO loop:
C     Add two 10 x 10 matrixes in double precision
C
      DOUBLE PRECISION a(10,10), b(10,10), d(10,10)
C
      DO 10 j = 1,10
          DO 10 i = 1,10
10      d(i,j) = a(i,j) + b(i,j)
C
C     VIS equivalent:
      CALL DVADD (a(1,1),1,b(1,1),1,d(1,1),1,10 * 10)
C     or
      CALL DVADD (a, 1, b, 1, d, 1, 10 * 10)
       .
       .
       .
      END
```

The parameters for DVADD are:

```
v1 = a  incr1 = 1   NumElements = [(10–1+1)/1]*[(10–1+1)/1] = 100
v2 = b  incr2 = 1
v3 = d  incr3 = 1
```

For multidimensional arrays and one FORTRAN DO loop, the NumElements parameter is calculated as for one dimensional arrays. For more than one nested FORTRAN DO loop, such as:

```
      DO 10 i = istart#1, iend#1, incr#1
      DO 20 j = istart#2, iend#2, incr#2
       .
       .
       .
20    CONTINUE
       .
       .
       .
10    CONTINUE
```

The NumElements parameter is the product of the number of elements for each loop.

```
NumElements = [(iend#1-istart#1+1)/incr#1] * [(iend#2-istart#2+1)/incr#2]
```

Example: Exchange two rows in a matrix.

```
C     FORTRAN DO loop:
C     Exchange rows
C
      DIMENSION a(5,100)
C
C     Exchange row 1 of a with row 3 of a
      DO 10 icol = 1,100
          temp = a(1,icol)
          a(1,icol) = a(3,icol)
          a(3,icol) = temp
10    CONTINUE
C
C     VIS equivalent:
      CALL VSWP (a(1,1),5,a(3,1),5,100)
      END
```

The parameters for VSWP are:

```
v1 = a(1,1)  incr1 = 5
v2 = a(3,1)  incr2 = 5

NumElements = 100–1+1/1
```

Nested DO Loops Example

To enhance program execution speed, it is not necessary to convert every DO loop. Converting the innermost DO loop is normally sufficient to decrease execution time. If the number of operations in the innermost loop is reasonably large, converting that DO loop should produce 90% of the speed enhancement.

This example multiplies two matrixes, a and b, by calculating dot products of the rows and columns and forming matrix c:

Example: Dot product of two matrixes.

```
C     FORTRAN nested DO loops
      DIMENSION a (25,10), b(10,20), c(25,20)
C
C     FORTRAN DO loops:
      DO 20 i = 1,25
         DO 20 j = 1,20
             c(i,j) = 0.0
             DO 30 k = 1,10
                 c(i,j) = c(i,j) + a(i,k) * b(k,j)
30       CONTINUE
20    CONTINUE
C
C
C     VIS equivalent:
      DO 20 i = 1,25
         DO 20 j = 1,20
C
C        Replace DO
         CALL VDOT (c(i,j), a(i,1), 25, b(1,j), 1, 10)
C
20    CONTINUE
```

In the VIS call, the subscript k is replaced by its initial value, 1. The subscripts i and j remain constant within the inner loop.

Combinations of Vector Instructions

Sometimes an expression exists within a DO loop that does not correspond to just one standard vector instruction. However, the expression can often be rearranged to yield a combination of VIS operations. The example that follows shows two vectors being multiplied by scalars. Those results are then added together to form another vector.

Example: Evaluate x(i) = s*v1(i) + t*v2(i).

```
      DO 10 i = 1,n
          x(i) = s*v1(i) + t*v2(i)
10 CONTINUE
```

can be rewritten as:

```
      DO 10 i = 1,n
          x(i) = s*v1(i)
          x(i) = x(i) + t*v2(i)
10 CONTINUE
```

The single loop can be broken into two separate loops:

```
      DO 10 i = 1,n
          x(i) = s*v1(i)
10 CONTINUE

      DO 20 i = 1,n
          x(i) = x(i) + t*v2(i)
20 CONTINUE
```

These loops can now be replaced by two vector instructions:

```
      CALL VSMY (s,v1,1,x,1,n)
      CALL VPIV (t,v2,1,x,1,x,1,n)
```

Increment Parameters Other Than One

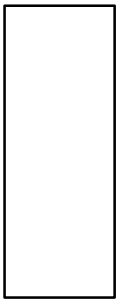
The incr1, incr2, or incr3 parameter of the calling sequence determines the next array elements to be used. An increment of 1 accesses every element in an array, an increment of 2 accesses every other element, and so on.

For two and three dimensional arrays in FORTRAN, elements are stored in column order. Refer to Chapter 8 for an explanation of arrays in memory. The spacing between elements in a row is determined by the number of rows as declared in the FORTRAN DIMENSION statement. Therefore, to access row elements, the increment should equal the number of rows in the array. With VIS, only one row can be accessed at a time. If NumElements is greater than the actual number of row elements, then the elements would be out of the array bounds. However, any number of columns can be accessed because elements are stored contiguously in column order.

For a three dimensional array, to increment from one plane to the next, the increment is the size of the array for one plane. For instance, for a 3x5x2 array to go from element 3,5,1 to 3,5,2, the increment is 3x5 or 15. (There are 15 elements in memory between the two elements.) The different array structures are depicted as follows:

one dimensional array

a(10)



two dimensional array

b(3x5)

	5 columns				
3 rows	1.0	4.0	7.0	10.0	13.0
	2.0	5.0	8.0	11.0	14.0
	3.0	6.0	9.0	12.0	15.0

three dimensional array

c(3x5x2)

	5 columns					
3 rows	1.0	4.0	7.0	10.0	13.0	(plane 1 for three dimensional array)
	2.0	5.0	8.0	11.0	14.0	
	3.0	6.0	9.0	12.0	15.0	

	5 columns					
3 rows	2.0	8.0	14.0	20.0	24.0	(plane 2 for three dimensional array)
	4.0	10.0	16.0	21.0	26.0	
	6.0	12.0	18.0	22.0	28.0	

The use of increment and number of elements for these arrays could be:

Increment	NumElements	Action
1	10	access every element of array a
1	15	access every element of array b in column order
1	30	access every element of array c in column order
1	3	access one column of array b
1	3	access one column of array c
1	6	access two consecutive columns of array b
1	6	access two consecutive columns of array c
2	5	access every other element of array c
3	5	access one row of array b
3	5	access one row of array c
3	10	access two rows of array c, the first row of each plane
15	2	access elements c(3,5,1) and c(3,5,2)

Zero Increment

A zero increment value causes the same element to be repeatedly accessed. This feature can be used to initialize arrays to scalar values.

With the VECTOR MOVE routines, VMOV or DVMOV, you can fill in an array with a constant:

Example: Initialize an array to a constant.

```
C     FORTRAN DO loop:
      DO 10 i = 1,n
10      a(i) = const
C
C     VIS equivalent:
      CALL VMOV(const,0,a,1,n)
```

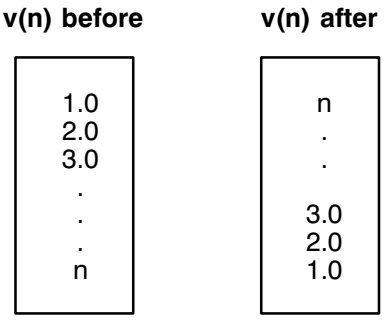
Const must be a floating point variable with precision matching array a. The precision of the scalar value can be modified as needed with the FORTRAN FLOAT, DBLE, or SNGL functions.

Negative Increment

The negative increment sequences through an array in reverse order. The VSWP instructions can also be used to reverse the order of the elements in an array. For example, transforming a vector whose values are arranged in ascending order into one of descending order is easily accomplished by the vector instruction:

Example: Swap elements in an array.

```
C     FORTRAN DO loop:
      j = n
      DO 10 i = 1,n/2
          temp = v(i)
          v(i) = v(j)
          v(j) = temp
10      j = j - 1
C
C     VIS equivalent:
      CALL VSWP (v,1,v(n),-1,n/2)
```



Here v(n) specifies the last element of the vector and has a corresponding increment of -1. If n is an odd number, the algorithm still works since the middle vector element is not moved.

Note If the starting array element (v1, v2, or v3 parameter) is actually the very first element of an array and is specified with a negative increment, VIS starts at that element and processes backward in memory out of the array's bounds. With negative increments, be careful to specify the starting array elements well within the array bounds.

Useful Applications

Here are some examples showing practical uses of VIS.

Initialize a Square Matrix

The identity matrix has ones on the main diagonal and zeros in all other elements. To create this matrix, two vector instructions are needed:

1. Fill the entire matrix with zeros.
2. Insert ones on the main diagonal.

For a double precision matrix a(n x n), the VIS instructions are:

Example: Initialize a square matrix in double precision.

```
C     FORTRAN DO loop:
      DO 10 i = 1,n
          DO 20 j = 1,n
              a(i,j) = 0.0D0
              IF (i .EQ.  j) a(i,j) = 1.0D0
20        CONTINUE
10    CONTINUE
C
C     VIS equivalent:
      CALL DVMOV (0.0D0, 0, a, 1, n*n)
      CALL DVMOV (1.0D0, 0, a, n+1, n)
```

Notice that the scalars are double precision and have increments of 0. In the first VIS instruction, a's increment is 1 to cause all elements to be zeroed. The second instruction uses an increment of n+1 for a to initialize ones on the main diagonal.

a(5x5)

	5 columns				
5 rows	1	0	0	0	0
	0	1	0	0	0
	0	0	1	0	0
	0	0	0	1	0
	0	0	0	0	1

Note that the elements on the main diagonal are 5+1 or 6 elements apart in memory.

Initialize an Array in a Certain Order

In FORTRAN, DO loops are often used to initialize an array with values in a certain order; VIS can easily do this function. The following array is initialized in ascending order from 1 to n:

Example: Initialize an array from 1 to n.

```
      array(1) = 1.0
      array(2) = 2.0
C     FORTRAN DO loop:
      DO 10 i = 1,n-2
10      array (i+2) = array(i) + 2.0
C
C     VIS equivalent:
      CALL VSAD (2.0, array, 1, array(3), 1, n-2)
```

The resulting array must start with array(3), because VIS is "pipelined"; that is, it accesses the next element while arithmetic is being performed on the current element. This affects operations where the next element's value depends on the previous elements. Therefore, although the following example may seem logically correct, it does not execute as above.

Example: Incorrect initialization of an array.

```
array(1) = 1.0
CALL VSAD (1.0, array, 1, array(2), 1, n-1)
```

Statistical Examples

Several basic statistical measures can be rapidly and easily calculated using VIS. The most obvious is the average or mean. Assuming that n sample values are stored in the vector v:

Example: Average or mean.

```
CALL VSUM (sum, v, 1, n)
avg = sum/n
```

A similar quantity is the root-mean-square (RMS) average. Here it is necessary to sum the squares of the sample values, which can easily be done by taking the dot product of the vector with itself. The RMS calculation then becomes:

Example: Root-mean-square.

```
CALL VDOT (dot, v, 1, v, 1, n)
rms = SQRT (dot/n)
```

Note The function SQRT is in the Scientific Instruction Set and thus also executes at microcoded speed.

Another useful statistical quantity is the standard deviation, which measures the distribution (or spread) of the sample values about the mean. First the average must be calculated and subtracted from each sample value. A temporary vector will probably be needed to hold the results of this subtraction. The algorithm for standard deviation then becomes:

Example: Standard deviation.

```
CALL VSUM (sum, v, 1, n)
avg = sum/n
CALL VSAD (-avg, v, 1, temp, 1, n)
CALL VDOT (dot, temp, 1, temp, 1, n)
stdev = SQRT(dot/(n-1))
```

To subtract the scalar avg from each element, VSAD is used with a negative value of avg.

The standard deviation can also be computed without a temporary vector using the following formula:

Example: More efficient standard deviation.

$$\text{STDEV} = \sqrt{\frac{\sum x^2 - \frac{(\sum x)^2}{n}}{n-1}}$$

$$S1 = \sum_{i=1}^{n} x_i \quad \text{use VSUM}$$

$$S2 = \sum_{i=1}^{n} x_i^2 \quad \text{use VDOT}$$

```
CALL VSUM (S1, v, 1, n)
CALL VDOT (S2, v, 1, v, 1, n)
var = (S2 - S1*S1/n)/(n-1)
stdev = SQRT(var)
```

Matrix Transposition

The transposition matrix atrans of the matrix a is formed by transferring the rows of a into the columns of atrans. A FORTRAN program to generate the matrix atrans could be written as:

```
DO 10 i = 1,n
   DO 10 j = 1,n
      atrans(i,j) = a(j,i)
10 CONTINUE
```

To form this operation rapidly with vector instructions, two cases must be considered. In the first case, the matrix atrans is distinct from matrix a; for the second case, the transposition is done in place.

Case 1 (Not in Place)

A vector move instruction with the proper increment values can move one row of matrix into a column of matrix atrans. Thus, a DO loop is required to move all of the rows:

```
      DO 10 i = 1,n
          CALL VMOV (a(i,1), n, atrans(1,i), 1, n)
10 CONTINUE
```

Case 2 (In Place)

Transposing a matrix in place is a little more complicated than the previous case. The vector instruction VMOV does not work, but the VSWP instruction can be used. It is easier to understand the algorithm by applying it to a specific case. Let us define the 4 x 4 matrix a and its desired transposition atrans:

a(4x4)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

atrans(4x4)

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Since the main diagonals of a and atrans are identical, only the off-diagonal terms need to be modified. To form the first row and column of atrans, swap row 1 of a with column 1 of a. (You could actually include the diagonal element, but that would be swapping a value with itself, which is a waste of time.) Thus, to form the first row and column of atrans:

```
CALL VSWP (a(1,2), 4, a(2,1), 1, 3)
```

The increment of the first vector (row) is 4 and the increment of the second vector (column) is 1. The number of elements is 3 instead of 4 because the diagonal need not be modified.

To perform the entire transposition in place, use a DO loop to similarly manipulate all of the rows and columns. As you move to each succeeding row/column, the number of elements to be exchanged decreases. Generalizing the example to the n x n case yields:

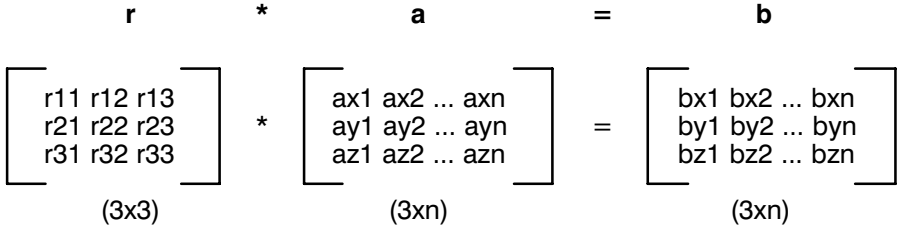
```
      DO 10 i = n-1
        CALL VSWP (a(i,i+1), n, a(i+1,i), 1, n-i)
10 CONTINUE
```

Only n-1 loop passes are required since the last row and column is the a(n,n) element.

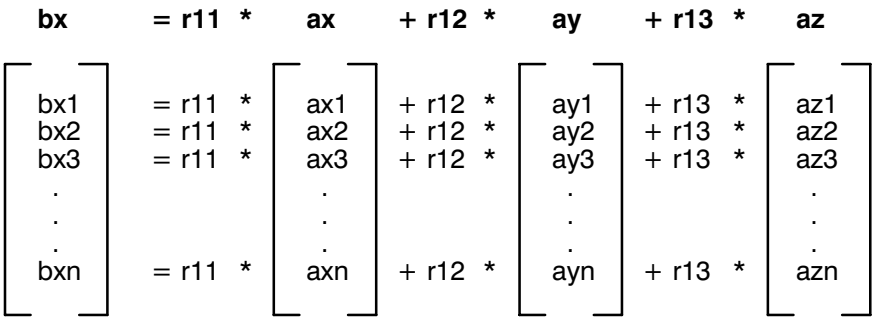
Graphics Coordinate Transformation

A basic operation in three dimensional graphics software is the coordinate transformation. Points in the graphics space are stored as 3 element vectors, whose values are the x, y, and z coordinates of each point. (The n points of the complete picture thus require n vectors, which are stored in a 3 x n array). Coordinate transformation involves rotating the graphical image and then translating the points to a new origin.

The rotation is accomplished by first building a 3 x 3 rotation matrix r. Each vector is then "passed through" the rotation matrix by matrix multiplication to create a new set of coordinate vectors. The operation can be expressed as:



This operation is matrix multiplication. However, since the matrixes a and b usually contain a large number of points (columns), n will be much greater than the number of rows (3) and a more efficient technique can be found. Instead of transforming each column vector, let us examine the equation for the x-coordinate of each point:



From these equations you see that bx, the vector of new x-coordinates, is a linear combination of the input coordinate vectors ax, ay, and az. Note that the rotation matrix coefficients (r11, r12, r13 just appear as scalars in these equations. Using the general notation bx, ax, ay, and az, the VIS equivalent of these equations becomes:

```
CALL VSMY (r11, ax, 3, bx, 3 n)
CALL VPIV (r12, ay, 3, bx, 3, bx, 3, n)
CALL VPIV (r13, az, 3, bx, 3, bx, 3, n)
```

All the increments are 3 because the coordinate vectors are rows of the matrixes a and b.

The same equations apply to the other dimensions y and z when the proper values of the rotation matrix are used. The final VIS algorithm for three dimensional rotation then becomes:

```
   DO 10 i = 1,3
       CALL VSMY (r(i,1), a(1,1), 3, b(i,1), 3, n)
       CALL VPIV (r(i,2), a(2,1), 3, b(i,1), 3, b(i,1), 3, n)
       CALL VPIV (r(i,3), a(3,1), 3, b(i,1), 3, b(i,1), 3, n)
10 CONTINUE
```

This algorithm generates 9 VIS calls, independent of the number of vectors (n). The matrix multiplication (VDOT) technique would generate 3*n VDOT calls of length 3. In an actual application, n can easily be in the thousands, making the speed advantage of this algorithm highly significant.

The translation process just involves adding a translation vector (tx, ty, tz) to each coordinate vector. As in rotation, this is best accomplished by doing all of the x-coordinates, then the y-coordinates, and then the z-coordinates:

```
CALL VSAD(tx, b(1,1), 3, b(1,1), 3, n)
CALL VSAD(ty, b(2,1), 3, b(2,1), 3, n)
CALL VSAD(tz, b(3,1), 3, b(3,1), 3, n)
```

Extended Memory Area (EMA) Considerations

When programming with EMA or VMA arrays and variables, check your programs for the following items:

1. Check the MSEG size in the $EMA directive.

   ```
   MSEG = 2*n-1
   ```

   where:

   n is the largest number of parameters declared in a VIS call.

   For example, in the following VIS routines, the MSEGs are:

   ```
   WSUM:  n = 1 (v1 only), MSEG = 2*1-1 = 1
   WDOT:  n = 2 (v1 and v2), MSEG = 2*2-1 = 3
   WDOT:  n = 3 (v1, v2, and v3) MSEG = 2*3-1 = 5
   ```

2. Check the VIS calling sequence for proper EMA and non-EMA parameters. With the EMA routines, only the v1, v2, and v3 parameters can be in EMA. All other parameters (incr1, incr2, incr3, and scalar) cannot be EMA call-by-reference variables. However, they can be call-by-value variables.

3. Check the actual name of the VIS routine for the correct EMA routine. Also, use the Y option in the FORTRAN control statement (FTN7X,Y) when using double precision routines. The initial letter determines the type of VIS routine:

   W = single precision, EMA routine
   DW = double precision, EMA routine

4. Do not use EMA transparency mode for addressing EMA/VMA variables.

Refer to the *RTE-A Programmer's Reference Manual*, part number 92077-90007, for a description of EMA and VMA in RTE-A. There are no special programming requirements for VMA other than those mentioned above.

EMA Call by Value and Call by Reference

An EMA variable or array can be passed "by value" or "by reference". (Refer to the EMA directive and EMA statement in the *FORTRAN 77 Reference Manual*, part number 92836-90001.) If passed by reference, an entire array or variable is available and can be changed by the called subroutine. If passed by value, only the value is available and the variable cannot be altered. Therefore, "by value" arguments can be used as operands but not as results. "By value" is noted by extra parentheses around the variable or by an arithmetic expression as shown with the variable scalar in the example:

```
CALL subr ((scalar), earray(1,1),1,10)
CALL subr (scalar+0.,earray(1,1),1,10)
```

For all EMA VIS routines, EMA arrays must be passed by reference for the v1, v2, and v3 parameters. All other parameters (scalar, incr1, incr2, incr3, and NumElements) cannot be in EMA. If an EMA variable is needed, it can be placed into a temporary non-EMA variable or be passed by value. Scalar results should not be "by value" elements in a calling sequence. Transparency mode for EMA/VMA addressing is not supported with VIS. The following examples illustrate using a temporary non-EMA element and a "call by value" element:

Example: Using an EMA VIS routine with temporary non-EMA element to add a variable to each element in an array.

```
$EMA(xyz,3)
      PROGRAM sump
      COMMON /xyz/ a(100,300), tsum(100)
C
C     Find sum of each row in a and put into tsum.
C
C     Without VIS:
      DO 10 i = 1,100
          tsum(i) = 0.0
          DO 20 j = 1,300
              tsum(i) = tsum(i) + a(i,j)
   20     CONTINUE
   10 CONTINUE
C
C     VIS equivalent:
      DO 10 i = 1,100
          CALL WSUM (sum, a(i,1), 100, 300)
          tsum(i) = sum
   10 CONTINUE
      END
```

Arrays a and tsum are in EMA. The variable sum is not in EMA and is used as the scalar result in wsum. sum's value is placed into tsum each time through the DO loop. With VIS, tsum does not have to be initialized to zero.

Example: Use an EMA element as call-by-value element to add a variable to each element in an array.

```
$EMA(xyz,3)
      COMMON /xyz/ a(500), b(500), c(300)
      EQUIVALENCE (var, c(1))
C
C     Add var to each element in b and store into a.
C
C     Without VIS:
      DO 20 i = 1, 500
          a(i) = var + b(i)
   20 CONTINUE
C
C     VIS equivalent:
      CALL WSAD ( (var), b, 1, a, 1, 500)
      END
```

Arrays a, b, and c are in EMA. Var is made equivalent to c(1), so it also is in EMA. In WSAD, var is used as the scalar operand by passing it "by value."

Obtaining Efficiency with Multidimensional Arrays

To optimize program execution times when manipulating EMA arrays, access array elements contiguously in memory (column order) with VIS routines.

With EMA arrays, different portions of an array can be mapped into the MSEG many times to access all the array elements. Since FORTRAN array elements are stored in column order, row elements cause more mapping since they are not contiguous in memory. Therefore, for more efficient execution, access EMA arrays by columns. For FORTRAN DO loops written for row access, modify the program for column access by the following:

1. Exchange array subscripts in matrix array declarations (DIMENSION, COMMON, or type specification statements) of the matrix being altered.

2. Exchange only the subscripts in all executable statements where the matrix is used.

3. Change the increment parameter associated with the matrix array to 1 in VIS calls.

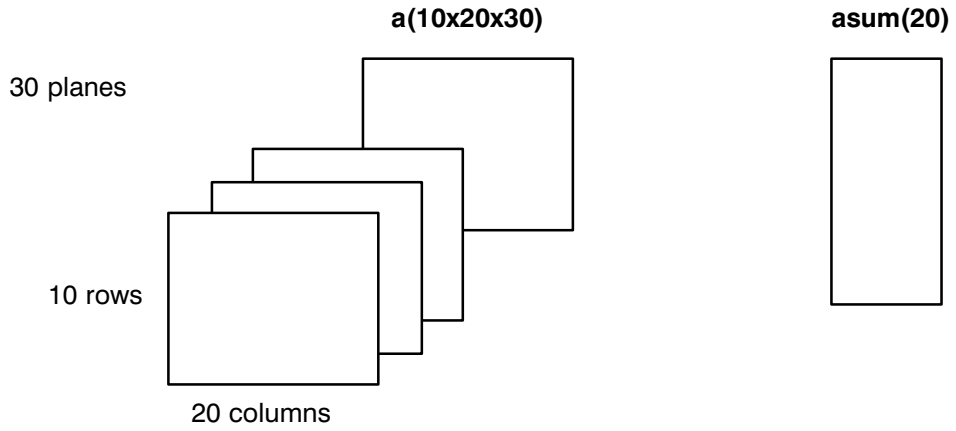
Program sump from above can be rewritten to calculate the sums of each column in a instead of each row:

Example: More efficient use of EMA arrays to find sum of each column in array a.

```
$EMA(xyz,1)
      PROGRAM sump1
      COMMON /xyz/ a(300,100), tsum(100)
C
C     Find sum of each column in a and put into tsum.
C
C     Without VIS:
      DO 10 i = 1,100
          tsum(i) = 0.0
          DO 20 j = 1,300
              tsum(i) = tsum(i) + a(i,j)
   20     CONTINUE
   10 CONTINUE
C
C     VIS equivalent:
      DO 10 i = 1,100
          CALL WSUM (sum, a(1,i), 1, 300)
          tsum(i) = sum
   10 CONTINUE
      END
```

In this example, the sum of each column is found. This is functionally equivalent, yet much faster than the previous sum example. The increment is 1 to access each contiguous column element. Only the subscripts of a and the increment were changed.

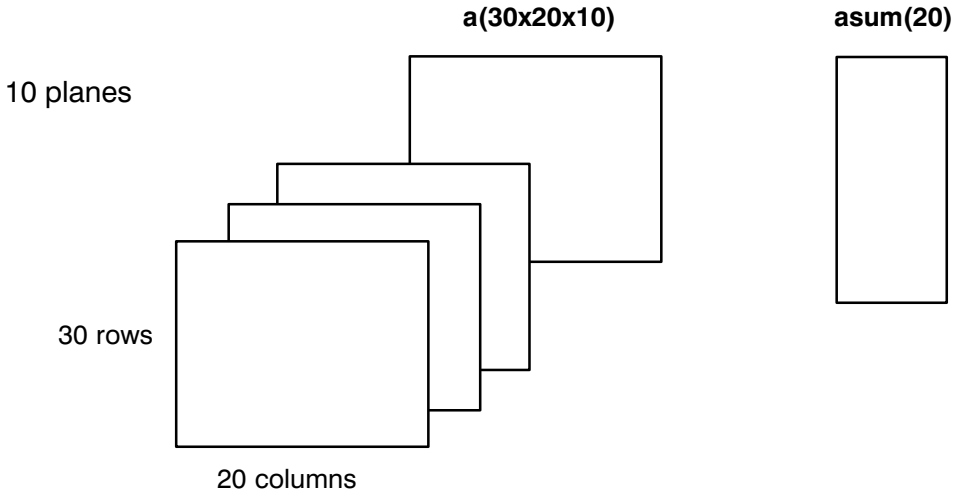
The next example computes 20 sums of a three dimensional array, a (10x20x30) in EMA into a non-EMA array, asum (20). Starting with each element in the first row, the sum is taken from every plane.



Example: Find sum of plane elements for row 1.

```
$EMA(area,1)
      PROGRAM plana
      COMMON/area/a
      DIMENSION a(10,20,30), asum(20)
C
C     Without VIS:
      DO 10 i = 1,20
          asum(i) = 0.0
          DO 20 j = 1,30
              asum(i) = asum(i) + a(1,i,j)
   20     CONTINUE
   10 CONTINUE
C
C     VIS equivalent:
      DO 10 i = 1,20
          CALL WSUM (asum(i), a(1,i,1), 10*20, 30)
   10 CONTINUE
      END
```

To be more efficient, exchange the dimensions of a to be a(30x20x10). Then find the sum of each column in plane 1.



Example: Find sum of elements for each column in plane 1.

```
$EMA(area,1)
      PROGRAM plana
      COMMON/area/a
      DIMENSION a(30,20,10), asum(20)
C
C     Without VIS:
      DO 10 i = 1,20
          asum(i) = 0.0
          DO 20 j = 1,30
              asum(i) = asum(i) + a(j,i,1)
   20     CONTINUE
   10 CONTINUE
C
C     VIS equivalent:
      DO 10 i = 1,20
          CALL WSUM (asum(i), a(1,i,1), 1, 30)
   10 CONTINUE
      END
```

Matrix Multiplication EMA Example

Matrix multiplication (a * b = c) is defined if the number of columns of a is equal to the number of rows in b:

```
  a     *     b     =   c
(M*N)       (N*P)     (M*P)
```

Each element c(i,j) is defined to be the dot product of the ith row of a and the jth column of b. Thus, a DO loop is required to generate each element c(i,j). A FORTRAN routine to form the entire product matrix c might be:

```
        DO 10 i =1,m
            DO 10 j = 1,p
                c(i,j) = 0.0
                DO 20 l = 1,n
                    c(i,j) = c(i,j) + a(i,l) * b(l,j)
    20          CONTINUE
    10  CONTINUE
```

The innermost loop can be replaced with a call to the VDOT routine:

```
        DO 10 i = 1,m
            DO 10 j = 1,p
C
C     Without VIS:
                c(i,j) = 0.0
                DO 20 l = 1,n
                    c(i,j) = c(i,j) + a(i,l) * b(l,j)
  20            CONTINUE
C     VIS equivalent:
                CALL VDOT (c(i,j), a(i,1), m, b(1,j), 1, n)
C
  10  CONTINUE
```

The increment on the row vector a(i,1) is m and the increment on the column vector b(1,j) is 1. The expression c(i,j) is actually a scalar as far as the VDOT instruction is concerned.

Recall that scalar parameters in VIS cannot be in the EMA address space. Thus, if the matrixes a, b, and c are all in EMA, an additional statement must be included to place the scalar result into the EMA area, c(i,j).

Example: Matrix multiplication with EMA arrays.

```
$EMA(xyz,3)
      PROGRAM ema1
      COMMON /xyz/ a, b, c
      DIMENSION a(100,200), b(200,300), c(100,300)
C
      m = 100
      p = 300
      n = 200
      DO 10 i = 1,m
          DO 10 j = 1,p
C
C     Without VIS:
              c(i,j) = 0.0
              DO 20 l = 1,n
                  c(i,j) = c(i,j) + a(i,l) * b(l,j)
   20         CONTINUE
      VIS equivalent:
              CALL WDOT (temp, a(i,1), m, b(1,j), 1, n)
              c(i,j) = temp
C
   10 CONTINUE
      END
```

Note that, as required, the scalar result temp is not in EMA. The dot product is calculated between the rows in a and the columns in b. To speed the array processing, the arrangement of rows and columns in a could be exchanged. This would cause the dot product of the columns in a and b to be calculated:

```
$EMA(xyz,3)
      PROGRAM ema2
      COMMON /xyz/ a, b, c
      DIMENSION a(200,100), b(200,300), c(100,300)
C
      m = 100
      p = 300
      n = 200
      DO 10 i = 1,m
          DO 10 j = 1,p
C
C     Without VIS:
              c(i,j) = 0.0
              DO 20 l = 1,n
   20           c(i,j) = c(i,j) + a(l,i) * b(l,j)
C     VIS equivalent:
              CALL WDOT (temp, a(1,i), 1, b(1,j), 1, n)
              c(i,j) = temp
C
   10 CONTINUE
      END
```

Here, the dot product of columns in a and b is calculated more quickly. The increment is 1 for a to access the contiguous column elements. Only the dimensioning and subscripts of a and the increment changed from the previous example.

Example VIS Programs

Calculating Prime Numbers: Sieve of Eratosthenes

The Sieve of Eratosthenes is a scheme for calculating prime numbers. The sieve is attributed to Eratosthenes, an ancient Greek scholar. First integers are written in consecutive order starting with 1. Then the nonprime numbers are eliminated in a certain order leaving the prime numbers.

In the program that follows, the prime numbers of the first 65,535 numbers are calculated. Since all even numbers except for 2 are not prime numbers, an array of odd numbers beginning with 3 is generated first. The sequence for eliminating nonprime numbers is:

1. Starting with the square of 3 (3*3 = 9), every third number is crossed off or zeroed.
2. The square of the next number 5 (5*5 = 25) is found. From that position, every fifth number is zeroed.
3. The square of the next number 7 (7*7 = 49) is found. From that position, every seventh number is zeroed.
4. This elimination continues until the square root of the largest number is reached.

Note Some numbers may be zeroed more than once. The numbers that are not crossed off (that is, not zeroed) are prime numbers.

```
$EMA(dan,3)
      PROGRAM sieve
      COMMON /dan/ prime(32767)
      DIMENSION ip(5),anum(10)
      EQUIVALENCE (ip,lu)
C
C***  This program generates prime numbers from 2 to 65535 ****
C                                                             *
C     incr   = increment through array prime                  *
C              And also the working prime number              *
C     ip     = global inputs                                  *
C     32767  = length of array prime                          *
C     lu     = output logical unit                            *
C     ncount = number of values to eliminate                  *
C     prime  = array of odd numbers, initially; then,         *
C              Array of prime numbers                         *
C     isqrln = index of the largest odd number .LE.           *
C              the SQRT of last number in array               *
C     istrt  = index of incr**2 which is 1st no. to eliminate *
C                                                             *
C*************************************************************
C     Get output logical unit
      CALL RMPAR(lu)
C
      isqrln = (SQRT(2.0*32767+1.0) – 1)/2
C***************************************
C     Initialize prime with odd numbers  *
C***************************************
      prime(1) = 3.0
      prime(2) = 5.0
C-------------------------------------------------------------
C     Without VIS:
      DO 10 i = 3,32767
   10   prime(i) = prime(i–2) + 4.0
C
C     With VIS:
      CALL WSAD(4.0,prime,1,prime(3),1,32767–2)
C-------------------------------------------------------------
C
C*********************************
C     Eliminate nonprime numbers  *
C*********************************
      DO 100 i = 1,isqrln
      IF (PRIME(I).EQ.0.0  GOTO 100
      incr = i + i + 1
      istrt = incr * incr/2.0
      ncount = (32767 – istrt)/incr + 1

C*********************************
C     Zero out nonprime numbers    *
C*********************************
```

```
C-------------------------------------------------------------------
C     Without VIS:
      j = istrt
      DO 20 k = 1,ncount
      prime(j) = 0.0
   20   j = j + incr
C
C     With VIS:
      CALL VWMOV(0.0,0,prime(istrt),incr,ncount)
C-------------------------------------------------------------------
  100 CONTINUE
C**********************************
C     List prime numbers           *
C                                  *
C     isum = total number of primes *
C     anum = 10 primes to print    *
C            per line              *
C**********************************
      isum = 0
      j    = 1
      anum(j) = 2.0
      DO 200 i = 1,32767
          IF (prime(i).EQ.0.0) GOTO 200
          isum = isum + 1
          j = j + 1
          anum(j) = prime(i)
          IF (j.LT.10) GOTO 200
          IF (j.NE.0)  WRITE(lu,250) (anum(k),k=1,j)
          j = 0
   200 CONTINUE
       WRITE (lu, 300) isum
   250 FORMAT(5X,10I7)
   300 FORMAT(///" Total primes =  ",I5)
       END
```

Solution of Linear Systems

a is a square matrix of order n and b is a given vector of n elements. To solve

```
a * x = b
```

for the unknown vector x, a popular method is to use Gaussian elimination with back substitution. To find the solution vector x, the elements in a are first eliminated to form an upper tridiagonal. Then back substitution is performed.

The program that follows uses VIS routines in two places. The execution time for VIS is about 10 times faster than the non-VIS version. The array subscripts are changed to column-row order, to obtain maximum speed efficiency. Note that subroutine gauss accepts EMA call-by-reference arguments.

```
$EMA(den,5)
     PROGRAM vsolv
     COMMON /den/ a(201,200),x(200),ip(200)
     DIMENSION itim1(5),itim2(5)
C***  This program solves a linear system using VIS routines ****
C
     lu = loglu(i)
C**************************************
C  Read in order of linear system       *
C**************************************
C
     WRITE(lu,50)
  50 FORMAT(//"Input order of linear system _")
     READ(lu,*) n
     IF(n .LE.  1) n=2
     IF(n .GT.  200) n=200
C
C**************************************
C* Form the test matrix                *
C* For example: 1x + 2y + 3z = 4       *
C*              2x + 3y + 4z = 1       *
C*              3x + 4y + 1z = 2       *
C**************************************
C
     DO 10 i=1,n
         k=i
         DO 10 j=1,n+1
             IF(k.GT.n+1) k=MOD(k,n+1)
             a(j,i) = FLOAT(k)
  10 k=k+1
C
C**************************************
C* Call gauss to solve linear system   *
C**************************************
C* Call for system time                *
C**************************************
C
     CALL EXEC(11,itim1)
     CALL gauss (a,x,ip,200,201,n,iflag)
```

```
     CALL EXEC(11,itim2)
     WRITE(lu,55)
  55 FORMAT(///"The solution vector x is"//)
     DO 70 i=1,n
         WRITE(lu,60) i,x(i)
  60     FORMAT(9X,"X(",I3,")   ",F11.5)
  70 CONTINUE
     CALL etime(itim1,itim2)
     END
C
C
     SUBROUTINE gauss (a,x,ip,msize,msizp1,n,iflag)
     EMA a,x,ip
     REAL a(msizp1,msize), x(msize)
     INTEGER ip(msize)
C
C*************************************************************
C* Initialize the ip vector to be used for pivoting         *
C*************************************************************
C
     iflag=1
     DO 10 i=1,n
  10   ip(i)=i
C
C*************************************************************
C* Begin Gaussian elimination by first pivoting the equations *
C*************************************************************
C
     DO 20 k=1,n-1
         DO 30 i=k+1,n
             IF (ABS(a(k,ip(k))) .GE  ABS(a(k,ip(i)))  GOTO 30
             itemp=ip(k)
             ip(k)=ip(i)
             ip(i)=itemp
  30     CONTINUE
C
C*****************************************
C* Check for a zero in the diagonal         *
C*****************************************
         IF (a(k,ip(k)) .NE.  0.  G  TO 35
         iflag=0
  35 CONTINUE
C
C*********************************
C  Perform the actual elimination *
C*********************************
C
         DO 20 i=k+1,n
             z1 = a(k,ip(i))
             IF (z1 .EQ.  0  GOTO 20
             z1 = z1/a(k,ip(k))
C-------------------------------------------------------------------
C    Without VIS:
             DO 25 j=k+1,n+1
```

```
  25         a(j,ip(i)) = a(j,ip(i)) – z1*a(j,ip(k))
C
C    With VIS:
         CALL WPIV (–z1, a(k+1,ip(k)), 1, a(k+1,ip(i)), 1,
     +           a(k+1,ip(i)), 1, (n–k+1))
C
20        CONTINUE
C————————————————————————————————————————————————————————————
C*********************************************************
C  End of elimination – now check to make sure the matrix *
C                       is not singular                   *
C*********************************************************
C
     IF (a(n,ip(n)) .EQ.  0) iflag=0
C
C*********************************
C  Now perform back substitution  *
C*********************************
C
     DO 50 k=n,1,–1
         s=0.0
         l=k+1
         IF (l .GT.  n) GOTO 50
C————————————————————————————————————————————————————————————
C        Without VIS:
         DO 55 j=l,n
  55     s = s + a(j,ip(k))*x(j)
C
C    With VIS:
         CALL WDOT (s, a(l,ip(k)), 1, x(l), 1, (n–l+1))
C————————————————————————————————————————————————————————————
  50 x(k) = (a(n+1,ip(k))–s)/a(k,ip(k))
     RETURN

  70 iflag = 0
     RETURN
     END
C
C
     SUBROUTINE etime(itim1,itim2)
     DIMENSION itim1(5),itim2(5)
C
C*************************************
C*    Calculate elapsed time          *
C*************************************
C Check centiseconds
      IF (itim2(1) .GE.  itim1(1)) GOTO 10
      itim2(2) = itim2(2) – 1
      itim2(1) = itim2(1) + 100
C Check seconds
   10 IF (itim2(2) .GE.  itim1(2)) GOTO 20
      itim2(3) = itim2(3) – 1
      itim2(2) = itim2(2) + 60
C Check minutes
```

```
   20 IF (itim2(3) .GE.  itim1(3)) GOTO 30
      itim2(4) = itim2(4) – 1
      itim2(3) = itim2(3) + 60
C Don’t bother with hours
   30 icens = 10*(itim2(1) – itim1(1))
      isecs = itim2(2) – itim1(2)
      imins = itim2(3) – itim1(3)
      ihour = itim2(4) – itim1(4)
      WRITE(1,55) ihour,imins,isecs,icens
   55 FORMAT(//”Execution time  =”,3(I2,”:”),I3)
      RETURN
      END
```

Matrix Inversion

Finding the inverse of a matrix is a problem often presented in numerical analysis. The following program does matrix inversion for square matrixes from 50 to 200 in increments of 50. The array subscripts are exchanged to specify column-row order instead of row-column. This exchange executes about 10 times faster.

```
$EMA(bill,5)
      PROGRAM chuck
      COMMON /bill/ a,test
      DIMENSION itemp1(5),itemp2(5),ipvt(200)
      REAL a(200,200),test(400),dot,norm
      DATA msize / 200 /
      CALL RMPAR(itemp1)
C     listlu = output device
C     icheck = 0, DO NOT compute norm; = 1, compute the norm
      listlu = itemp1(1)
      icheck = itemp1(2)
C*********************************************************
C
C Major loop: calculate inverse of square matrixes
C from 50 to 200 in increments of 50
      DO 100 n = 50,200,50
C
C*********************************************************
C
C     Construct test matrix
          DO 10 i = 1,2*n
              test(i) = iabs(i–n)
   10     CONTINUE
C
          DO 15 j = 1,n
C———————————————————————————————————————
C           Without VIS:
C             DO 15 i = 1,n
C                 a(i,j) = test(n–j+i)
C           VIS equivalent:
                  CALL WMOV(test(n–j+1),1,a(1,j),1,n)
C———————————————————————————————————————
```

```
   15     CONTINUE

C*************************************************************
C
C     Set itemp1 to the cpu time in seconds
C
      CALL EXEC(11,itemp1)
C*************************************************************
C
C     Invert the matrix
C
      CALL invrt(a,msize,n,ipvt,ierr)
C*************************************************************
C
C     Set itemp2 to the cpu time in seconds
      CALL EXEC(11,itemp2)
C
C     Calculate elapsed time
      DO 25 i = 1,4
         itemp2(i) = itemp2(i) – itemp1(i)
   25 CONTINUE
      IF (itemp2(4) .LT.  0) itemp2(4) = itemp2(4) + 24
      time = itemp2(4)*3600.+itemp2(3)*60.+itemp2(2)
     +       +itemp2(1)/100.
C*************************************************************
C
C     Check the answer
C
      norm = 0.0
      IF (icheck .EQ.  0) GOTO 40
C
      DO 20 i = 1,n
         DO 20 j = 1,n
C————————————————————————————————————————————————————————————
C           Without VIS:
C           dot = 0.0
C           DO 30 k = 1,n
C  30         dot = dot + a(k,i) * test(n–j+k)
C           With VIS:
            CALL WDOT(dot,a(1,i),1,test(n–j+1),1,n)
C————————————————————————————————————————————————————————————
            IF (i .EQ.  j)   dot = dot – 1.0
            norm = norm + ABS(dot)
   20    CONTINUE
C
C*************************************************************
C
C     Print the results
C
      WRITE (listlu,90)
90    FORMAT(///" Matrix size",11X,"Norm",9X,"Inversion time"//)
40    WRITE (listlu,91) n,norm,time
91    FORMAT(4X,I3,10X,E12.5,5X,F12.2)
C*************************************************************
```

```
C
C     End of major loop
  100 CONTINUE
      END

      SUBROUTINE invrt(a,msize,n,ipvt,ierr)
C Matrix inversion
      EMA a
      REAL a(msize,n),pivot,t
      INTEGER ipvt(n)
      DATA eps / 1E-6 /
C*************************************************************
C
C     Initialize interchange pointer
C
      DO 5 i = 1,n
          ipvt(i) = i
   5  CONTINUE
C*************************************************************
C
C     Start loop through main diagonal
C
      DO 100 irow = 1,n
C*************************************************************
C
C     Search for pivot
C
         icol = ipvt(irow)
C-------------------------------------------------------------------
C        Without VIS:
C        imax = irow
C        pivot = a(icol,irow)
C        IF (irow .EQ.  n) GOTO 20
C        DO 10 jrow = irow+1, n
C           t = a(icol,jrow)
C           IF (ABS(t) .LE.  pivot) GOTO 10
C           imax = jrow
C           pivot = t
C  10    CONTINUE
C        With VIS:
         CALL WMAB(imab,a(icol,irow),msize,n-irow+1)
         imax = imab + irow - 1
         pivot = a(icol,imax)
C-------------------------------------------------------------------
   20    IF (ABS(pivot) .LT.  eps) GOTO 99
C
C*************************************************************
C
C     Interchange columns
C
         jmax = ipvt(imax)
         IF (imax .EQ.  irow) GOTO 40
C
         ipvt(irow) = ipvt(imax)
```

```
         ipvt(imax) = icol
C
C-------------------------------------------------------------------
C        Without VIS:
C        DO 25 jcol = 1,n
C            T = a(jcol,irow)
C            a(jcol,irow) = a(jcol,imax)
C            a(jcol,imax) = t
C  25    CONTINUE
C        With VIS:
         CALL WSWP(a(1,irow),1,a(1,imax),1,n)
C
C-------------------------------------------------------------------
C*******************************************************
C
C  Zero out elements in this column and form column of inverse.
C
   40      a(icol,irow) = a(jmax,irow)
           a(jmax,irow) = 1.0
C-------------------------------------------------------------------
C        Without VIS:
C        DO 45 jcol = 1,n
C           a(jcol,irow) = a(jcol,irow)/pivot
C  45    CONTINUE
C        With VIS:
         CALL wsmy(1.0/pivot,a(1,irow),1,a(1,irow),1,n)
C-------------------------------------------------------------------
C
         DO 50 jrow = 1,n
            IF (jrow .EQ.  irow) GOTO 50
            t = -a(icol,jrow)
            a(icol,jrow) = a(jmax,jrow)
C-------------------------------------------------------------------
C           Without VIS:
C           DO 60 kcol = 1,n
C              a(kcol,jrow) = a(kcol,jrow) + t*a(kcol,irow)
C  60       CONTINUE
C           With VIS:
            CALL wpiv(t,a(1,irow),1,a(1,jrow),1,a(1,jrow),1,n)
C-------------------------------------------------------------------
            a(jmax,jrow) = t/pivot
   50    CONTINUE
C
C*******************************************************
C
C     End of main loop
  100 CONTINUE
      ierr = 0
      RETURN
C  Error condition, ierr = column with no acceptable pivot.
   99 ierr = irow
      RETURN
      END
```

VIS Online Diagnostic

The program VISOD is the online diagnostic for the Vector Instruction Set on an HP 1000 A990, A900, A700, or F-Series Computer. It is designed to verify the proper installation and operation of read-only-memories (ROMs). The diagnostic contains operands for each VIS instruction that have been chosen to exercise the firmware through all of the internal algorithm paths.

Required Hardware and Software

The required hardware for each computer is described below:

1. HP 1000 A990 or A900 Computer with RTE-A.
2. HP 1000 A700 Computer with the HP 12156A floating-point board and RTE-A.
3. For the F-Series:
   a. HP 1000 F-Series Computer with an RTE-6/VM Operating System.
   b. HP 12791A Firmware Expansion Module (FEM). The FEM must reside in select code 10 or 11 of the CPU I/O backplane and is cabled to the CPU and FAB with a 50-pin flat ribbon cable.
   c. VIS firmware (part numbers 12824-80007 through 80009) installed on a FEM.

The software required to run VISOD consists of an RTE Operating System and the following VIS relocatables:

1. The Firmware Interface Library, $VLB6A (RTE-6/VM) or $VLBA1 (RTE-A).
2. The VIS Online Diagnostic, %VISO6 (RTE-6/VM) or %VISOA (RTE-A).

VISOD requires a 13-page partition to run on RTE-A and a 15-page partition to run on RTE-6/VM. The program must be assigned to a suitably-sized partition at load time. VISOD should not be loaded at system generation time. A duplicate entry point error will occur.

Test Sections

The VISOD program is divided into three main sections:

1. Self-test (F-Series only).
2. Non-privileged.
3. Privileged.

Self-Test Section

The VIS self-test section in VISOD is for the F-Series computer only. If the self-test section does not return the correct results, the following message is displayed on the output device and the program stops:

```
**** SELFTEST FAILURE
```

Non-Privileged Section

This section contains all of the tests that can be performed in a non-privileged user environment. These include all of the non-EMA VIS instructions and .ERES which resolves EMA call by reference addressing.

An error in any VIS instruction causes the following message to be displayed on the output device and the program to stop:

```
**** ERROR IN INSTRUCTION xxxxx
```

where *xxxxx* is the VIS instruction name. A successful pass displays:

```
VIS ON-LINE DIAGNOSTIC SUCCESSFUL COMPLETION
```

on the output device. If the privileged section is not to be tested, the message:

```
WARNING - PRIVILEGED INSTRUCTIONS NOT TESTED
```

precedes the above completion message.

Privileged Section

This section tests the VIS instructions .ESEG and .VSET by temporarily entering a privileged environment. In addition, the program must lock itself into memory for this portion of the testing. In this privileged mode, malfunctions in VIS firmware could cause catastrophic system failure. Therefore, it is recommended that this section be executed only on an otherwise inactive system.

The last test in the privileged section executes the VIS EMA sum routine, WSUM, to ensure that individually tested instructions correctly perform together.

An error in any of the instructions, .ESEG, .VSET, or WSUM causes this message to be displayed on the system console:

```
**** ERROR IN INSTRUCTION xxxxx
```

where *xxxxx* is the VIS instruction name.

Running the Diagnostic

Before operating the diagnostic, the RTE Operating System should be up and running. The proper entry points for VIS firmware should be specified during generation or declared to the loader with a separate RPL file.

The diagnostic program is supplied as a relocatable file which is loaded using the RTE loader. The main program to be loaded and the library to search are:

	RTE-A	**RTE-6/VM**
Main	%VISOA	%VISO6
Library	$VLBA1	$VLB6A

Once the diagnostic has been loaded, it can be executed as follows:

```
CI> VISOD[,lu][,#passes][,priv]
```

where:

lu is the output logical unit number. Specifies where messages are to be output. If 0 or not specified, defaults to user terminal.

#passes is the number of passes to be run. If 0 or not specified, 1 pass is run.

priv is the privileged section flag. If 0 or not specified, do not run the privileged section test. If 1, run the privileged section test.

Examples:

`CI> VISOD`	Print messages on user terminal, run 1 pass, do not run privileged section test.
`CI> VISOD,,1,0`	Print messages on user terminal, run 1 pass, do not run privileged section test.
`CI> VISOD,0G,3,1`	Print messages on user terminal, run 3 passes, run all sections.
`CI> VISOD,6,1`	Print messages on LU 6, run 1 pass, do not run privileged section test.
`CI> VISOD,,0,1`	Print messages on user terminal, run 1 pass, run privileged section test.

Caution When executing the privileged section, malfunctions in VIS firmware could cause overwriting of the operating system. It is recommended that other critical tasks not be executing concurrently.

Figure 9-1 is a troubleshooting flowchart to follow if errors occur while running the VIS online diagnostic.

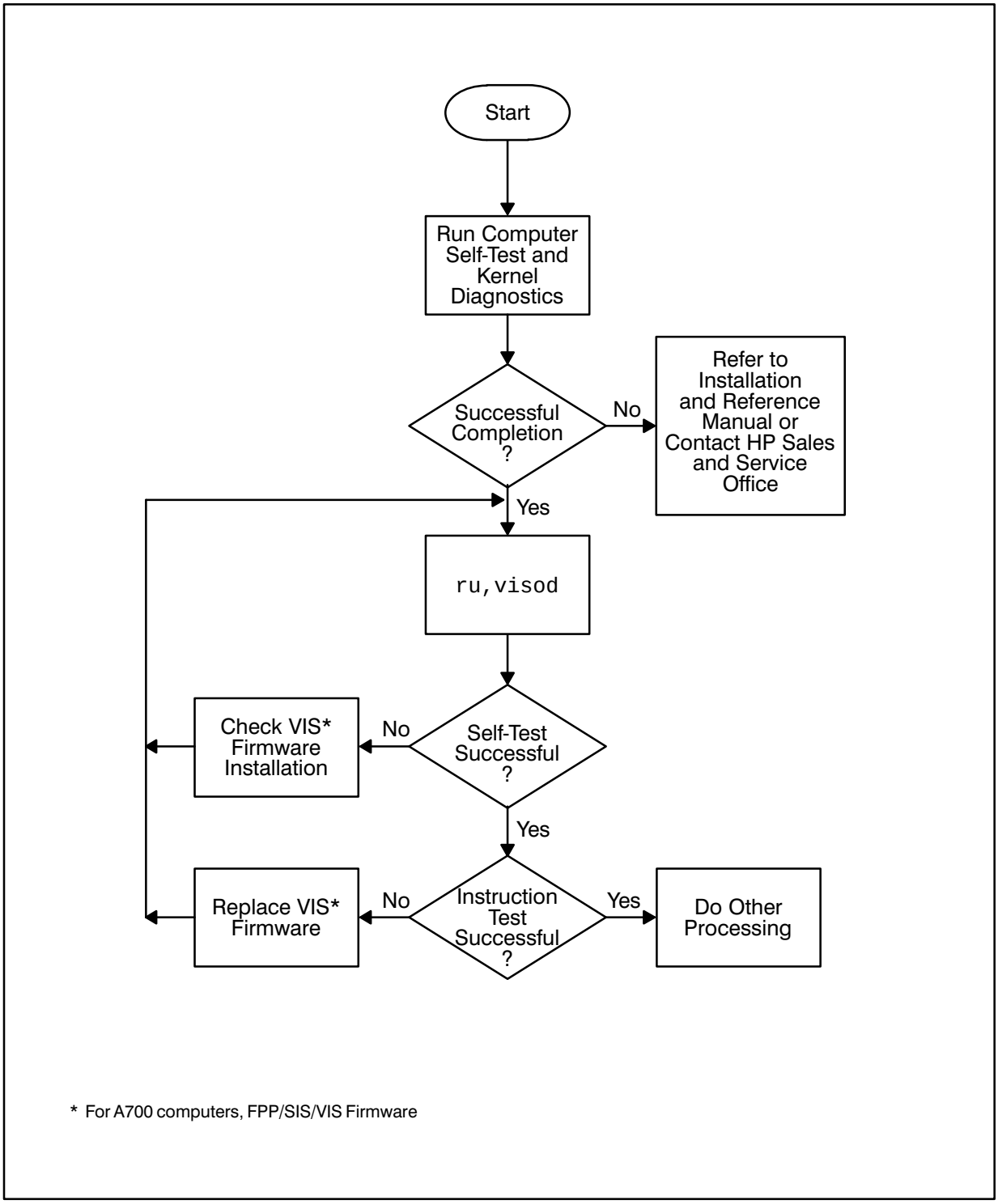


Figure 9-1. Troubleshooting Flowchart

FORTRAN Equivalents for VIS

The Software Equivalents Library contains FORTRAN equivalents for all VIS routines. Program results may not be exactly the same using the software equivalents in place of the firmware. For reference, the source of the single precision, non-EMA routines are presented. The following notation is used:

`V1,V2,V3`	= starting array elements
`INCR1,INCR2, INCR3`	= increments to obtain next elements
`S,IMAX,IMIN`	= scalar operand or result
`N`	= number of elements to process

VADD: addition

```
      SUBROUTINE VADD(V1,INCR1,V2,INCR2,V3,INCR3,N)
      REAL V1(1),V2(1),V3(1)
      IF(N.LE. 0) RETURN
      J1 = 1
      J2 = 1
      J3 = 1
      DO 10 I = 1,N
      V3(J3) = V1(J1) + V2(J2)
      J1 = J1 + INCR1
      J2 = J2 + INCR2
   10 J3 = J3 + INCR3
      RETURN
      END
```

VSUB: subtraction

```
      SUBROUTINE VSUB (V1,INCR1,V2,INCR2,V3,INCR3,N)
      REAL V1(1),V2(1),V3(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      J3 = 1
      DO 10 I = 1,N
      V3(J3) = V1(J1) - V2(J2)
      J1 = J1 + INCR1
      J2 = J2 + INCR2
   10 J3 = J3 + INCR3
      RETURN
      END
```

VMPY: multiplication

```
      SUBROUTINE VMPY(V1,INCR1,V2,INCR2,V3,INCR3,N)
      REAL V1(1),V2(1),V3(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      J3 = 1
      DO 10 I = 1,N
      V3(J3) = V1(J1) * V2(J2)
      J1 = J1 + INCR1
      J2 = J2 + INCR2
   10 J3 = J3 + INCR3
      RETURN
      END
```

VDIV: division

```
      SUBROUTINE VDIV(V1,INCR1,V2,INCR2,V3,INCR3,N)
      REAL V1(1),V2(1),V3(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      J3 = 1
      DO 10 I = 1,N
      V3(J3) = V1(J1) / V2(J2)
      J1 = J1 + INCR1
      J2 = J2 + INCR2
   10 J3 = J3 + INCR3
      RETURN
      END
```

VSAD: scalar-vector addition

```
      SUBROUTINE VSAD(S,V1,INCR1,V2,INCR2,N)
      REAL S,V1(1),V2(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      DO 10 I = 1,N
      V2(J2) = S + V1(J1)
      J1 = J1 + INCR1
   10 J2 = J2 + INCR2
      RETURN
      END
```

VSSB: scalar-vector subtraction

```
      SUBROUTINE VSSB(S,V1,INCR1,V2,INCR2,N)
      REAL S,V1(1),V2(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      DO 10 I = 1,N
      V2(J2) = S – V1(J1)
      J1 = J1 + INCR1
   10 J2 = J2 + INCR2
      RETURN
      END
```

VSMY: scalar-vector multiplication

```
      SUBROUTINE VSMY (S,V1,INCR1,V2,INCR2,N)
      REAL S,V1(1),V2(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      DO 10 I = 1,N
      V2(J2) = S * V1(J1)
      J1 = J1 + INCR1
   10 J2 = J2 + INCR2
      RETURN
      END
```

VSDV: scalar-vector division

```
      SUBROUTINE VSDV(S,V1,INCR1,V2,INCR2,N)
      REAL S,V1(1),V2(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      DO 10 I = 1,N
      V2(J2) = S / V1(J1)
      J1 = J1 + INCR1
   10 J2 = J2 + INCR2
      RETURN
      END
```

VPIV: pivot operation

```
      SUBROUTINE VPIV(S,V1,INCR1,V2,INCR2,V3,INCR3,N)
      REAL S,V1(1),V2(1),V3(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      J3 = 1
      DO 10 I = 1,N
      V3(J3) = S * V1(J1) + V2(J2)
      J1 = J1 + INCR1
      J2 = J2 + INCR2
   10 J3 = J3 + INCR3
      RETURN
      END
```

VDOT: dot product

```
      SUBROUTINE VDOT(S,V1,INCR1,V2,INCR2,N)
      REAL S,V1(1),V2(1)
      DOUBLE PRECISION DS
      IF (N.LE.0) RETURN
      DS = 0.0D0
      J1 = 1
      J2 = 1
      DO 10 I = 1,N
      DS = DS + DBLE(V1(J1)) * DBLE(V2(J2))
      J1 = J1 + INCR1
   10 J2 = J2 + INCR2
      S = SNGL(DS)
      RETURN
      END
```

VABS: absolute value

```
      SUBROUTINE VABS(V1,INCR1,V2,INCR2,N)
      REAL V1(1),V2(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      DO 10 I = 1,N
      V2(J2) = ABS(V1(J1))
      J1 = J1 + INCR1
   10 J2 = J2 + INCR2
      RETURN
      END
```

VSUM: sum

```
      SUBROUTINE VSUM(S,V1,INCR1,N)
      REAL S,V1(1)
      DOUBLE PRECISION DS
      IF (N.LE.0) RETURN
      DS = 0.0D0
      J1 = 1
      DO 10 I = I,N
      DS = DS + DBLE(V1(J1))
   10 J1 = J1 + INCR1
      S = SNGL (DS)
      RETURN
      END
```

VNRM: sum of absolute values

```
      SUBROUTINE VNRM(S,V1,INCR1,N)
      REAL S,V1(1)
      DOUBLE PRECISION DS
      IF (N.LE.0) RETURN
      DS = 0.0D0
      J1 = 1
      DO 10 I = 1,N
      DS = DS + DABS(DBLE(V1(J1)))
   10 J1 =J1 + INCR1
      S = SNGL(DS)
      RETURN
      END
```

VMAX: maximum value

```
      SUBROUTINE VMAX(IMAX,V1,INCR1,N)
      REAL V1(1),MAX,TMAX
      IF (N.LE.1) RETURN
      IMAX = 1
      MAX = V1(1)
      J1 = 1 + INCR1
      DO 10 I = 2,N
      TMAX = V1(J1)
      IF(TMAX.LE.MAX) GO TO 10
      IMAX=I
      MAX=TMAX
   10 J1 =J1 + INCR1
      RETURN
      END
```

VMAB: maximum absolute value

```
      SUBROUTINE VMAB(IMAX,V1,INCR1,N)
      REAL V1(1),MAX,TMAX
      IF (N.LE.1) RETURN
      IMAX = 1
      MAX = ABS(V1(1))
      J1 = 1 + INCR1
      DO 10 I = 2,N
      TMAX = ABS(V1(J1))
      IF(TMAX.LE.MAX) GO TO 10
      IMAX=I
      MAX=TMAX
   10 J1 = J1 + INCR1
      RETURN
      END
```

VMIN: minimum value

```
      SUBROUTINE VMIN (IMIN,V1,INCR1,N)
      REAL V1(1),MIN,TMIN
      IF (N.LE.1) RETURN
      IMIN = 1
      MIN = V1(1)
      J1 = 1 + INCR1
      DO 10 I = 2,N
      TMIN = V1(J1)
      IF(TMIN.GE.MIN) GO TO 10
      IMIN=I
      MIN=TMIN
   10 J1 = J1 + INCR1
      RETURN
      END
```

VMIB: minimum absolute value

```
      SUBROUTINE VMIB(IMIN,V1,INCR1,N)
      REAL V1(1),MIN,TMIN
      IF (N.LE.1) RETURN
      IMIN = 1
      MIN = ABS(V1(1))
      J1 = 1 + INCR1
      DO 10 I = 2,N
      TMIN = ABS(V1(J1))
      IF(TMIN.GE.MIN) GO TO 10
      IMIN=I
      MIN=TMIN
   10 J1 = J1 + INCR1
      RETURN
      END
```

VMOV: move

```
      SUBROUTINE VMOV(V1,INCR1,V2,INCR2,N)
      REAL V1(1),V2(1)
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      DO 10 I = 1,N
      V2(J2) = V1(J1)
      J1 = J1 + INCR1
   10 J2 = J2 + INCR2
      RETURN
      END
```

VSWP: swap

```
      SUBROUTINE VSWP(V1,INCR1,V2,INCR2,N)
      REAL V1(1),V2(1),TEMP
      IF (N.LE.0) RETURN
      J1 = 1
      J2 = 1
      DO 10 I = 1,N
      TEMP = V1(J1)
      V1(J1) = V2(J2)
      V2(J2) = TEMP
      J1 = J1 + INCR1
   10 J2 = J2 + INCR2
      RETURN
      END
```

Assembly Language Opcodes

A990, A900, and A700

The A990, A900, and A700 VIS uses the opcodes listed below.

VIS ROUTINES WITH ONE-WORD OPCODES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	DOUBLE PRECISION ROUTINE
105001	VADD	105021	DVADD
105003	VSUB	105023	DVSUB
105004	VMPY	105024	DVMPY
105005	VDIV	105025	DVDIV
105006	VSAD	105026	DVSAD
105007	VSSB	105027	DVSSB
105010	VSMY	105030	DVSMY
105011	VSDV	105031	DVSDV
105101	VPIV	105121	DVPIV
105103	VABS	105123	DVABS
105105	VSUM	105125	DVSUM
105107	VNRM	105127	DVNRM
105110	VDOT	105130	DVDOT
105111	VMAX	105131	DVMAX
105112	VMAB	105132	DVMAB
105113	VMIN	105133	DVMIN
105115	VMIB	105135	DVMIB
105116	VMOV	105136	DVMOV
105117	VSWP	105137	DVSWP

VECTOR ARITHMETIC ROUTINES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	DOUBLE PRECISION ROUTINE
105001	VADD	105021	DVADD
105003	VSUB	105023	DVSUB
105004	VMPY	105024	DVMPY
105005	VDIV	105025	DVDIV

Calling sequence:

```
CALL subr (v1, incr1, v2, incr2, v3, incr3, #elements)
```

These routines are each nine-word instructions:

```
word 1 = first word of opcode
word 2 = address of next instruction
word 3 = address of array,           v1
word 4 = address of increment,       incr1
word 5 = address of array,           v2
word 6 = address of increment,       incr2
word 7 = address of array,           v3
word 8 = address of increment,       incr3
word 9 = address of no. of elements, #elements
```

VECTOR-SCALAR ARITHMETIC ROUTINES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	DOUBLE PRECISION ROUTINE
105006	VSAD	105026	DVSAD
105007	VSSB	105027	DVSSB
105010	VSMY	105030	DVSMY
105011	VSDV	105031	DVSDV

Calling sequence:

```
CALL subr (scalar, v1, incr1, v2, incr2, #elements)
```

These routines are each eight-word instructions:

```
word 1 = first word of opcode
word 2 = address of next instruction
word 3 = address of scalar,          scalar
word 4 = address of array,           v1
word 5 = address of increment,       incr1
word 6 = address of array,           v2
word 7 = address of increment,       incr2
word 8 = address of no. of elements, #elements
```

ABSOLUTE VALUE ROUTINE

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	DOUBLE PRECISION ROUTINE
105103	VABS	105123	DVABS

Calling sequence:

```
CALL subr (v1, incr1, v2, incr2, #elements)
```

These routines are each seven-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of array,           v1
word 4 = address of increment,       incr1
word 5 = address of array,           v2
word 6 = address of increment,       incr2
word 7 = address of no. of elements, #elements
```

SUM ROUTINES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	DOUBLE PRECISION ROUTINE
105105	VSUM	105125	DVSUM
105107	VNRM	105127	DVNRM

Calling sequence:

```
CALL subr (scalar, v1, incr1, #elements)
```

These routines are each six-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of scalar,          scalar
word 4 = address of array,           v1
word 5 = address of increment,       incr1
word 6 = address of no. of elements, #elements
```

DOT PRODUCT ROUTINE

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	SINGLE PRECISION ROUTINE
105110	VDOT	105130	DVDOT

Calling sequence:

```
CALL subr (scalar, v1, incr1, v2, incr2, #elements)
```

These routines are each eight-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of scalar,          scalar
word 4 = address of array,           v1
word 5 = address of increment,       incr1
word 6 = address of array,           v2
word 7 = address of increment,       incr2
word 8 = address of no. of elements, #elements
```

PIVOT ROUTINE

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	SINGLE PRECISION ROUTINE
105101	VPIV	105121	DVPIV

Calling sequence:

```
CALL subr (scalar, v1, incr1, v2, incr2, v3, incr3, #elements)
```

These routines are each ten-word instructions:

```
word  1 = opcode
word  2 = address of next instruction
word  3 = address of scalar,          scalar
word  4 = address of array,           v1
word  5 = address of increment,       incr1
word  6 = address of array,           v2
word  7 = address of increment,       incr2
word  8 = address of array,           v3
word  9 = address of increment,       incr3
word 10 = address of no. of elements, #elements
```

MAX/MIN ROUTINES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	SINGLE PRECISION ROUTINE
105111	VMAX	105131	DVMAX
105112	VMAB	105132	DVMAB
105113	VMIN	105133	DVMIN
105115	VMIB	105135	DVMIB

Calling sequence:

```
CALL subr (scalar, v1, incr1, #elements)
```

These routines are each six-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of integer scalar,  scalar
word 4 = address of array,           v1
word 5 = address of increment,       incr1
word 6 = address of no. of elements, #elements
```

MOVE ROUTINES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	SINGLE PRECISION ROUTINE
105116	VMOV	105123	DVABS
105117	VSWP	105137	DVSWP

Calling sequence:

```
CALL subr (v1, incr1, v2, incr2, #elements)
```

These routines are each seven-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of array,           v1
word 4 = address of increment,       incr1
word 5 = address of array,           v2
word 6 = address of increment,       incr2
word 7 = address of no. of elements, #elements
```

F-Series

The firmware address space 06000B to 07777B is assigned to VIS with the opcodes 101460 through 101477 and 105460 through 105477. The following VIS routines use two-word opcodes, where the second word of the opcode (the sub-opcode) replaces the return address. .VSRP and .VDRP are assembly language routines (located in the Firmware Interface Library) that install the two-word opcodes for FORTRAN programs at run time.

VIS ROUTINES WITH TWO-WORD OPCODES

TWO-WORD OPCODE	SINGLE PRECISION ROUTINE	TWO-WORD OPCODE	DOUBLE PRECISION ROUTINE
101460 000000	VADD	105460 004002	DVADD
101460 000020	VSUB	105460 004022	DVSUB
101460 000040	VMPY	105460 004042	DVMPY
101460 000060	VDIV	105460 004062	DVDIV
101460 000400	VSAD	105460 004402	DVSAD
101460 000420	VSSB	105460 004422	DVSSB
101460 000440	VSMY	105460 004442	DVSMY
101460 000460	VSDV	105460 004462	DVSDV

The assignment of one-word opcodes and the equivalent VIS routine are listed as follows. These entry points are declared in the Parameter Input Phase of RTE generation.

VIS ROUTINES WITH ONE-WORD OPCODES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	DOUBLE PRECISION ROUTINE
101460	.VECT	105460	.DVCT *
101461	VPIV	105461	DVPIV
101462	VABS	105462	DVABS
101463	VSUM	105463	DVSUM
101464	VNRM	105464	DVNRM
101465	VDOT	105465	DVDOT
101466	VMAX	105466	DVMAX
101467	VMAB	105467	DVMAB
101470	VMIN	105470	DVMIN
101471	VMIB	105471	DVMIB
101472	VMOV	105472	DVMOV
101473	VSWP	105473	DVSWP
101474	.ERES**		
101475	.ESEG** EMA interface routines		
101476	.VSET**		
		105477	self-test

* (first word of two-word opcodes as above.)
** (used under RTE-IVB only)

VECTOR ARITHMETIC ROUTINES

TWO-WORD OPCODE	SINGLE PRECISION ROUTINE	TWO-WORD OPCODE	DOUBLE PRECISION ROUTINE
101460 000000	VADD	105460 004002	DVADD
101460 000020	VSUB	105460 004022	DVSUB
101460 000040	VMPY	105460 004042	DVMPY
101460 000060	VDIV	105460 004062	DVDIV

Calling sequence:

```
CALL subr (v1, incr1, v2, incr2, v3, incr3, #elements)
```

These routines are each nine-word instructions:

```
word 1 = first word of opcode
word 2 = second word of opcode
word 3 = address of array,           v1
word 4 = address of increment,       incr1
word 5 = address of array,           v2
word 6 = address of increment,       incr2
word 7 = address of array,           v3
word 8 = address of increment,       incr3
word 9 = address of no. of elements, #elements
```

VECTOR-SCALAR ARITHMETIC ROUTINES

TWO-WORD OPCODE	SINGLE PRECISION ROUTINE	TWO-WORD OPCODE	DOUBLE PRECISION ROUTINE
101460 000400	VSAD	105460 004402	DVSAD
101460 000420	VSSB	105460 004422	DVSSB
101460 000440	VSMY	105460 004442	DVSMY
101460 000460	VSDV	105460 004462	DVSDV

Calling sequence:

```
CALL subr (scalar, v1, incr1, v2, incr2, #elements)
```

These routines are each eight-word instructions:

```
word 1 = first word of opcode
word 2 = second word of opcode
word 3 = address of scalar,          scalar
word 4 = address of array,           v1
word 5 = address of increment,       incr1
word 6 = address of array,           v2
word 7 = address of increment,       incr2
word 8 = address of no. of elements, #elements
```

ABSOLUTE VALUE ROUTINE

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	DOUBLE PRECISION ROUTINE
101462	VABS	105462	DVABS

Calling sequence:

```
CALL subr (v1, incr1, v2, incr2, #elements)
```

These routines are each seven-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of array,           v1
word 4 = address of increment,       incr1
word 5 = address of array,           v2
word 6 = address of increment,       incr2
word 7 = address of no. of elements, #elements
```

SUM ROUTINES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	DOUBLE PRECISION ROUTINE
101463	VSUM	105463	DVSUM
101464	VNRM	105464	DVNRM

Calling sequence:

```
CALL subr (scalar, v1, incr1, #elements)
```

These routines are each six-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of scalar,          scalar
word 4 = address of array,           v1
word 5 = address of increment,       incr1
word 6 = address of no. of elements, #elements
```

DOT PRODUCT ROUTINE

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	SINGLE PRECISION ROUTINE
101465	VDOT	105465	DVDOT

Calling sequence:

```
CALL subr (scalar, v1, incr1, v2, incr2, #elements)
```

These routines are each eight-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of scalar,          scalar
word 4 = address of array,           v1
word 5 = address of increment,       incr1
word 6 = address of array,           v2
word 7 = address of increment,       incr2
word 8 = address of no. of elements, #elements
```

PIVOT ROUTINE

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	SINGLE PRECISION ROUTINE
101461	VPIV	105461	DVPIV

Calling sequence:

```
CALL subr (scalar, v1, incr1, v2, incr2, v3, incr3, #elements)
```

These routines are each ten-word instructions:

```
word  1 = opcode
word  2 = address of next instruction
word  3 = address of scalar,          scalar
word  4 = address of array,           v1
word  5 = address of increment,       incr1
word  6 = address of array,           v2
word  7 = address of increment,       incr2
word  8 = address of array,           v3
word  9 = address of increment,       incr3
word 10 = address of no. of elements, #elements
```

MAX/MIN ROUTINES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	SINGLE PRECISION ROUTINE
101466	VMAX	105466	DVMAX
101467	VMAB	105467	DVMAB
101470	VMIN	105470	DVMIN
101471	VMIB	105471	DVMIB

Calling sequence:

```
CALL subr (scalar, v1, incr1, #elements)
```

These routines are each six-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of integer scalar,  scalar
word 4 = address of array,           v1
word 5 = address of increment,       incr1
word 6 = address of no. of elements, #elements
```

MOVE ROUTINES

ONE-WORD OPCODE	SINGLE PRECISION ROUTINE	ONE-WORD OPCODE	SINGLE PRECISION ROUTINE
101472	VMOV	105472	DVABS
101473	VSWP	105473	DVSWP

Calling sequence:

```
CALL subr (v1, incr1, v2, incr2, #elements)
```

These routines are each seven-word instructions:

```
word 1 = opcode
word 2 = address of next instruction
word 3 = address of array,           v1
word 4 = address of increment,       incr1
word 5 = address of array,           v2
word 6 = address of increment,       incr2
word 7 = address of no. of elements, #elements
```

Firmware Interface Routines, .VSRP and .VDRP

.VSRP and .VDRP are firmware interface routines included in the VIS Firmware Interface Library. These routines install the two-word opcodes for FORTRAN programs at run time. (See the previous section for opcodes.) They are used only on an F-Series computer. VIS on an A-Series computer does not have two-word opcodes.

The following assembly language routine .VSRP interfaces between FORTRAN programs and the following single-precision operations:

VADD	VSUB	VMPY	VDIV
VSAD	VSSB	VSMY	VSDV

by replacing the sequence

```
JSB  VADD      with      101460
DEF  *+8                 000000
```

and so forth for the other operations.

```
ASMB,L
      HED ".VSRP" - RPL'ING OF:  VADD VSUB VMPY VDIV
*                                VSAD VSSB VSMY VSDV
      NAM .VSRP,7 12824-16001 REV.1926 790403
*
      EXT .VECT
      ENT VADD,VSUB,VMPY,VDIV,VSAD,VSSB,VSMY,VSDV
*
A     EQU 0
B     EQU 1
*
*     .VSRP REPLACES CALLS TO VADD...VSDV WITH THE APPROPRIATE
*     TWO-WORD OPCODE. IF THE MAIN OPCODE IS NOT RPL'D, IT
*     FAKES A JSB INSTEAD OF REPLACING THE ORIGINAL JSB.
*
VADD  NOP            ADDITION
      JSB COM
      OCT 000000
*
VSUB  NOP            SUBTRACTION
      JSB COM
      OCT 000020
*
VMPY  NOP            MULTIPLICATION
      JSB COM
      OCT 000040
*
VDIV  NOP            DIVISION
      JSB COM
      OCT 000060
*
VSAD  NOP            VECTOR-SCALAR ADDITION
      JSB COM
      OCT 000400
```

```
*
VSSB  NOP            VECTOR-SCALAR SUBTRACTION
      JSB COM
      OCT 000420
*
VSMY  NOP            VECTOR-SCALAR MULTIPLICATION
      JSB COM
      OCT 000440
*
VSDV  NOP            VECTOR-SCALAR DIVISION
      JSB COM
      OCT 000460
*
COM   NOP
      LDA COM,I      A = SUB-OPCODE.
      LDB COM        GET ADDRESS + 1 OF ORIGINAL JSB.
      ADB =D-2
      LDB B,I
      STA B,I        OVERLAY ORIGNAL RTN PTR WITH SUB-OPCODE.
      LDA OPCOD      A = MAIN OPCODE.
      SSA,RSS        SOFTWARE ?
      JMP SOFT       YES, SPECIAL CASE.
      ADB =D-1       NO. OVERLAY ORIGINAL JSB WITH MAIN OPCODE.
      STA B,I
      JMP B,I        GO EXECUTE THE OPCODE.
*
SOFT  STB A,I        FAKE A JSB TO THE SOFTWARE. FIRST, RTN ADDR.
      INA            THEN,
      JMP A,I         THE ENTRY.
*
OPCOD DEF .VECT+0
      END
```

The following assembly language routine .VDRP interfaces between FORTRAN programs and the following double-precision operations:

DVADD	DVSUB	DVMPY	DVDIV
DVSAD	DVSSB	DVSMY	DVSDV

by replacing the sequence

```
JSB DVADD       with      105460
DEF *+8                   004002
```

and so forth for the other operations.

```
ASMB,L
      HED ".VDRP" - RPL'ING OF:  DVADD DVSUB .... DVSMY DVSDV .
      NAM .VDRP,7 12824-16001 REV.1926 790403
*
      EXT .DVCT
      ENT DVADD,DVSUB,DVMPY,DVDIV,DVSAD,DVSSB,DVSMY,DVSDV
*
A     EQU 0
B     EQU 1
*
*     .VDRP REPLACES CALLS TO DVADD...DVSDV WITH THE APPROPRIATE
*     TWO-WORD OPCODE.  IF THE MAIN OPCODE IS NOT RPL'D, IT FAKES
*     A JSB INSTEAD OF REPLACING THE ORIGINAL JSB.
*
DVADD NOP           ADDITION
      JSB COM
      OCT 004002
*
DVSUB NOP           SUBTRACTION
      JSB COM
      OCT 004022
*
DVMPY NOP           MULTIPLICATION
      JSB COM
      OCT 004042
*
DVDIV NOP           DIVISION
      JSB COM
      OCT 004062
*
DVSAD NOP           VECTOR-SCALAR ADDITION
      JSB COM
      OCT 004402
*
DVSSB NOP           VECTOR-SCALAR SUBTRACTION
      JSB COM
      OCT 004422
*
DVSMY NOP           VECTOR-SCALAR MULTIPLICATION
      JSB COM
      OCT 004442
*
```

```
DVSDV NOP           VECTOR-SCALAR DIVISION
      JSB COM
      OCT 004462
*
COM   NOP
      LDA COM,I     A = SUB-OPCODE.
      LDB COM       GET ADDRESS + 1 OF ORIGINAL JSB.
      ADB =D-2
      LDB B,I
      STA B,I       OVERLAY ORIGINAL RTN PTR WITH SUB-OPCODE.
      LDA OPCOD     A = MAIN OPCODE.
      SSA,RSS       SOFTWARE ?
      JMP SOFT      YES, SPECIAL CASE.
      ADB =D-1      NO. OVERLAY ORIGINAL JSB WITH MAIN OPCODE.
      STA B,I
      JMP B,I       GO EXECUTE THE OPCODE.
*
SOFT  STB A,I       FAKE A JSB TO THE SOFTWARE.  FIRST, RTN ADDR.
      INA           THEN,
      JMP A,I        THE ENTRY.
*
OPCOD DEF .DVCT+0
      END
```

Adding Your Own EMA Routines

You may create your own routines involving non-EMA arrays either in software or microcode. To extend your routines to include EMA arrays, you may use the VIS firmware interface routine .WCOM. .WCOM is included in the Firmware Interface Library. This program calls the non-EMA VIS routine using the EMA arrays. Note that .WCOM can only be used with three arrays or less.

The calling sequence for your VIS program must be in the assembly language form:

```
JSB subr                 subr is your non-EMA VIS routine
DEF return address       = current address +2+i+2*j
DEF misc. parameter1
     .
     .
DEF misc. parameteri     "i" misc. parameters
DEF v1                   "j" arrays in EMA
DEF incr1                with increments
     .
     .
DEF vj
DEF incrj
DEF #elements
```

which can be generated from FORTRAN with the following call:

```
CALL subr (misc.1,...,misc.i,v1,incr1,...,vj,incrj, #elements)
```

A maximum of 16 parameters (any combination) can be in the parameter list.

All the parameters have the same definitions as stated in Chapter 8, which describes the general calling sequence. Therefore, when creating EMA routines, only the arrays can be in EMA and all other arguments must be in non-EMA. The EMA routine calls .WCOM, which interfaces between the EMA and the non-EMA routine and has the following form:

```
ASMB,L
      NAM name,7              name is the EMA routine
      ENT name
      EXT subr,.WCOM          subr is the equivalent non-EMA routine
name  NOP
      JSB .WCOM               .WCOM is the interface routine
      DEF subr + 0            + 0 guarantees a direct address
      BYT i,j                 i= # of misc. parameters,j= #arrays
      ABS (1024/#words per element)-1 + bit 13
      END
```

When bit 13 = 0:

After calling “subr”, .WCOM returns directly to the program that called “name”. See the integer vector add example.

When bit 13 = 1:

After each call to “subr”, .WCOM calls the subroutine whose entry point must follow the ABS instruction in “name”. The form of this call is:

```
JSB  <ABS + 1>
DEF  <first parameter DEF in ”subr” call>
DEC  <0 if last call; –1 if not last call>
```

See the integer vector sum example.

Example 1: Integer Vector Add Instruction Example

NON-EMA VIS ROUTINE:

Generate a VIS integer add routine for non-EMA arrays. Add integer elements from two arrays and place results into a third array.

```
      SUBROUTINE VIADD (I,IX,J,JX,K,KX,N)
C
C     I = OPERAND ARRAY    IX = INCREMENT FOR I
C     J = OPERAND ARRAY    JX = INCREMENT FOR J
C     K = RESULT ARRAY     KX = INCREMENT FOR K
C     N = NO. OF ELEMENTS TO ADD
C
      INTEGER I(1), J(1), K(1)
      IA = 1
      JA = 1
      KA = 1
      DO 10 L = 1,N
      K(KA) = I(IA) + J(JA)
      IA = IA + IX
      JA = JA + JX
      KA = KA + KX
10    CONTINUE
      RETURN
      END
```

Subroutine WIADD is the EMA routine equivalent to VIADD. It has the same calling sequence as VIADD except that I, J, and K must be EMA arrays.

```
ASMB,L
      NAM WIADD,7          PROGRAM TYPE = 7
      ENT WIADD
      EXT VIADD,.WCOM      EXTERNAL ROUTINES
WIADD NOP                  EMA VIS ROUTINE FOR INTEGER ADD
      JSB .WCOM            CALL EMA INTERFACE ROUTINE
      DEF VIADD+0
      BYT 0,3              0 MISC. PARAMETERS,3 ARRAYS
      ABS 1023+0           (1024/1 WORD PER ELEMENT)–1 + 0
      END
```

Each integer array element is one word. Bit 13 is not set, so return from .WCOM normally.

Example 2: Integer Vector Sum Instruction Example

NON-EMA VIS ROUTINE:

Generate a VIS integer sum routine for non-EMA arrays that calculates the sum of integer elements.

```
      SUBROUTINE VISUM (ISUM, I, INCR, N)
C
C     ISUM = INTEGER SUM
C     I    = INTEGER ARRAY     INCR = INCREMENT FOR I
C     N    = NO. OF ELEMENTS TO ADD
C
      INTEGER ISUM, I(1)
      IA = 1
      ISUM = 0
      DO 10 L = 1,N
      ISUM = ISUM + I(IA)
      IA = IA + INCR
10    CONTINUE
      RETURN
      END
```

Subroutine WISUM is the EMA routine equivalent to VISUM. WISUM has the same calling sequence except that Array I must be in EMA.

```
ASMB,L
      NAM WISUM,7          PROGRAM TYPE = 7
      ENT WISUM
      EXT VISUM,.WCOM      EXTERNAL ROUTINES
A     EQU 0                A REGISTER
WISUM NOP                  EMA VIS ROUTINE FOR INTEGER SUM
      JSB .WCOM            CALL EMA INTERFACE ROUTINE
      DEF VISUM+0
      BYT 1,1              1 MISC. PARAMETER,1 ARRAY
      ABS 1023+20000B      (1024/1 WORD PER ELEMENT)-1 + BIT 13 SET
*
EACH  NOP           ENTRY AFTER EACH .WCOM CALL TO VISUM
      DLD EACH,I    A = ADDR OF PARAMETER LIST
*                   B = LAST TIME FLAG
      STB LAST      SAVE LAST TIME FLAG
      ISZ EACH
      ISZ EACH
      LDA A,I       A = ADDR OF PARAMETER "ISUM"
      LDB A,I       B = SUM FROM CURRENT VISUM CALL
      ADB SUM       B + SUM OF PREVIOUS CALLS
      STB SUM       = NEW TOTAL SUM
      ISZ LAST      SKIP IF NOT LAST TIME
      RSS
      JMP EACH,I    NOT LAST TIME, JUST EXIT
      STB 0,I       SET ISUM = TOTAL SUM
      CLA           SET RUNNING SUM = 0 FOR NEXT TIME
      STA SUM
      JMP EACH,I
*
SUM   NOP           RUNNING SUM
```

```
LAST  NOP           = -1 IF NOT LAST TIME
*                   =  0 IF LAST TIME
      END
```

.WCOM calls the non-EMA routine, VISUM, repeatedly to obtain the sum of each group of EMA array elements. Because bit 13 is set, .WCOM returns to WISUM with the sum from the current VISUM call and a last time flag. WISUM must then take that sum and add it to the running total, SUM. If the last time flag, FLAG, is 0, then WISUM clears the running sum, stores the final sum and exits. If FLAG is −1, then EACH returns to .WCOM to get the next sum.

Error Messages

The format for error messages displayed on LU 6 is:

name : *NN XX @address*

where:

name = user program name
NN XX = error code
@address = address from within a VIS interface routine

ERROR CODE	EXPLANATION	ACTION
20 EM	An array is specified with incorrect subscripts: negative subscripts, negative dimensions, subscript < lower bound.	Check array declarations and array subscripts.
21 EM	MSEG in the $EMA directive is not specified correctly.	Specify MSEG = (2^n)-1 where n is the maximum number of EMA arrays used in a VIS instruction.
22 VI	The program is not an EMA program. An EMA VIS routine is called instead of a non-EMA VIS routine.	Check $EMA directive for correctness. Check VIS routine calls.

10

Decimal String Arithmetic Subroutines

The Hewlett-Packard Decimal String Arithmetic Routines (%DECAR) is a group of subroutines that provide solutions to business applications for users of Hewlett-Packard FORTRAN, BASIC, and Assembler Programming languages. Routines in the Decimal String Arithmetic Package perform tasks such as:

- Arithmetic functions performed on decimal data strings. Strings can be as long as desired.
- Code conversion for data manipulation.
- Editing for the preparation of output in special formats including insertion of commas, decimal points, dollar signs, minus signs, asterisks, and zero suppression.

Using the DCAR Routines

The Decimal String Arithmetic Routines are executed through a calling sequence from either BASIC, FORTRAN, or Assembly Language programs. You select the desired routine by using the routine name in the calling sequence. Parameters accompanying the subroutine call control subroutine operation. Arithmetic operations performed by the routines are performed using string variables. String variables are created by defining a one dimensional integer array. ASCII characters are then loaded into the array (using the SPUT subroutine, for example). The number of string characters stored in the array depends upon the format chosen by you for the data.

All arithmetic performed by arithmetic routines in the package is done using integer numbers (without fractions). For example, rather than deal in dollars and cents when multiplying monetary values, you'll deal in cents only, as:

$350.56 = 35056 cents

Later the result of arithmetic operations can be output with leading dollar signs and decimal points inserted by the SEDIT routine. A decimal number used in an arithmetic calculation using one of the arithmetic routines can be as long as desired. You can process the entire string defined in the array or any smaller substring within the array.

DCAR Data Formats

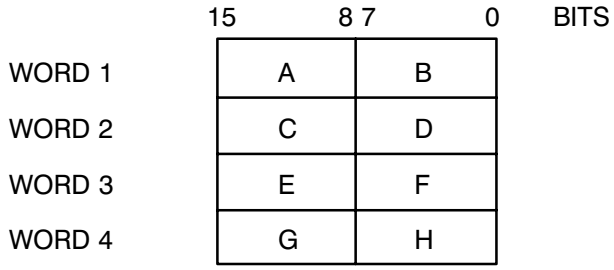
Data is stored in several different formats in integer arrays, depending upon the requirements of the Decimal Arithmetic Routines and the user's needs. Data can be stored in one format into integer arrays, using the appropriate subroutine, and then converted into a different format using the conversion routines supplied as part of the package.

A2 Format

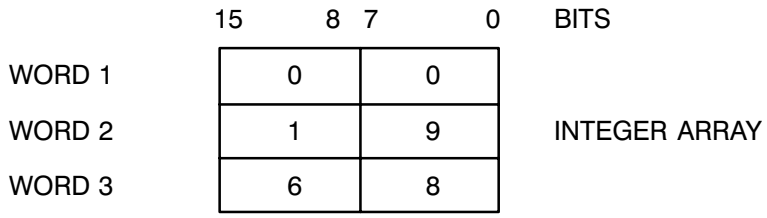
Character strings stored in A2 format are stored two characters per 16 bit computer word. The characters are represented in 8-bit ASCII code; for example, to reserve space in memory for an 8-character string, the user must define an integer array four words in length. In FORTRAN, arrays are defined by a DIMENSION statement:

```
DIMENSION IX(4)
```

An 8-character string is stored into the integer array, IX, in the following manner:



If a number is stored in A2 format (two ASCII digits per computer word), then the sign of the number (indicating whether it is positive or negative) is indicated in the rightmost digit of the string. Positive numbers are indicated by no sign at all. For example, the number 001968 is stored in an integer array as follows:

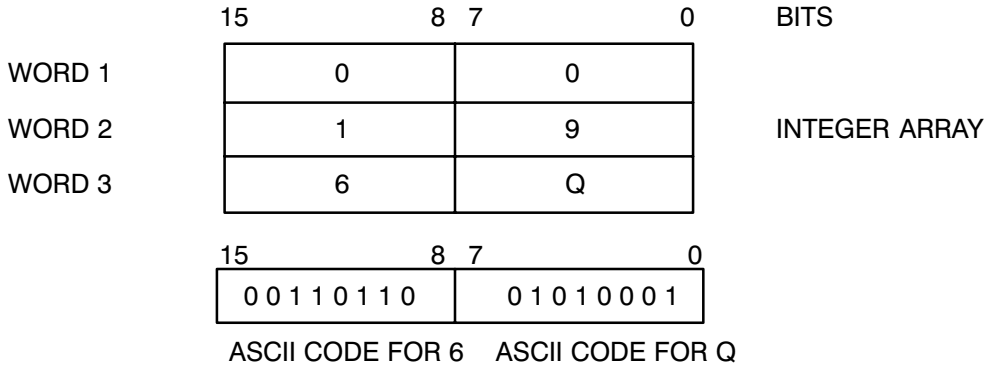


If a substring number has a negative sign, the rightmost character of the string must be represented as an 11 zone character. For example, if the rightmost character of a negative number is a 0, then the zero is changed to a minus sign to reflect the negative sign of the number. A rightmost character equal to 1 is changed to a J, and so on. Table 10-1 below shows the zoned character which must appear as the rightmost digit of a negative string, depending upon the value of the rightmost digit of the string.

Table 10-1. Zoned Characters for Negative Strings

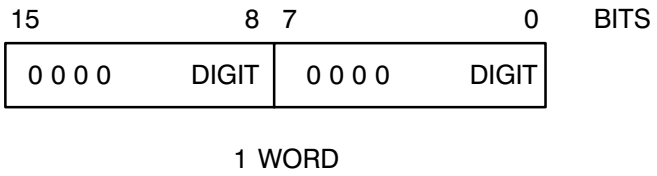
If the sign of the substring is negative and the rightmost digit is a:	The programmer must represent the rightmost digit as a:
0	–
1	J
2	K
3	L
4	M
5	N
6	O
7	P
8	Q
9	R

According to Table 10-1, the string −001968 is represented in an integer array as 00196Q:



D2 Format

The D2 format is used to store numbers (and only numbers) in memory, and consists of two digits per 16-bit computer word:

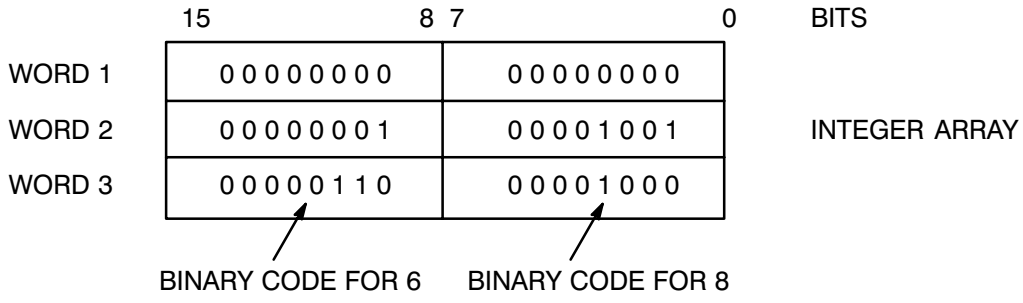


Unlike A2 format, each number is represented in binary code (as opposed to ASCII code for A2), the number is right justified in the appropriate half-word (eight bits), and unused bits are set to zero. Table 10-2 shows the binary code for the digits 0 through 9.

Table 10-2. Binary Representation of Decimal Digits

Decimal Digit	Binary Representation
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111

For example, the number 001968 is stored in an integer array in D2 format as follows:

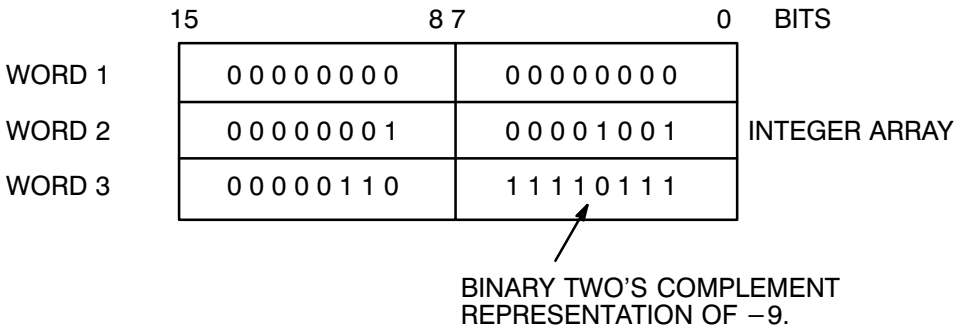


If a number is stored in D2 format, the sign of the number is indicated by the rightmost digit. Positive numbers are indicated by no sign at all. For example, the positive number 001968 is stored as shown in the previous figure. If a number has a negative sign, the negative number is indicated in the rightmost digit. If the rightmost digit of a negative number is a 0, the user must represent the rightmost digit as a −1. A rightmost character equal to 1 is changed to −2 to reflect the negative sign, and so on. Table 10-3 shows the digit which must appear as the rightmost digit of a negative number, depending upon the value of the rightmost digit of the number.

Table 10-3. Rightmost Digit for Negative Numbers

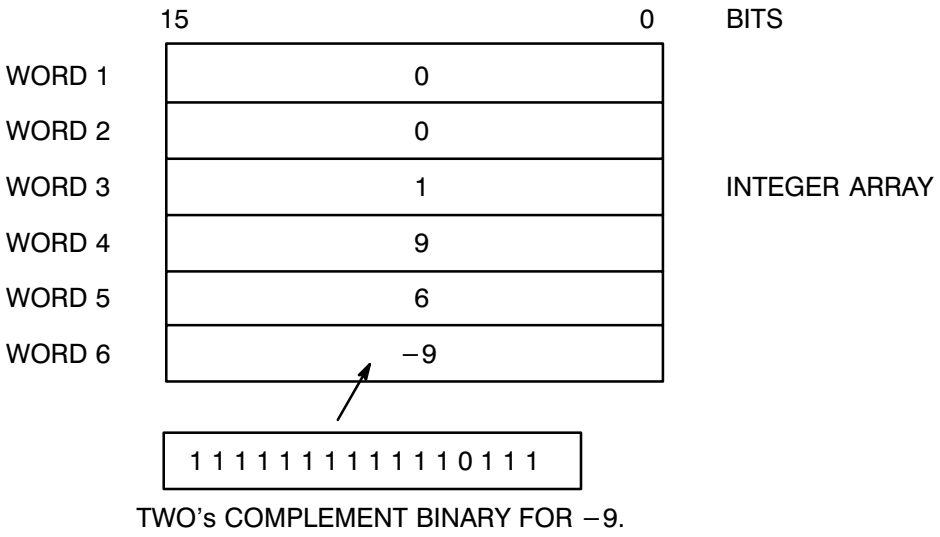
If the sign of the number is negative and the rightmost digit is a:	The rightmost digit of the number is represented as:
0	−1
1	−2
2	−3
3	−4
4	−5
5	−6
6	−7
7	−8
8	−9
9	−10

For example, the negative number −001968 is represented in an integer array as:



D1 Format

D1 format is the same as D2 format except that one digit is stored in one computer word. Negative numbers are represented in the same way as in D2 format (the rightmost digit of the number is changed according to Table 10-3). For example, the number −001968 would be stored in six elements (one word per element) of an integer array as follows:



String Utilities Routines

JSCOM, Substring Character Compare

JSCOM, a function subprogram that can be used in any arithmetic expression, compares two variable length data substrings in A2 format according to the ASCII collating sequence and sets the result to a negative number, zero, or a positive number.

```
JSCOM (jstr,jbeg,jend,kstr,kbeg,ierr)
```

where:

jstr names a one dimensional integer string array defined in a DIMENSION statement. This array contains the first data field to be compared, in A2 format, two characters per word.

jbeg is an integer constant, integer variable, or integer expression defining the position of the first character in *jstr* to be compared (beginning of substring).

jend is an integer constant, integer variable, or integer expression defining the position of the last character in *jstr* to be compared (end of substring). *jend* must be greater than or equal to *jbeg*.

kstr names a one dimensional integer string array defined in a dimension statement. This array contains the second data field to be compared, in A2 format, two characters per word.

kbeg is an integer constant, integer variable, or integer expression defining the position of the first character in *kstr* to be compared (beginning of substring).

ierr is an integer variable used as an error indicator. The value of *ierr* following execution of JSCOM indicates whether an invalid character was encountered.

Error: If any character in *jstr* or *kstr* to be compared is not a valid printable ASCII character, *ierr* is set to the position of the current character in *jstr*, and JSCOM is set to one; otherwise, *ierr* remains unchanged.

Example:

```
      DIMENSION ITEMA (5), ITEMB (6)
      IERR=0
       IF (JSCOM(ITEMA,1,10,ITEMB,3,IERR))1,2,3
1     ITEMA substring is less than ITEMB substring
2     ITEMA substring is equal to ITEMB substring
3     If (IERR)5,4,5
4     ITEMA substring is greater than ITEMB substring
      .
      .
      .
ITEMA          0001335689
ITEMB          000001335791
```

ITEMA, from positions 1 through 10, is compared character by character with ITEMB, positions 3 through 12. If the ITEMA field is less than the ITEMB field, control goes to statement 1.

If the ITEMA field is equal to the ITEMB field, control goes to statement 2. If the ITEMA field is greater than the ITEMB field or if an illegal character was encountered, control goes to statement 3, where a test may be made for the error condition.

Comments: The collating sequence used in the comparison given in Appendix A is in ascending order and constitutes the entire set of valid ASCII characters.

Corresponding characters in *jstr* and *kstr* are compared logically according to the collating sequence given in Appendix A. Comparison starts with the *jbeg* and *kbeg* positions and proceeds from left to right. The comparison is finished with the first pair of characters that do not match, or when the character at *jstr* (*jend*) has been compared.

JSCOM is set when the comparison terminates according to the following:

JSCOM	Result of Comparison
− (minus)	*jstr* substring is less than *kstr* substring
0 (zero)	*jstr* substring is equal to *kstr* substring
+ (plus)	*jstr* substring is greater than *kstr* substring

Note JSCOM does not set, test, and reset *ierr*.

SFILL, Substring Fill

SFILL fills a specified area in a substring array with a specified character.

```
CALL SFILL (jstr, jbeg, jend, jcd)
```

where:

jstr names a one dimensional integer string array containing the area of the substring to be filled. The array must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable, or integer expression defining the position of the first character in *jstr* to be filled (beginning of substring).

jend is an integer constant, integer variable, or integer expression defining the position of the last character in *jstr* to be filled (end of substring). *jend* must be greater than or equal to *jbeg*.

jcd is an integer constant, integer variable, or integer expression containing the ASCII code for the fill character.

Errors: None.

Example:

```
DIMENSION IPRIN (13)
JCB=000052B
CALL SFILL (IPRIN,9,15,JCD)
```

Before:

	Word	1		2		3		4		5		6		7		8		9		10		11		12		13	
IPRIN	Data	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
	String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

After:

	Word	1		2		3		4		5		6		7		8		9		10	
IPRIN	Data	0	1	2	3	4	5	6	7	*	*	*	*	*	*	*	5	6	7	8	9
	String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

The array IPRIN is filled with asterisks from positions 9 through 15. To fill the array IPRIN with blanks, the following code and parameters are specified.

```
ICD = 000040B
CALL  SFILL(IPRIN<1,26,ICD)
```

SGET, Substring Get

SGET gets a specified character from a substring.

```
CALL SGET (jstr,j,jhold)
```

where:

jstr names a one dimensional integer string array containing the area of the requested character. The array must be defined in a DIMENSION statement.

j is an integer constant, integer variable, or integer expression defining the position of the specified character in *jstr*.

jhold is an interger variable or integer expression containing the specified character, zero filled, right justified (after SGET is executed).

Errors: None.

Comments: The character in position J of *jstr* is returned in *jhold*, right justified, zero filled.

Example:

```
DIMENSION IPRIN(10)
  .
  .
  .
CALL SGET (IPRIN,6,NCHAR)
```

Before:

IPRIN

Word	1		2		3		4		5		6		7		8		9		10	
Data	0	2	4	6	8	3	5	7	9	A	B	C	D	E	F	G	H	I	J	K
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

After:

IPRIN unchanged
NCHAR – 000063_8 (ASCII 3)

SMOVE, Substring Move

SMOVE moves data from one string array to another

```
CALL SMOVE (jstr, jbeg, jend, kstr, kbeg)
```

where:

jstr names a one dimensional integer string array containing the data to be moved. The array must be defined in a DIMENSION statement. The data may be any format that is two characters per word.

jbeg is an integer constant, integer variable, or integer expression defining the position of the first character to be moved (beginning of substring).

jend is an integer constant, integer variable, or integer expression defining the position of the last character in *jstr* to be moved, (end of substring). *jend* must be greater than or equal to *jbeg*.

kstr names a one dimensional integer array, in any format that is two characters per word, into which the data from *jstr* is moved. It must be defined in a DIMENSION statement.

kbeg is an integer constant, integer variable, or integer expression defining the first character position in *kstr* to which data from *jstr* is moved (beginning of substring).

Errors: None.

Comments: Each character in *jstr* beginning with position *jbeg* and ending with *jend* is moved to *kstr* beginning at position *kbeg*.

Example:

```
DIMENSION  ICARD(80),ILINE(120)
I =  2
J = 13
K = 10
CALL  SMOVE(ICARD,I,J,ILINE,K)
```

Before:

	Word	1		2		3		4		5		6		7		8		9		10		11		12		13	
ICARD	Data	0	1	X	Y	Z	A	B	C	0	0	0	5	7	8	1	3	7	6	5	Δ	Δ	0	0	7	3	9
	String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

	Word	1		2		3		4		5		6		7		8		9		10		11		12		13	
ILINE	Data	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
	String	1	2	3	4	A	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

After:

ICARD No Change

	Word	1		2		3		4		5		6		7		8		9		10		11		12		13	
ILINE	Data	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	1	X	Y	Z	A	B	C	0	0	0	5	7	Δ	Δ	Δ	Δ	Δ
	String	1	2	3	4	A	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

The field in the array ICARD beginning at character 2 and ending with character 13 as defined by the variables I and J, is moved to ILINE starting with character 10 as defined by the variable K.

In all, 12 characters were moved.

SPUT, Substring Put

SPUT puts a specified character in a specified position of a substring.

```
CALL SPUT (jstr,j,jhold)
```

where:

jstr names a one dimensional integer string array into which the requested character is to be placed. The array must be defined in a dimension statement.

j is an integer constant, integer variable, or integer expression defining the position in *jstr* where the specified character is to be placed.

jhold is an integer variable, or integer expression, containing the character to be transferred, right justified and zero filled.

Errors: None.

Comments: *jhold* remains unchanged after the transfer.

Example:

```
DIMENSION IPRIN(5)
NCHAR=000060
CALL SPUT(IPRIN,7,NCHAR)
```

Before:

IPRIN

Word	1		2		3		4		5	
Data	0	2	4	6	8	1	5	3	7	9
String	1	2	3	4	5	6	7	8	9	10

NCHAR = 000060

After:

IPRIN

Word	1		2		3		4		5	
Data	0	2	4	6	8	1	0	3	7	9
String	1	2	3	4	5	6	7	8	9	10

NCHAR = 000060

SZONE, Substring Zone

SZONE finds the zone punch of a character, sets a code to indicate what the zone is, and provides a new zone.

```
CALL SZONE(jstr,jbeg,nez,noz)
```

where:

jstr names a one dimensional integer string array containing the character whose zone is to be tested and modified. It must be defined in a DIMENSION statement. The character must be in A2 format, two characters per word.

jbeg is an integer constant, integer variable, or integer expression defining the position of the character in *jstr* to be tested and modified.

nez is an integer constant, integer variable, or integer expression specifying a code for the new zone.

noz is an integer variable which is set to a code indicating the original zone of the character.

Errors: None.

Comments: First, the zone of the character *jbeg* is retrieved and *noz* is set as follows:

NOZ	Original Zone	Character
1	12-zone	A−I
2	11-zone	−,J−R
3	0-zone	/,S−Z
4	no zone	+,0−9
more than 4		special

A new zone is then inserted as specified by *nez* as follows:

NEZ	New Zone	Character
1	12-zone	A−I
2	11-zone	−,J−R
3	0-zone	/,S−Z
4	no zone	+,0−9
more than 4		special

No change is made to the zone when the character is a special character.

The minus sign or hyphen (- or an 11 zone punch) is not treated as a special character. It is assumed to be a negative zero. The only modification that can be made to a − (minus, or negative zero) is to change it to an unsigned zero with a no zero code. Zero (0) and + (plus) are treated as no-zone characters; however, the only modification that can be made to a zero (0) or plus (+) is to change it to a minus (−) upon request for an 11 zone punch. Plus is changed to zero upon request for a no-zone punch. Upon request for any other zero punch, zero (0) and plus (+) remain unchanged. These are the only exceptions among the special characters.

Example:

```
DIMENSION ICHAR(80)
CALL NZONE (ICHAR,8,1,I)
```

Before:

```
ICHAR(8) = R (11–9 punch)
I=0
```

After:

```
ICHAR(8) = I (12–9 punch)
I=2
```

Table 10-4. SZONE Conversion

Old Character	New Character NEZ					NOZ
	1	2	3	4	5	
A	A	J	/	1	A	1
B	B	K	S	2	B	1
C	C	L	T	3	C	1
D	D	M	U	4	D	1
E	E	N	V	5	E	1
F	F	O	W	6	F	1
G	G	P	X	7	G	1
H	H	Q	Y	8	H	1
I	I	R	Z	9	I	1
J	A	J	/	1	J	2
K	B	K	S	2	K	2
L	C	L	T	3	L	2
M	D	M	U	4	M	2
N	E	N	V	5	N	2
O	F	O	W	6	O	2
P	G	P	X	7	P	2
Q	H	Q	Y	8	Q	2
R	I	R	Z	9	R	2
/	A	J	/	1	/	3
S	B	K	S	2	S	3
T	C	L	T	3	T	3
U	D	M	U	4	U	3
V	E	N	V	5	V	3
W	F	O	W	6	W	3
X	G	P	X	7	X	3
Y	H	Q	Y	8	Y	3
Z	I	R	Z	9	Z	3
0	0	–	0	0	0	4
+	+	–	+	0	+	4
–	–	–	–	0	–	2
1	A	J	/	1	1	4
2	B	K	S	2	2	4
3	C	L	T	3	3	4
4	D	M	U	4	4	4
5	E	N	V	5	5	4
6	F	P	W	6	6	4
7	G	P	X	7	7	4
8	H	Q	Y	8	8	4
9	I	R	Z	9	9	4
special character	same special character					5

String Arithmetic Routines

SADD, Substring Decimal Add

SADD adds two character substrings of arbitrary length and stores the result in the second substring.

```
CALL SADD(jstr,jbeg,jend,kstr,kbeg,kend,ierr)
```

where:

jstr names a one dimensional integer string array containing the first character substring to be added. The contents of the array must be in A2 format, two characters per word. *jstr* must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable or integer expression indicating the position of the first character in *jstr* to be added (beginning of substring).

jend is an integer constant, integer variable or integer expression giving the position of the last character in *jstr* to be added (end of substring). It must be greater than or equal to *jbeg*.

kstr names a one dimensional integer string array containing the data to which the data in *jstr* is added. It will contain the result following addition in A2 format, two characters per word. *kstr* is defined in a DIMENSION statement.

kbeg is an integer constant, integer variable or integer expression giving the position of the first character in *kstr* (beginning of substring).

kend is an integer constant, integer variable, or integer expression giving the position of the last character in *kstr* (end of substring). It must be greater than or equal to *kbeg*.

ierr is an integer variable used as an error indicator. The value *ierr* following execution of SADD indicates whether arithmetic overflow occurred; if an overflow occurred, *ierr* is set equal to *kend*. The programmer must initialize, test and reset *ierr*.

Errors: *ierr* is set when:

- There was arithmetic overflow; if *kstr* is not large enough to contain the sum, the *kstr* field is filled with 9s and *ierr* is made equal to *kend*.
- *jstr* is longer than *kstr*, neither field is altered, but *ierr* is set equal to *kend*.
- Either substring of *jstr* and/or *kstr* does not contain all ASCII numeric characters (except the rightmost character), or *ierr* is set equal to −1.

Comments: *jstr* and *kstr* can be any length up to the maximum space available; *kstr* must, however, be greater than *jstr* in order to avoid an overflow condition.

The characters in *jstr* and *kstr* must all be ASCII numeric, 0−9, except the rightmost character, JLAST or KLAST, which may be an 11 zone character, indicating a negative digit.

Note SADD does not initialize, test, or reset *ierr*.

Example:

```
DIMENSION  IFLDA(8),IFLDB(10)
IE = 0
CALL  SADD(IFLDA,1,15,IFLDB,1,20,IE)
```

Before:

IFLDA

Word	1		2		3		4		5		6		7		8	
Data	Δ	Δ	Δ	Δ	Δ	3	7	1	4	1	0	0	2	5	1	6
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

IFLDB

Word	1		2		3		4		5		6		7		8		9		10	
Data	Δ	Δ	Δ	1	5	3	4	6	7	8	9	3	5	0	0	0	9	8	7	6
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

IE = 0(zero)

After:

IFLDA No Change

IFLDB

Word	1		2		3		4		5		6		7		8		9		10	
Data	0	0	0	1	5	3	4	6	7	9	3	0	6	4	1	1	0	1	2	7
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

IE = 0(zero)

The data field IFLDA is added to IFLDB and the result placed in IFLDB. The error indicator IE is unchanged since no overflow occurred.

Note At the conclusion of SADD, the rightmost character in *kstr*, *kend*, carries the sign of the sum. Thus, if the sum is negative, the rightmost character will be an 11 zone character. However, if the sum is zero, the rightmost character may be either 0 (zero) or − (minus sign).

SDIV, Substring Decimal Division

SDIV divides arbitrary length substring *kstr* by another such substring *jstr*, placing the quotient and the remainder in *kstr*.

```
CALL SDIV (jstr,jbeg,jend,kstr,kbeg,kend,ierr)
```

where:

jstr names a one dimensional integer string array used as the divisor. It must contain data in A2 format, two characters per word. *jstr* must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable, or integer expression giving the position of the first digit of *jstr* (beginning of substring).

jend is an integer consant, integer variable, or integer expression giving the position of the last digit of *jstr* (end of substring). *jend* must be greater than or equal to *jbeg*.

kstr names a one dimensional integer string array used as the dividend. It will contain the quotient and the remainder, extended to the left, following division. The data is in A2 format, two digits per word.

kbeg is an integer constant, integer variable, or integer expression giving the position of the first digit of *kstr* (beginning of substring).

kend is an integer constant, integer variable, or integer expression giving the position of the last digit of *kstr* (end of substring). It must be greater than or equal to *kbeg*.

ierr is an integer variable used as an error indicator. After SDIV is executed, it indicates whether division by zero was attempted, or whether the field *kstr* was too small to contain quotient and remainder.

Errors: *ierr* is set in one of four circumstances:

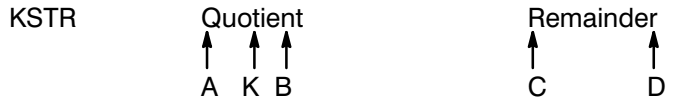
1. If division by zero was attempted, *ierr* is set to *kend*.
2. If either substring of *jstr* and/or *kstr* does not contain all ASCII numerics, except the rightmost character, *ierr* is set to −1.
3. If insufficient space was allocated to extend *kstr* to the left, *ierr* is set to *kend*.
4. If the length of the divisor is greater than the length of the dividend, *ierr* is set to *kend*.

In all above cases, neither *kstr* nor *jstr* is modified.

Comments: *jstr* and *kstr* can be any length up to the maximum space available. Sufficient space must be allocated to *kstr* to allow for its extension. At least (*kend-kbeg*+1) + 2(*jend-jbeg*+1) positions must be provided between the beginning of *kstr* and the first dividend position *kbeg*. For instance, if *jend*=6, *jbeg*=2 (the divisor has 5 positions) and the dividend has 7 positions, then *kbeg* must be at least 18 positions from the beginning of *kstr*.

Note SDIV does not initialize, test, or reset the error indicator, *ierr*.

The quotient and the remainder will both be located in the extended *kstr* field according to the diagram below:



A is the position: KBEG-(JEND-JBEG+1)
K is the position: KBEG
B is the position: KEND-(JEND-JBEG+1)
C is the position: KEND-(JEND-JBEG)
D is the position: KEND

The SDIV arithmetic is decimal arithmetic using whole numbers only, with no decimal point alignment. Therefore, the numbers should have an assumed decimal point following the rightmost digit.

See also "Short-String Routine" in the SMPY description.

Example: Divide 7943074 by −42135

```
DIMENSION  IDIVR(3),IDVD(12)
IE=0
CALL  SDIV(IDIVR,2,6,IDVD,18,24,IE)
```

Before:

IDIVR

Word	1		2		3	
Data	A	4	2	1	3	N
String	1	2	3	4	5	6

Note: 11 – zone 5, (N) stands for – 5 in A2 formats.

IDVD

Word	1		2		3		4		5		6		7		8		9		10		11		12	
Data	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0	0	0	7	9	4	3	0	7	4
String	1	2	3	4	A	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

After:

IDIVR No Change

IDVD

Word	1		2		3		4		5		6		7		8		9		10		11		12	
Data	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	8	Q	2	1	6	9	4
String	1	2	3	4	A	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Quotient Remainder

Answer: – 188, remainder 21694

IE = 0(zero)

The numeric field IDVR was divided by the numeric field IDIVD with the quotient and remainder placed in IDVD. The field IDVD has been extended 17 places to the left and filled with zeros. The remainder is in the 5 low order positions of IDVD, the quotient in positions 13 through 19.

Note See SMPY for a routine that enables you to provide a shorter string for the dividend.

SMPY, Substring Decimal Multiply

SMPY multiplies two character data substrings and places the result in the second substring. The substrings may be any length.

```
CALL SMPY (jstr,jbeg,jend,kstr,kbeg,kend,ierr)
```

where:

jstr names a one dimensional integer string array containing the data to be multiplied. The array must be defined in a DIMENSION statement. The data is in A2 format, two characters per word.

jbeg is an integer constant, integer variable, or integer expression defining the position of the first character in *jstr* to be multiplied (beginning of substring).

jend is an integer constant, integer variable, or integer expression defining the position of the last character in *jstr* to be multiplied (end of substring). *jend* must be greater than or equal to *jbeg*.

kstr names a one dimensional integer string array containing the multiplicand. After multiplication, it will contain the product extended to the left. The data, before and after multiplication, is in A2 format, two characters per word.

kbeg is an integer constant, integer variable, or integer expression defining the position of the first character in the multiplicand (beginning of substring).

kend is an integer constant, integer variable, or integer expression defining the position of the last character in both the multiplicand and the product (end of substring). *kend* must be greater than or equal to *kbeg*.

ierr is an integer variable used as an error indicator. It is set to *kend* when *kstr* is not large enough to contain the product.

Errors: If *kstr* does not have enough positions to allow for its extension to the left in order to receive the product, *ierr* is set equal to *kend*. The subroutine terminates at that point. If *jstr* or *kstr* contain a non-numeric or non-blank character in other than the last position, *ierr* is set to −1. In either case, neither *jstr* nor *kstr* is modified. The user is responsible for testing and resetting *ierr*.

Comments: The data is converted from ASCII to numeric within SMPY.

jstr and *kstr* can be any length up to the maximum space available. Sufficient space must be allocated to *kstr* to allow for its extension. At least (*kend-kbeg*+1) + 2(*jend-kbeg*+1 positions must be provided between the beginning of *kstr* and the first multiplicand position *kbeg*. That is, if *jstr* has five positions (for example, *jend*=6,*jbeg*=2) and the multiplicand has 7 positions, then *kbeg* must be at least 18 positions from the beginning of *kstr*; *kbeg* would be greater than or equal to 18.

The SMPY arithmetic is decimal arithmetic using whole numbers only.

The product of SMPY is located in *kstr* beginning at position *kbeg* and ending at position *kend*.

Example:

```
DIMENSION  MULTR(3)0,MLCND(13)
IE = 0
CALL  SMPY(MULTR,2,6,MLCND,18,24,IE)
```

Before:

MULTR

Word	1		2		3	
Data	0	0	1	5	4	0
String	1	2	3	4	5	6

MLCND

Word	1		2		3		4		5		6		7		8		9		10		11		12		13	
Data	V	W	X	Y	Z	C	D	E	F	G	H	I	J	K	L	R	S	0	8	6	5	8	3	2	A	B
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

IE = 0

After:

MULTR No Change

MLCND

Word	1		2		3		4		5		6		7		8		9		10		11		12		13	
Data	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	3	3	3	8	1	2	8	0	A	B
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

IE = 0

The numeric data fields MULTR and MLCND are multiplied and the result placed in MLCND. The field MLCND has been extended to the left 17 characters and filled with zeros. IE has not been changed. The result starts the number of positions in MULTR to the left of *kbeg*, but the field was extended and zero filled 2*(*jend-jbeg*+1) + (*kend-kbeg*+1) positions.

Short-String Routine

If you do not wish to provide such a long string of *kstr*, you can use the following instructions with SMPY:

```
MAINLINE
.
.
.
N1=2+(JEND-JBEG+1)+(KEND-KBEG+1)+1
N2=N1+(KEND-KBEG)
CALL  SMOVE(LSTR,KBEG,KEND,STEMP,N1)
CALL  SMPY(JSTR,JBEG,JEND,KTEMP,N1,N2,IERR)
N3=N1–(JEND-JBEG+1)
N4=KBEG-(JEND-JBEG+1)
CALL  SMOVE  (KTEMP,N3,N2,KSTR,N4)
.
.
.
```

kstr must be dimensioned, and at least (*jend-jbeg*+1) positions must be provided between the beginning of *kstr* and the first multiplicand position, *kbeg*, to allow for the extension of the product. That is, if *jstr* has 5 positions (for example, *jend*=6, *jbeg*=2), and the multiplicand has 7 positions, then *kbeg* must be greater than or equal to 6. KTEMP is a temporary buffer to which the multiplicand is moved to allow for its expansion during SMPY. It must be dimensioned by the user, and must consist of at least 2 (*kend-kbeg*+1) + 2(*jend-jbeg*+1) positions.

Note The short-string routine also can be used with SDIV.

Example:

```
DIMENSION  MULTR(3),MLCND(6),MTEMP(12)
IE=0
JBEG=2
JEND=6
KBEG=6
KEND=12
N1=2*(JEND-JBEG+1)+(KEND-KBEG+1)+1
N2=N1+(KEND-KBEG)
CALL  SMOVE(MLCND,KBEG,KEND,MTEMP,N1)
CALL  SMPY(MULTR,JBEG,JEND,MTEMP,N1,N2,IE)
N3=N1-(JEND-JBEG+1)
N4=KBEG-(JEND-JBEG+1)
CALL  SMOVE(MTEMP,N3,N2,MLCND,N4)
```

Before:

MULTR

Word	1		2		3	
Data	0	0	1	5	4	0
String	1	2	3	4	5	6

MLCND

Word	1		2		3		4		5		6	
Data	J	K	L	R	S	0	8	6	5	8	3	2
String	1	2	3	4	A	6	7	8	9	10	11	12

IE = 0

After:

MLCND

Word	1		2		3		4		5		6	
Data	0	0	1	3	3	3	3	8	1	2	8	0
String	1	2	3	4	A	6	7	8	9	10	11	12

IE = 0

SSUB, Substring Subtract

SSUB subtracts one substring from a second substring and places the result in the second substring. Both substrings may be of any length.

```
CALL SSUB(jstr,jbeg,jend,kstr,kbeg,kend,ierr)
```

where:

jstr names a one dimensional integer string that is to be subtracted from a second array. The array must be defined in a DIMENSION statement. The contents of the array must be in A2 format, two characters per word.

jbeg is an integer constant, integer variable, or integer expression defining the position of the first character to be subtracted (beginning of substring).

jend is an integer constant, integer variable, or integer expression defining the position of the last character to be subtracted (end of substring). *jend* must be greater than or equal to *jbeg*.

kstr names a one-dimensional integer string array containing the data from which the data in *jstr* is subtracted. It will contain the result following subtraction. The array must be defined in a DIMENSION statement.

kbeg is an integer constant, integer variable, or integer expression defining the position of the first character in *kstr* (beginning of substring).

kend is an integer constant, integer variable, or integer expression defining the position of the last character in *kstr* (end of substring). *kend* must be greater than or equal to *kbeg*.

ierr is an integer variable used as an error indicator. Upon completion of SSUB, *ierr* indicates whether arithmetic overflow has occurred.

Errors: If there was arithmetic overflow (*kstr* was not large enough to contain the result), *ierr* is set to *kend*, *kstr* is filled with 9s.

If *jstr* is longer than *kstr*, neither field is altered, but *ierr* is set equal to *kend* and SSUB terminates.

If either data field, except *jend* and *kend*, is not numeric ASCII, *ierr* is set to −1 and SSUB terminates.

Comments: See comments for SADD.

Example:

```
DIMENSION  IFLDA(8),IFLDD(10)
IE = 0
CALL  SSUB(IFLDA,1,8,IFLDB,1,16,IE)
```

Before:

IFLDA

Word	1		2		3		4		5		6		7		8	
Data	1	5	6	4	3	0	5	5	D	D	D	D	D	D	D	D
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

IFLDB

Word	1		2		3		4		5		6		7		8		9		10	
Data	0	0	0	0	7	2	3	5	7	9	8	3	4	0	5	0	0	0	0	0
String	1	2	3	4	A	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

IE = 0

After:

IFLDA No Change

IFLDB

Word	1		2		3		4		5		6		7		8		9		10	
Data	0	0	0	0	7	2	3	5	6	4	1	9	0	9	9	5	0	0	0	0
String	1	2	3	4	A	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

IE = 0

The decimal data field IFLDA is subtracted from the decimal data field IFLDB and the result placed in IFLDB. Because IFLDA is positive, it is made negative and then added to IFLDB producing the result.

The error indicator IE is unchanged since no overflow occurred.

Output Editing Routine, SEDIT

SEDIT edits data in one substring array using an edit mask in a second substring array and places the edited data in the second substring array.

```
CALL SEDIT(jstr,jbeg,jend,kstr,kbeg,kend)
```

where:

jstr names a one dimensional integer string array containing the data to be edited. The array must be defined in a DIMENSION statement. The data to be edited, called the source field, is in A2 format, two charactes per word.

jbeg is an integer constant, integer variable, or integer expression defining the position of the first character of *jstr* to be edited (beginning of substring).

jend is an integer, integer variable, or integer expression defining the position of the last character of *jstr* to be edited (end of substring). *jend* must be greater than or equal to *jbeg*.

kstr names a one dimensional integer string array containing the edit mask and into which the data is edited. The edit mask, called the mask field, is in A2 format, two characters per word.

kbeg is an integer constant, integer variable, or integer expression defining the first position of the mask field (beginning of substring).

kend is an integer constant, integer variable, or integer expression defining the last position of the mask field (end of substring). It must be greater than *kbeg*.

Alphanumeric Editing

X (Alphanumeric Replacement Holder)

Alphanumeric edit masks are used to edit character substring and consists of Xs as replacement holders and any other characters as insertion characters. Characters are placed in the edit mask from right to left. Each replacement holder (X) in the edit mask is replaced in the display result with a character from the substring. Each insertion character (anything other than X) in the edit mask appears unmodified in the display result. If the end of the mask is reached before the end of the character substring, the remaining characters in the elements are not displayed. If the end of the character substring is reached first, the remainder of the display is replaced by asterisks. The character substring must be defined in ASCII if using the alphanumeric edit mask.

Examples:

```
Character Substring     Edit Mask       Edited Result
MNRZ                    "X-XX-X"        M-NR-Z
MNRZ                    "XXX"           NRZ
MNRZ                    "XX/XX/XX"      **/MN/RZ
```

Numeric Editing

Numeric edit masks are used to edit ASCII numeric, 0−9. Numeric edit masks consist of replacement holders, sign characters, and insertion characters.

Replacement

9 (Numeric Replacement Holder)

Each 9 in the edit mask is replaced by a decimal digit in the corresponding position of the numeric substring.

Z (Zero Suppression Replacement Holder)

The position of the Z in the edit mask is replaced by a decimal digit in the corresponding position of the numeric substring. Zeros to the left of the first significant position in the substring are replaced by blanks.

* (Asterisk Replacement Holder)

Asterisks rather than blanks are inserted to the left of the first significant decimal digit in the substring.

$ (Dollar Sign Replacement Holder)

A dollar sign is inserted to the left of the first significant decimal digit in the substring, and is to the left of the position that defined the zero suppression. Any zero in the remaining non-significant positions are replaced by blanks.

Sign Characters

Cr (Credit)

These two characters are placed in the rightmost positions of the edit mask. If the decimal substring is negative, the characters remain in the edited output. If the substring value is positive, CR is replaced by two blanks. When CR is present in the edit mask, no data is edited into the last two positions but only into the edit characters to the left.

– (Minus)

This character placed in the rightmost position of the edit mask is treated similarly to CR. It remains if the substring value is negative; is replaced by a blank when the substring value is positive. A minus elsewhere in the edit mask remains in that position in the edited output.

Insertion Characters

All other characters in the edit mask not defined above are insertion characters.

Operation of SEDIT

The characters are placed in the edit mask right to left. Only the characters 9, *, and S are replaced by decimal characters in the substring.

If the characters CR or a minus are in the rightmost position or positions, they are made blank for a positive substring value and left unchanged for a negative substring value.

If all the substring characters have not been placed in the edit mask when the end of the edit mask is reached, the entire edited output is filled with asterisks and editing terminates. Zero suppression proceeds from left to right of the edit mask. Any of the edit mask characters: 9, Z, X, (decimal point), or, (comma) is replaced by a blank unless the zero suppression character is an asterisk, in which case it is replaced by an asterisk.

Rules Governing Creation of Edit Mask

There must be no more than one decimal point in a numeric edit mask. Zero suppression is used when the edit mask contains a Z (zero), * (asterisk), or $ (dollar sign) and:

- A Z may not appear anywhere after a 9, *, or $ which is not the first holder in the edit mask.
- An * may not appear anywhere after a 9, Z or $ which is not the first holder in the edit mask.
- A $ may not appear anywhere after a Z, 9, or *.

In editing a numeric data substring through a numeric edit mask, the digits that represent the value of the substring are exchanged for the replacement holder. The decimal point remains in the edited output where it was placed in the edit mask. If, however, zero suppression is also requested, it is replaced by a blank if it is to the left of the last character to be suppressed.

Any insertion character appears unmodified in the display unless it is a decimal point or comma with zero suppression.

Examples:

Substring Value	Edit Mask	Edited Result
0059	"$$$,999"	$059
1024	"ZZZ,ZZZ"	1,024
010555	"$$,$$$.99CR"	$105.55
01055N (-0l0555)	"$$,$$$.99CR"	$105.55CR
01055N (-010555)	"$$,$$$.99-"	$105.55-
010555	"$$,$$$.99-"	$105.55
15039250	"$,$$$,$$$.99CR"	$150,392.50
139R (-1399)	"*,***.99CR"	***13.99CR
044240474	"999-99-9999"	044-24-0474
214N(-2145)	"$,$$$.99"	$21.45
24	"999.99"	000.24
24	"9.99.9"	***0.24
1234	"X.XX.X"	1.23.4

Errors

When the number of characters in the source field is greater than the number of characters in the mask substring, the mask substring is filled with asterisks and editing terminates.

In numeric edits, if more than one decimal point is encountered, the mask substring will be filled with stars from the place of the second decimal point to the leftmost position of the substring.

Each execution of SEDIT destroys the mask field by replacing it with the edited result. It is, therefore, advisable to move the mask to the output area and perform the edit function in the output area.

Internal Routines

SA2DE, Substring A2 Format to Decimal

SA2DE converts a field from A2 format to decimal format; A2 format is two characters per word; decimal format is two digits per word.

```
CALL SA2DE (jstr,jbeg,jend,ierr)
```

Note This routine is not normally called by user programs. It is used by the variable length decimal string arithmetic subroutines: SADD, SSUB, SMPY, and SDIV.

---

where:

jstr names the one dimensional integer string array in A2 format that is to be converted to decimal. The array must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable, or integer expression defning the first character position in *jstr* to be converted (beginning of substring).

jend is an integer constant, integer variable, or integer expression defining the last character position in *jstr* to be converted (end of substring). *jend* must be greater than or equal to *jbeg*.

ierr is an integer variable used as an error indicator. If all characters are valid, *ierr* is unchanged; otherwise, it is set to the last invalid character found during conversion.

Errors: When an invalid character is found, the position of the character is placed in *ierr*. (A nonnumeric or nonblank character is invalid; an 11 zone character representing a sign in the *jend* position of *jstr* is valid.) If more than one invalid character is found, *ierr* is set to the most recent position and processing continues.

Comments: Only the last invalid character is an indicator in *ierr* when conversion is complete. Other invalid characters may have been found in preceding positions.

Blanks are converted to zeros.

Zone punches can be used to indicate conditions. These punches can be removed with the SZONE routine as shown in the following example.

Example:

```
DIMENSION INFL(10)
IE = 0
CALL SA2DE (INVL,7,17,IE)
```

INFL is originally in A2 format. After execution of SA2DE, positions 7−17 of INFL have been converted to decimal format (blanks are converted to zeros). Since no invalid characters are found, IE is unchanged. The field to be converted was originally:

```
bbbb012345J
```

and the field after conversion is:

```
00000123451
```

Example: In order to remove zone punches, use the following routine:

```
      MAINLINE
      .
      .
      .
11    CALL SA2DE (JARY,JBEG,JEND,IERR)
      IF (IERR)22,22,32
22    (CONTINUE MAINLINE)
      .
      .
      .
32    (ERROR ROUTINE)
      .
      .
      .
      CALL SZONE (JARY,IERR,4,N1)
      NI = 0
      CALL SA2DE (JARY,IERR,IERR,N1)
      IF (N1)50,50,40
40    STOP 999
50    CALL SDEA2 (JARY,JBEG,JEND,IERR)
      IERR = 0
      GO TO 11
```

When *ierr* is greater than zero, control transfers to statement 32. Unless the zone is a special character, it is removed with the SZONE routine and converted to decimal. If the character was a special character (truly invalid), the program halts at statement 40. Otherwise, control goes to statement 50 where the field is returned to A2 format. Control then returns to statement 11 where the field is again converted to decimal in an attempt to find other invalid characters. This process continues until no more errors are found or a truly invalid character is encountered. The error indicator is not reset by this routine but must be reset by the programmer.

SCARY, Substring D2 Decimal Carry

SCARY examines a specified D2 decimal substring for carries, resolves the carries in the next higher substring, and saves any carry from the high order digit of the substring.

```
CALL SCARY (jstr,jbeg,jend,kout)
```

Note This routine normally is not called by a user program.

---

where:

jstr names a one dimensional integer string array which is interrogated for carries. It must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable, or integer expression indicating the first digit in *jstr* (beginning of substring).

jend is an integer constant, integer variable, or integer expression indicating the position of the last digit in *jstr* (end of substring). *jend* is greater than or equal to *jbeg*.

kout identifies an integer variable used to hold any carry from the high order position of *jstr* after execution of SCARY. If there is no carry, *kout* is set to zero.

Errors: None.

Comments: Generally, this routine is not called by a user program, since carries are resolved within the arithmetic routines SADD, SSUB, SMPY, and SDIV. SADD and SSUB call SCARY to resolve carries.

Example:

```
DIMENSION JDIGT(10)
M = 17
CALL CARRY(JDIGT,1,10,M)
```

Before:

JDIGT

Word	1		2		3		4		5		6		7		8		9		10	
Data	0	0	72	6	$2\overline{7}$	5	1	$\overline{8}$	1	1	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

M = 17

After:

JDIGT

Word	1		2		3		4		5		6		7		8		9		10	
Data	0	7	2	3	3	5	0	2	1	1	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
String	1	2	3	4	A	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

M = 0

As a result of multiple arithmetic operations, JDIGT originally has positions 3, 5, and 8 as shown before execution of SCARY. Following execution of SCARY, a 1 has been borrowed from the 7th position to resolve the −8 condition, a 3 was borrowed from the 4th position to resolve the condition at position 5, and the 7 from 72 is now in position 2.

SDCAR, Substring D1 Decimal Carry

SDCAR examines a specified D1 decimal substring for carries, resolves the carries in the next higher substring, and saves any carry from the high order digit of the substring.

```
CALL SDCAR (jstr,jbeg,jend,kout)
```

Note This routine normally is not called by a user program.

---

where:

jstr names a one dimensional integer string array in D1 decimal format (one digit per word) which is interrogated for carries. It must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable, or integer expression indicating the word postion of the first digit to be carried in *jstr* (beginning of substring).

jend is an integer constant, integer variable, or integer expression indicating the word position of the last digit to be carried in *jstr* (end of substring). *jend* is greater than or equal to *jbeg*.

kout identifies an integer variable used to hold any carry from the high order position of *jstr* after execution of SDCAR. If there is no carry, *kout* is set to zero.

Errors: None.

Comments: Generally, this routine is not called by the user since carries are resolved within the arithmetic routines SADD, SSUB, SMPY, and SDIV. SMPY and SDIV call SDCAR to resolve carries in D1 format substrings.

Example:

```
DIMENSION JDIGT(10)
M = 17
CALL SDCAR JDIGT(0,9,M)
```

Before:

JDIGT

Word	1		2		3		4		5		6		7		8		9		10	
Data	0		0		72		6		$2\bar{7}$		5		1		$\bar{8}$		1		1	
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

M = 17

After:

JDIGT

Word	1		2		3		4		5		6		7		8		9		10	
Data	0		7		2		3		3		5		0		2		1		1	
String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

M = 0 (zero)

SDEA2, Substring Decimal to A2 Format

SDEA2 converts a substring from D2 format to A2 format.

```
CALL SDEA2 (jstr,jbeg,jend,ierr)
```

Note This routine normally is not called by user program. It is used by the variable length decimal string arithmetic subroutines: SADD, SSUB, SMPY and SDIV.

---

where:

jstr names a one dimensional integer string array containing the substring to be converted; it must be in decimal format, two digits per word before conversion. The array must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable, or integer expression defining the position of the first digit of *jstr* to be converted (beginning of substring).

jend is an integer constant, integer variable, or integer expression defining the position of the last digit in *jstr* to be converted (end of substring). It must be greater than or equal to *jbeg*.

ierr is an integer variable used as an error indicator. It is set when a digit is greater than nine or is negative unless the negative digit is at position *jend* which, as the sign digit, can be negative.

Errors: The error indicator *ierr* is set equal to the position of the last invalid digit encountered. An invalid digit is one outside the range 0−9 except for a signed digit in the last position.

Comments: Only the last invalid digit is indicated by the error indicator. Other invalid digits may have been encountered to the left of the digit noted.

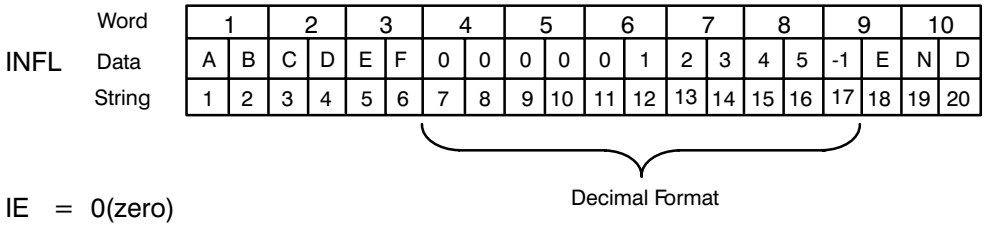
Errors should not occur since the arithmetic routines (SADD, SDIV, SMPY, and SSUB) resolve carries. If an error does occur, the user's program should indicate it.

Note SDEA2 does not set, test, or reset the error indicator.

Example:

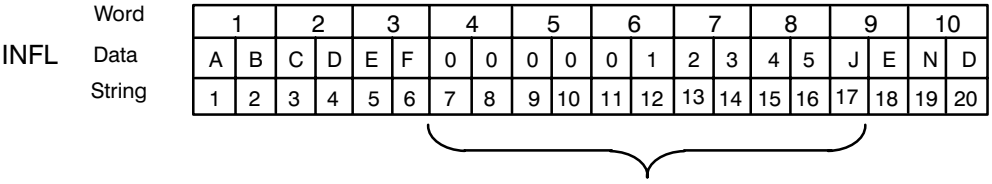
```
DIMENSION INFL(10)
IE = 0
CALL SDEA2(INFL,7,18,IE)
```

Before:



IE = 0(zero)

After:



IE = 0(zero)

SD1D2, Substring Decimal D1 Format to Substring Decimal D2 Format

SD1D2 converts a substring from D1 format (1 digit per word) to D2 format (2 digits per word).

```
CALL SD1D2 (jstr,jbeg,jend,diff)
```

Note This routine normally is not called by a user program. It is used by the variable length decimal string arithmetic subroutines SMPY and SDIV.

---

where:

jstr names a one dimensional integer string array containing the substring to be converted; it must be in D1 format, 1 digit per word before conversion. The array must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable, or integer expression defining the first position of *jstr* after conversion to D2 format.

jend is an integer constant, integer variable, or integer expression defining the last position of *jstr* after conversion to D2 format. It must be greater than or equal to *jbeg*.

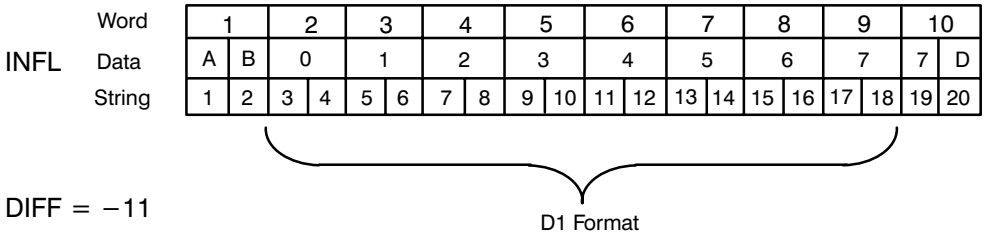
diff is an integer constant, integer variable, or integer expression defining the bias to be added to any index or position pointer for D2 format to obtain an index for D1 format. It is calculated by SD2D1.

Errors: None.

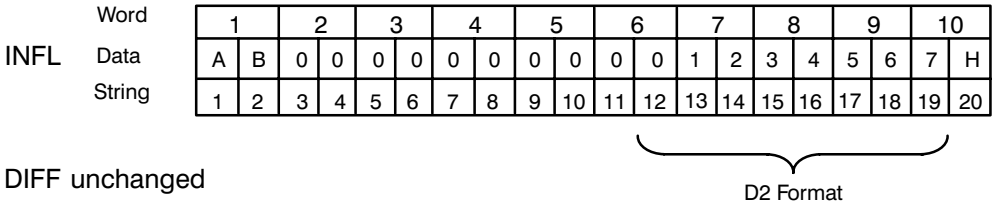
Example:

```
DIMENSION INFL(10)
DIFF = -11
CALL  SD1D2(JSTR,12,19,DIFF)
```

Before:



After:



SD2D1, Substring Decimal D2 Format to Substring Decimal D1 Format

SD2D1 converts a substring from D2 format (2 digits per word) to D1 format (1 digit per word).

```
CALL SD2D1 (jstr,jbeg,jend,diff)
```

Note This routine normally is not called by a user program. It is used by the variable length decimal string arithmetic subroutines SMPY and SDIV to accommodate large numbers.

---

where:

jstr names a one dimensional integer string array containing the substring to be converted; it must be in D2 decimal format, two digits per word before conversion. The array must be defined in a DIMENSION statement.

jbeg is an integer constant, integer variable, or integer expression defining the position of the first digit of *jstr* to be converted (beginning of substring).

jend is an integer constgant, integer variable, or integer expression defining the position of the last digit in *jstr* to be converted (end of substring). It must be greater than or equal to *jbeg*.

diff is an integer constant, integer variable, or integer expression defining the bias to be added to any index or position pointer for D2 to obtgain an index for D1 format. It is calculated according to the formula:

```
DIFF = -((jend+1)/2+1)
```

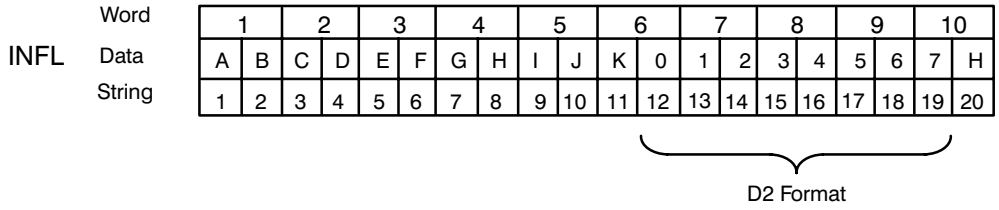
Note 2(*jend-jbeg*+1) positions in *jstr* must be available, so that no digit would ever be moved to a position preceding *jstr*(1).

Errors: None.

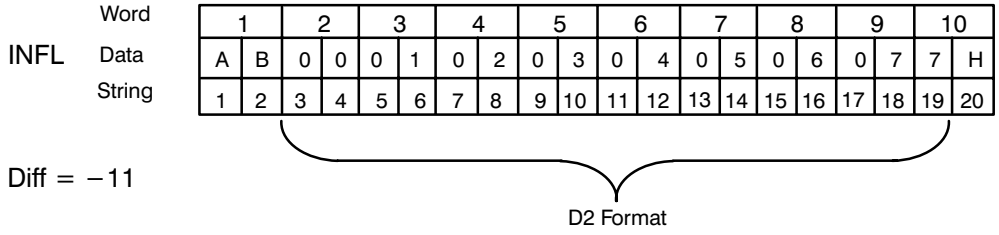
Example:

```
DIMENSION INFL(100)
CALL SD2D1 (JSTR,12,19,DIFF)
```

Before:



After:



SSIGN, Substring Sign

SSIGN finds the sign of a number, sets a code to indicate this sign and gives the number a new sign (as specified).

```
CALL SSIGN (jstr,jbeg,news,nolds)
```

Note This routine normally is not called by a user program. It is used by the variable length decimal string arithmetic subroutines: SADD, SSUB, SMPY, SDIV, JSCOM.

---

where:

jstr names a one dimensional string array containing the character whose sign is to be tested and modified. The array array must be defined in a DIMENSION statement. The character must be decimal format, two digits per word (D2).

jbeg is an integer constant, integer variable, or integer expression defining the position of the character to be tested and modified.

news is an integer constant, integer variable, or integer expression containing the code for the new sign.

nolds is an integer variable which is set to a code specifying the old (original) sign of the character.

Method: First the sign of the character at *jbeg* is retrieved and *nolds* is set as follows:

NOLDS	Original Sign
+1	non-negative
−1	negative

Next, the character is given a new sign according to the code specified by *news*. The following shows the sign depending on the value of *news*:

NEWS	New Sign
+1	positive
0	opposite of original sign
−1	negative

Errors: None.

Comments: The character to be processed must be in decimal format, two digits per word (D2 format), or the result is meaningless.

Example:

```
DIMENSION IDGT(10)
CALL SSIGN(IDGT,20+1,NS)
```

Before:

```
IDGT(20) = +8
NS = 0
```

After:

```
IDGT(20) = +8   (no change)
NS = +1
```

11

Floating Point Conversion Subroutines

The floating point conversion routines convert between HP 1000 and IEEE standard floating point formats. These subroutines are part of the $MATH library. Each function must be declared as a 32-bit integer, the same type that it returns.

The subroutines in this chapter are presented in the following format:

The name of the subroutine, a statement of the use of the subroutine, followed by the subroutine's syntax, a description of the parameters, and then returns, if any.

If a parameter is underlined in a subroutine call description, the value is a variable returned or modified by the system subroutine.

DFCHI

DFCHI is a function that converts from HP 1000 format double precision floating point to IEEE standard format.

error = DFCHI(*hpfp*,*i3efp*)

```
integer*4 error,DFCHI
real*8 hpfp,i3efp
```

where:

hpfp is the 64-bit double precision floating point real to be converted to IEEE standard format.

i3efp is the 64-bit variable in which the IEEE standard double precision floating point real will be stored.

Returns: A 32-bit integer with the following meanings:

0 Successful conversion.
−3 Value of *hpfp* is denormalized number; *i3efp* unchanged.

FCHI

FCHI is a function that converts from HP 1000 format single precision floating point to IEEE standard format.

```
error = FCHI(hpfp,i3efp)

integer*4 error,FCHI
real*4 hpfp,i3efp
```

where:

hpfp is the 32-bit single precision floating point real to be converted to IEEE standard format.

i3efp is the 32-bit standard single precision floating point variable in which the IEEE standard single precision floating point real will be stored.

Returns: A 32-bit integer with the following meanings:

0	Successful conversion.
−3	Value of *hpfp* is denormalized number; *i3efp* unchanged.
−5	Underflow occurred; *i3efp* unchanged.

DFCIH

DFCIH is a function that converts from IEEE standard format double precision floating point to HP 1000 format.

```
error = DFCIH(i3efp,hpfp)

integer*4 error,DFCIH
real*8 i3efp,hpfp
```

where:

i3efp is the 64-bit IEEE standard format double precision floating point real to be converted to HP 1000 format.

hpfp is the 64-bit standard single precision floating point variable in which the HP 1000 double precision floating point real will be stored.

Returns: A 32-bit integer with the following meanings:

0	Successful conversion.
−1	Value of *i3efp* is not a number; *hpfp* unchanged.
−2	Value of *i3efp* is signed infinity; *hpfp* unchanged.
−3	Value of *i3efp* is denormalized number; *hpfp* unchanged.
−4	Overflow occurred; *hpfp* unchanged.

FCIH

FCIH is a function that converts from IEEE standard format single precision floating point to HP 1000 format.

error = FCIH(*i3efp*,*hpfp*)

```
integer*4 error,FCIH
real*4 i3efp,hpfp
```

where:

i3efp is the 32-bit IEEE standard format single precision floating point real to be converted to HP 1000 format.

hpfp is the 32-bit standard single precision floating point variable in which the HP 1000 single precision floating point real will be stored.

Returns: A 32-bit integer with the following meanings:

0	Successful conversion.
−1	Value of *i3efp* is not a number; *hpfp* unchanged.
−2	Value of *i3efp* is signed infinity; *hpfp* unchanged.
−3	Value of *i3efp* is denormalized number; *hpfp* unchanged.
−4	Overflow occurred; *hpfp* unchanged.
−5	Underflow occurred; *hpfp* unchanged.

12

HpCrt Library Routines

The subroutines in this chapter are presented in the following format:

The name of the subroutine, a statement of the use of the subroutine, followed by the subroutine's syntax, a description of the parameters, and then returns, if any.

If a parameter is underlined in a subroutine call description, the value is a variable returned or modified by the system subroutine.

A_Register, B_Register, A_B_Registers, ABREG

These are CDS compatible replacements for the ABREG call. To use these functions, you must declare them to be direct so that the standard .ENTR calling sequence will be omitted. They must be declared as Integer*2 for A_Register and B_Register and Integer*4 for A_B_Registers.

```
$alias  A_Register,direct
$alias  B_Register,direct
$alias  A_B_Registers,direct

Integer*2  A_Register,B_Register
Integer*4 A_B_Registers        (Real*4 will work also)
```

As an example call, you could do the following:

```
CALL  Exec(1,401b,ibuf,-80)      ! read the terminal
Jboth  = A_B_Registers()         ! save both registers
Istat  = A_Register()            ! save A only
Length = B_Register()            ! save B only
```

You must call A_B_Registers or A_Register first, as B_Register copies the B-Register contents into the A-Register. The destination variables must be simple variables, that is, not subscripted, as the subscript resolution process alters the registers. This same restriction applies to the HP ABREG call.

The assembly language generated by the previous example is:

```
      jsb exec
      def rtn
      def =d1
      def =b401
      def ibuf
      def =d-80
rtn   equ *
      jsb A_B_registers   (converted to a NOP at load time)
      dst Jboth
      jsb A_register      (converted to a NOP at load time)
      sta Istat
      jsb B_register      (converted to LDA B at load time)
      sta length
```

Note The usual restrictions upon ABREG apply; that is, do not use variables, EMA variables, or anything else that will alter the registers before they can be stored.

ClearBitMap

This routine clears the specified bit in a bit map.

```
CALL ClearBitMap(ibuf,ibit)

integer*2 ibuf(*),ibit
```

where:

ibuf is an integer array of up to 64K bits (4096 words).

ibit is a one-word integer representing the bit number to clear, where 0 equals the most significant bit of the first word.

When accessing bits above 32K in FORTRAN, you must use negative numbers, as there is no unsigned integer data type.

CompareBufs

This routine compares two buffers. It is similar to the CompareWords routine except that it returns the offset of the mismatch.

```
flag = CompareBufs(buf1,buf2,numwds,badwd)

logical*2 flag,CompareBufs
integer*2 buf1(*),buf2(*),numwds,badwd
```

where:

flag is the logical return flag. It will be −1 if the buffers are identical, which is a logical true in FORTRAN or Pascal; or 0 if they differ, which is a logical false in both languages.

buf1 specifies the first buffer.

buf2 specifies the second buffer.

numwds is the number of words to be compared.

badwd is the offset of the word from the beginning of the buffer that was found to be different.

The buffers can be of any data type except FTN77 characters. The number of words to be compared must be computed accordingly. For example to compare two buffers of 12 double precision floating point (Real*8), the number of words would be 8/2*12=48.

CompareWords

This routine compares two buffers.

```
flag = CompareWords(buf1,buf2,numwds)

logical*2 flag,CompareWords
integer*2 buf1(*),buf2(*),numwds
```

where:

flag is the logical return flag. It will be −1 if the buffers are identical, which is a logical true in FORTRAN or Pascal; or 0 if they differ, which is a logical false in both languages.

buf1 specifies the first buffer.

buf2 specifies the second buffer.

numwds is the number of words to compare.

The buffers can be of any data type except FTN77 characters. The number of words to be compared must be computed accordingly. For example to compare two buffers of 12 double precision floating point (Real*8), the number of words would be 8/2*12=48.

CompressAsciiRLE

This function moves bytes from the input buffer to the output buffer, compressing the data by replacing repeated characters with a repeat count. The first character in a sequence of like characters is copied to the output, but all successive characters are replaced by the 2's complement of the repeat count. If more than 128 characters appear in the run, repeats of −128 are used as necessary. Characters which are not repeated are not altered. If data that already has the 8th bit set is encountered, such as binary data, Kanji, or extended ASCII for national character sets, then the error flag will be set.

flag = CompressAsciiRLE(*ibuf*,*length*,*obuf*,*newlength*)

```
logical*2 flag,CompressAsciiRLE
integer*2 ibuf(*),length,obuf(*),newlength
```

where:

flag	is the logical error indicator. Equals .TRUE. when an error occurs.
ibuf	is the input data buffer.
length	is the number of bytes in the input buffer.
obuf	returns the output data buffer.
newlength	contains the number of bytes that were moved to *obuf*.

For example, if the routine is called with the following values for *ibuf* and *length*:

ibuf = A B C C C D : : : E E E E E E E E E E I
length = 26

The output values of *obuf* and *newlength* will be:

obuf = A B C 376b D : 375b E 367b . 371b I
newlength = 12

A buffer length of zero is acceptable and does not cause an error. It is not possible for the output to be longer than the input.

It is permissible for the output buffer to be the same as the input buffer so that in-place compression is done. For example, the following call is valid:

flag = CompressAsciiRLE(*ibuf*,*length*,*ibuf*,*length*)

See also ExpandAsciiRLE.

ExpandAsciiRLE

This logical*2 function processes run length encoded ASCII data to expand it back to the original uncompressed contents. If the expanded text will be longer than the stated size of the output buffer, conversion terminates and the routine returns with a .TRUE. error condition. Note that the output can be much larger than the input.

errorflag = ExpandAsciiRLE(*ibuf*,*ilength*,*obuf*,*olength*,*newlength*)

```
logical*2 errorflag,ExpandAsciiRLE
integer*2 ibuf(*),ilength,obuf(*),olength,newlength
```

where:

errorflag is the error indicator; will be true if an error occurs.

ibuf is the input data buffer.

ilength is the number of bytes in the input buffer.

obuf returns the output data buffer.

olength is the size of the output buffer.

newlength contains the number of bytes that were moved to *obuf*.

For example, if the routine is called with the following values for *ibuf* and *ilength*:

ibuf = X 377b Y 374b $ @ * 370b A B C 375b A
ilength = 13

The output values of *obuf* and *newlength* will be:

obuf = X X Y Y Y Y Y $ @ * * * * * * * * * A B C C C C A
newlength = 25

See also CompressAsciiRLE.

FakeSpStatus

This routine creates a status buffer similar to what a Special Status Read returns. FakeSpStatus is used by HpZPrintPort to obtain as much port status information as it can in the event that the device LU is busy. The FakeSpStatus routine retrieves information statically from the system status tables that would normally be returned dynamically from a Special Status Read call issued to the driver (refer to the serial driver documentation). Therefore, some of the values returned by FakeSpStatus in words 1 through 32 of *statebuffer* are invalid as indicated in Table 12-1.

```
CALL FakeSpStatus(lu,statebuffer)

integer*2 lu,statebuffer(32)
```

where:

lu is the LU number of the port (in the range 1..255).

statebuffer is a 32-word buffer to hold the current state of the port.

This call can be made only for an LU known to be a DDC0x LU.

See also HpCrtSavePort, HpCrtRestore, and HpZPrintPort.

Table 12-1. Contents of the State Buffer after FakeSpStatus or HpCrtSavePort Call

Word	Returned Value
1..3*	Device driver name
4 *	Device driver Revision code
5	DVT word 6 or EQT word 5
6	DVT (or EQT) address
7..9*	Interface driver name
10 *	Interface driver Revision code
11 *	Firmware Revision code
12..14	Primary Interrupt program name
15..17	Secondary Interrupt program name
18	Echo back of CN 17 parameter
19	Echo back of CN 22 parameter
20	Echo back of CN 30 parameter
21	Echo back of CN 31 parameter
22 *	Echo back of CN 33 parameter
23	Echo back of CN 34 parameter
24	Address of IFT (RTE-A only)
25..27	Echo back of CN 42 parameters (RTE-A only)
28	Echo back of CN 16 parameters (RTE-A only)
29..31	Zero
32	Driver communications word

* Words 3, 4, 9 - 11, and 22 are not valid after a FakeSpStatus call.

FillBuffer

This routine fills a buffer with either null characters or a specified value.

```
CALL FillBuffer(ibuf,length[,value])

integer*2 ibuf(*),length,value
```

where:

ibuf	returns the buffer that is filled.
length	specifies the length of the buffer in words.
value	is an optional value to store in *ibuf*. The value to store is located in the low byte of *value*. The default is zero (null).

Examples:

```
call FillBuffer(ibuf,20,2h .)    ! fill 1st 20 words of ibuf
                                 ! with ASCII decimal points
call FillBuffer(ibuf,50,2h  )    ! fill 1st 50 words of ibuf
                                 ! with blanks (020040b)
call FillBuffer(ibuf,30)         ! fill 1st 30 words of ibuf
                                 ! with nulls (000000b)
```

FirstCharacter

This function returns the first character of a buffer. The character is in the left byte with an ASCII blank character in the right byte.

```
value = FirstCharacter(buffer)

integer*2 value,buffer(*),FirstCharacter
```

For example:

```
if ( FirstCharacter(buffer).eq.2h+ ) then ... endif
```

will test to see if the first character of “buffer” is a plus sign.

GetBitMap

This integer*2 function retrieves the value of the indicated bit from a bit map (packed array of bits). When accessing bits above 32k, you must use negative numbers, as there is no unsigned integer data type in FORTRAN on the HP 1000.

bit = GetBitMap(*ibuf*,*index*)

integer*2 *bit*,*ibuf*(*),*index*,GetBitMap

where:

bit is the value of the bit indicated by *index*.

ibuf is the bit map, an array of up to 64K bits (4096 words).

index is the bit number to be read, where 0 equals the MSB of the first word of *ibuf*.

See also SetBitMap, TestBitMap, TestSetBitMap, and PutBitMap.

GetByte

This routine gets a byte from a packed array of bytes. The leftmost byte of the first word of *array* is byte number 0. The array can be up to 32K words, so the byte index can be from 0 to 65,535. Addresses above 32,767 look like negative numbers because there is no unsigned integer data type in FORTRAN on the HP 1000.

byte = GetByte(*array*,*index*)

integer*2 *byte*,*array*(*),*index*,GetByte

where:

byte is the value of the indicated byte in *array*. The value is contained in the low byte of *byte*; null is contained in the high byte.

array is the array that contains the byte to be read.

index is the index into the array indicating the byte to be read where byte 0 is the left byte of the first word of *array*.

See also PutByte.

GetDibit

This routine gets a dibit (bit pair) from a packed array of dibits. The leftmost two bits of the first word of *array* is dibit number 0. The index is limited to 16 bits, which limits the length of the array to 8K words. Dibit addresses above 32,767 look like negative numbers because there is no unsigned integer data type in FORTRAN on the HP 1000.

twobitvalue = GetDibit(*array*,*index*)

integer*2 *twobitvalue*,*array*(*),*index*,GetDibit

where:

twobitvalue is the value of the indicated dibit in *array*. The value is contained in the lower 2 bits of *twobitvalue*; upper bits are null.

array is the array that contains the dibit to be read.

index is the index into the array indicating the dibit to be read where dibit number 0 is the leftmost two bits of the first word of *array*.

See also PutDibit.

GetNibble

This routine gets a nibble (4 bits) from a packed array of nibbles. The leftmost four bits of the first word of *array* is nibble number 0. The index is limited to 16 bits, which limits the length of the array to 16K words. Nibble addresses above 32,767 look like negative numbers because there is no unsigned integer data type in FORTRAN on the HP 1000.

nibble = GetNibble(*array*,*index*)

integer*2 *nibble*,*array*(*),*index*,GetNibble

where:

nibble is the value of the indicated nibble in *array*. The value is contained in the lower 4 bits of *nibble*; the upper 12 bits are null.

array is the array that contains the nibble to be read.

index is the index into the array indicating the nibble to be read where nibble number 0 is the leftmost four bits of the first word of *array*.

See also PutNibble.

GetRunString

This integer*2 or logical*2 function retrieves the runstring that was used to schedule a program.

```
IF (GetRunString(ibuf,ibuflen,length,ip)) THEN
  no_runstring_passed
ENDIF

integer*2 ibuf(*),ibuflen,length,ip
logical*2 GetRunString
```

where:

ibuf	returns the input buffer.
ibuflen	specifies the length of the buffer, positive number of words or negative number of bytes.
length	returns the number of characters passed back by the EXEC 14 call including "RU,*progname*,".
ip	returns the pointer to the first character past "RU,*progname*," in the buffer. This value is not valid for a .TRUE. return.

If it is desired to manipulate the buffer using the FORTRAN character routines, use StrDsc to build a character descriptor after calling this function.

Algorithm used:

An EXEC 14 call reads the runstring, if any is present, into the caller's buffer. *length* is updated to show the length in bytes. The buffer is then scanned until the second comma is found, to skip the "RU,*progname*," that is present in the buffer when programs are run interactively. If the program is not going to be run interactively, do not use this routine; use an EXEC 14 call instead.

The position just following the second comma is then passed back in *ip*, where 0 indicates the first byte in the buffer, the left byte of the first word. This is done to facilitate the use of routines such as NAMR or the HpZ routines to parse the remaining contents of the buffer.

If no string was passed, the function returns with A=−1 (.TRUE.). If a string was passed, it returns with A=*ip* (a .FALSE. value).

The values passed back are in the correct form for a call to HpZDefIbuf.

See also GETST and HpZDefIbuf (this manual); Exec 14 (documented in the *RTE-A Programmer's Reference Manual)*; RCPAR and RHPAR (documented in the *FORTRAN 77 Reference Manual)*.

GetString

This subroutine copies a string or a constructed string descriptor (refer to the StrDsc call) to the indicated output string. The number of characters is given by the third parameter rather than using a length imbedded in *instring*.

```
CALL GetString(outstring,instring,stringlength)

character*(*) outstring
integer*4 instring
integer*2 stringlength
```

where:

outstring is the output string.

instring is the input string. This could be the output of StrDsc.

stringlength is the number of characters to be copied to *outstring*.

This routine is sometimes required because FORTRAN does not have a mechanism to copy the contents of a constructed string descriptor.

See also HpZMoveString and ConCat.

HpCrtCharMode

This call sends the escape sequences to the terminal that place it in line mode and character mode with forms mode disabled. It also does the control call to the driver that is required after such strap changes.

```
CALL HpCrtCharMode(lu)

integer*2 lu
```

where:

lu is the LU number(1..255) of the terminal to be strapped for character mode.

The escape sequence generated by this call is:

```
Esc & k 0 a 0 B Esc & s 0 d 0 g 0 H Esc X Esc b
```

It is the caller's responsibility to ensure that the LU is legal, that the terminal is capable of page mode operation, and that the terminal driver is capable of block mode transfers (such as DDC00 and DVC05).

See also HpCrtPageMode and HpCrtLineMode.

HpCrtCheckStraps

This routine checks the port and the terminal to ensure that screen mode operation (as used by EDIT/1000 and the Command Stack routines) is available.

```
IF (HpCrtCheckStraps(crtlu,error,memsize,linewidth)) THEN
  screen mode is usable
ENDIF

integer*2 crtlu,error,memsize,linewidth
logical*2 HpCrtCheckStraps
```

where:

crtlu is the LU number of the terminal.

error is the returned error code as given below.

memsize is the returned amount of display memory in the terminal.

linewidth is the returned number of columns that can be displayed at any one time. To determine the linewidth, position the cursor to column 1, then cursor left. This is done to be compatible with the Command Stack routines.

The error code is defined as:

−1 successful
0 not an HP 2645 or compatible terminal
1 EXEC call failed
2 one of the terminal straps is incorrect even though this routine tried to set it to the following states:
character mode (not page mode and not block mode); transmit functions disabled.

When error 2 is returned, the calling program should print an advisory message telling the user what strapping is needed.

See also HpCrtScreenSize.

HpCrtCRC16_F, HpCrtCRC16_S

A Cyclic Redundancy Check (CRC) is an error checking method based upon feedback shift registers for the hardware implementation. The mathematical derivation is that it is a synthetic division of the message by a given polynomial.

Two implementations are provided, HpCrtCRC16_F which uses a 256-word lookup table, and HpCrtCRC16_S which uses a 32-word table and runs correspondingly slower.

icheck = HpCrtCRC16_F(*buffer*,*bytelength*,*start*)

integer*2 *icheck*,*buffer*,*bytelength*,*start*,HpCrtCRC16_F

where:

icheck returns a 16-bit integer result (2 bytes in A2 format).

buffer is the buffer containing data in A2 format (cannot be character data type).

bytelength is the number of bytes to include in the Cyclic Redundancy Check bytes.

start is the starting position in *buffer*, where 0 is the left byte of the first word.

This routine uses the polynomial 120001 octal as its generator. This is commonly called a CRC-16, and is the CRC used by HDLC, BiSync, and other industry standard protocols. Refer to the standard textbooks on data communications for further information.

The algorithm used by these routines was first described by Stuart Wecker of Digital Equipment Corporation (1974).

HpCrtGetCursor

This call returns the coordinates of the cursor of an HP CRT. The reply is in the form of the ASCII escape string that would be needed to redirect the cursor to its current position.

```
IF (HpCrtGetCursor(crt,reply[,screenrelative])) THEN
  good_reply
ENDIF

integer*2 crt,reply(6),screenrelative
logical*2 HpCrtGetCursor
```

where:

crt is the LU number of the terminal; in the range 1..255 for RTE-A and 1..63 for RTE-6/VM systems.

reply is a 6-word long buffer to receive the reply in one of the following forms:

If *screenrelative* is not passed or equals zero: "<esc>&a000c000R<cr>"

If *screenrelative* is nonzero: "<esc>&a000c000Y<cr>" or "<esc>&a000x000Y<cr>"

screenrelative is an optional parameter that indicates screen relative coordinates should be returned. If *screenrelative* is not passed or equals zero, memory relative coordinates are returned. If it is nonzero, screen relative coordinates are returned.

Note that the column number may not be limited to 0...79. As an example, the HP 2393 can be configured for up to 160 columns. The maximum row number is also terminal dependent.

See also HpCrtGetCursorXY.

HpCrtGetCursorXY

This call returns the coordinates of the cursor of an HP CRT. The reply is in the form of the column and row numbers needed to redirect the cursor to its current position.

```
IF (HpCrtGetCursorXY(crt,x,y[,screenrelative])) THEN
  good_XY
ENDIF

integer*2 crt,x,y,screenrelative
logical*2 HpCrtGetCursorXY
```

where:

crt	is the LU number of the terminal; in the range 1..255 for RTE-A and 1..63 for RTE-6/VM systems.
x	returns cursor column position (0..n).
y	returns cursor row position (0..n).
screenrelative	is an optional parameter that indicates screen relative coordinates should be returned. If *screenrelative* is not passed or equals zero, memory relative coordinates are returned. If it is nonzero, screen relative coordinates are returned.

Note that the column number may not be limited to 0..79. As an example, the HP 2393 can be configured for up to 160 columns. The maximum row number will depend upon the amount of memory installed.

See also HpCrtGetCursor.

HpCrtGetfield_I

This integer*2 function retrieves the *n*th field from an integer*2 buffer that contains the data read from an HP terminal in block page mode. The fields are separated by US (Unit Separator, octal 37) characters. The function returns the number of characters in the given field. The newer HP terminals have the capability to return modified fields only, which results in two adjacent US characters for unmodified fields. This routine will return a length of zero in that case.

There is another version of this routine, HpCrtGetfield_S, that is used when the destination buffer is a FORTRAN character data type.

olen = HpCrtGetfield_I(*ibuf*,*ilength*,*field#*,*obuf*,*olength*)

integer*2 *olen*,*ibuf*(*),*ilength*,*field#*,*obuf*(*),*olength*,HpCrtGetfield_I

where:

olen returns the number of characters present in the *n*th field.

ibuf specifies the input buffer from a page mode read.

ilength specifies the number of bytes in the input buffer.

field# specifies the ordinal number of the unprotected field containing the data that is to be moved to the output string.

obuf is the buffer to receive the data; it should be as long as the protected fields you are reading.

olength is the length of the output buffer in bytes.

The input buffer is assumed to contain something similar to the following:

'datadatada<US>tadatadat<US><US>adatadatadat<US>adata...'

the first field contains	'datadatada'	length = 10
the second contains	'tadatadat'	length = 9
the third is empty	''	length = 0
the fourth contains	'adatadatadat'	length = 12
and so on.		

If the terminal is not capable of operating in "altered fields only" mode, every field will contain something, even if just blanks.

HpCrtGetfield_S

This logical*2 function retrieves the *n*th field from an integer*2 buffer that contains the data read from an HP terminal in block page mode. The fields are separated by Unit Separator characters. For the newer HP terminals, fields which were not modified by the user will return two adjacent Unit Separators. This subroutine detects such fields and returns a .FALSE., while modified fields will return a .TRUE. to indicate the data is valid.

The destination string will be blank padded, as is usual for character data type. There is another version of this routine, HpCrtGetfield_I, that returns the data to integer*2 buffers with a byte count.

flag = HpCrtGetfield_S(*ibuf*,*ilength*,*field#*,*deststring*)

```
logical*2 flag,HpCrtGetfield_S
integer*2 ibuf(*),ilength,field#
character*(*) deststring
```

where:

flag returns true if data is present in the *n*th field.

ibuf specifies the input buffer from a page mode read.

ilength specifies the number of bytes in the input buffer.

field# specifies the ordinal number of the unprotected field containing the data that is to be copied to the destination string.

deststring returns the FORTRAN character data type buffer to receive the data; it should be at least as long as the unprotected field.

The input buffer is assumed to contain something similar to the following:

```
'datadatada<US>tadatadat<US><US>adatadatad<US>ata...'
```

the first field contains	'datadatada'
the second contains	'tadatadat '
the third is empty	' '
the fourth contains	'adatadatad'
and so on.	

If the terminal is not capable of operating in "altered fields only" mode, every field will contain something, even if just blanks, so that the routine will return .TRUE. for all fields.

HpCrtGetLine_Pos

This routine returns the cursor position, the contents of the line where the cursor was located when a given delimiter is entered, and the delimiter that caused the routine to return.

bytelen = `HpCrtGetLine_Pos(`*crt*`,`*buf*`,`*length*`,`*pos*`,`*delimiters*`,`*stopchar*`)`

`integer*2` *bytelen*`,`*crt*`,`*buf*`(*),`*length*`,`*pos*`,`*delimiters*`,`*stopchar*`,HpCrtGetLine_Pos`

where:

bytelen returns the number of characters read from the screen.

crt is the LU number of the User's terminal (in the range 1..255).

buf is the name of the return buffer; it must be large enough to hold the width of the screen *plus* any non-displayable characters.

length specifies the size of *buf* in bytes.

pos returns the position that the column cursor was in when the user typed the delimiter (0-n).

delimiters defines up to two delimiters that will cause the routine to return. If only one delimiter is needed, the high byte and low byte must be set equal to each other.

stopchar returns the delimiter that was entered.

This function performs a single character read on the indicated LU until one of the delimiters passed in the call is received. It then reads the absolute cursor position on the line followed by reading the line from the screen where the cursor is currently located. The function returns the number of characters in the current line. The routine attempts to do the single character reads using class I/O to allow the calling program to be swappable.

HpCrtGetMenuItem

This routine returns a menu item from the screen. Put the cursor on the item and press the carriage return key. This routine returns, in *buf*, the item under (or just in front of) the cursor. The character length is also returned. For generality, the result is not upshifted. It is assumed that the item is delimited by one of the passed delimiters or by the start or the end of the line.

```
bytelen    = HpCrtGetMenuItem(crt,buf,length,delimiters)
fakestring = HpCrtGetMenuItem(crt,buf,length,delimiters)

integer*2 crt,buf(*),length
character*(*) delimiters
```

where:

bytelen *fakestring*	if this routine is declared integer*2, it returns the length in the A-Register. if this routine is declared integer*4, it returns a string descriptor in the A- and B-Registers.
crt	is the LU number of the user's terminal (in the range 1..255).
buf	is the name of the return buffer; it must be large enough to hold the width of the screen *plus* any non-displayable characters.
length	specifies the size of *buf* in bytes.
delimiters	is the string that defines the delimiters.

HpCrtHardReset

This function performs a hard reset on an HP terminal connected to the given LU. A hard reset sets the terminal back to the state that it had on power-up. The results are specific to the terminal model, but a hard reset usually causes loss of all soft configurations, such as margins, screen memory, and softkeys. This is a drastic way to set a terminal to a known state.

```
CALL HpCrtHardReset(lu)

integer*2 lu
```

where:

lu	is the LU of the terminal on which to perform a hard reset (in the range 1..255).

Algorithm:

1. Transmit 'Esc Z Esc E' to the terminal.
2. Use an EXEC 12 call to pause 1 second.
3. Perform an XLuEx CN25 call to the port to re-sense the terminal straps.

Because this routine uses an EXEC 12 call, the calling program will be removed from the time list if it happens to be there.

HpCrtLineMode

This call sends the escape sequences to the terminal that place it in block line mode. It also does the control call to the driver that is required for line mode operation.

```
CALL HpCrtLineMode(lu,flag)

integer*2 lu
logical*2 flag
```

where:

lu is the LU number of the terminal to be strapped for line mode.

flag specifies a flag to control forms mode:

```
.TRUE.  = enable protected forms mode.
.FALSE. = disable protected forms mode.
```

The escape sequence generated by this call is:

```
Esc & k 1 a 1 B Esc & s 0 d 0 g 0 H Esc X Esc b
```

It is the caller's responsibility to ensure that the LU is legal, that the terminal is capable of page mode operation, and that the terminal driver is also capable of block mode transfers (DDC00, DVC05, for example).

See also HpCrtPageMode and HpCrtCharMode.

HpCrtMenu

This routine prints multiple character strings to an LU. Each string is printed by a separate EXEC call, in the same way as HpCrtSendChar.

```
CALL HpCrtMenu(lu, '1st line of menu',
>                  '2nd line'
>                  '3rd line'
                     ...
>                  'last line')
```

The number of strings passed is unlimited, and either quoted strings (literals) or FTN7X character variables can be passed.

See the documentation for HpCrtSendChar for a caution about certain printers.

See also HpCrtXSendChar, HpCrtXReadChar, HpCrtSendChar, and HpCrtXMenu.

HpCrtNlsMenu

This routine performs the HpCrtMenu function from the NLS module.

```
CALL HpCrtNlsMenu(lu,start[,end])

integer*2 lu,start,end
```

where:

lu is the LU number for the output; a single integer in the form expected by EXEC calls.

start specifies the starting string index.

end is an optional parameter that specifies the ending string index. *end* must be equal to or greater than the starting string index. It defaults to be equal to the start so that only one line is displayed.

Prior to this call, an initialization call to HpZNlsSubset must have been issued.

When used with HpZNlsSubset, the index number of the string must be in the range 0..N, where N is the last message number in the catalog.

See also HpZNlsSubset and HpCrtNlsXMenu.

HpCrtNlsXMenu

This routine performs the HpCrtXMenu function from the NLS module.

```
CALL HpCrtNlsXMenu(lu,start[,end])

integer*4 lu
integer*2 start,end
```

where:

lu is the LU number for the output; a double integer in the form expected by XLuEx calls.

start specifies the starting string index.

end is an optional parameter that specifies the ending string index. *end* must be equal to or greater than the starting string index. It defaults to be equal to the start so that only one line is displayed.

Prior to this call, an initialization call to HpZNlsSubset must have been issued. When used with HpZNlsSubset, the index number of the string must be in the range 0..N, where N is the last message number in the catalog. Perhaps a different initialization routine will be written in the future, but it is unlikely.

See also HpZNlsSubset.

HpCrtPageMode

This call sends the escape sequences to the terminal that place it in block page mode. It also does the control call to the driver that is required for page mode operation.

```
CALL HpCrtPageMode(lu,flag)

integer*2 lu
logical*2 flag
```

where:

lu is the LU number of the terminal to be strapped for line mode.

flag specifies a flag to control forms mode:

```
.TRUE.  = enable protected forms mode.
.FALSE. = disable protected forms mode.
```

The escape sequence generated by this routine is:

```
Esc & k 1 a 1 B Esc & s 1 d 0 g 0 H Esc X Esc b
```

It is the caller's responsibility to ensure that the LU is legal, that the terminal is capable of page mode operation, and that the terminal driver is also capable of block mode transfers (DDC00, DVC05, for example).

See also HpCrtReadPage, HpCrtLineMode, and HpCrtCharMode.

HpCrtParityChk

This logical*2 function performs a parity check on a data buffer. It will return .TRUE. if there are NO parity errors in the buffer.

```
IF (HpCrtParityChk(ibuf,length,par)) THEN
  no_parity_error
ENDIF

integer*2 ibuf(*),length,par
logical*2 HpCrtParityChk
```

where:

ibuf is the input buffer.

length is the number of characters in the input buffer.

par specifies odd/even parity select as follows:

```
set to 0 for even parity
set to 1 for odd parity
```

The companion routine HpCrtParityGen can be used to compute parity in a user buffer.

HpCrtParityGen

This subroutine computes and sets the parity bits in a data buffer. It works on word boundaries, so the byte following the data will be altered if an odd number of data bytes is specified.

```
CALL HpCrtParityGen(ibuf,length,par)

integer*2 ibuf(*),length,par
```

where:

ibuf specifies the input and output buffer. The conversion is done in place.

length is the number of characters in the input buffer.

par specifies odd/even parity as follows:

set to 0 for even parity.
set to 1 for odd parity.

The companion routine HpCrtParityChk can be used to check for correct parity in a buffer.

HpCrtQTDPort7

This integer*2 function returns the LU of the seventh port when given the LU of one of the modem ports on a Rev. D 8-channel MUX. The LU of Port 7 is needed to perform writes for auto-dial when the MUX is connected to an HP 37214 modem card cage. Port 7 must have been initialized before this call is made, *lu* must be configured as a modem (bit 13 was set when the LU was initialized with a CN 30B), and it must be a Rev. D MUX.

```
port7lu = HpCrtQTDPort7(lu,workbuffer)

integer*2 port7lu,lu,workbuffer(*),HpCrtQTDPort7
```

where:

port7lu returns the LU number of Port 7.

lu is the LU number of one of the other ports on that 8-channel MUX. It must be configured as a modem port.

workbuffer is used by this routine to do a Special Status read. It must be at least 32 words long.

If for any reason the Port 7 LU cannot be determined, this routine returns a zero.

HpCrtReadChar

This function inputs directly from an LU to a character data type variable. As is usual for character data types, it will be blank filled if the user enters less data than the full length allowed. The variable can be even or odd length, and need not start on a word boundary. The input is done via REIO to allow for swapping.

```
CALL HpCrtReadChar(lu,charvariable,status,bytelen)

integer*2 lu,status,bytelen
character*(*) charvariable
```

where:

lu	is the LU number from which to read; it must be in the same format as an EXEC 1 call.
charvariable	returns the data read from the specified LU.
status	returns the status (A-Register).
bytelen	returns the transmission log (B-Register).

The caller must set the bits in the LU word in the same way that they would be set in an EXEC call; for example, bit 8 must be set to enable echo.

The status and bytelen variables returned are the A- and B-Register returns from the REIO call, which is done with the no-abort option. For normal call completion, the sign bit of the status will be set and the low byte will have the usual I/O status bits, including the timeout bit in bit 1. *bytelen* will be valid. For an abnormal call completion, *status* and *bytelen* (A- and B-Registers) will contain the ASCII error code, such as “IO13”.

See also HpCrtXSendChar, HpCrtXMenu, HpCrtSendChar, and HpCrtMenu.

HpCrtReadPage

This integer*2 function does a page mode write/read call. The write portion does a keyboard unlock. The read is done without echo and with the auto-home bit set. Thus the keyboard is again locked when the routine exits.

iostatus = HpCrtReadPage(*lu*,*buffer*,*bufferlen*,*bytelen*)

integer*2 *iostatus*,*lu*,*buffer*(*),*bufferlen*,*bytelen*,HpCrtReadPage

where:

iostatus returns the status word from the EXEC call. Refer to the driver documentation for the bit assignements.

lu is the LU number of the HP terminal; for RTE-A *lu* can range from 1 to 255.

buffer returns the input buffer; the size must be sufficient to receive the data from the largest screen that is to be read.

bufferlen specifies the length of the input buffer; positive number of words or negative number of bytes.

bytelen returns the number of bytes that were read from the screen.

The programmer must ensure the following conditions exist before this routine is called:

- The port driver is compatible with block/page/auto-home operation. Call HpCrtSSRCDriver to verify.
- The driver in use is configured for driver 5 operation. Verify by executing an EXEC 13.
- The port is legal for use. Verify by executing an EXEC 13 call with the no-abort bit.
- To operate in “altered fields only” mode you must send the appropriate CRT escape sequences. Refer to the terminal reference manuals. Use either formatted writes, EXEC calls, or HpCrtXSendChar.
- The terminal screen has been “painted” with the proper form. Refer to the terminal reference manuals.
- The terminal is in block/page mode and a CN25 call has been done. Use HpCrtPageMode to do this.

After this call, the programmer should perform the following checks:

- Bit 1 of the status is set if the request timed out. The usual course of action is to simply call this routine again.
- The transmission log as given by *bytelen* cannot be zero unless the terminal was reset by the user or the read timed out. Normally you would loop back in the program to the point where the terminal was set up, including sending the forms again.

If the results of the above checks are good, the programmer can then use the data that was read from the screen. HpCrtGetField is a useful subroutine for this purpose.

HpCrtRestorePort

This subroutine resets a port to the conditions that were in effect when HpCrtSavePort was called.

```
CALL HpCrtRestorePort(lu,statebuffer)

integer*2 lu,statebuffer(32)
```

where:

lu is the LU number of the port (in the range 1..255).

statebuffer specifies a 32-word buffer that contains the original state of the port that is to be restored.

Note This routine temporarily enables the primary and secondary programs during the process of restoring the port. This occurs even if 1) both the primary and secondary programs were disabled prior to calling HpCrtRestorePort and 2) the state being restored has them disabled. If this affects your application, it is suggested that you disable the primary or secondary programs by passing the CN 20 or CN 40 control requests an invalid program name, for example "−", instead of using CN 21 or CN 41. Refer to the serial driver documentation in the *RTE-A Driver Reference Manual*, part number 92077-90011.

HpCrtSavePort

This logical*2 function reads the current state of a serial port that is driven by one of the Special Status Read compatible drivers. Refer to Table 12-1 for the definitions of words 1 through 32 of the state buffer that is returned by an HpCrtSavePort call. The contents of *statebuffer* is described in the Special Status Read section of the serial driver documentation.

```
error = HpCrtSavePort(lu,statebuffer)

logical*2 error,HpCrtSavePort
integer*2 lu,statebuffer(32)
```

where:

error returns .TRUE. if the call failed.

lu is the LU number of the port (in the range 1..255).

statebuffer is a 32-word buffer to hold the current state of the port.

statebuffer should not be altered if the state is to be returned to the current state by calling HpCrtRestorePort. It is a good idea to declare *statebuffer* in SAVE address space.

HpCrtRestorePort should not be invoked if this routine returns an error.

See also FakeSpStatus.

HpCrtSchedProg, HpCrtSchedProg_S

This function passes the name of the program to be scheduled on unexpected interrupt to the terminal drivers. In RTE-6/VM, only the primary program name can be set.

```
CALL HpCrtSchedProg(lu,5hprmry[,2hpr])      ! pass primary name
CALL HpCrtSchedProg(lu,5hsecnd,2hse)        ! pass secondary name
```

An alternative form of the routine is available for use with FORTRAN character data type variables:

```
CALL HpCrtSchedProg_S(lu,'prmry','pr')
CALL HpCrtSchedProg_S(lu,'secnd','se')
```

HpCrtScreenSize

This function returns the width and height (in characters) of the visible surface of the screen of an HP terminal. The width and height parameters are not modified if an error occurs.

```
IF (HpCrtScreenSize(crt,width,height)) THEN
  Good_Width_Height
ENDIF

integer*2 crt,width,height
logical*2 HpCrtScreenSize
```

where:

crt is the LU number of the terminal.

width is the screen width (in number of characters).

height is the screen height (in number of characters).

See also HpCrtCheckStraps.

HpCrtSendChar

This routine calls EXEC to print a FORTRAN character variable or literal. It computes the word address of the buffer and handles the case of a buffer starting on an odd byte boundary.

```
    CALL HpCrtSendChar(lu,'characters to print')
```
or
```
    CALL HpCrtSendChar(lu,character_variable)

    integer*2 lu
```

The caller must set the bits in the LU in the same way that they would be set in an EXEC call; for example, bit 10 must be set to inhibit the Carriage Return and Line Feed that are normally appended to the buffer by the drivers.

Because EXEC calls can be done only from word (even byte) addresses, this routine must handle the occasional odd byte case in a special way. The word that contains the first byte of the specified buffer is saved inside the subroutine while the left byte is replaced by a null for the duration of the EXEC call. After the EXEC call completes, the original contents of the word is restored. The null byte does not display on terminals, so the user is not aware of the extra byte preceding the data. However, there are some devices, such as the HP 2608 lineprinter, which do print nulls. The programmer should be aware of this, especially when using the subroutine to display substrings. The FORTRAN compiler does not produce odd byte addresses for character literals. In fact, almost the only time that odd byte addresses are generated by FTN7X is for the substrings mentioned above.

See also HpCrtXMenu, HpCrtMenu, and HpCrtXSendChar.

HpCrtSSRCDriver, HpCrtSSRCDriver?

This logical function determines if the driver for a given LU will respond to a Special Status Read that is supported by the serial drivers. Once this capability is known, the Special Status Read can be used to retreive any further information needed by the application.

```
logical*2 HpCrtSSRCDriver     ! must be declared as a logical function

If (HpCrtSSRCDriver(lu)) THEN
   continue                   ! The LU is driven by a Special Status
                              ! Read Compatible driver.
ELSE
   continue                   ! The driver does not do Special Status
                              ! Reads, or there was an error generated
                              ! by the XLuEx call.
ENDIF
```

The method used is to issue an XLUEX control call 6, with the optional parameter equal to −2. The new drivers return 123456b in the B-Register. Because the driver must be entered, this call will suspend if the driver is busy. Note that because XLUEX is used and the *lu* parameter is used as the first word of the *cntwd* array when XLUEX is called, you may set the OV bit (bit 15) or the OS bit (bit 14) in the *lu* parameter to override LU 1 mapping or spooling, respectively.

For special purpose programs that must avoid suspension, another version of the call is available, "HpCrtSSRCDriver?". The method used by this call is to ensure that the driver type is 0, 5, 6, or 12 and the sign bit of DVT word 20 has been cleared. Prior serial drivers such as DD.00 and ID.00 did not change this bit. Unfortunately, Hewlett-Packard has no knowledge of what all customers have done, so it is possible that a customer written driver may cause this routine to be in error. Because the method is less robust, the subroutine is not directly callable from FORTRAN. It is available by using the $ALIAS construct. Note that the call will fail even for a compatible driver until the driver has been entered the first time.

Use caution when calling either of these routines for non-serial driver LUs. For example, HP-IB LUs may hang in response to this call.

HpCrtStatus

This call does an XLuEx write/read call to read the status of an HP terminal. The returned status is unpacked, one byte per word.

```
CALL HpCrtstatus(lu,flag,statusbuffer)

integer*2 lu,flag,statusbuffer(7)
```

where:

lu	is the LU number of the terminal (in the range 1..255).
flag	specifies a flag equal to 2hPR for primary status or 2hSE for secondary status.
statusbuffer	is a 7-word buffer to receive the terminal status bytes, one byte per word. Refer to the terminal manual for definitions of the status bytes.

It is the responsibility of the programmer to ensure that the terminal is an HP terminal which responds to the type of status requested. The programmer must also ensure that the interface card and driver are capable of burst transmissions, such as DDC00 and DVC05.

This call produces a write/read call to the given LU. The write portion contains “Esc Z Esc ~” or “Esc Z Esc ^”. The reply is of the form “Esc \ 4 0 0 ? 4 2 0 Cr”. The “400?420” is then unpacked into the caller’s buffer, one byte per word with the upper byte of each set to null.

HpCrtStripChar

This routine deletes characters from a buffer. Its primary use is to remove the Unit Separator characters that separate the lines of data read from a terminal in block mode for backwards compatibility with the RTE-6/VM driver DVR05/DVA05. DVR05 stripped the Unit Separators automatically, but no other drivers do so.

newlength = HpCrtStripChar(*ibuf*,*num*,*char*)

integer*2 *newlength*,*ibuf*(*),*num*,*char*,HpCrtStripChar

where:

newlength returns the new buffer length in number of characters.

ibuf specifies the buffer containing the data to modify and returns the modified data. The conversion is done in place.

num specifies the number of characters to process.

char specifies the character to remove (in the low byte).

To convert a program that was written for DVA05 to use the new drivers, the screen input buffer must be increased in size to accomodate one more character per unprotected field and the read request length must be increased accordingly. After the read completes, this routine should be called to remove the Unit Separator characters (octal 37). For example:

bytelen = HpCrtStripChar(*ibuf*,*bytelen*,37b)

At this point, the program can proceed as if DVA05 were still in use.

HpCrtStripCntrls

This routine strips non-displayable characters from a string. It deletes all characters less than blank (40b). When the control character is Escape (33b), it attempts to eliminate the rest of the sequence as well.

newlen = HpCrtStripCntrls(*ibuf*,*bytelen*)
fakestring = HpCrtStripCntrls(*ibuf*,*bytelen*)

integer*2 ibuf(*),bytelen

where:

newlen if this routine is declard integer*2, it returns the length in the A-Register.
fakestring if this routine is declard integer*4, it returns a string descriptor in the A- and B-Registers.

ibuf is the data buffer to be modified; modification is done in place.

bytelen is the number of characters to be passed into the routine.

See also HpCrtStripChar.

HpCrtXMenu

This routine prints multiple character strings to an LU. Each string is printed by a separate XLuEx call, in the same way as HpCrtXSendChar.

```
CALL HpCrtXMenu(lu,'1st line of menu',
>                  '2nd line'
>                  '3rd line'
                     ...
>                  'last line')
```

The number of strings passed is unlimited, and either quoted strings (literals) or FTN7X character variables can be passed.

Refer to the documentation for HpCrtXSendChar for a caution about certain printers.

The LU parameter is a double word value as is used in an XLuEx call.

See HpCrtXSendChar, HpCrtXReadChar, and HpCrtMenu.

HpCrtXReadChar

This function inputs directly from an LU into a character data type variable. As is usual for a character data type, it will be blank filled if the user enters less data than the full length allowed. The variable can be even or odd length and need not start on a word boundary. The input is done via XREIO to facilitate swapping.

```
CALL HpCrtXReadChar(lu,charvariable,status,bytelen)

integer*2 lu(2),status,bytelen
character*(*) charvariable
```

where:

lu	is the LU number, a double word value in the same format as an XLUEX call.
charvariable	returns the data read from the terminal.
status	returns status (A-Register).
bytelen	returns status (B-Register).

The caller must set the bits in the CONWD (the second word of the *lu* parameter) in the same way that they would be set in an XLUEX call; for example, bit 8 must be set to enable echo.

The status and bytelen variables returned are the A- and B-Register returns from the XREIO call, which is done with the no-abort option. For normal call completion, the sign bit of the status will be set and the low byte will have the usual I/O status bits, including the timeout bit in bit 1. The bytelen will be valid. For an abnormal call completion, the status and bytelen (A and B) will contain the ASCII error code, such as 'IO13'.

See also HpCrtReadChar, HpCrtXSendChar, and HpCrtXMenu.

HpCrtXSendChar

This routine calls XLUEX to print a FTN7X character variable or literal. It computes the word address of the buffer and handles the case of a buffer starting on an odd byte boundary.

```
    CALL  HpCrtXSendChar(lu,'characters to print')
```
or
```
    CALL  HpCrtXSendChar(lu,character_variable)
```

where:

lu is the LU number used in the XLUEX call.

The caller must set the bits in the CONWD (the second word of the LU parameter) in the same way that they would be set in an XLUEX call; for example, bit 10 must be set to inhibit the Carriage Return and Line Feed that are normally appended to the buffer by the drivers.

Because EXEC and XLUEX calls can be done only from word (even byte) addresses, this routine must handle the occasional odd byte case in a special way. The word which contains the first byte of a specified buffer is saved inside the subroutine while the left byte is replaced by a null for the duration of the EXEC call. After the EXEC call completes, the original contents of the word are restored. The null byte does not display on terminals, so the user is not aware of the extra byte preceeding the data. However, there are some devices, such as the HP 2608 Lineprinter, which do print nulls. The programmer should be aware of this, especially when using the subroutine to display substrings. The FORTRAN compiler does not produce odd byte addresses for character literals. In fact, almost the only time that odd byte addresses are generated by FTN7X is for the substrings mentioned above.

See also HpCrtXMenu and HpCrtXReadChar.

HpLowerCaseName

This subroutine changes the name of the program that calls it to the lowercase equivalent. Once this is done, the program is safe from user commands entered through any of the standard RTE user interfaces, as they all do upshifting. It is still possible to affect the program programmatically because the EXEC calls themselves do not do casefolding. Note that the $ALIAS command is required because this call is not .ENTR compatible.

```
$ALIAS  HpLowerCaseName,DIRECT

CALL  HpLowerCaseName()
```

For example, if a program named “XYZ12” calls this routine, the ID segment name field is altered to “xyz12”.

This routine should be called by any program which could adversely affect the system if it were killed.

HpRteA

This logical*2 function determines whether the calling program is running on an RTE-A system. The method used is to read $OPSY and to check if it is in the range −33 to −128 as defined for RTE-A. Note that the $ALIAS command is required because this routine is not .ENTR compatible.

```
$ALIAS HpRteA,DIRECT

logical*2 HpRteA

IF (HpRteA()) THEN
  CALL HpCrtSendChar(1,'This is an RTE-A system')
ELSE
  CALL HpCrtSendChar(1,'This is not an RTE-A system')
ENDIF
```

See also HpRte6.

HpRte6

This logical*2 function determines whether the calling program is running on an RTE-6/VM system. The method used is to read $OPSY and check to see whether it is in the range −17 to −28 as defined for RTE-6/VM. Note that the $ALIAS command is required because this routine is not .ENTR compatible.

```
$ALIAS HpRte6,DIRECT

logical*2 HpRte6

IF (HpRte6()) THEN
  CALL HpCrtSendChar(1,'This is an RTE-6 system')
ELSE
  CALL HpCrtSendChar(1,'This is not an RTE-6 system')
ENDIF
```

See also HpRteA.

HpZ, Mini-Formatter

The mini-formatter is a set of routines designed to provide formatted text I/O for systems programs. It can be called from FORTRAN in place of the FTN77 formatter, or from other languages.

Certain routines manipulate the pointers so that you can “erase” output characters or back up to re-parse input characters. You can even “clear” the line if you start building a line and then discover that you do not need it. Some routines can be used to inquire about the current positions of the pointers.

Because it was designed primarily for the systems programmer, it has much less extensive error checking, and dispenses with support of floating point conversions entirely. Its advantages are that it provides support for a wider range of representations, such as octal, hexadecimal, transparency ASCII, mnemonic ASCII, and inverse assembly, as well as being much smaller and faster code. It is more difficult to use, but provides greater flexibility.

How to Use the Mini-Formatter to Do Output

Call HpZDefOBuf to define the buffer that will be used by the mini-formatter. The buffer can be of any size, but it is better to make it bigger than you think you will need (rather than smaller), because the mini-formatter does no error checking when it adds characters to the output buffer (you can overwrite your code!).

Now “write” data into the buffer from left to right, just as if you were writing on paper. Each element of the line will require a call to one of the various format routines. The routine puts its output into the output buffer at the current position indicated by an internal pointer and then updates the pointer. Because the output line is being built in memory, you can use DO loops, IF statements, or other control structures as needed to make the line look the way you desire. When you have finished building a line, write it to a device, a file, or a buffer with the appropriate calls. This action sets the internal pointer back to the start of the buffer so you can start building the next line of output.

How to Use the Mini-Formatter to Do Input

After a line of input text is read from a device or a file, call HpZDefIbuf to tell the input routines where the input text is located and the valid length. Then call the various parsing routines to extract tokens from left to right as the internal pointer moves across the line.

Precautions

Because of the use of internal pointers and other related reasons, the routines HpZDefOBuf and HpZDefIbuf must be called in the main at least once when the routines are used in a segmented or CDS program. By the same token, the buffers should be either in the main, or in common, or in SAVE address space so that they are available at all times.

It is very easy to add new output formats to this package, so if a format becomes too awkward to do with individual calls, write a subroutine to accomplish the same task.

A sample program:

```
Ftn7x,q,s
     program sample
     implicit none

     integer*2                  ! variables
     > Crt,                     ! LU of user's terminal
     > Parameter,               ! counter for number of parameters
     > Type,                    ! type of a parameters
     > Bytelen,                 ! length of data in input buffer
     > Ipntr,                   ! pointer to first byte of runstring
                                ! following "ru,sample,"
     > Index                    ! do loop index for subparameters

     integer*2                  ! arrays
     > OutputBuffer(0:127),     ! output buffer for HpZ mini-formatter
     > InputBuffer(0:127),      ! input buffer for HpZ parse routines
     > Result(0:14)             ! buffer to receive parse results

     logical*2                  ! variables
     > EndOfString              ! true when last parameter processed

     logical*2                  ! functions
     > GetRunString,            ! true if NO runstring
     > HpZParse                 ! true if parameter parsed

     data
     > Crt /1/
 *---------------------------------------------------------------

     call HpZDefOBuf(OutputBuffer)     ! initialize mini-formatter output

     if ( GetRunString(InputBuffer,-256,ByteLen,Ipntr) ) then
        call HpCrtSendChar(Crt,'No runstring provided')

     else
        EndOfString = .false.
        parameter = 1

        call HpZDefIbuf(InputBuffer,ByteLen,Ipntr)

        do while ( HpZParse(Result) )     ! loop until all parameters
                                          ! have been parsed
          Call HpZmvs('Parameter #')      ! show parameter number
          Call HpZdecv(Parameter)
          Call HpZmvs(' type word = ')    ! and its type word
          Call HpZWriteLu(Crt)

          Type = ibits(Result(0),0,2)     ! extract type of 1st param
```

```
      call HpZmvs(' Subparameter 1 = ')
      if ( type.eq.3 ) then           ! if ASCII,
        call HpZmvc(Result(1),8)      ! print as 8 alphabetic characters
      else
        call HpZdecv(Result(1))       ! print as a numeral
      endif
      call HpZWriteLu(Crt)            ! display it

      type = Result(0)
      do index = 9,14                  ! show the remaining subparameter
        call HpZmvs(' Subparameter ')
        call HpZdecv(index-7)
        type = ishft(type,-2)
        if ( ibits(type,0,2).eq.3 ) then
          call HpZmvc(Result(index))
        else
          call HpZdecv(Result(index))
        endif
        call HpZWriteLu(Crt)
      enddo
      call HpZWriteLu(Crt)

    enddo
endif
end
```

HpZAscii64

This subroutine copies the characters from the given input buffer to the output buffer defined by a prior call to HpZDefOBuf. As the characters are copied, they are examined to ensure that they have values in the range 40b through 137b, the TTY compatible subset of the ASCII characters. If the character is not in that range, the low byte of the replacement value given in the call is substituted.

```
CALL HpZAscii64(ibuf,length,replacechar)

integer*2 ibuf(*),length,replacechar
```

where:

ibuf	specifies the buffer containing the characters to be copied.
length	specifies the length of the input buffer in bytes.
replacechar	specifies the replacement value.

The output length will be the number of bytes specified in the call.

See also HpZAscii95, HpZAsciiHpEnh, and HpZAsciiMne3.

HpZAscii95

This subroutine copies the characters from the input buffer to the output buffer defined by a prior call to HpZDefOBuf. As the characters are copied, they are examined to ensure that they have values in the range from 40b through 176b, the printable subset of the ASCII characters. If the character is not in that range, the low byte of the replacement value given in the call is substituted.

```
CALL HpZAscii95(ibuf,length,replacechar)

integer*2 ibuf(*),length,replacechar
```

where:

ibuf	specifies the buffer containing the characters to be copied.
length	specifies the length of the input buffer in bytes.
replacechar	specifies the replacement value.

The output length will be the number of bytes specified in the call.

See also HpZAscii64, HpZAsciiHpEnh, and HpZAsciiMne3.

HpZAsciiHpEnh

This subroutine copies characters from the input buffer to the output buffer defined by a prior call to HpZDefOBuf. As the characters are copied, they are processed according to the following algorithm:

- The character enhancement is set to none.
- If the parity bit (bit 7) is set, underlining is turned on and 200b is subtracted from the character.
- If the character is a control code (0..37b), inverse video is turned on and 100b is added to the character.
- If the character is 177b (Delete or rub-out), half bright inverse video is turned on and the character is replaced by a blank.

The new enhancement generated by the process above is compared to the current enhancement value as generated by preceeding characters; if different, the new enhancement is emitted. The character as altered by the process above is emitted (it will be in the range 40b..176b).

After the last character is processed, if any video enhancement is currently in effect, the "Esc & d @" sequence is appended to turn off all enhancements.

```
CALL HpZAsciiHpEnh(ibuf,length)

integer*2 ibuf(*),length
```

where:

ibuf specifies the buffer containing the characters to be displayed.

length specifies the number of bytes to output.

For an input string of N bytes, the output length will be in the range of N to 5N, so be sure to allow sufficient space in the buffer that is declared by HpZDefOBuf. It is also a good idea to be aware that enhancements are cancelled when the line wraps around on the CRT terminals, but retained on the hardcopy terminal (HP 2635) and lineprinters. It is best to restrict the line to fewer than 80 displayable characters.

See also HpZAscii95, HpZAscii64, and HpZAsciiMne4.

HpZAsciiMne3

This routine copies the characters in the input buffer into the output buffer defined by a previous call to HpZDefOBuf. As each character is copied, it is examined to see if it falls in the 95 printable character subset of ASCII. If it does, it is copied without change, and two blanks are appended. If it does not, the mnemonic for the control character is placed in the buffer instead. Thus, three characters are output for each character of input.

```
CALL HpZAsciiMne3(buffer,count)

integer*2 buffer(*),count
```

where:

buffer specifies the input buffer containing the characters which are to be formatted.

count specifies the length of *buffer* in bytes.

The parity bit (bit 7 of each character) is masked by this routine.

For example, if the input contains:

```
040502b,060542b,034001b,006412b,077614b,020040b
```

the output will be:

```
A  B  a  b  BelSohCr Lf DelFfSpcSpc
```

See also HpZAsciiMne4, HpZAscii95, and HpZAsciiHpEnh.

HpZAsciiMne4

This routine copies the characters in the input buffer into the output buffer defined by a previous call to HpZDefOBuf. As each character is copied, it is examined to see if it falls in the 95 printable character subset of ASCII. If it does, it is copied without change and three blanks are appended. If it does not, the mnemonic for the control character is placed in the buffer instead and a single trailing blank is appended. Four characters are output for each character of input.

```
CALL HpZAsciiMne4(buffer,count)

integer*2 buffer(*),count
```

where:

buffer specifies the input buffer containing the characters which are to be formatted.

count specifies the length of *buffer* in bytes.

The parity bit (bit 7 of each character) is ignored by this routine.

For example, if the input contains:

```
040502b,060542b,034001b,006412b,077614b,020040b
```

the output will be:

```
A   B   a   b   Bel Soh Cr  Lf  Del Ff  Spc Spc
```

See also HpZAsciiMne3, HpZAscii95, and HpZAsciiHpEnh.

HpZBackSpaceIbuf

This routine backs up the character pointer to the input buffer defined by a prior call to HpZDefIbuf.

```
CALL HpZBackSpaceIbuf
```

See also HpZGetNextStrDsc and HpZGetNextToken.

HpZBinc

This routine converts a number to binary. The output length is 1..15 bytes, as specified in the call.

```
CALL HpZBinc(ivalue,numdigits)

integer*2 ivalue,numdigits
```

where:

ivalue is the number to be converted to binary. It is stored into the output buffer defined by a prior call to HpZDefOBuf.

numdigits is the number of digits to produce; must be in the range 1..15.

See also HpZBino.

HpZBino

This subroutine converts the passed value to its binary ASCII representation at the current position in the output buffer defined by a prior call to HpZDefOBuf. The output length is 16 bytes.

```
CALL HpZBino(ivalue)

integer*2 ivalue
```

HpZDeco

This subroutine stores the ASCII decimal representation of an integer*2 number at the current position in the output buffer defined by a prior call to HpZDefOBuf. The output will be 6 characters wide with leading zeros and a leading blank or negative sign as required.

```
CALL HpZDeco(ivalue)

integer*2 ivalue
```

Examples: `'-00001', '-32768', ' 00005', ' 00000'`

See also HpZDecv, HpZDecc, HpZUdeco, HpZUdecv, and HpZDicv.

HpZDecv

This subroutine converts an integer*2 number into ASCII decimal representation, suppressing leading zeroes. The conversion is done to the buffer defined by a prior call to HpZDefOBuf. The output length is in the range of 1 to 6 bytes.

```
CALL HpZDecv(ivalue)
```

See also HpZDeco, HpZDecc, HpZUdeco, HpZUdecv, and HpZDicv.

HpZDecc

This routine converts the given value to ASCII numerals at the current position in the output buffer defined by a prior call to HpZDefOBuf. The conversion is to **unsigned** decimal constant width format with either leading zeroes or leading blanks as specified by the sign of *fieldwidth*. *fieldwidth* must be in the range 1..5 or −1..−5.

```
CALL HpZDecc(value,[-]fieldwidth)

integer*2 value,fieldwidth
```

where:

value is the value to be represented in ASCII decimal.

fieldwidth is the minimum width of the output; it must be in the range 1..5 or −5..−1.

Examples:

```
call HpZDecc(52,5) → '00052'    call HpZDecc(52,-5) → '   52'
call HpZDecc(52,4) → '0052'     call HpZDecc(52,-4) → '  52'
call HpZDecc(52,3) → '052'      call HpZDecc(52,-3) → ' 52'
call HpZDecc(52,2) → '52'       call HpZDecc(52,-2) → '52'
call HpZDecc(52,1) → '2'        call HpZDecc(52,-1) → '2'
```

See also HpZDecv, HpZDeco, HpZUDeco, and HpZUDecv.

HpZDefIBuf

This routine is used to declare the attributes of the input buffer that is referenced by the various parsing routines such as HpZParse.

```
CALL HpZDefIBuf(buffer,ilength,position)

integer*2 buffer(*),ilength,position
```

where:

buffer specifies the buffer where the string to be examined is located.

ilength specifies the number of valid characters in the input buffer.

position specifies the starting character position in the buffer. Zero indicates the first character (leftbyte of the first word).

This routine must be loaded with the main in a segmented or CDS program. The buffer should be in the main, in common, or in SAVE address space.

See also HpZDefOBuf, HpZDParse, and HpZGetNextToken.

HpZDefIString

This routine defines a string as the input for HpZ routines.

```
CALL HpZDefIString(stringbuffer)

character*(*) stringbuffer
```

where:

stringbuffer is a character variable (or literal) that contains data to parse with the HpZ routines.

See also HpZDefIBuf and HpZParse.

HpZDefOBuf

This routine defines the output buffer for the Mini-Formatter.

```
CALL HpZDefOBuf(workbuffer)

integer*2 workbuffer(*)
```

where:

workbuffer is a buffer to receive the characters from the various formatting routines.

The work buffer should be in the main, in common, or in SAVE address space so that it is always available.

See also HpZDefIBuf and HpZPushOBuf.

HpZDicv

This routine puts the ASCII decimal representation of a double integer value (integer*4) into the output buffer previously defined by a call to HpZDefOBuf. The routine stores the minimum representation, so no leading blanks or zeros will be produced. The output length will be from 1 to 11 bytes.

```
CALL HpZDicv(ivalue)

integer*4 ivalue
```

where:

ivalue is the number to be converted to ASCII decimal representation.

See also HpZDecv.

HpZDParse

This logical*2 function parses the next occurring token in the input buffer defined by a prior call to HpZDefIbuf.

```
IF ( HpZDParse(obuf,olength,ocount,type,offset) ) THEN something_was_parsed ENDIF

integer*2 obuf(*),olength,ocount,type,offset
logical*2 HpZDParse
```

where:

obuf is a buffer that holds the result of the parse. For maximum utility, equivalences should be defined to allow access to *obuf* as integer*2 and integer*4 as well as an integer array of characters.

olength is the number of characters allowed in *obuf*, must be at least 4.

ocount is the number of characters written to *obuf*, if *type* = 3.

type is a flag indicating the type of data found in the input buffer:

0 Null parameter
1 16-bit integer, signed or unsigned
2 32-bit integer, signed or unsigned
3 ASCII string

offset if non-zero, this parameter indicates the offset where a subparameter starts. The input buffer can then be reparsed by another call to HpZDParse to retrieve the next subparameter.

HpZDParse returns .TRUE. if something was found in the input buffer that could be parsed; otherwise, .FALSE. is returned.

The parse is performed as follow:

- skips leading white space (blanks and tabs).
- ends on a comma, trailing white space, or the end of the buffer.
- defines a subparameter when it encounters a colon (:) or an equal sign (=).

When a subparameter is parsed, the offset is noted and the end of the parameter is then sought. In this way the input buffer can be reparsed by another call to HpZDParse to retrieve the next subparameter (or ignored).

To reparse, use the offset value as a parameter to the HpZIBufReset call.

Examples:

Input	Type Returned	Offset Returned	Interpretation
123	1	0	decimal 123
123b	1	0	octal 123
0123z	1	0	hexadecimal 123
0123zz	3	0	an alpha string ‘0123zz’
a123z	3	0	an alpha string ‘a123z’
0a123z	1	0	hexadecinal a123
123bb	3	0	an alpha string ‘123bb’
foo::2	3	4	an alpha string ‘foo’
	0	5	reparse yields a null
	1	0	reparse yields decimal 2.
mt=8	3	3	an alpha string ‘mt’
	1	0	reparse yields decimal 8
277777b	2	0	double integer octal 277777 or decimal 98303
1000000	2	0	double integer one million

Example program:

```
ftn7x,q,s
       program ptest
       implicit none

       integer*2 ! variables
      > Type,
      > ByteLen,
      > OutLen,
      > Offset

       integer*2 ! arrays
      > Ibuf(0:127),
      > Obuf(0:127),
      > Result(0:39)

       logical*2 ! variables
      > More

       logical*2 ! functions
      > GetRunString,
      > HpZDParse

  ------------------------------------------------------------------

       call HpZDefObuf(Obuf)

       if ( GetRunString(Ibuf,-256,ByteLen,Offset) ) then
         call HpCrtMenu(1,'Nothing to parse!')
```

```
        go to 9999
      endif

      call HpZDefIbuf(Ibuf,ByteLen,Offset)

      Offset = 0
      More = .true.
      do while ( More )
        if ( Offset.ne.0 ) then
          call HpZmvs('  subparameter ')
          call HpZIbufReset(Offset)
        endif
         More = HpZDParse(Result,80,OutLen,Type,Offset)
        if ( .not.More ) then
          call HpZmvs('End of buffer')

        else
          if ( Type.eq.0 ) then
            call HpZmvs('Null')

          elseif ( Type.eq.1 ) then
            call HpZmvs('Single Integer ')
            call HpZdecv(Result(1))

          elseif ( Type.eq.3 ) then
            call HpZmvs('Ascii ')
            call HpZmvc(Result(0),OutLen)

          elseif ( Type.eq.2 ) then
            call HpZmvs('Double Integer ')
            call HpZDicv(Result(0))

          else
            call HpZmvs('HpZDParse defective, type = ')
            call HpZdecv(Type)
            more = .false.
          endif
        endif
        call HpZWriteLu(1)
      enddo
      call HpZWriteLu(1)

9999  end
```

See also HpZParse, NAMR, and HpZGetNextToken.

HpZDumpBitMap

This routine is useful for debugging, to display a bit map. The bit map can contain up to 32,768 bits (8K words). The output consists of a display of the bit numbers in ASCII.

```
CALL HpZDumpBitMap(lu,bitmap,lastbit[,width])

integer*2 lu,bitmap(*),lastbit,width
```

where:

lu is the LU number of the output device. It must be in the range of 1 to 63.

bitmap is a variable length array containing the bit map.

lastbit is a one-word integer representing the last bit to be displayed.

width is an optional one-word integer representing the field width for the numbers as specified for HpZDecc. The default is 0 which specifies a variable width.

If a bit is set, the number will be displayed in inverse video. The bits in the map are numbered from 0 to N, where the MSB (most significant bit) of the first word is bit 0.

For example, if the bit map buffer contains only one 16-bit number, 64703B, the output would look like:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Because the output of this routine uses the terminal escape sequences for inverse video, the output device should be a terminal, not a printer.

This call uses the output buffer defined by a prior call to HpZDefOBuf as its work buffer. The display lines are up to 79 displayable characters wide, but the invisible escape sequences which turn inverse video on and off add to that to make the worst case line 230 characters total. Therefore, the work buffer must be at least 115 words long.

See also HpZDefOBuf, HpZDumpBuffer, and SetBitMap.

HpZDumpBuffer

This routine is useful for debugging, to dump a buffer in many different formats at once. It uses the HpZ mini-formatter routines, so they must have been initialized by calling HpZDefOBuf with a buffer of at least 80 words (160 characters).

```
CALL HpZDumpBuffer(crt,ibuf,wordlen[,format])

integer*2 crt,ibuf(*),wordlen,format
```

where:

crt is the LU number of the list device.

ibuf specifies the buffer to be displayed.

wordlen specifies the length of the buffer in words.

format is an optional parameter which specifies the display routines to be used to format the data for printing. The bits are assigned the following meanings:

Bit	Format	Example
0	Decimal conversion	' 00001 00031 16737'
1	Octal conversion	'000001 000037 040541'
2	Hexadecimal conversion	' 0001 001F 4161'
3	ASCII mnemonics	'NulSoh Nul Us A a'
4	ASCII 95 character subset	' .. .. Aa'
5	ASCII HP video enhancements	' @A @_ Aa'
6	Hexadecimal bytes	' 00 01 00 1F 41 61'
13..7	Reserved for future enhancements	
14	Emit a blank line before and after the dump	
15	Emit a blank line between 10-word groups	

If *format* is not supplied, it defaults to 47b, which displays in decimal, octal, hexadecimal, and HP video ASCII.

See also HpZDeco, HpZOcto, HpZHexo, HpZHexc, HpZAsciiHpEnh, HpZAscii95, and HpZAsciiMne3.

HpZFieldDefine

This call issues up to three escape sequences to define a field in a block mode screen and optionally set display enhancements.

```
CALL HpZFieldDefine(onoff,fieldattribute)

integer*2 onoff,fieldattribute
```

where:

onoff specifies the flag to define the field as protected or unprotected.

fieldattribute specifies two characters to define the edit checks and video enhancements for the field.

Up to three escape sequences can be emitted by the routine.

For *onoff* = `.True.` the sequence is "`Esc ] Esc & d 'x'`".

The character to be used for 'x' is passed in the right byte of *fieldattribute*. Its value should be chosen from the following:

	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
half bright									x	x	x	x	x	x	x	x
underline					x	x	x	x					x	x	x	x
inverse video			x	x			x	x			x	x			x	x
blinking		x		x		x		x		x		x		x		x
end enhancement	x															

If a blank is passed, the enhancement select escape sequence is suppressed so that only the "`Esc ]`" is emitted.

For *onoff* = `.False.` the sequence is "`Esc & d 'x' Esc '#' Esc [`".

The right byte of *fieldattribute* selects 'x', as described above with the addition of S to the possible enhancements for security fields to inhibit the display of any characters.

The left byte of *fieldattribute* controls the second escape sequence and should be chosen from the following set:

blank	supress the escape sequence
`'6'`	start an alphabetic only field
`'7'`	start a numeric only field
`'8'`	start an alphanumeric only field

Examples:

```
CALL HpZFieldDefine(.True.,2h D)  ! protected field, underlined
CALL HpZFieldDefine(.False.,2h B) ! unprotected field, inverse video
CALL HpZFieldDefine(.False.,2h7J) ! unprotected numeric only field,
                                  ! in half intensity inverse video
CALL HpZFieldDefine(.True.,2h @)  ! protected field, normal video
CALL HpZFieldDefine(.True.,2h )   ! protected field
```

The output of the routine is to the current position in the HpZ output buffer declared in a prior call to HpZDefOBuf. The maximum length of output is 8 characters.

HpZFmpWrite

This subroutine writes the current contents of the buffer defined by a prior call to HpZDefOBuf to the file specified. The current buffer pointer is then reset to the beginning of the buffer, thus "erasing" the buffer. The length returned by the function call is the same as for an FmpWrite call.

```
CALL HpZFmpWrite(dcb,error)
length = HpZFmpWrite(dcb,error)

integer*2 dcb(*),error,length,HpZFmpWrite
```

where:

dcb specifies an open file Disk Control Block (DCB) as expected by an FmpWrite call.

error returns the error code.

length returns the number of bytes written to the file or a negative error code.

Before the write is performed, if the buffer currently ends on an odd byte, a blank will be put into the right byte. This is because the file write occurs on word boundaries. The record length will NOT be altered, so the length word in the file will not include the added blank.

If the buffer length is known to be less than 32,768 bytes, the following construct allows easier error checking. HpZFmpWrite must be declared as logical*2 in this case:

```
IF (HpZFmpWrite(dcb,error)) THEN
  error_occurred
ENDIF
```

See also HpZWriteLu, HpZDefOBuf, HpZdecv, HpZMesss, and HpZWriteToString.

HpZGetNextChar HpZPeekNextChar

These logical*2 functions extract the next character from the input buffer defined by a prior call to HpZDefIBuf. The character is returned in the right byte with a blank pad in the left byte. HpZPeekNextChar differs from HpZGetNextChar in that it does not consume the character, that is, the character is still the next character in the input buffer.

```
IF (HpZGetNextChar(ichar)) THEN no_characters ENDIF
IF (HpZPeekNextChar(ichar)) THEN no_characters ENDIF

integer*2 ichar
```

where:

ichar is a variable that receives the character from the buffer.

It is possible for no characters to be remaining in the input buffer, either because it was empty to begin with or because all the characters have already been parsed. When this happens, this function returns a −1 (`.true.` in both FORTRAN and Pascal) and *ichar* is set to null.

See also HpZGetNextStrDsc, HpZGetNextToken, and HpZBackSpaceIbuf.

HpZGetNextStrDsc

This integer*2 function builds a string descriptor for the next token in the input buffer defined by a prior call to HpZDefIBuf. To find the token, the function skips leading blanks and tabs. The end of the token is the first occurrence of a blank, a tab, a comma, or the end of the input buffer.

```
length = HpZGetNextStrDsc(stringdescriptor)

integer*2 length
integer*4 stringdescriptor
```

where:

length returns the number of characters in the token.

stringdescriptor returns the manufactured string descriptor.

The action of this routine is similar to that the StrDsc call and the same limitations apply. Refer to "StrDsc" in this manual.

It is possible for no characters to be remaining in the input buffer, either because it was empty to begin with or because all the tokens have already been parsed. When this happens, this function builds a zero-length string descriptor and returns a value of zero. Because the FORTRAN definition of a character string does not allow zero-length strings, it is the programmer's responsibility to check the function return to trap such cases. If this is not done, the FORTRAN string handling routines will get a runtime error when they encounter the zero-length strings.

See also HpZGetNextToken, GetString, StrDsc, MinStrDsc, HpZParse, HpZDParse, and HpZGetRemStrDsc.

HpZGetNextToken

This integer*2 function copies the next token in the input buffer defined by a prior call to HpZDefIBuf to the output string. Tokens are delimited by either commas, blanks, tabs, or the end of the input buffer. Leading blanks and tabs are skipped.

```
length = HpZGetNextToken(string)

integer*2 length
character*(*) string
```

where:

length returns the number of characters in the token.

string is the variable in which the token is returned.

It is possible for no characters to be remaining in the input buffer, either because it was empty to begin with or because all the tokens have already been parsed. When this happens, this function builds an empty string and returns a value of zero. It is the programmer's responsibility to check the function return to trap such cases. If this is not done, FORTRAN runtime errors will result.

See also HpZGetNextStrDsc, HpZParse, and HpZDParse.

HpZGetNumD2 HpZGetNumO2 HpZGetNumB2 HpZGetNumD4 HpZGetNumO4 HpZGetNumB4

These logical*2 functions extract a signed or unsigned number from the input buffer defined by a prior call to HpZDefIBuf. The number is converted to either a *2 or *4 integer as per the ending routine number. The type of conversion is defined by the D (decimal), O (octal), or B (decimal unless there is a B suffix, then octal) in the function name.

```
IF (HpZGetNumD2(number)) THEN no_number ENDIF
IF (HpZGetNumO2(number)) THEN no_number ENDIF
IF (HpZGetNumB2(number)) THEN no_number ENDIF
IF (HpZGetNumD4(number)) THEN no_number ENDIF
IF (HpZGetNumO4(number)) THEN no_number ENDIF
IF (HpZGetNumB4(number)) THEN no_number ENDIF
```

where:

number returns the converted number.
Declare as integer*2 when using HpZGetNumD2, HpZGetNumO2, or HpZGetNumB2.
Declare as integer*4 when using HpZGetNumD4, HpZGetNumO4, or HpZGetNumB4.

The routine returns true if a number is not found. In this case the next character available to Ibuf routines will be the first nonblank character found in the conversion attempt. Leading blanks are allowed in the number as well as a leading + or −. After the first nonblank character is found, any non-numeric character is taken as the terminator. No blanks may be imbedded in the number or between the sign and the number.

See also HpZParse, HpZGetNextToken, and HpZGetNextStrDsc.

HpZGetNumStrDsc

This function returns a string descriptor for a signed or unsigned number, beginning at the current position in the input buffer and ending just before first non-numeric after possible leading +/− sign. Leading blanks are skipped. Blanks after the sign or first digit terminate the number. If no number is present, a zero length is returned and the Ibuf pointer will point at the first nonblank beyond where it started.

length = HpZGetNumStrDsc(*StrDsc*)

```
integer*2 length,HpZGetNumStrDsc
integer*4 StrDsc
```

or

StrDsc = HpZGetNumStrDsc(*StrDsc*)

```
integer*4 StrDsc,HpZGetNumStrDsc
```

where:

length is the length of the string when HpZGetNumStrDsc is declared as integer*2. *length* is also returned in the A-Register.

StrDsc returns the string descriptor when HpZGetNumStrDsc is declared as integer*4. *StrDsc* is also returned in the A- and B-Registers.

See also HpZGetNumD2, HpZGetNumD4, HpZGetNumO2, HpZGetNumO4, etc.

HpZGetNumX

This routine converts digits to internal representation. It assumes that the actual conversion can be done by a function (such as DecimalToInt) that accepts a character string as the first parameter and returns the result in A or A and B; and has a second parameter that is an error flag indicating an error if not equal to zero. The conversion function is passed in along with an indicator of the size of the result. It can handle any conversion of pure digits with a possible sign and either integer*2 or integer*4 result.

```
EXTERNAL conversion                    (declare conversion function)
    :
IF (HpZGetNumX(number,conversion,size) THEN

logical HpZGetNumX
```

where:

conversion is the external conversion function.

number returns the conversion result. Declare as integer*2 if *size* = −1, otherwise declare as integer*4.

size indicates the size of the result. If *size* = −1, *number* is integer*2, otherwise *number* is integer*4.

See also HpZGetNumD2, HpZGetNumO2, HpZGetNumB2, HpZGetNumD4, HpZGetNumO4, and HpZGetNumB4.

HpZGetRemStrDsc

Returns a string descriptor to the portion of the HpZ input buffer that has not yet been consumed by the other HpZ calls such as HpZGetNextChar or HpZParse.

```
length  = HpZGetRemStrDsc(StrDsc)

integer*2 length,HpZGetRemStrDsc
integer*4 StrDsc
```

or

```
StrDsc  = HpZGetRemStrDsc(StrDsc)

integer*4 StrDsc,HpZGetRemStrDsc
```

where:

length is the length of the string when HpZGetRemStrDsc is declared as integer*2. *length* is also returned in the A-Register.

StrDsc returns the string descriptor when HpZGetRemStrDsc is declared as integer*4. *StrDsc* is also returned in the A- and B-Registers.

It is possible for no characters to be remaining in the input buffer, either because it was empty to begin with or because all the tokens have already been parsed. When this happens, this function builds a zero-length string descriptor and returns a value of zero. Because the FORTRAN definition of a character string does not allow zero-length strings, it is the programmer's responsibility to check the function return to trap such cases. If this is not done, the FORTRAN string handling routines will get a runtime error when they encounter the zero-length strings.

See also HpZIBufUseStrDsc and HpZGetNextToken.

HpZHexc

This routine converts a number to hexadecimal. The output length will be 1 to 3 bytes, as specified in the call.

```
CALL HpZHexc(ivalue,numdig)

integer*2 ivalue,numdig
```

where:

ivalue is the number to be converted to hexadecimal and is stored in the output buffer defined by a prior HpZDefOBuf call.

numdig is the number of digits to produce; must be 1, 2, or 3.

See also HpZHexo.

HpZHexi

This function parses hexadecimal ASCII integers from the HpZ input buffer defined by a prior call to HpZDefIBuf. The conversion terminates when the buffer is exhausted, the *max* number of characters are converted, or a character other than 0-9, A-F, or a-f is found. The current position in the buffer is then updated. This routine ignores leading blanks, and does not count them in the *max* limitation.

```
IF (HpZHexi(number[,max])) THEN
  no_number_found
ENDIF

integer*2 number,max
logical*2 HpZHexi
```

where:

number is the number found in the input buffer.

max is the maximum number of characters to convert. *max* can be in the range of 1..4, with the default value equal to 4.

Example:

Assume the input buffer '`Ibuf`' contains '`Blarg, A0FE, F2BAAD`'.

```
CALL HpZdefIbuf(Ibuf,19,0)       ! declare input buffer

Temp = HpZGetNextToken(DString) ! parse 'Blarg' and consume
                                ! trailing commas

if ( HpZHexi(num,4) ) then       ! parse up to 4 hex characters
 :
 :                               ! code to handle parse failure
endif
 :                               ! in this example, the parse succeeds
 :                               ! with 'num' = 0xA0FE (41214 decimal);
 :                               ! the leading blank is skipped, but
 :                               ! the trailing comma is NOT consumed

Temp = HpZGetNextToken(DString) ! consume the trailing comma
                                ! Temp will be 0

if ( HpZHexi(num,2) ) then       ! parse only 2 hex characters this
 :                              ! time
endif
 :                               ! 'num' will be 0xF2 (242 decimal);
 :                               ! the leading blank is skipped
if ( HpZHexi(num,4) ) then       ! parse remaining 4 characters
 :
endif
 :                               ! 'num' will be 0xBAAD (47789 decimal)
```

See also HpZParse, HpZGetNextToken, and HpZGetNextChar.

HpZHexo

This subroutine stores the ASCII hexadecimal representation of a number into the current position in the output buffer that was defined by a prior call to HpZDefOBuf. The output is 4 characters wide with leading zeros.

```
CALL HpZHexo(ival)

integer*2 ival
```

Examples: `'00A5', 'FFFF', '8000', '0000'`

See also HpZHexc and HpZDeco.

HpZIBufRemain

This integer*2 function returns the number of bytes remaining from the current position to the end of the input buffer defined by a prior call to HpZDefIBuf.

```
count = HpZIBufRemain()

integer*2 count,HpZIBufRemain
```

See also HpZIBufUsed.

HpZIBufReset

This subroutine resets the current input position to the start of the input buffer defined by a prior call to HpZDefIBuf. Optionally, the position can be set to a value passed in by the caller.

```
CALL HpZIBufReset([position])

integer*2 position
```

where:

position is an optional parameter to specify the new pointer position, where 0 equals the start of the buffer.

HpZIBufUsed

This integer*2 function returns the current byte offset in the input buffer defined by a prior call to HpZDefIBuf. It can be used to see how much of the HpZ input buffer has been parsed.

```
count = HpZIBufUsed()

integer*2 count,HpZIBufUsed
```

See also HpZIBufRemain.

HpZIBufUseStrDsc

Returns a string descriptor for the portion of the input buffer (as defined by a call to HpZDefIbuf) that has already been passed over. This is useful if it is desired to output the buffer to the current position as part of an error message.

```
length = HpZIBufUseStrDsc(StrDsc)

integer*2 length,HpZIBufUseStrDsc
integer*4 StrDsc
```

or

```
StrDsc = HpZIBufUseStrDsc(StrDsc)

integer*4 StrDsc,HpZIBufUseStrDsc
```

where:

length is the length of the string when HpZIBufUseStrDsc is declared as integer*2. *length* is also returned in the A-Register.

StrDsc returns the string descriptor when HpZIBufUseStrDsc is declared as integer*4. *StrDsc* is also returned in the A- and B-Registers.

It is possible for no characters to be remaining in the input buffer, either because it was empty to begin with or because all the tokens have already been parsed. When this happens, this function builds a zero-length string descriptor and returns a value of zero. Because the FORTRAN definition of a character string does not allow zero-length strings, it is the programmer's responsibility to check the function return to trap such cases. If this is not done, the FORTRAN string handling routines will get a runtime error when they encounter the zero-length strings.

See also HpZGetRemStrDsc.

HpZInsertAtFront

This routine inserts the given data in front of the data that is currently in the buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZInsertAtFront(buffer,numchars)

integer*2 buffer(*),numchars
```

where:

buffer is the data to be copied into the buffer previously defined by HpZDefOBuf.

numchars is the number of characters to copy.

For example, suppose that the current contents of the output buffer is "`abcdef`".

```
call HpZInsertAtFront(2h* ,2)   ! stuff '* ' into output
```

The contents will now be "`* abcdef`" and the output buffer pointers will be updated accordingly.

HpZmbt

This routine copies the indicated bytes from a buffer to the current position in the output buffer defined by a prior call to HpZDefOBuf. *offset* = 0 describes the first byte in the buffer.

```
CALL HpZmbt(buffer,offset,bytes)

integer*2 buffer(*),offset,bytes
```

where:

buffer is a packed array containing the bytes to be copied.

offset is the offset into the buffer (previously defined by HpZDefOBuf) for the first byte to be copied.

bytes is the number of bytes to copy.

See also HpZmvc, HpZmvs, and HpZsbt.

HpZMesss

This function delivers a command to the operating system's operator interface section. Any system command can be given. If the system returns a reply message, it does so into the HpZ buffer and this routine sets the pointers accordingly. If not, the pointers will be reset just as if HpZWriteLu had been called. Note that HpZDefOBuf must have been called with a buffer at least 72 bytes long.

length = HpZMesss([*lu*])

integer*2 *length*,*lu*,HpZMesss

where:

length is the number of bytes in the message returned from the operating system, if any.

lu is an optional LU number used to form the session number if the command is an "RU" or "XQ".

Example program:

```
ftn7x,q,s
      program hpzmessstest
      implicit none

      integer*2 ! variables
     > obuf(0:39),
     > temp

     integer*2 ! functions
     > HpZMesss

_____________________________________________________

      call HpZDefOBuf(Obuf)
      call HpZmvs('TM ')
      Temp = HpZmesss()
      if ( Temp.ne.0 ) then
        call HpZmvs(' <=== time from HpZMesss call')
        call HpZWriteLu(1)
      endif
      end
```

See also HpZInsertAtFront, HpZWriteLu, and Messs.

HpZMoveString

This routine copies strings without FORTRAN limitations. This routine does the same thing as the FORTRAN statement "*to* = *from*"; where *to* and *from* are strings.

```
CALL HpZMoveString(from,to)

character*(*) from,to
```

where:

from is the "from" string descriptor.

to is the "to" string descriptor.

FORTRAN does not allow the "*to* = *from*" construct if the strings are manufactured string descriptors, such as the output from StrDsc, which are really type Integer*4. This routine also handles zero-length strings. The destination string is blank padded in the usual manner.

This routine does not do overlap checking. It moves from low to high, so that you can ripple fill.

Note that this routine does NOT use the HpZ mini-formatter input or output buffers.

See also ConCat and GetString.

HpZmvc

This routine copies characters from an integer buffer to the current position in the output array defined by a prior call to HpZDefOBuf.

```
CALL HpZmvc(buffer,numchar)

integer*2 buffer(*),numchar
```

where:

buffer is the buffer to be copied.

numchar is the number of characters in *buffer*.

For an input buffer of *n* bytes, the output length will be *n* bytes.

See also HpZmvs, HpZsbt, HpZmbt, and HpZInsertAtFront.

HpZmvs

This routine copies a string to the current position in the output array defined by a prior call to HpZDefOBuf.

```
CALL HpZmvs('string to move')
```

or

```
CALL HpZmvs(string_variable)
```

where:

`string to move`	is a literal data string to move.
string_variable	is a variable declared as a character string that contains the string to move.

For an input string of *n* bytes, the output length will be *n* bytes.

Note that it is less efficient, both in time and space, to use string concatenation before calling HpZmvs.

For example,

```
Str1 = 'String number 1'
Str2 = 'String number 2'

call HpZmvs(Str1)
call HpZmvs(Str2)
```

is preferable to

```
call HpZmvs(Str1//Str2)
```

See also HpZmvs_Escape, HpZmvs_Control, HpZmvc, and HpZsbt.

HpZmvs_Control

This call moves the string passed by the user to the current position in the output buffer defined by a prior call to HpZDefOBuf. As the string is copied, the underscore character (octal 137) is interpreted to mean that the next character should be altered by subtracting 100 octal (modulo 200 octal). The uppercase portion of the ASCII character set (octal 100..137) will be downshifted to the control portion (octal 0..37), and "delete" (octal 177) will be produced by `'_?'`.

This call is very useful to generate control characters that cannot be manipulated by the editor or language compilers.

```
    CALL HpZmvs_Control(charvariable)
```
or
```
    CALL HpZmvs_Control('_M_[&dB Inverse _[&d@ Non-inverse _L')
```

For an input string of *n* bytes, the output length will be in the range of *n*/2 to *n*.

```
_@ ==> Nul    _A ==> Soh    _B ==> Stx    _C ==> Etx
_D ==> Eot    _E ==> Enq    _F ==> Ack    _G ==> Bel
_H ==> Bs     _I ==> Tab    _J ==> Lf     _K ==> Vt
_L ==> Ff     _M ==> Cr     _N ==> So     _O ==> Si
_P ==> Dle    _Q ==> Dc1    _R ==> Dc2    _S ==> Dc3
_T ==> Dc4    _U ==> Nak    _V ==> Syn    _W ==> Etb
_X ==> Can    _Y ==> Em     _Z ==> Sub    _[ ==> Esc
_\ ==> Fs     _] ==> Gs     _^ ==> Rs     __ ==> Us
_? ==> Del
```

See also HpZmvs, HpZmvs_Escape, and HpZDefOBuf.

HpZmvs_Escape

This call moves the string passed by the user to the current position in the output buffer defined by a prior call to HpZDefOBuf. As the string is copied, the underscore character (octal 137) is replaced by an escape character (octal 33). This call is useful for generating the escape sequences needed to control HP terminals that are difficult to manipulate with the editor. Note that the character following an underscore is not examined, therefore "__" is translated to escape underscore.

```
CALL HpZmvs_Escape(charvariable)
```

or

```
CALL HpZmvs_Escape('_&dB Inverse _&d@ Non-inverse')
```

An example string to perform:

memory lock off
home up
clear screen
turn on inverse video
address the cursor to line 24, column 12

is:

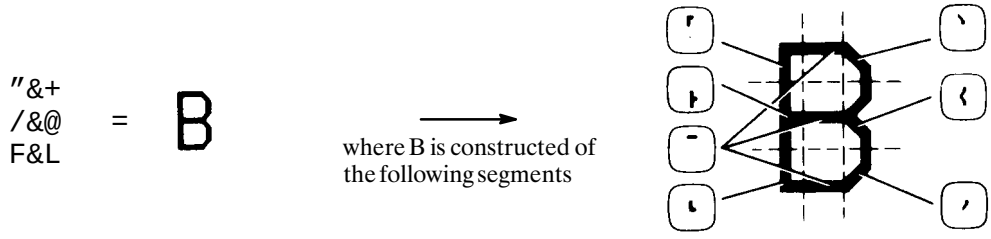
```
CALL HpZmvs_Escape('_m_H_J_&dB_a24r12C')
```

For an input string of *n* bytes, the output length is *n* bytes.

See also HpZmvs, HpZmvs_Control, and HpZDefOBuf.

HpZmvs_Large

This subroutine creates large characters in a 3-by-3 character cell using line segments (nine character segments) in the HP 264x alternate character set. An example of constructing the letter "B" using the large character set is as follows:



If your terminal does not reproduce the above characters as a large letter "B", it is not capable of producing the large character set. This capability is standard on some terminals, and optional or unavailable on others.

Example program segment:

```
call HpZDefOBuf(Obuf)       ! make it BIG, each char needs 26 bytes
call HpZStartLarge          ! must call at start of each "line" of
                            ! large letters (emits 18 bytes)
call HpZmvs_Large('String') ! emit large characters
call HpZEndLarge            ! must call to terminate line
                            ! (emits 5 bytes)
call HpZWriteLu(Crt)        ! flush the buffer
```

HpZNlsMvs

This routine moves an NLS string to the HpZ output buffer. Prior to this call, initialization calls to HpZNlsSubset and HpZDefOBuf must have been issued.

```
CALL HpZNlsMvs(stringnumber)

integer*2 stringnumber
```

where:

stringnumber is the index number of the string to be moved.

When used with HpZNlsSubset, the index number of the string must be in the range 0 to *n*, where *n* is the last message number in the catalog.

See also HpZNlsSubset and HpZDefOBuf.

HpZNlsSubset

This routine sets up the linkage from NLS to HpZ routines. It allows access to the binary relocatable module that is the output of the GENCAT program.

```
CALL HpZNlsSubset(modulename)
```

This routine implements only a restricted subset of the NLS features in that it assumes that the messages in the catalog are numbered from 0 to *n* with no skipped values.

See also HpCrtNlsMenu and HpZNlsMvs.

HpZOBufReset

This subroutine resets the current position to the start of the output buffer defined by a prior call to HpZDefOBuf. Optionally, the position can be set to the value passed in by the caller.

```
CALL HpZOBufReset([position])

integer*2 position
```

where:

position is an optional parameter that specifies the new pointer position, where 0 equals the start of the buffer.

HpZOBufUsed

This integer*2 function returns the current byte offset in the output buffer defined by a prior call to HpZDefOBuf.

```
count = HpZOBufUsed()

integer*2 count,HpZOBufUsed
```

HpZOBufUseStrDsc

This integer*2 function returns the current byte offset in the output buffer defined by a prior call to HpZDefOBuf.

```
length = HpZOBufUseStrDsc(StrDsc)

integer*2 length,HpZOBufUseStrDsc
integer*4 StrDsc
```

or

```
StrDsc = HpZOBufUseStrDsc(StrDsc)

integer*4 StrDsc,HpZOBufUseStrDsc
```

where:

length is the length of the string when HpZOBufUseStrDsc is declared as integer*2. *length* is also returned in the A-Register.

StrDsc is an optional variable that returns the manufactured string descriptor for the filled portion of the HpZ output buffer when HpZOBufUseStrDsc is declared as integer*4. *StrDsc* is also returned in the A- and B-Registers.

It is possible for no characters to be remaining in the input buffer, either because it was empty to begin with or because all the tokens have already been parsed. When this happens, this function builds a zero-length string descriptor and returns a value of zero. Because the FORTRAN definition of a character string does not allow zero-length strings, it is the programmer's responsibility to check the function return to trap such cases. If this is not done, the FORTRAN string handling routines will get a runtime error when they encounter the zero-length strings.

HpZOctc

This routine converts the passed value to its octal ASCII representation at the current position in the output buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZOctc(ivalue,numdigits)

integer*2 ivalue,numdigits
```

where:

ivalue is the number to be converted to octal and is stored in the output buffer defined by a prior HpZDefOBuf call.

numdigits is the number of digits to produce, must be in the range 1..5.

The output length will be 1 to 5 bytes, as specified in the call.

See also HpZOctv and HpZOcto.

HpZOctd

This routine converts the given double integer value to its octal representation. The result will be stored at the current position in the output buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZOctd(value)

integer*4 value
```

where:

value is the number to be converted to octal.

The output will be eleven characters wide with leading zeros as needed.

Examples:

```
             0 → 00000000000
            -1 → 37777777777
           255 → 00000000377
         65535 → 00000177777
       8640000 → 00040753000
    2147483647 → 17777777777
```

HpZOcto

This subroutine converts the passed value to its octal ASCII representation at the current position in the output buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZOcto(value)

integer*2 value
```

where:

value is the number to be converted to octal ASCII representation.

The output length will be 6 bytes.

See also HpZOctv and HpZOctc.

HpZOctv

This routine converts the passed value to its octal ASCII representation at the current position in the output buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZOctv(value)

integer*2 value
```

where:

value is the number to be converted to octal suppressing leading zeros. The result will be stored at the current position in the output buffer defined by a previous call to HpZDefOBuf.

The output length will be from 1 to 6 bytes.

See also HpZOctc and HpZOcto.

HpZPadToCount

This subroutine puts the specified number of blanks at the current position in the output buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZPadToCount(numblanks)

integer*2 numblanks
```

where:

numblanks is the variable or constant indicating the number of blanks to output.

The output length will be the number of bytes specified in the call.

See also HpZPadToPosition.

HpZPadToPosition

This subroutine puts blanks into the output buffer defined by a prior call to HpZDefOBuf until the desired position is reached. The first character position in the buffer is position zero. Note that HP terminals also number rows and columns starting from 0. If the current position in the buffer is greater than or equal to the desired position, no blanks are emitted.

```
CALL HpZPadToPosition(position)

integer*2 position
```

where:

position is the variable or constant indicating the desired position in the buffer.

The output length will be in the range from zero to *position* −1 bytes.

For example:

```
call HpZsbt(2h *)
call HpZPadToPosition(6)
call HpZsbt(2h *)
```

would put

```
'*     *'
 0123456
```

in the buffer.

See also HpZPadToCount.

HpZParse

This is a parse routine for 16-character parameters.

```
IF (HpZParse(result)) THEN
  parse succeeded
ENDIF

integer*2 result(15)
logical*2 HpZParse
```

where:

result returns the result of parsing the contents of the buffer defined by a prior call to HpZDefIBuf. The parse starts at the current position in the buffer, extracts a parameter (possibly with subparameters) then updates the current position in the buffer. Multiple calls to HpZParse can be made to parse successive parameters.

Parsing rules:

- Parameters (a namr) are separated by commas (`','`) or spaces (`' '`).
- Subparameters (name, scode, etc) are separated by colons (`':'`) or equal signs (`'='`).
- Leading negative signs of numeric arguments are significant, but are just another character for alphabetic arguments.
- Octal numeric constants can be expressed with a B suffix.
- Anything that has a non-numeric character in it is assumed to be alpha, except for the B suffix for octal.

Contents of *result* after a call:

word 1 – a flag word, see below for explanation
word 2 – a numeric value if the first subparameter is numeric, or the first 2 ASCII characters of a name if alphabetic
word 3 – 0 or characters 3 & 4 of a name
word 4 – 0 or characters 5 & 6
word 5 – 0 or characters 7 & 8
word 6 – 0 or characters 9 & 10
word 7 – 0 or characters 11 & 12
word 8 – 0 or characters 13 & 14
word 9 – 0 or characters 15 & 16
word 10 – the second subparameter (security code)
word 11 – the third subparameter (cartridge reference number)
word 12 – the fourth subparameter (file type)
word 13 – the fifth subparameter (file size in blocks)
word 14 – the sixth subparameter (record length)
word 15 – the seventh subparameter (DS node number)

Words 2 through 9 default to blanks for alphabetic characters or null for numeric.

Example:

```
&SOURCE_FILE_NAME:1:40:2:200:1000,6  ::22 File=Fred:AT:AT T7::22
```

will be parsed into the following arrays in 5 successive calls:

	First	Second	Third	Fourth	Fifth	
						← call
1	02527B	000001B	000010B	000377B	000603B	← flag word
2	&S	6	0	Fi	T7	name or numeric value
3	OU	0	0	le	2 blanks	
4	RC	0	0	2 blanks	2 blanks	
5	E_	0	0	2 blanks	2 blanks	
6	FI	0	0	2 blanks	2 blanks	
7	LE	0	0	2 blanks	2 blanks	
8	_N	0	0	2 blanks	2 blanks	
9	AM	0	0	2 blanks	2 blanks	
10	00001	0	0	Fr	0	scode
11	00040	0	00022	AT	00022	crn
12	00002	0	0	AT	0	type
13	00200	0	0	0	0	blks
14	01000	0	0	0	0	reclen
15	0	0	0	0	0	node#

The flag word contains a bit pattern to describe the type of each of the subparameters, with bits 0 and 1 for the name, 2 and 3 for the scode, and so on, where:

```
00 = null, no parameter supplied (this is not the same as a zero)
01 = numeric, a decimal, or octal constant (zero is numeric, :0:)
10 = not used
11 = an ASCII value, stored 2 bytes per word, left justified, blank filled
```

See also NAMR, PARSE, INAMR, HpZReScan, and HpZDParse.

HpZPlural

This routine conditionally makes a string plural depending upon a count. This routine emits an "s" to the HpZ output buffer if the given count is not equal to 1. Thus it follows the English language rules about forming most plurals.

```
CALL HpZPlural(count)

integer*2 count
```

where:

count indicates whether or not to emit an "s". If *count* = 1, an "s" is NOT emitted, otherwise an "s" is emitted.

Example:

```
                                   ErrorCount = 0          ErrorCount = 1
call HpZdecv(ErrorCount)      →    '0'                     '1'
call HpZmvs(' Error')         →    '0 error'               '1 error'
call HpZplural(ErrorCount)    →    '0 errors'              '1 error'
call HpZmvs(' detected')      →    '0 errors detected'     '1 error detected
```

HpZPrintPort

This routine displays the port status (using a special status read) of *port_lu* to *display_lu*. If *port_lu* is equal to 1, this routine converts it to the true LU number. It verifies that the LU is not locked, not down, and not busy. If it is, a message is output to the calling LU. Before calling this routine, you must call HpZDefOBuf with a buffer of at least 80 words (160 characters).

```
CALL HpZPrintPort(port_lu,display_lu,waitflag)

integer*2 port_lu,display_lu,waitflag
```

where:

port_lu is the LU number of the port to display (in the range 1..255) in the format of the first word of the XLUEX control word.

display_lu is the LU number of the display LU (in the range 1..255) in the format of the first word of the XLUEX control word. If it is not in the valid range, it will default to the call LU.

waitflag specifies whether to get real or "fake" status in the event that *port_lu* is busy. If *waitflag* is 0, the true status is displayed. If *waitflag* is nonzero, a call to FakeSpStatus is made. See the FakeSpStatus section earlier in this chapter.

This subroutine is designed to be a simple addition to a program such as:

```
Ftn7x,l
      program HpCrt
      integer params(5),obuf(0:79)
      call rmpar(params)
      call HpZDefOBuf(Obuf)
      call HpZPrintPort(params(1),params(2),params(3))
      end
```

The resulting display is similar to the following:

```
Status for LU 32:
Device Driver:    DDC01 Rev. 5.00    Driver type = 05
Interface Driver: ID800 Rev. 5.00
  Firmware:             Rev.  4.10
CN20: Primary Program:    PROMT (Enabled)
CN40: Secondary Program:        (Disabled)
CN17: 000000B No user defined terminator
CN22:  32767  Timeout = 327.39 seconds
CN30: 010132B Frame=8/1 No par. BRG1 9600 baud Port 2
CN31: 000000B
CN33: 000000B
CN34: 000002B ENQ/ACK Protocol
DV20: 000077B Character mode
 DVT Address: 44227B;  IFT Address: 47203B
```

HpZPushObuf and HpZPopObuf

Declare a new output buffer for the HpZ mini-formatter routines, pushing the old one.

```
call HpZPushObuf(NewObuf)

integer*2 NewObuf(*)
```

where:

NewObuf is a buffer to receive the characters from the various formatting routines

The *NewObuf* parameter should be in the main, in common, or in SAVE address space so that it is always available. Also, it should be large enough to accomodate the expected output, as the HpZ output routines do not perform limit checks.

This call is used to declare a new output buffer for the HpZ mini-formatter routines. It pushes the declaration of the output buffer currently in use, if any, into the first two words of the buffer provided in the call. The call is legal even if it is not preceeded by a call to HpZDefObuf, so it is useful for system routines in which you are not sure if the user is calling the HpZ library or not.

The HpZPopObuf routine is the inverse of HpZPushObuf.

HpZQandA

This subroutine allows the user to ask a question and read a reply. The question is written in transparency mode to keep the cursor on the prompt line. The reply is read with REIO, with echo enabled.

```
CALL HpZQandA(lu,prompt,ibuf,ilength,status,length)

integer*2 lu,ibuf(*),ilength,status,length
character*(*) prompt
```

where:

lu	is the LU number of the user's CRT.
prompt	is the prompt string to be issued to the user. The cursor remains on the same line.
ibuf	returns the reply buffer.
ilength	is the size of the input buffer (positive number of words or negative number of characters).
status	is the status word returned from the driver after the read is completed. Refer to the driver reference manual for the driver you are using for a definition of the bits in the status word.
length	returns the number of bytes in the user's reply.

In most cases, this routine should be followed by a call to upshift the reply.

Note that this routine does NOT use the HpZ mini-formatter input or output buffers.

See also HpZYesOrNo, HpCrtSendChar, and HpCrtGetChar.

HpZReScan

This subroutine resets the internal pointers used by the HpZ parse routines to the start of the second subparameter preceeding the current position in the buffer defined by a prior call to HpZDefIbuf. This is useful so that constructs such as `'File=Data::20'` can be reparsed after the `'File='` is recognized to extract the `'Data::20'` portion of the string.

```
CALL HpZReScan()
```

See also HpZParse and HpZIbufReset.

HpZRomanNumeral

This routine places the Roman numeral equivalent for the given value at the current position in the output buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZRomanNumeral(value,case)

integer*2 value,case
```

where:

value is the value to be output in Roman numerals.

case is a flag to indicate uppercase or lowercase output. If *case* is 0, output will be uppercase; if *case* is nonzero, output will be lowercase.

Note The Roman numeral system does not provide for a zero or for negative numbers. This implementation is limited to a maximum input value of 3999. If a number outside the range 1 to 3999 is passed to this routine, it will pass it on to HpZdecv for display as a decimal number.

The minimum output emitted is 1 character and the maximum is 15 characters for the value `3888 (MMMDCCCLXXXVIII)`.

The values used are:

```
I  1
V  5
X  10
L  50
C  100
D  500
M  1000
```

A lower value number to the left subtracts, (for example, ix = 9). A number to the right adds (for example, xii = 12).

HpZsbt

This routine stores the lower byte of the passed value into the current position defined by a prior call to HpZDefOBuf.

```
CALL HpZsbt(value)

integer*2 value
```

where:

value is the variable or constant whose lower byte is to be stored.

The output length is one byte.

Example:

```
CALL HpZsbt(7)        ! Put an ASCII 'BELL' in the output buffer.
```

See also HpZmvc.

HpZStripBlanks

This routine adjusts the internal pointer to the output buffer for the HpZ routines to "erase" trailing blanks before calling HpZWriteLu, HpZFmpWrite, or any of the other routines that flush the output buffer. See HpZ.

```
CALL HpZStripBlanks
```

HpZUdeco

This subroutine stores the unsigned decimal representation of a number at the current position in the output buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZUdeco(ivalue)

integer*2 ivalue
```

where:

ivalue is the number to be stored.

The output will be 5 characters wide with leading zeros as required.

Examples: `'65536', '32768', '00005', '00000'`

See also HpZUdecv and HpZDecc.

HpZUdecv

This subroutine converts a number into unsigned decimal representation, suppressing leading zeros. The conversion is done to the buffer defined by a prior call to HpZDefOBuf.

```
CALL HpZUdecv(ivalue)

integer*2 ivalue
```

The output length will be in the range of 1 to 6 bytes.

See also HpZDeco.

HpZWriteExec14

Perform an EXEC 14 call from the HpZ mini-formatter buffer.

```
call HpZWriteExec14
```

This routine writes the current contents of the HpZ output buffer to the parent process via an EXEC 14 call. Just as in an HpZWriteLu call, the buffer pointers are reset to clear the buffer.

HpZWriteLU

This subroutine writes the current contents of the output buffer defined by a prior call to HpZDefOBuf to the LU specified. The current buffer pointer is then reset to the beginning of the buffer. The write is done with an EXEC call so the LU must be in the range 0 to 63.

```
CALL HpZWriteLu(lu)

integer*2 lu
```

See also HpZWriteXLU, HpZFmpWrite, HpZDefOBuf, HpZDecv, HpZMesss, and HpZWriteToString.

HpZWriteXLU

This subroutine writes the current contents of the output buffer defined by a prior call to HpZDefOBuf to the LU specified. The current buffer pointer is then reset to the beginning of the buffer. The write is done with an XLUEX call so the LU can be greater than 63. The LU parameter should be a double word in the form expected by XLUEX.

```
CALL HpZWriteXLu(lu)

integer*2 lu
```

See also HpZFmpWrite, HpZDefOBuf, and HpZDecv.

HpZWriteToString

This routine copies the contents of the HpZ output buffer to a string. The output string will be padded with blanks, as necessary.

```
length = HpZWriteToString(string)

integer*2 length
character*(*) string
```

where:

length is the number of bytes moved to *string*; it will be the minimum of the string length and the occupied length of HpZ output buffer.

string is the string that receives the current contents of the HpZ output buffer.

It is possible for no characters to be remaining in the input buffer, either because it was empty to begin with or because all the tokens have already been parsed. When this happens, this function builds a zero-length string descriptor and returns a value of zero. Because the FORTRAN definition of a character string does not allow zero-length strings, it is the programmer's responsibility to check the function return to trap such cases. If this is not done, the FORTRAN string handling routines will get a runtime error when they encounter the zero-length strings.

See also HpZWriteLU, HpZFmpWrite, HpZOBufStrDsc, and HpZMesss.

HpZYesOrNo

This routine is used to ask questions which are to be answered with a yes or no response only. The cursor remains on the prompt line after the question. The answer is read with XREIO, with echo enabled. The user can answer the prompt with any of [y ye Ye YE yE n no No NO nO], any other string that starts with those characters, or a carriage return by itself.

```
IF (HpZYesOrNo(crt,prompt,ians)) THEN answer_was_yes ENDIF

integer*2 crt,ians
character*(*) prompt
logical*2 HpZYesOrNo

ians = 2hNO                    (set default answer to NO; must be all uppercase)
```

where:

crt is the LU number of the user's terminal (in the range 1..255).

prompt is the prompt string for the user.

ians returns the user's answer to the question, if entered (that is, the user did not hit return). By setting *ians* before the call, you can control the action that occurs when the user types only a carriage return.

initial value for *ians*	resulting action if user hits carriage return only
`2hYE`	function returns `.true.`
`2hNO`	function returns `.false.`
anything else	the user is forced to type a yes or no response; function will be `.true.` or `.false.` accordingly

In all cases, *ians* contains either 2hYE or 2hNO after the call completes, even if the user typed a lowercase answer. The question is repeated if the read times out.

Caution Do not pass a constant for the *ians* parameter because it is altered by the HpZYesOrNo call. For example, do not do the following:

```
if (HpZYesOrNo(Crt,'prompt',2hYE) then
```

To make it obvious to the user what the defaults are, the prompt string can contain escape sequences to display the default answer in inverse video. The cursor can then be positioned back on the default so it will be overwritten if the user does not want the default value. For example:

```
ians = 2hYE
if (HpZYesOrNo(crt,'OK? ^[&dBY ^[&d@^[D^[D',ians) ...
```

The above prompt string in this case is "O K ? Esc & d B Y Space Esc & d @" to display the Y in inverse video, followed by "Esc D Esc D" to put the cursor under the Y.

Note that this routine does NOT use the HpZ mini-formatter input or output buffers.

See also HpZQandA and HpCrtSendChar.

MinStrDsc

This routine builds a string descriptor that describes a trimmed substring of the string that is passed to it. Leading and trailing blanks are not included in the output string descriptor.

fakestring = MinStrDsc(*string*)

integer*4 *fakestring*,MinStrDsc
character*(*) *string*

where:

fakestring returns the manufactured string descriptor.

string is the input string.

It is possible for no characters to be remaining in the input buffer, either because it was empty to begin with or because all the tokens have already been parsed. When this happens, this function builds a zero-length string descriptor and returns a value of zero. Because the FORTRAN definition of a character string does not allow zero-length strings, it is the programmer's responsibility to check the function return to trap such cases. If this is not done, the FORTRAN string handling routines will get a runtime error when they encounter the zero-length strings.

See also StrDsc.

PutBitMap

This subroutine copies the LSB of the indicated word to a bit map (a packed array of bits). When accessing bits above 32 Kbytes, you must use negative numbers, as there is no unsigned integer data type in FORTRAN on the HP 1000.

CALL PutBitMap(*newbit*,*bitmap*,*index*)

integer*2 *newbit*,*array*(*),*bitmap*

where:

newbit is the word whose LSB is to be copied into the bit map.

bitmap an array of up to 64K bits (4096 words).

index is the bit number to be altered; where 0 is the MSB of the first word of *bitmap.*

See also SetBitMap, TestBitMap, TestSetBitMap, and GetBitMap.

PutByte

This routine writes a byte into a packed array of bytes. The leftmost byte of the first word of *array* is byte number 0. The array can be up to 32K words, so the byte index can be from 0 to 65,535. Addresses above 32,767 look like negative numbers because there is no unsigned integer data type in FORTRAN.

```
CALL PutByte(newbyte,array,index)

integer*2 newbyte,array(*),index
```

where:

newbyte is the value of the byte to be stored into *array*. The value is contained in the low byte of *newbyte*; the high byte is ignored.

array is the array to be modified.

index is the index into *array* indicating the byte to be altered where byte 0 is the left byte of the first word of *array*.

See also GetByte.

PutDibit

This routine writes a dibit (bit pair) into a packed array of dibits. The leftmost two bits of the first word of *array* is dibit number 0. The index is limited to 16 bits, which limits the length of the array to 8K words. Dibit addresses above 32,767 look like negative numbers because there is no unsigned integer data type in FORTRAN.

```
CALL = PutDibit(twobitvalue,array,index)

integer*2 twobitvalue,array(*),index
```

where:

twobitvalue is the value of the dibit to be stored into *array*. The value is contained in the lower 2 bits of *twobitvalue*; upper bits are null.

array is the array to be modified.

index is the index into the array indicating the dibit to be written where dibit number 0 is the leftmost two bits of the first word of *array*.

See also GetDibit.

PutNibble

This routine writes a nibble (4 bits) into a packed array of nibbles. The leftmost four bits of the first word of *array* is nibble number 0. The index is limited to 16 bits, which limits the length of the array to 16K words. Nibble addresses above 32,767 look like negative numbers because there is no unsigned integer data type in FORTRAN on the HP 1000.

```
CALL PutNibble(newnibble,array,index)

integer*2 newnibble,array(*),index
```

where:

newnibble is the value of the nibble to be stored into *array*. The value is contained in the lower 4 bits of *newnibble*; the upper 12 bits are null.

array is the array that will be modified.

index is the index into the array indicating the nibble to be written where nibble number 0 is the leftmost four bits of the first word of *array*.

See also GetNibble.

SetBitMap

This routine sets a bit in a bit map. When accessing bits above 32 Kbytes, you must use negative numbers, as FORTRAN does not have an unsigned integer data type on the HP 1000.

```
CALL SetBitMap(ibuf,ibit)

integer*2 ibuf(*),ibit
```

where:

ibuf is an array of up to 64K bits (4096 words).

ibit is the bit number to set, where 0 is the MSB of the first word of *ibuf*.

SetPriority

This function sets the priority of the currently executing program to a given value. It returns the existing priority for later use in restoring the priority to the current value. It can also be used to read the current priority without setting a new value by calling the routine with *newpriority* equal to 0.

```
oldpriority = SetPriority(newpriority)

integer*2 oldpriority,newpriority,SetPriority
```

where:

oldpriority returns the existing priority of the program.

newpriority is the priority to which you want to set the calling program.

TestBitMap

This routine tests a bit in a bit map. When accessing bits above 32K, you must use negative numbers, as there is no unsigned integer data type on the HP 1000.

```
IF (TestBitMap(ibuf,ibit)) THEN
  bit was set
ENDIF

integer*2 ibuf(*),ibit
logical*2 TestBitMap
```

where:

ibuf specifies an array of up to 64 K bits (4096 words).

ibit specifies the bit to test, where 0 is the MSB of the first word of *ibuf.*

Test_PutByte

This routine puts a byte into an array with a test for zero. This logical*2 function return `.true.` if the existing byte in the array is not zero before it is modified.

```
IF (Test_PutByte(newbyte,array,index)) THEN
  byte was not zero
ENDIF

integer*2 newbyte,array(*),index
logical*2 Test_PutByte
```

where:

newbyte is the byte to be put into the array.

array is the array where the byte is to be stored.

index is the index into the byte array.

See also GetByte and PutByte.

Test_SetBitMap

This routine tests and sets a bit in a bit map. For either the true or the false return, the bit referenced will be set. When accessing bits above 32K, you must use negative numbers, as there is no unsigned integer data type on the HP 1000.

```
IF (Test_SetBitMap(ibuf,ibit)) THEN
   bit was already set
ENDIF

integer*2 ibuf(*),ibit
logical*2 Test_SetBitMap
```

where:

ibuf is an array of up to 64 K bits (4096 words).

ibit is the bit to test, where 0 is the MSB of the first word of *ibuf*.

Index

Symbols

..CCM, 3-132
..DCM, 3-133
..DLC, 3-134
..FCM, 3-135
..MAP, 5-41
..TCM, 3-136
.ABS, 3-62
.ATAN, 3-63
.ATN2, 3-64
.BLE, 3-65
.CADD, 3-66
.CDBL, 3-67
.CDIV, 3-68
.CFER, 3-69
.CHEB, 3-70
.CINT, 3-71
.CMPY, 3-72
.CMRS, 3-73
.COS, 3-74
  no error return, /COS, 3-167
  range reduction, /CMRT, 3-168
.CPM, 3-75
.CSUB, 3-76
.CTBL, 3-77
.CTOI, 3-78
.DADS, 4-3
.DCO, 4-4
.DCPX, 3-79
.DDE, 4-5
.DDI, .DDR, 4-6
.DDS, 4-7
.DFER, 3-80
.DIN, 4-8
.DINT, 3-81
.DIS, 4-9
.DMP, 4-10
.DNG, 4-11
.DTBL, 3-82
.DTOD, 3-83
.DTOI, 3-84
.DTOR, 3-85
.ENTC, 5-28
.ENTN, 5-28
.ENTP, 5-29
.ENTR, 5-29
.ENTR call sequence, 2-1
.EXP, 3-86
  no error return, /EXP, 3-169
  range reduction, /CMRT, 3-168
.FAD, 3-87
.FDV, 3-88
.FIXD, 4-12
.FLTD, 4-13
.FLUN, 3-89
.FMP, 3-90
.FMUI, 5-32
.FMUO, 5-32
.FMUP, 5-32
.FMUR, 5-34
.FPWR, 3-91
.FSB, 3-87
.GOTO, 5-35
.ICPX, 3-92
.IDBL, 3-93
.IENT, 3-94
.ITBL, 3-95
.ITOI, 3-96
.LOG, 3-97
  no error return, /LOG, 3-171
.LOG0, 3-98
  no error return, /LOG0, 3-172
.MANT, 3-99
.MAP, 5-36
.MAX1, 3-100
.MIN1, 3-100
.MOD, 3-101
.MPY, 3-102
.NGL, 3-103
.OPSY, 5-37
.PACK, 3-104
.PAUS, 5-38
.PCAD, 5-39
.PWR2, 3-105
.RTOD, 3-106
.RTOI, 3-107
.RTOR, 3-108
.RTOT, 3-109
.SIGN, 3-110
.SIN, 3-111
.SQRT, 3-112
  no error return, /SQRT, 3-174
.TADD, 3-113
.TAN, 3-114
  no error return, /TAN, 3-175
  range reduction, /CMRT, 3-168
.TANH, 3-115
  range reduction, /CMRT, 3-168
.TAPE, 5-40
.TCPX, 3-116
.TDBL, 3-117
.TDIV, 3-113
.TENT, 3-118
.TFTD, 4-14
.TFXD, 4-15
.TINT, 3-119, 3-176
.TMPY, 3-113

.TPWR, 3-120
.TSUB, 3-113
.TTOI, 3-121
.TTOR, 3-122
.TTOT, 3-123
.XADD, 3-124
.XCOM, 3-125
.XDIV, 3-126
.XFER, 3-127
.XFTD, 4-16
.XFXD, 4-17
.XMPY, 3-128
.XPAK, 3-129
.XPLY, 3-130
.XSUB, 3-124
.YINT, 3-131
#COS, 3-137
#EXP, 3-138
#LOG, 3-139
#SIN, 3-140
$EXP, 3-141
$LOG, 3-142
$LOGT, 3-143
$SETP, 5-42
$SQRT, 3-144
$TAN, 3-145
%ABS, 3-146
%AN, 3-147
%AND, 3-148
%ANH, 3-149
%BS, 3-150
%FIX, 3-151
%IGN, 3-152
%IN, 3-153
%INT, 3-154
%LOAT, 3-155
%LOG, 3-156
%LOGT, 3-157
%NT, 3-158
%OR, 3-159
%OS, 3-160
%OT, 3-161
%QRT, 3-162
%SIGN, 3-163
%SSW, 5-43
%TAN, 3-164
%XP, 3-165
/ATLG, 3-166
/CMRT, 3-168
/COS, 3-167
/EXP, 3-169
/EXTH, 3-170
/LOG, 3-171
/LOG0, 3-172
/SIN, 3-173
/SQRT, 3-174
/TAN, 3-175
/TINT, 3-176

A

A_B_Registers, 12-1
A_Register, 12-1
A- and B-Registers, ABREG, 5-2
A2 to decimal converison, SA2DE, 10-31
A990
  reading real-time clock, 7-40
  setting real-time clock, 7-54
ABREG, 5-2, 12-1
ABS, 3-2
ABS entry (call-by-name), %BS, 3-150
absolute value
  double real, 3-62
  extended real, 3-20
  integer, 3-43
  of a real, 3-2
  of complex (real), 3-12
  routine
    DVABS (double precision), 8-13
    DWABS (EMA double precision), 8-13
    VABS (single precision), 8-13
    WABS (EMA single precision), 8-13
AccessLU, check for LU access, 6-2
accounting limits, user and group, SetAcctLimits, 6-17
actual address, array element, .MAP, 5-36
add
  complex to complex, 3-66
  double integer, 4-3
  DVADD (double precision), 8-9
  DWADD (EMA double precision), 8-9
  extended real, 3-124
  real, 3-87
  VADD (single precision), 8-9
  WADD (EMA single precision), 8-9
address
  actual, array element, .MAP, 5-36
  array element, .MAP, 5-41
  true, of parameter, .PCAD, 5-39
address transfer
  .ENTC, 5-28
  .ENTN, 5-28
  .ENTP, 5-29
  .ENTR, 5-29
AddressOf, 7-1
AIMAG, 3-3
AINT, 3-4
AINT entry, %INT, 3-154
ALOG, 3-5
ALOG entry (call-by-name), %LOG, 3-156
ALOGT, 3-6
ALOGT entry (call-by-name), %LOGT, 3-157
alternate returns, 2-3
AMAX0, 3-7
AMAX1, 3-8
AMIN0, 3-7
AMIN1, 3-8

AMOD, 3-9
AND entry, logical (call-by-name), %AND, 3-148
applications (VIS), 9-11
arctangent
  extended real, 3-21
  of a real, 3-10
  quotient of two double reals, 3-64
  quotient of two extended reals, 3-22
  quotient of two reals, 3-11
arithmetic, double real, 3-113
arithmetic routines, 8-9
array initialization (VIS), 9-12
arrays in memory, 8-2
ASCII, digit to internal numeric conversion, .FMUI, 5-32
ASCII to double integer conversion
  DecimalToDint, 7-21
  OctalToDint, 7-37
ASCII to single integer conversion
  DecimalToInt, 7-22
  OctalToInt, 7-38
ATACH, attach to session, 6-3
ATAN, 3-10
ATAN entry (call-by-name), %TAN, 3-164
ATAN2, 3-11
ATCRT, attach a CRT, 6-4
attach a CRT, ATCRT, 6-4
attach to session, ATACH, 6-3

B

B_Register, 12-1
backspace tape, .TAPE, 5-40
bit map manipulation routines, 7-2
  ChangeBits, 7-2
  CheckBits, 7-2
  ClearBitMap, 12-2
  FindBits, 7-3
  GetBitMap, 12-8
  HpZDumpBitMap, 12-48
  PutBitMap, 12-82
  SetBitMap, 12-84
  Test_SetBitMap, 12-86
  TestBitMap, 12-85
bits
  change, 7-2
  check, 7-2
  find free bits, 7-3
BlankString, 7-4
block and sector to track and sector, 7-4
BlockToDisc, 7-4
buffer, zero, ClearBuffer, 7-6
byte manipulation, GetByte, 12-8

C

CABS, 3-12
calculate sign
  real or integer times integer, 3-50
  real or integer times real, 3-54
call LOGOF, CLGOF, 6-5
call LOGON, CLGON, 6-6
calling conventions, 2-1
CaseFold, convert lowercase to uppercase, 7-5
CDS programs, 2-4
CEXP, 3-13
CEXP entry, #EXP, 3-138
ChangeBits, 7-2
character strings, 2-5
characters in arrays, compare, CharsMatch, 7-6
CharFill, 7-5
CharsMatch, 7-6
Chebyshev series, evaluate, 3-70
check if user is in group (RTE-A only), Member, 6-13
check system session table address, FromSySession, 6-8
CheckBits, 7-2
CLCUC, convert lowercase to uppercase, 7-7
ClearBitMap, 12-2
ClearBuffer, zero a passed buffer, 7-6
CLGOF, call LOGOF, 6-5
CLGON, call LOGON, 6-6
CLOG, 3-14
CLOG entry, #LOG, 3-139
CMNDO
  example program, 7-10
  routines, 7-8
    HpReadCmndo, 7-9
    HpStartCmndo, 7-8
    HpStopCmndo, 7-10
CmndStackInit, initialize command stack, 7-12
CmndStackMarks, check for marked lines, 7-13
CmndStackPush, add line to command stack, 7-13
CmndStackRestore, restore command stack, 7-14
CmndStackSaveP, CmndStackRstrP, save and reset command stack, 7-15
CmndStackScreen, 7-16
CmndStackStore, store command stack contents in a file, 7-17
CmndStackUnmark, remove marks from command stack lines, 7-17
CMPLX, 3-15
combinations of vector instructions, 9-7
command line editing, RteShellRead routine, 7-45
command stack example program, 7-18
commas, PutInCommas, 7-39
common logarithm
  double real, 3-98
  extended real, 3-29
  real, 3-6
compare, double integer, 4-4
compare characters in arrays, CharsMatch, 7-6
CompareBufs, 12-3
CompareWords, 12-3
complement
  complex, 3-132
  double real unpacked mantissa, 3-125
  real, 3-135
complex, 3-15

conjugate, 3-16
exponential, 3-13
extract real, 3-53
return extended precision, 3-67
natural logarithm, 3-14
CompressAsciiRLE, 12-4
Concat, concatenate strings, 7-20
ConcatSpace, concatenate strings with embedded blanks, 7-20
CONJG, 3-16
conjugate, of complex, 3-16
control transfer, computed GOTO, .GOTO, 5-35
conventions, calling, 2-1
conversion
ASCII digit to internal numeric, .FMUI, 5-32
ASCII to double integer, OctalToDint, 7-37
ASCII to double integer conversion, DecimalToDint, 7-21
ASCII to single integer, OctalToInt, 7-38
ASCII to single integer conversion, 7-22
block and sector to track and sector, 7-4
complex real to double real, 3-77
double integer to ASCII
DintToDecimal, 7-22
IntToDecimal, 7-32
double integer to double real, 4-14
double integer to extended real, 4-16
double integer to octal
DintToOctal, 7-23
IntToOctal, 7-34
double integer to real, 4-13
double length record number to real, 4-2
double precision to integer, /TINT, 3-176
double real to complex real, 3-116
double real to double integer, 4-15
double real to extended real without rounding, 3-117
double real to integer, 3-81, 3-119
double real to real, 3-103
extended real to complex, 3-79
extended real to double integer, 4-17
extended real to double real, 3-82
extended real to real, 3-56
without rounding, 3-57
HP 1000 single precision floating point to IEEE, FCHI, 11-2
IEEE standard format double precision to HP 1000, DFCIH, 11-2
IEEE standard format single precision to HP 1000, FCIH, 11-3
integer to complex, 3-92
integer to double real, 3-95
integer to extended real, 3-93
integer to real, 3-42
internal to normal format, .FMUP, 5-32
lowercase to uppercase
CaseFold, 7-5
CLCUC, 7-7
numeric to ASCII, .FMUO, 5-32
real part of complex to integer, 3-71
real to double integer, 4-12
real to double real, 3-65
real to extended, 3-23
real to integer, 3-47
segment address to program name and LU number, IdAddToName, 7-30
segment address to segment number, IdAddToNumber, 7-30
segment number to segment address, IdNumberToAdd, 7-31
signed mantissa into normalized real format, 3-104
track, sector, to double integer block number, 7-24
converting FORTRAN DO loops, with VIS, 9-1
copy one string to another, StringCopy, 7-50
copy routine
DVSWP (double precision), 8-24
DWSWP (EMA double precision), 8-24
VSWP (single precision), 8-24
WSWP (EMA single precision), 8-24
COS, 3-17
COS entry (call-by-name), %OS, 3-160
cosine
#COS, 3-137
complex, 3-18
double precision, 3-74
extended real, 3-24
real, 3-17
CSNCS, 3-18
CSQRT, 3-19
Cyclic Redundancy Check (CRC), 12-13

D

D1 decimal substring carries, SDCAR, 10-34
D1 format, 10-6
D1 to D2 decimal format conversion, SD1D2, 10-37
D2 decimal substring carries, SCARY, 10-33
D2 format, 10-4
D2 to A2 substring conversion, SDEA2, 10-36
D2 to D1 decimal substring conversion, SD2D1, 10-38
DABS, 3-20
DATAN, 3-21
DATN2, 3-22
DayTime, seconds since January 1, 1970, 7-21
DBLE, 3-23
DCOS, 3-24
DDINT, 3-25
deallocate ID segment, IDCLR, 7-30
decimal string arithmetic subroutines, 10-1
DecimalToDint, ASCII to double integer conversion, 7-21
DecimalToInt, ASCII to single integer conversion, 7-22
decrement double integer, 4-5
(and skip if zero), 4-7
default parameters, 2-2

detach from session, DETACH, 6-7
DEXP entry, 3-26
  $EXP, 3-141
DFCHI, HP 1000 double precision floating point to IEEE conversion, 11-1
DFCIH, IEEE standard format double precision to HP 1000 conversion, 11-2
difference, positive real, 3-27
DIM, 3-27
DintToDecimal, double integer to ASCII conversion, 7-22
DintToDecimalr, double integer to ASCII conversion, 7-23
DintToOctal, double integer to octal conversion, 7-23
DintToOctalr, double integer to octal conversion, 7-24
direct address, AddressOf, 7-1
direct calls, 2-3
DiscSize, tracks and sector per track, 7-25
DiscToBlock, 7-24
divide
  complex by complex, 3-68
  double integer, 4-6
  DVDIV (double precision), 8-9
  DWDIV (EMA double precision), 8-9
  extended real by extended real, 3-126
  real, 3-88
  substrings, SDIV, 10-19
  VDIV (single precision), 8-9
  WDIV (EMA single precision), 8-9
DLOG entry, 3-28
  $LOG, 3-142
DLOGT entry, 3-29
  $LOGT, 3-143
DMAX1, DMIN1, 3-30
DMOD, 3-31
DO loops, converting, with VIS, 9-1
dot product routine, 8-17
  DVDOT (double precision), 8-17
  DWDOT (EMA double precision), 8-17
  VDOT (single precision), 8-17
  WDOT (EMA single precision), 8-17
double integer to ASCII conversion
  DintToDecimal, 7-22
  DintToDecimalr, 7-23
double integer to octal conversion
  DintToOctal, 7-23
  DintToOctalr, 7-24
double precision floating point conversion
  DFCHI, 11-1
  DFCIH, 11-2
double precision to integer conversion, /TINT, 3-176
double real
  arithmetic, 3-113
  remainder, 3-101
DPOLY, 3-32
DS, programmatic logon, 6-4
DSIGN, 3-34
DSIN, 3-35
DSQRT entry, 3-36
  $SQRT, 3-144
DTACH, detach from session, 6-7
DTAN, 3-37
  no error, $TAN, 3-145
DTANH, 3-38
DVABS, absolute value routine (double precision), 8-13
DVADD, vector add (double precision), 8-9
DVDIV, vector divide (double precision), 8-9
DVDOT, vector dot product routine (double precision), 8-17
DVMAB, vector largest value (absolute) (double precision), 8-20
DVMAX, vector largest value (double precision), 8-20
DVMIB, vector smallest value (absolute) (double precision), 8-20
DVMIN, vector smallest value (double precision), 8-20
DVMOV, vector move routine (double precision), 8-24
DVMPY, vector multiply (double precision), 8-9
DVNRM, vector sum routine (absolute) (double precision), 8-14
DVPIV, vector pivot routine (double precision), 8-18
DVSAD, vector-scalar add (double precision), 8-11
DVSDV, vector-scalar divide (double precision), 8-11
DVSMY, vector-scalar multiply (double precision), 8-11
DVSSB, vector-scalar subtract (double precision), 8-11
DVSUB, vector subtract (double precision), 8-9
DVSUM, vector sum routine (double precision), 8-14
DVSWP, vector copy routine (double precision), 8-24
DVWMV
  vector non- EMA to EMA copy routine (double precision), 8-26
  vector non-EMA to EMA move routine (double precision), 8-26
DWABS, absolute value routine (EMA double precision), 8-13
DWADD, vector add (EMA double precision), 8-9
DWDIV, vector divide (EMA double precision), 8-9
DWDOT, vector dot product routine (EMA double precision), 8-17
DWMAB, vector largest value (absolute) (EMA double precision), 8-20
DWMAX, vector largest value (EMA double precision), 8-20
DWMIB, vector smallest value (absolute) (EMA double precision), 8-20
DWMIN, vector smallest value (EMA double precision), 8-20

DWMOV, vector move routine (EMA double precision), 8-24
DWMPY, vector multiply (EMA double precision), 8-9
DWNRM, vector sum routine (absolute) (EMA double precision), 8-14
DWPIV, vector pivot routine (EMA double precision), 8-18
DWSAD, vector-scalar add (EMA double precision), 8-11
DWSDV, vectro-scalar divide (EMA double precision), 8-11
DWSMY, vector-scalar multiply (EMA double precision), 8-11
DWSSB, vector-scalar subtract (EMA double precision), 8-11
DWSUB, vector subtract (EMA double precision), 8-9
DWSUM, vector sum routine (EMA double precision), 8-14
DWSWP, vector copy routine (EMA double precision), 8-24
DWVMV, vector EMA/non-EMA move routine (double precision), 8-26

E

ElapsedTime, 7-25
EMA
  call by value and call by reference (VIS), 9-17
  considerations (VIS), 9-16
  copy routine, DVWMV (double precision), 8-26
  variables, 2-5
EMA/non-EMA, move routine, 8-26
  DVWMV (double precision), 8-26
  VWMOV (single precision), 8-26
  WVMOV (single precision), 8-26
EMA/non-EMA move routine, DWVMV (double precision), 8-26
end-of-file, perform on tape, .TAPE, 5-40
ENTIE, 3-39
ENTIER
  extended real, 3-40
  real, 3-39
ENTIX, 3-40
ER0.E, 5-3
ERRLU, 5-4
ERRO, 5-5
error code
  for ERO.E, 5-5
  for ERRLU, 5-4
error messages (VIS), 9-62
ETime, 7-25
Euclidean norm, 8-17
example, using CMNDO routines, 7-10
example VIS programs, 9-24
EXP, 3-41
EXP entry (call-by-name), %XP, 3-165
ExpandAsciiRLE, 12-5
exponential, extended real, 3-26
exponentiate
  double real to double real power, 3-123
  double real to integer power, 3-121
  double real to unsigned integer power, 3-120
  integer to integer power, 3-96
  real to double real power, 3-109
  real to integer power, 3-107
  real to real power, 3-108
  real to unsigned integer power, 3-91
exponentiate e
  double real power, 3-86
  real power, 3-41
extend complement, real, 3-133
extended real
  to integer, truncate, 3-25, 3-46
  to real, conversion, 3-56
    without rounding, 3-57
extract real from complex, 3-53
  return extended precision, 3-67

F

FakeSpStatus, 12-6
Fast FORTRAN Processor (FFP), 2-6
FCHI, HP 1000 single precision floating point to IEEE conversion, 11-2
FCIH, IEEE standard format single precision to HP 1000 conversion, 11-3
Fgetopt routine, 7-26
FillBuffer, 12-7
FindBits, 7-3
FirstCharacter, 12-7
FLOAT, 3-42
FLOAT entry (call-by-name), %LOAT, 3-155
FLTDR, 4-2
format of routines, 3-1, 4-1, 5-1
FORTRAN
  DO loops, converting, with VIS, 9-1
  routines callable from, 2-6
FromSySession, 6-8
FTRAP, 5-6

G

get
  redirection commands, 7-28
  runstring option, 7-26
get a character, SGET, 10-10
get session number, GETSN, 6-11
GetAcctInfo, access user and group accounting, 6-8
GetBitMap, 12-8
GetByte, 12-8
GetDibit, 12-9
GetFatherIdNum, 7-28
GetNibble, 12-9
GetOwnerNum, 6-10
GetRedirection routine, 7-28
GetResetInfo, access/reset user accounting, 6-10
GetRteTime, 7-29
GetRunString, 12-10

GETSN, get session number, 6-11
GETST, 5-9
GetString, 12-11
GPNAM, 6-11
graphics coordinate transformation (VIS), 9-15
greatest integer
  (ENTIER), real, 3-39
  double real, 3-118
  real, 3-94
group ID
  GroupToId, 6-12
  OwnerToId, 6-14

H

HexToInt, 7-29
HMSCtoRteTime, 7-29
HP 1000, single precision floating point to IEEE conversion, FCHI, 11-2
HPCRT library routines, 12-1
HpCrtCharMode, 12-11
HpCrtCheckStraps, 7-12, 12-12
HpCrtCRC16_F, 12-13
HpCrtCRC16_S, 12-13
HpCrtGetCursor, 12-14
HpCrtGetCursorXY, 12-15
HpCrtGetfield_I, 12-16
HpCrtGetfield_S, 12-17
HpCrtGetLine_Pos, 12-18
HpCrtGetMenuItem, 12-19
HpCrtHardReset, 12-19
HpCrtLineMode, 12-20
HpCrtMenu, 12-20
HpCrtNlsMenu, 12-21
HpCrtNlsXMenu, 12-21
HpCrtPageMode, 12-22
HpCrtParityChk, 12-22
HpCrtParityGen, 12-23
HpCrtQTDPort7, 12-23
HpCrtReadChar, 12-24
HpCrtReadPage, 12-25
HpCrtRestorePort, 12-26
HpCrtSavePort, 12-26
HpCrtSchedProg, 12-27
HpCrtSchedProg_S, 12-27
HpCrtScreenSize, 12-27
HpCrtSendChar, 12-28
HpCrtSSRCDriver, 12-29
HpCrtSSRCDriver?, 12-29
HpCrtStatus, 12-30
HpCrtStripChar, 12-31
HpCrtStripCntrls, 12-31
HpCrtXMenu, 12-32
HpCrtXReadChar, 12-32
HpCrtXSendChar, 12-33
HpLowerCaseName, 12-33
HpReadCmndo, 7-9
HpRte6, 12-34
HpRteA, 12-34
HpStartCmndo, 7-8
HpStopCmndo, 7-10
HpZ, mini-formatter, 12-35
HpZAscii64, 12-38
HpZAscii95, 12-38
HpZAsciiHpEnh, 12-39
HpZAsciiMne3, 12-40
HpZAsciiMne4, 12-41
HpZBackSpaceIbuf, 12-41
HpZBinc, 12-42
HpZBino, 12-42
HpZDecc, 12-43
HpZDeco, 12-42
HpZDecv, 12-42
HpZDefIBuf, 12-43
HpZDefIString, 12-44
HpZDefOBuf, 12-44
HpZDicv, 12-44
HpZDParse, 12-45
HpZDumpBitMap, 12-48
HpZDumpBuffer, 12-49
HpZFieldDefine, 12-50
HpZFmpWrite, 12-51
HpZGetNextChar, 12-51
HpZGetNextStrDsc, 12-52
HpZGetNextToken, 12-52
HpZGetNumB2, 12-53
HpZGetNumB4, 12-53
HpZGetNumD2, 12-53
HpZGetNumD4, 12-53
HpZGetNumO2, 12-53
HpZGetNumO4, 12-53
HpZGetNumStrDsc, 12-54
HpZGetNumX, 12-55
HpZGetRemStrDsc, 12-56
HpZHexc, 12-56
HpZHexi, 12-57
HpZHexo, 12-58
HpZIBufRemain, 12-58
HpZIBufReset, 12-58
HpZIBufUsed, 12-58
HpZIBufUseStrDsc, 12-59
HpZInsertAtFront, 12-60
HpZmbt, 12-60
HpZMesss, 12-61
HpZMoveString, 12-62
HpZmvc, 12-62
HpZmvs, 12-63
HpZmvs_Control, 12-64
HpZmvs_Escape, 12-65
HpZmvs_Large, 12-65
HpZNlsMvs, 12-66
HpZNlsSubset, 12-66
HpZOBufReset, 12-66
HpZOBufUsed, 12-67
HpZOBufUseStrDsc, 12-67
HpZOctc, 12-68
HpZOctd, 12-68
HpZOcto, 12-69
HpZOctv, 12-69
HpZPadToCount, 12-70

HpZPadToPosition, 12-70
HpZParse, 12-71
HpZPeekNextChar, 12-51
HpZPlural, 12-73
HpZPopObuf, 12-75
HpZPrintPort, 12-74
HpZPushObuf, 12-75
HpZQandA, 12-76
HpZReScan, 12-76
HpZRomanNumeral, 12-77
HpZsbt, 12-78
HpZStripBlanks, 12-78
HpZUdeco, 12-78
HpZUdecv, 12-79
HpZWriteExec14, 12-79
HpZWriteLU, 12-79
HpZWriteToString, 12-80
HpZWriteXLU, 12-79
HpZYesOrNo, 12-80
hyperbolic tangent
  double real, 3-115
  extended real, 3-38
  real, 3-61

I

IABS, 3-43
IABS entry (call-by-name), %ABS, 3-146
IAND, 3-44
ID segment, deallocate, IDCLR, 7-30
IdAddToName, convert segment address to program name and LU number, 7-30
IdAddToNumber, convert segment address to segment number, 7-30
IDCLR, 7-30
IDIM, 3-45
IDINT, 3-46
IdNumberToAdd, convert segment number to segment address, 7-31
IdToGroup, 6-12
IdToOwner, 6-13
IEEE standard format
  double precision to HP 1000 conversion, DFCIH, 11-2
  single precision to HP 1000 conversion, FCIH, 11-3
IFIX, 3-47
IFIX entry (call-by-name), %FIX, 3-151
IGET, 5-10
imaginary part, extraction of, 3-3
INAMR routine, 5-11
inclusive OR entry
  (call-by-name), %OR, 3-159
  integer, 3-51
  logical, 3-49
increment double integer, 4-8
  (and skip if zero), 4-9
increment parameters other than one (VIS), 9-8
IND.E, 5-14
initialize a square matrix (VIS), 9-11
initialize an array in a certain order (VIS), 9-12
input buffer, read, 5-18
INT, 3-48
INT entry (call-by-name), %NT, 3-158
integer inclusive OR, 3-51
integer to ASCII conversion
  IntToDecimal, 7-32
  IntToDecimalr, 7-32
integer to octal conversion
  IntToOctal, 7-34
  IntToOctalr, 7-34
internal routines, 10-31
internal to normal format conversion, .FMUP, 5-32
IntString, 7-31
IntToDecimal, integer to ASCII conversion, 7-32
IntToDecimalr, integer to ASCII conversion, 7-32
IntToHex, 7-33
IntToHexR, 7-33
IntToOctal, integer to octal conversion, 7-34
IntToOctalr, integer to octal conversion, 7-34
inverse tangent, double real, 3-63
InvSeconds, 7-35
IOR, 3-49
ISIGN, 3-50
ISIGN entry (call-by-name), %SIGN, 3-163
ISSR, 5-15
ISSW, 5-16
ISSW entry (call-by-name), %SSW, 5-43
IXGET, 5-10
IXOR, 3-51

J

JSCOM, substring compare, 10-7

L

L2 norm, 8-17
language instruction set, 2-6
largest value
  DVMAX (double precision), 8-20
  DWMAX (EMA double precision), 8-20
  VMAX (single precision), 8-20
  WMAX (EMA single precision), 8-20
largest value (absolute)
  DVMAB (double precision), 8-20
  DWMAB (EMA double precision), 8-20
  VMAB (single precision), 8-20
  WMAB (EMA single precision), 8-20
LastMatch, 7-35
LeapYear, 7-35
load and complement, real, 3-134
locked LU, 7-36
  WhoLockedLu, 7-53
locked resource number, WhoLockedRn, 7-53
logical AND entry (call-by-name), %AND, 3-148
logical inclusive OR, 3-49
logical product, integer, 3-44
logon, programmatic, 6-4
lowercase to uppercase, CaseFold, 7-5

LU, locked, WhoLockedLu, 7-53
LuLocked, 7-36
LUSES, get user table address, 6-13

M

magnetic tape, position, PTAPE, 5-24
magnetic tape utility functions, MAGTP, 5-17
MAGTP, 5-17
mantissa
  complement, double real unpacked, 3-125
  normalized, rounded, packed (double real), 3-129
  real, extract, 3-99
matrix
  inversion (VIS), 9-30
  multiiplication EMA example, 9-22
  transposition (VIS), 9-13
MAX/MIN routines, 8-20
MAX0, 3-7
MAX1, 3-8
maximum
  double real value, 3-100
  extended real, 3-30
  integer value, 3-7
  real value, 3-8
Member, check if user is in group, 6-13
microcoded routines (RPLs), 2-6
MIN0, 3-7
MIN1, 3-8
minimum
  double real value, 3-100
  extended real, 3-30
  integer value, 3-7
  real value, 3-8
MinStrDsc, 12-82
MOD, 3-52
modulus, of complex (real), 3-12
move
  complex to complex, 3-69
  extended real to extended real, 3-127
  name of program from ID segment, PNAME, 5-23
move routines, 8-24
  DVMOV (double precision), 8-24
  DWMOV (EMA double precision), 8-24
  VMOV (single precision), 8-24
  WMOV (EMA single precision), 8-24
MoveWords, 7-36
multidimensional arrays, efficiency (VIS), 9-19
multiply
  by 2 to integer power, 3-105
  complex by complex, 3-72
  double integer, 4-10
  extended real by extended real, 3-128
  hardware, 3-102
  real, 3-90
  substrings, SMPY, 10-22
multiply routines
  DVMPY (double precision), 8-9
  DWMPY (EMA double precision), 8-9
  VMPY (single precision), 8-9
  WMPY (EMA single precision), 8-9
MyIdAdd, return segment address, 7-36

N

NAMR routine, 5-18
natural logarithm
  complex, 3-14
  double real, 3-97
  extended real, 3-28
  real, 3-5
negate
  double integer, 4-11
  double real, 3-136
negative increment (VIS), 9-10
nested DO loops example (VIS), 9-6
NOT function (call-by-name), %OT, 3-161
numeric to ASCII conversion, .FMUO, 5-32
NumericTime, 7-37

O

obtaining efficiency with multidimensional arrays (VIS), 9-19
OctalToDint, ASCII to double integer conversion, 7-37
OctalToInt, ASCII to single integer conversion, 7-38
one dimensional array examples (VIS), 9-2
operating system determination, .OPSY, 5-37
OR
  integer inclusive, 3-51
  logical inclusive, 3-49
OR entry, inclusive (call-by-name), %OR, 3-159
output editing routine, SEDIT, 10-28
output pause message, PAU.E, 5-22
overflow bit, OVF, 5-21
OVF, 5-21
OwnerToId, return user ID and group ID, 6-14

P

parameters, default, 2-2
parameters from ID segment, RMPAR, 5-25
parse string
  SplitCommand, 7-47
  SplitString, 7-48
Pascal, routines callable from, 2-7
PAU.E, 5-22
pause message
  PAU.E, 5-22
  print, 5-38
PCAL call sequence (CDS), 2-4
pivot routine, 8-18
  DVPIV (double precision), 8-18
  DWPIV (EMA double precision), 8-18
  VPIV (single precision), 8-18
  WPIV (EMA single precision), 8-18

PNAME, 5-23
pointers, list of, $SETP, 5-42
polynomial
  extended real, 3-130
  quotient double precision, 3-32
position magnetic tape, PTATE, 5-24
positive difference, integer, 3-45
print pause message, .PAUS, 5-38
printing library error messages, ER0.E, 5-3
priority, ProgramPriority, 7-38
processor, Fast FORTRAN (FFP), 2-6
ProgIsSuper, check for super program, 6-14
programmatic logon, 6-4
ProgramPriority, 7-38
ProgramTerminal, 7-39
PTAPE, 5-24
put a character, SPUT, 10-13
PutBitMap, 12-82
PutByte, 12-83
PutDibit, 12-83
PutInCommas, 7-39
PutNibble, 12-84

Q

quotient, polynomial, double precision, 3-32

R

raise
  complex to integer power, 3-78
  double real to double real power, 3-83
  double real to real power, 3-85
  extended real to integer power, 3-84
  real to double real power, 3-106
  real to real power, 3-122
range reduction for SIN, .COS, .TAN, .EXP, .TANH; /CMRT, 3-168
read input buffer, 5-18
read memory address
  IGET, 5-10
  IXGET, 5-10
ReadA990Clock routine, 7-40
real
  to double real, conversion, 3-65
  to integer, conversion, 3-47
real remainder, 3-9
REAL routine, 3-53
real to integer, truncate, 3-48
reduce argument, for SIN, COS, TAN, EXP, 3-73
regular expression, routines, 7-41
remainder
  double real, 3-101
  extended real, 3-31
  integer, 3-52
  real, 3-9
remove trailing blanks, TrimLen, 7-52
ResetAcctTotals, reset user and group accounting totals, 6-15
ResetTimer, 7-40
resouce number, locked, WhoLockedRn, 7-53
return address adjust
  .ENTC, 5-28
  .ENTN, 5-28
  .ENTP, 5-29
  .ENTR, 5-29
return direct address, AddressOf, 7-1
return group ID
  GroupToId, 6-12
  OwnerToId, 6-14
return group name
  GPNAM, 6-11
  IdToGroup, 6-12
return owner's ID, GetOwnerNum, 6-10
return segment address, MyIdAdd, 7-36
return session number
  RTNSN, 6-16
  USNUM, 6-20
return user ID, OwnerToId, 6-14
return user name
  IdToOwner, 6-13
  SessnToOwnerName, 6-16
  USNAM, 6-19
return user table address, LUSES, 6-13
rewind tape, .TAPE, 5-40
Rex routines, 7-41
RexBuildPattern, 7-42
RexBuildSubst, 7-42
RexExchange, 7-43
RexMatch, 7-44
RMPAR, 5-25
round, real, 3-39
rounding of digit string produced by .FMUO, .FMUR, 5-34
routines
  callable from FORTRAN, 2-6
  callable from Pascal, 2-7
  format of, 3-1, 4-1, 5-1
RPL, 2-6
RT_ER, 5-26
RteDateToYrDoy, 7-44
RteShellRead, 7-45
RteTimeToHMSC, 7-46
RTNSN, return session number, 6-16
RTRAP, 5-6
run length encoding, 12-4, 12-5
runstring
  option, getting, 7-26
  redirection command, getting, 7-28

S

S-Register set, ISSR, 5-15
SA2DE, A2 to decimal conversion, 10-31
SADD, substring add, 10-17
SamInfo (RTE-A only), 7-46
scalar-vector arithmetic routines, 8-11
SCARY, D2 decimal substring carries, 10-33
SD1D2, D1 to D2 decimal format conversion, 10-37

SD2D1, D2 to D1 decimal substring conversion, 10-38
SDCAR, D1 decimal substring carries, 10-34
SDEA2, D2 to A2 substring conversion, 10-36
SDIV, divide substrings, 10-19
Seconds routine, 7-47
seconds since 12 AM January 1, 1970, 7-47
  DayTime, 7-21
SEDIT, output editing routine, 10-28
segment address
  MyIdAdd, 7-36
  to program name an LU number conversion, IdAddToName, 7-30
  to segment number, IdAddToNumber, 7-30
segment number to segment address, IdNumberToAdd, 7-31
session number
  RTNSN, 6-16
  USNUM, 6-20
SessnToOwnerName, return user name, 6-16
set user and group accounting limits, SetAcctLimits, 6-17
SetAcctLimits, set user and group accounting limits, 6-17
SetBitMap, 12-84
SetPriority, 12-84
SFILL, substring fill, 10-9
SGET, get a character, 10-10
SIGN, 3-54
  entry (call-by-name), %IGN, 3-152
sign
  bit, S-Register, ISSW, 5-16
  change, SSIGN, 10-40
  real or integer times integer, calculate, 3-50
  real or integer times real, calculate, 3-54
  transfer, extended real, 3-34
SIN, 3-55
  entry (call-by-name), %IN, 3-153
  range reduction, /CMRT, 3-168
sine
  #SIN call, 3-140
  complex, 3-18
  double precision, 3-111
  double real (no error return), /SIN, 3-173
  extended real, 3-35
  real, 3-17, 3-55
single precision floating point conversion
  FCHI, 11-2
  FCIH, 11-3
smallest value
  DVMIN (double precision), 8-20
  DWMIN (EMA double precision), 8-20
  VMIN (single precision), 8-20
  WMIN (EMA single precision), 8-20
smallest value (absolute)
  DVMIB (double precision), 8-20
  DWMIB (EMA double precision), 8-20
  VMIB (single precision), 8-20
  WMIB (EMA single precision), 8-20
SMOVE, substring move, 10-11
SMPY, multiply substrings, 10-22
SNGL, 3-56
SNGM, 3-57
solution of linear systems (VIS), 9-27
SplitCommand, parse string, 7-47
SplitString, parse string, 7-48
SPOLY, 3-58
SPUT, put a character, 10-13
SQRT, 3-59
  entry (call-by-name), %QRT, 3-162
square matrix, initialize (VIS), 9-11
square root
  complex, 3-19
  double real, 3-112
  extended real, 3-36
  real, 3-59
SSIGN, sign change, 10-40
SSUB, subtract substrings, 10-26
stack, 2-5
statistical examples (VIS), 9-12
StrDsc, 7-49
string
  arithmetic routines, 10-17
  utilities routines, 10-7
string manipulation
  BlankString, 7-4
  Concat, 7-20
  ConcatSpace, 7-20
  GetRunString, 12-10
  GetString, 12-11
StringCopy, copy one string to another, 7-50
subroutines, double integer, 4-1
substring
  add, SADD, 10-17
  compare, JSCOM, 10-7
  fill, SFILL, 10-9
  move, SMOVE, 10-11
subtract
  complex from complex, 3-76
  double integer, 4-3
  DVSUB (double precision), 8-9
  DWSUB (EMA double precision), 8-9
  extended real, 3-124
  real, 3-87
  substrings, SSUB, 10-26
  VSUB (single precision), 8-9
  WSUB (EMA single precision), 8-9
sum routine, 8-14
  DVSUM (double precision), 8-14
  DWSUM (EMA double precision), 8-14
  VSUM (single precision), 8-14
  WSUM (EMA single precision), 8-14
sum routine (absolute)
  DVNRM (double precision), 8-14
  DWNRM (EMA double precision), 8-14
  VNRM (single precision), 8-14
  WNRM (EMA single precision), 8-14
super program, check for, ProgIsSuper, 6-14
superuser, check for/if
  SuperUser, 6-18

UserIsSuper, 6-19
SYCON, 6-18
system console, write to, SYCON, 6-18
SystemProcess, check for/if system process, 6-19
SZONE, zone punch, 10-14

T

TAN, 3-60
TAN entry (call-by-name), %AN, 3-147
tangent
  double real, 3-114
  extended real, 3-37
  real, 3-60
TANH, 3-61
TANH entry (call-by-name), %ANH, 3-149
tape operations, .TAPE, 5-40
terminal, ProgramTerminal, 7-39
Test_PutByte, 12-85
Test_SetBitMap, 12-86
TestBitMap, 12-85
TIMEF, 7-51
TIMEI, TIMEO, 5-27
TimeNow, 7-52
timer, reset, ResetTimer, 7-40
track, sector, to double integer block number conversion, 7-24
tracks and sectors per track, DiscSize, 7-25
trailing blanks, remove, TrimLen, 7-52
transfer
  control, computed GOTO, .GOTO, 5-35
  extended real, 3-80
  sign
    double real to double real, 3-110
    extended real, 3-34
transformation, graphics coordinate (VIS), 9-15
TrimLen, remove trailing blanks, 7-52
true address of parameter, .PCAD, 5-39
true address transfer
  .ENTC, 5-28
  .ENTN, 5-28
  .ENTP, 5-29
  .ENTR, 5-29
truncate
  extended real to integer, 3-25, 3-46
  fractional part of double real, 3-131
  real, 3-4
  real to integer, 3-48
two dimensional array examples (VIS), 9-4

U

unpack, real, 3-89
use applications (VIS), 9-11
user ID, OwnerToId, 6-14
user name
  SessnToOwnerName, 6-16
  USNAM, 6-19
  verify, VFNAM, 6-20
user table address, LUSES, 6-13
UserIsSuper, check for/if superuser, 6-19
USNAM, return user name, 6-19
USNUM, return the session number, 6-20
utility functions, magnetic tape, MAGTP, 5-17

V

VABS, absolute value routine (single precision), 8-13
VADD, vector add (single precision), 8-9
VDIV, vector device (single precision), 8-9
VDOT, vector dot product routine (single precision), 8-17
vector add
  DVADD (double precision), 8-9
  DWADD (EMA double precision), 8-9
  VADD (single precision), 8-9
  WADD (EMA single precision), 8-9
vector arithmetic routines, 8-9
vector copy routine
  DVSWP (double precision), 8-24
  DWSWP (EMA double precision), 8-24
  VSWP (single precision), 8-24
  WSWP (EMA single precision), 8-24
vector divide
  DVDIV (double precision), 8-9
  DWDIV (EMA double precision), 8-9
  VDIV (single precision), 8-9
  WDIV (EMA single precision), 8-9
vector dot product routine
  DVDOT (double precision), 8-17
  DWDOT (EMA double precision), 8-17
  VDOT (single precision), 8-17
  WDOT (EMA single precision), 8-17
vector EMA copy routine, DVWMV (double precision), 8-26
vector EMA/non-EMA, move routine
  DVWMV (double precision), 8-26
  VWMOV (single precision), 8-26
  WVMOV (single precision), 8-26
vector EMA/non-EMA move routine, DWVMV (double precision), 8-26
vector instructions, combinations, 9-7
vector largest value
  DVMAX (double precision), 8-20
  DWMAX (EMA double precision), 8-20
  VMAX (single precision), 8-20
  WMAX (EMA single precision), 8-20
vector largest value (absolute)
  DVMAB (double precision), 8-20
  DWMAB (EMA double precision), 8-20
  VMAB (single precision), 8-20
  WMAB (EMA single precision), 8-20
vector move routine
  DVMOV (double precision), 8-24
  DWMOV (EMA double precision), 8-24
  VMOV (single precision), 8-24
  WMOV (EMA single precision), 8-24
vector multiply
  DVMPY (double precision), 8-9

DWMPY (EMA double precision), 8-9
VMPY (single precision), 8-9
WMPY (EMA single precision), 8-9
vector pivot routine
DVPIV (double precision), 8-18
DWPIV (EMA double precision), 8-18
VPIV (single precision), 8-18
WPIV (EMA single precision), 8-18
vector smallest value
DVMIN (double precision), 8-20
DWMIN (EMA double precision), 8-20
VMIN (single precision), 8-20
WMIN (EMA single precision), 8-20
vector smallest value (absolute)
DVMIB (double precision), 8-20
DWMIB (EMA double precision), 8-20
VMIB (single precision), 8-20
WMIB (EMA single precision), 8-20
vector subtract
DVSUB (double precision), 8-9
DWSUB (EMA double precision), 8-9
VSUB (single precision), 8-9
WSUB (EMA single precision), 8-9
vector sum routine
DVSUM (double precision), 8-14
DWSUM (EMA double precision), 8-14
VSUM (single precision), 8-14
WSUM (EMA single precision), 8-14
vector sum routine (absolute)
DVNRM (double precision), 8-14
DWNRM (EMA double precision), 8-14
VNRM (single precision), 8-14
WNRM (EMA single precision), 8-14
vector-scalar add
DVSAD (double precision), 8-11
DWSAD (EMA double precision), 8-11
VSAD (single precision), 8-11
WSAD (EMA single precision), 8-11
vector-scalar divide
DVSDV (double precision), 8-11
DWSDV (EMA double precision), 8-11
VSDV (single precision), 8-11
WSDV (EMA single precision), 8-11
vector-scalar multiply
DVSMY (double precision), 8-11
DWSMY (EMA double precision), 8-11
VSMY (single precision), 8-11
WSMY (EMA single precision), 8-11
vector-scalar subtract
DVSSB (double precision), 8-11
DWSSB (EMA double precision), 8-11
VSSB (single precision), 8-11
WSSB (EMA single precision), 8-11
verify user name, VFNAM, 6-20
VFNAM, verify user name, 6-20
VIS programs, examples, 9-24
VMAB, vector largest value (absolute) (single precision), 8-20
VMAX, vector largest value (single precision), 8-20
VMIB, vector smallest value (absolute) (single precision), 8-20
VMIN, vector smallest value (single precision), 8-20
VMOV, vector move routine (single precision), 8-24
VMPY, vector multiply (single precision), 8-9
VNRM, vector sum routine (absolute) (single precision), 8-14
VPIV, vector pivot routine (single precision), 8-18
VSAD, vector-scalar add (single precision), 8-11
VSDV, vector-scalar divide (single precision), 8-11
VSMY, vector-scalar multiply (single precision), 8-11
VSSB, vector-scalar subtract (single precision), 8-11
VSUB, vector subtract (single precision), 8-9
VSUM, vector sum routine (single precision), 8-14
VSWP, vector copy routine (single precision), 8-24
VWMOV, vector non-EMA to EMA move routine (single precision), 8-26

W

WABS, absolute value routine (EMA single precision), 8-13
WADD, vector subtract (EMA single precision), 8-9
WDIV, vector divide (EMA single precision), 8-9
WDOT, vector dot product routine (EMA single precision), 8-17
WhoLockedLu, 7-53
WhoLockedRn, 7-53
WMAB, vector largest value (absolute) (EMA single precision), 8-20
WMAX, vector largest value (EMA single precision), 8-20
WMIB, vector smallest value (absolute) (EMA single precision), 8-20
WMIN, vector smallest value (EMA single precision), 8-20
WMOV, vector move routine (EMA single precision), 8-24
WMPY, vector multiply (EMA single precision), 8-9
WNRM, vector sum routine (absolute) (EMA single precision), 8-14
WPIV, vector pivot routine (EMA single precision), 8-18
write to system console, SYCON, 6-18
WriteA990Clock routine, 7-54
WSAD, vector-scalar add (EMA single precision), 8-11
WSDV, vector-scalar divide (EMA single precision), 8-11
WSMY, vector-scalar multiply (EMA single precision), 8-11
WSSB, vector-scalar subtract (EMA single precision), 8-11

WSUB, vector subtract (EMA single precision), 8-9
WSUM, vector sum routine (EMA single precision), 8-14
WSWP, vector copy routine (EMA single precision), 8-24
WVMOV, vector EMA to non-EMA move routine (single precision), 8-26

X

XPOLY, 3-130

Y

YrDoyToMonDom, 7-54
YrDoyToRteDate, 7-55

Z

zero a passed buffer, ClearBuffer, 7-6
zero increment (VIS), 9-9
zone punch, SZONE, 10-14