

Texas Instruments



TI-83 Plus Developer Guide

Version 1.0

Important information

Texas Instruments makes no warranty, either expressed or implied, including but not limited to any implied warranties of merchantability and fitness for a particular purpose, regarding any programs or book materials and makes such materials available solely on an “as-is” basis.

In no event shall Texas Instruments be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of the purchase or use of these materials, and the sole and exclusive liability of Texas Instruments, regardless of the form of action, shall not exceed the purchase price of this product. Moreover, Texas Instruments shall not be liable for any claim of any kind whatsoever against the use of these materials by any other party.

The latest version of this Guide, along with all other up-to-date information for developers, is available at www.ti.com/calc.

© 1999 Texas Instruments Incorporated

Z80 is a trademark of ZiLOG, Inc.

IBM is a registered trademark of International Business Machines.

Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries.

Table of Contents

Chapter 1: Introduction

TI-83 Plus Developer Guide.....	1
Conventions Used in this Guide.....	1
Purpose of this Guide	2
Structure of this Guide	2

Chapter 2: TI-83 Plus Specific Information

Architecture.....	3
Hardware Layer	4
Z80 CPU and Memory	4
Z80 RAM Structure	5
System RAM.....	6
User RAM.....	6
Temporary RAM.....	6
Floating Point Stack	6
Free RAM	7
Operator Stack.....	7
Symbol Table	7
Hardware Stack	7
Flash ROM Structure	8
Boot (Code) Area.....	9
Certification Area	9
Operating System (OS) Area	9
Certificate List Area.....	9
User APPS (Calculator Software Applications)/Data Area.....	9
Swap Area/User APPS/Data Area	10
System Development Environment.....	10
System Routines.....	10
RST Routines.....	11
System RAM Areas.....	11
System Flags	11
OP1 through OP6 RAM Registers.....	16
Safe RAM Locations for Application Use	17
System Variables Area.....	18

Table of Contents (cont.)

System Variables that are Both Input and Output.....	18
System Variable Characteristics	18
Storing and Recalling System Variable Values.....	19
System Variables that Are Output Only	21
User RAM	21
Variable Data Structures	21
Numeric Based Data Types	21
Real Data Type Structure.....	22
Complex Data Type Structure	22
Real List Data Type Structure	22
Complex List Data Type Structure	23
Matrix Data Type Structure	23
Token Based Data Types	24
TI-83 Plus Tokens	24
Program, Protected Program, Equation, New Equation, and String Data Type Structures.....	24
Screen Image Data Type Structure	24
Graph Database Data Type Structure	25
Unformatted AppVar Data Type Structure	25
Guidelines for AppVar Usage.....	25
Variable Naming Conventions	25
Variable Name Spellings.....	27
Predefined Variable Names	27
Variables: A – Z and θ	27
List Variables: L1 – L6.....	27
Matrix Variables: [A] – [J]	28
Equation Variables: Y1 – Y0, X1t – X6t, Y1t – X1t, r1 – r6, u(n), v(n), w(n)	28
String Variables: Str1 – Str0.....	29
Picture Variables: Pic1 – Pic0	29
Graph Database Variables: GDB1 – GDB0	30
Variable: Ans	30
User-Defined Variable Names	30
User-Named Lists.....	31
User-Named Programs	31
User-Named AppVars	32
Accessing User Variables Stored In RAM — (Unarchived).....	32
Accessing Variables that Are Not Programs or AppVars	32
Accessing Programs and AppVar Variables	33

Table of Contents (cont.)

Output from a Variable Search on the Symbol Table	33
Creating Variables.....	35
Storing to Variables	38
Recalling Variables.....	39
Deleting Variables	40
Archiving and Unarchiving.....	42
Related Routines	43
Accessing Archived Variables without Unarchiving	44
Manipulation Routines	48
List Element Routines	48
Matrix Element Routines.....	48
Resizing AppVar, Program, and Equation Variables	49
Symbol Table Structure.....	51
Floating Point Stack (FPS).....	56
Naming Convention.....	57
General Use Rules.....	57
FPS System Routines	58
FPS Allocation Routines	58
FPS Deallocation Routines	59
Copy Data To and From Existing FPS Entries.....	60
Drivers Layer	63
Keyboard.....	63
Display	70
Displaying Using System Routines.....	70
Display Utility Routines.....	70
Displaying Text.....	71
Formatting Numeric Values for Display	75
Entry Points.....	75
Modifying Display Format Settings.....	76
Writing Directly to the Display Driver	76
Reading the Display Driver After Setting X or Y Coordinates	78
Contrast Control.....	80
Split Screen Modes.....	80
Graphing and Drawing — What's the difference?	82
Drawing.....	82
Graphing	82
Graphing and Drawing Utility Routines.....	82
Drawing Routine Specifics	83
Graphing Routine Specifics.....	86

Table of Contents (cont.)

Graph WINDOW Settings.....	86
Graphing in a Split Screen	86
Graphing Routines and System Flags	87
Run (Busy) Indicator	89
APD™ (Automatic Power Down™)	90
Link Port.....	91
Tools and Utilities Layer.....	97
Error Handlers.....	97
Nested Error Handlers	99
Utility Routines	100
Floating-Point Math.....	100
Miscellaneous Math Functions	102
Floating-Point Math Functions that Output Complex Results.....	102
Complex Math.....	103
Other Math Functions	105
Function Evaluation.....	106
Parse Routine	106
Temporary Variables.....	108
Using Temporary Variables.....	109
Managing Temporary Variables	109
Deleting Temps and Setting (pTempCnt)	110
Working with TI Language Localization Applications	112
Entering and Exiting an Application Properly	113
Stand-alone	113
Start-up Code.....	113
Exit Code	114
Stand-alone with Put Away Notification.....	115
Start-up Code.....	116
Put Away Code	118

Chapter 3: Application Development Process

Programming Layer	120
TI-BASIC Programs	120
ASM Programs.....	120
Applications.....	121
ASM versus Applications	121

Development System	121
Using the Simulator System — Requirements for Getting Started.....	121
Creating an Application for Debugging — One-Page and Multi-Page Apps.....	122
A Brief Overview of Certificates and Application Signing	122
Creating Applications that Fit On One Page	122
The Hello Application	123
Accessing System Resources	123
Application Headers	123
Header Creation	123
Calling System Routines	123
Accessing System Variables	123
Defining a String.....	124
Erasing the Screen.....	124
Printing Text to the Screen.....	124
Copying the String.....	124
System RAM Registers	124
Reading a Key Press.....	125
Exiting an Application	125
Creating a Multiple Page Application	125
Branch Table Entries.....	125
Branch Table Placement.....	126
Branch Table Equate File.....	126
Making Off-Page Calls and Jumps.....	126
Creating a Zilog Developer Studio Project.....	127
Creating the Project	127
Adding Files to the Project	127
Project Settings.....	127
Building the Application.....	128
Loading the Application into the Simulator.....	129
Debugging the Application	131
Preparing an Application for Site Testing.....	134
Signing the Application	135
Downloading the App.....	135
Preparing for Public Release	135

Chapter 4: Development Tools

Development Architecture.....	136
Z80 Development System.....	136
Installation	136
TI Software Simulator and Debugger	136
Introduction	136
Installation.....	137
Getting Started.....	137
Breakpoints	141
Trace Options	141
CPU View Window	142
Disassembly View Window	143
Flash View Window.....	144
RAM View Window.....	144
Memory Map Window	145
Calculator Simulator Window	145
Trace Log Window	146
Loading Applications, Operating System, and RAM Files.....	148
Terminating a Session	150
Support in Writing Applications	150

GLOSSARY	151
-----------------------	------------

Appendix A — System Routines.....	156
--	------------

Display	157
Bit_VertSplit	159
CheckSplitFlag.....	160
ClearRow	161
ClrLCD	162
ClrLCDFull	163
ClrOP2S.....	164
ClrScrn.....	165
ClrScrnFull	166
ClrTxtShd.....	167
DispDone	168
DispHL	169
DisplayImage	170

Table of Contents (cont.)

DispOP1A	172
EraseEOL	173
FormBase	174
FormDCplx.....	176
FormEReal.....	178
FormReal	179
LoadPattern	180
Load_SFont	181
OutputExpr.....	182
PutC.....	183
PutMap	184
PutPS.....	185
PutS.....	187
PutTokString	189
RestoreDisp	190
RunIndicOff.....	191
RunIndicOn.....	192
SaveDisp	193
SetNorm_Vals.....	194
SFont_Len	195
SStringLength	196
VPutMap	197
VPutS.....	198
VPutSN	200
Edit.....	202
CloseEditBufNoR	203
CursorOff	204
CursorOn	205
DispEOL	206
KeyToString	207
Error	208
ErrArgument	209
ErrBadGuess	210
ErrBreak.....	211
ErrD_OP1_0	212
ErrD_OP1_LE_0	213
ErrD_OP1Not_R	214
ErrD_OP1NotPos.....	215
ErrD_OP1NotPosInt.....	216

Table of Contents (cont.)

ErrDataType	217
ErrDimension	218
ErrDimMismatch	219
ErrDivBy0.....	220
ErrDomain.....	221
ErrIncrement	222
ErrInvalid.....	223
ErrIterations	224
ErrLinkXmit	225
ErrMemory	226
ErrNon_Real	227
ErrNonReal	228
ErrNotEnoughMem	229
ErrOverflow	230
ErrSignChange	231
ErrSingularMat.....	232
ErrStat	233
ErrStatPlot	234
ErrSyntax.....	235
ErrToITooSmall	236
ErrUndefined.....	237
JError	238
JErrorNo	239
Floating Point Stack.....	240
AllocFPS	241
AllocFPS1	242
CpyStack	243
CpyO1ToFPST, CpyO1ToFPS1, CpyO1ToFPS2, CpyO1ToFPS3, CpyO1ToFPS4, CpyO1ToFPS5, CpyO1ToFPS6, CpyO1ToFPS7, CpyO2ToFPST, CpyO2ToFPS1, CpyO2ToFPS2, CpyO2ToFPS3, CpyO2ToFPS4, CpyO3ToFPST, CpyO3ToFPS1, CpyO3ToFPS2, CpyO5ToFPS1, CpyO5ToFPS3, CpyO6ToFPST, CpyO6ToFPS2....	244
CpyTo1FPST, CpyTo1FPS1, CpyTo1FPS2, CpyTo1FPS3, CpyTo1FPS4, CpyTo1FPS5, CpyTo1FPS6, CpyTo1FPS7, CpyTo1FPS8, CpyTo1FPS9, CpyTo1FPS10, CpyTo1FPS11, CpyTo2FPST, CpyTo2FPS1, CpyTo2FPS2, CpyTo2FPS3, CpyTo2FPS4, CpyTo2FPS5, CpyTo2FPS6, CpyTo2FPS7, CpyTo2FPS8, CpyTo3FPST, CpyTo3FPS1, CpyTo3FPS2, CpyTo4FPST, CpyTo5FPST, CpyTo6FPST, CpyTo6FPS2, CpyTo6FPS3	245
CpyToFPST	246

Table of Contents (cont.)

CpyToFPS1	247
CpyToFPS2	248
CpyToFPS3	249
CpyToStack	250
PopOP1, PopOP3, PopOP5	251
PopReal	252
PopRealO1, PopRealO2, PopRealO3, PopRealO4, PopRealO5, PopRealO6.....	253
PushOP1, PushOP3, PushOP5	254
PushReal	255
PushRealO1, PushRealO2, PushRealO3, PushRealO4, PushRealO5, PushRealO6.....	256
Graphing and Drawing	257
AllEq	259
BufClr.....	260
BufCpy	261
CircCmd.....	262
ClearRect.....	264
CLine	265
CLineS.....	267
ClrGraphRef.....	269
CPoint.....	270
CPointS.....	272
DarkLine	274
DarkPnt.....	276
Disp	278
DrawCirc2	279
DrawCmd.....	281
DrawRectBorder	282
DrawRectBorderClear	283
EraseRectBorder	284
FillRect.....	285
FillRectPattern	287
GrBufClr.....	289
GrBufCpy	290
GrphCirc	291
HorizCmd.....	292
IBounds.....	293
IBoundsFull	294

Table of Contents (cont.)

ILine	295
InvCmd	297
InvertRect	298
IOffset	299
IPoint	300
LineCmd	302
PDspGrph	304
PixelTest	305
PointCmd	306
PointOn	308
Regraph	309
SetAllPlots	310
SetFuncM	311
SetParM	312
SetPolM	313
SetSeqM	314
SetTblGraphDraw	315
TanLnF	316
UCLineS	317
UnLineCmd	318
VertCmd	319
VtoWHLDE	320
Xftol	321
Xitof	322
Yftol	323
ZmDecml	324
ZmFit	325
ZmInt	326
ZmPrev	327
ZmSquare	328
ZmStats	329
ZmTrig	330
ZmUsr	331
ZooDefault	332
Interrupt	333
DivHLBy10	334
DivHLByA	335

Table of Contents (cont.)

IO	336
AppGetCalc	337
AppGetCbl	338
Rec1stByte	339
Rec1stByteNC	340
RecAByteIO	341
SendAByte	342
Keyboard.....	343
ApdSetup	344
CanAlphIns	345
GetCSC	346
GetKey.....	349
List	350
AdrLEle	351
AdrMEle	352
ConvDim	353
ConvLcToLr	354
ConvLrToLc	355
DelListEl.....	356
Find_Parse_Formula.....	357
GetLToOP1.....	358
InclstSize	359
InsertList	361
PutToL	363
Math	364
AbsO1O2Cp	368
AbsO1PAbsO2.....	369
ACos	370
ACosH	371
ACosRad	372
Angle.....	373
ASin	374
ASinH.....	375
ASinRad.....	376
ATan	377
ATan2	378
ATan2Rad.....	379
ATanH.....	380
ATanRad.....	381

Table of Contents (cont.)

CAbs	382
CAdd.....	383
CDiv	384
CDivByReal.....	385
CEtoX	386
CFrac.....	387
CIntgr.....	388
CkInt	389
CkOdd.....	390
CkOP1C0.....	391
CkOP1Cplx	392
CkOP1FP0.....	393
CkOP1Pos	394
CkOP1Real.....	395
CkOP2FP0.....	396
CkOP2Pos	397
CkOP2Real.....	398
CkPosInt	399
CkValidNum	400
CLN	401
CLog	402
ClrLp	403
ClrOP1S.....	404
CMltByReal	405
CMult	406
Conj	407
COP1Set0.....	408
Cos	409
CosH.....	410
CpOP1OP2.....	411
CpOP4OP3.....	412
CRecip	413
CSqRoot	414
CSquare.....	415
CSub.....	416
CTenX.....	417
CTrunc.....	418
Cube	419
CXrootY	420

Table of Contents (cont.)

CYtoX	421
DecO1Exp	422
DToR	423
EToX.....	424
ExpToHex	425
Factorial	426
FPAdd.....	427
FPDiv	428
FPMult	429
FPRecip	430
FPSquare.....	431
FPSub.....	432
Frac	433
HLTimes9	434
HTimesL	435
Int.....	436
Intgr	437
InvOP1S	438
InvOP1SC.....	439
InvOP2S	440
InvSub.....	441
LnX	442
LogX	443
Max.....	444
Min.....	445
Minus1	446
OP1ExpToDec.....	447
OP1Set0, OP1Set1, OP1Set2, OP1Set3, OP1Set4, OP2Set0, OP2Set1, OP2Set2, OP2Set3, OP2Set4, OP2Set5, OP2Set60, OP3Set0, OP3Set1, OP3Set2, OP4Set0, OP4Set1, OP5Set0	448
OP2Set8	449
OP2SetA.....	450
Plus1	451
PtoR.....	452
RandInit	453
Random	454
RName.....	455
RndGuard	456
RnFx	457

Table of Contents (cont.)

Round	458
RToD	459
RToP.....	460
Sin	461
SinCosRad.....	462
SinH.....	463
SinHCosH	464
SqRoot.....	465
Tan	466
TanH.....	467
TenX	468
ThetaName	469
Times2	470
TimesPt5.....	471
TName	472
ToFrac	473
Trunc	474
XName.....	475
XRootY	476
YName.....	477
YToX.....	478
Zero16D.....	479
ZeroOP	480
ZeroOP1, ZeroOP2, ZeroOP3	481
Matrix	482
AdrMRow	483
GetMToOP1.....	484
PutToMat	485
Memory	486
Arc_Unarc.....	488
ChkFindSym	489
CleanAll	491
CloseProg	492
CmpSyms	493
Create0Equ.....	494
CreateAppVar	495
CreateCList.....	496
CreateCplx.....	497
CreateEqu.....	498

Table of Contents (cont.)

CreatePair.....	499
CreatePict.....	500
CreateProg	501
CreateProtProg	502
CreateReal.....	503
CreateRList.....	504
CreateRMat	505
CreateStrng	506
DataSize	507
DataSizeA.....	508
DeallocFPS	509
DeallocFPS1	510
DelMem	511
DelVar.....	513
DelVarArc	514
DelVarNoArc.....	515
EditProg	516
EnoughMem	517
Exch9.....	518
ExLp	519
FindAlphaDn	520
FindAlphaUp	522
FindApp	524
FindAppNumPages.....	525
FindAppDn.....	526
FindAppUp.....	527
FindSym.....	528
FixTempCnt	530
FlashToRam	531
InsertMem.....	532
LoadCIndPaged	534
LoadDEIndPaged.....	535
MemChk	536
PagedGet.....	537
RclGDB2.....	538
RclIN.....	539
RclVarSym.....	540
RclX	541
RclY	542

Table of Contents (cont.)

RedimMat	543
SetupPagedPtr.....	544
SrchVLstDn, SrchVLstUp.....	545
StMatEl	546
StoAns	547
StoGDB2.....	548
StoN.....	549
StoOther	550
StoR.....	552
StoSysTok	553
StoT	554
StoTheta	555
StoX.....	556
StoY	557
Parser	558
BinOPExec	559
FiveExec.....	561
FourExec	563
ParseInp	565
RclSysTok.....	567
ThreeExec	568
UnOPExec	570
Screen.....	572
ForceFullScreen.....	573
Statistics.....	574
DelRes	575
OneVar	576
Rcl_StatVar.....	577
Utility	578
AnsName	580
ConvDim00	581
CpHLDE.....	582
DisableApd.....	583
EnableApd	584
EOP1NotReal	585
Equ_or_NewEqu.....	586
GetBaseVer	587
GetTokLen	588
JForceCmdNoChar	589

Table of Contents (cont.)

JForceGraphKey	590
JForceGraphNoKey	591
MemClear	592
MemSet	593
Mov7B, Mov8B, Mov9B, Mov10B, Mov18B.....	594
Mov9OP1OP2.....	595
Mov9OP2Cp	596
Mov9ToOP1.....	597
Mov9ToOP2.....	598
MovFrOP1	599
OP1ExOP2, OP1ExOP3, OP1ExOP4, OP1ExOP5, OP1ExOP6, OP2ExOP4, OP2ExOP5, OP2ExOP6, OP5ExOP6	600
OP1ToOP2, OP1ToOP3, OP1ToOP4, OP1ToOP5, OP1ToOP6, OP2ToOP1, OP2ToOP3, OP2ToOP4, OP2ToOP5, OP2ToOP6, OP3ToOP1, OP3ToOP2, OP3ToOP4, OP3ToOP5, OP4ToOP1, OP4ToOP2, OP4ToOP3, OP4ToOP5, OP4ToOP6, OP5ToOP1, OP5ToOP2, OP5ToOP3, OP5ToOP4, OP5ToOP6, OP6ToOP1, OP6ToOP2, OP6ToOP5.....	601
PosNo0Int.....	602
RclAns	603
ReloadAppEntryVecs	604
SetXXOP1	605
SetXXOP2	606
SetXXXXOP2.....	607
StoRand.....	608
StrCopy.....	609
StrLength	610
Miscellaneous	611
ConvOP1	612

Appendix B — TI-83 Plus “Large” Character Fonts.....	613
---	------------

Appendix C — TI-83 Plus “Small” Character Fonts.....	620
---	------------

Reference List — System Routines.....	628
--	------------

Figures

Fig. 2.1: TI-83 Plus Architecture.....	3
Fig. 2.3: Z80 Memory Space.....	4
Fig. 2.2: TI-83 Plus RAM.....	5
Fig. 2.4: TI-83 Plus RAM Structure	5
Fig. 2.5: TI-83 Plus Flash ROM Structure	8
Fig. 2.6: Symbol Table Structure.....	51
Fig. 2.7: TI-83 Plus System RAM.....	56
Fig. 2.8: Calculator Scan Code	64
Fig. 2.9: Home Screen Display Mapping	71
Fig. 2.10: Pen Display Mapping	73
Fig. 2.11: Command Values.....	76
Fig. 2.12: Pixel Coordinates	83
Fig. 2.13: Graph WINDOW Setting	86
Fig. 2.14: Error Flow.....	99
Fig. 2.15: TI-83 Plus System RAM.....	109
Fig. 2.16: Control Flow	113
Fig. 2.17: Event Sequence.....	114
Fig. 2.18: Application Loader Process.....	116
Fig. 3.1: Application Development Flow	119

Tables

Table 2.1: System Flags	12
Table 2.2: OP Registers	16
Table 2.3: Transfer one OP register to another (11 byte operation)	16
Table 2.4: Exchange one OP register with another (11 byte operation)	17
Table 2.5: Load a floating-point value into an OP register (9 byte operation)	17
Table 2.6: Miscellaneous floating-point utility routines in OP registers	17
Table 2.7: Set an OP register to all zeros (11 byte operation)	17
Table 2.8: Variable Name, RAM Equate, and SysTok Value	20
Table 2.9: Floating-Point Number Format	21
Table 2.10: Variable Name Format	26
Table 2.11: Format of Archive Stored Variables	44
Table 2.12: Format of Archive Stored Variables	47
Table 2.13: Program, AppVar, Group	52
Table 2.14: Lists	52
Table 2.15: Real, Cplx, Mat, Equ, GDB, Pict	52
Table 2.16: Formula Example	54
Table 2.17: Floating-Point Basic Math Functions	100
Table 2.18: Trigonometric and Hyperbolic Functions	101
Table 2.19: Floating-Point Power and Logarithmic Math Functions	101
Table 2.20: Floating-Point Miscellaneous Math Functions	102
Table 2.21: Complex Math Basic Math Functions	103
Table 2.22: Complex Math Power and Logarithmic Math Functions	104
Table 2.23: Complex Math Miscellaneous Math Functions	104
Table 2.24: Temporary Variables Example	108
Table 2.25: Language Table	112

1

Introduction

TI-83 PLUS DEVELOPER GUIDE

This guide contains information necessary to develop applications for the TI-83 Plus calculator. It addresses basic environmental specifics and development guidelines. This guide covers TI-83 Plus calculator specific information, processes, and development tools.

The TI-83 Plus Developer Guide is one of a set of documents supporting the TI-83 Plus calculator. The set includes:

- *TI-83 Plus Graphing Calculator Guidebook* — Describes how to use the calculator (provided with the TI-83 Plus calculator).
- *TI-83 Plus Tutorial* — Provides examples that introduce the developer to application creation.
- *TI-83 Plus User Interface Guide* — Provides information on the design and construction of the user interface.

To access these guides visit our web site.

Conventions Used in this Guide

The following conventions were adopted for this guide to help make the material easier to read.

Program text: All of the program examples are in a non-proportional font that can be distinguished from the guide text.

```
LD          HL,L1name
B_CALL      Mov9ToOP1      ; OP1 = list L1 name
;
B_CALL      FindSym        ; look up list variable in OP1
```

Syntax: Program instructions (commands and directives) are in all upper case letters.

Example:

```
B_CALL      routine
```

Optional parameters: These parameters are enclosed in square brackets. Part of a program instruction may be in italics to describe the type of information.

Example:

```
[label][:]  operation  [operands]  [; comment]
```

Program layout: The program statements appear in columns.

Example:

```
ThisIsALabel:
    LD      A,5
    B_CALL  SystemRoutine    ; call to a system routine
    DEC     A
    JR      NZ,ThisIsALabel
    RET
```

Purpose of this Guide

The types of programs that can be created on the TI-83 Plus calculator include RAM-based TI-BASIC programs, RAM-based assembly programs, and Flash ROM-based applications. This guide addresses Flash ROM-based application development and RAM-based assembly programs.

Structure of this Guide

- Chapter 2 provides an in-depth view of the TI-83 Plus physical and logical memory structures, and the various drivers, tools, and utilities available to the developer.
- Chapter 3 presents several processes including the application development process, the signature process, the testing process, and the release/distribution process.
- Chapter 4 provides a view of the various development tools.

2

TI-83 Plus Specific Information

ARCHITECTURE

Fig. 2.1 represents the TI-83 Plus architecture, which is composed of several layers and elements.

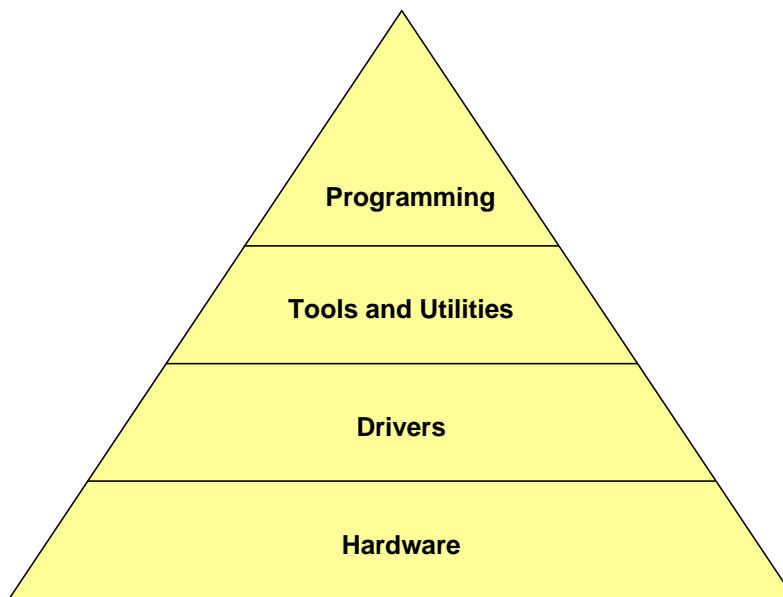


Fig. 2.1: TI-83 Plus Architecture

The **Hardware** layer contains the functional components of the unit — the Z80 processor, Random Access Memory (RAM), Flash ROM (also called Flash), Read Only Memory (ROM), and TI-BASIC (not included in this guide).

The **Drivers** layer contains assembly language-controlled functions such as the keypad, battery, display, and input/output.

The **Tools and Utilities** layer contains the elements that provide text, drawing tools, and utility routines.

The **Programming** layer contains the user interface — screen, keyboard, and the basic unit functionality. In addition, it provides the capability to load TI-BASIC programs (keystroke), assembly programs that execute in RAM, and application programs that execute in Flash ROM.

This chapter explains the Hardware layer, Drivers layer, and Tools and Utilities layer. Chapter 3 explains the Programming layer.

HARDWARE LAYER

Loading and debugging an application requires a general understanding of the memory layout of the calculator.

Other manuals and guides cover TI-83 Plus operation including keys, screens, menus, etc. This discussion covers the TI-83 Plus internal hardware components — Zilog Z80™ CPU, RAM, and Flash ROM.

Z80 CPU and Memory

The TI-83 Plus uses a Z80 processor with a 64K byte logical address space. To provide more than 64K bytes of physical RAM, this logical memory space is divided into four 16K byte pages (see Fig. 2.3). Physical memory is also divided into two 16K byte pages (see Fig. 2.3), and a physical page is mapped into each logical page as it is needed.

There are two types of physical memory in the calculator — Z80 RAM and Flash ROM. The following sections address the composition, structure, and uses of these memory types.

- Z80 Logical Memory Space

The Z80 logical memory size is 64K bytes, which is divided into four 16K byte pages — 0000h to 3FFFh, 4000h to 7FFFh, 8000h to BFFFh, and C000h to FFFFh. A physical memory page is mapped into each logical page.

0000h		
4000h	16K Always Flash ROM Page 0	3FFFh
8000h	16K RAM Page 0,1 or Flash ROM Pages 0-31	7FFFh
C000h	16K RAM Page 0,1 or Flash ROM Pages 0-31	BFFFh
	16K Always RAM Page 0	FFFFh

Fig. 2.3: Z80 Memory Space

The 16K byte address space from 0000h to 3FFFh is ROM page 0 from the Flash ROM. It is always present.

The 16K byte address space from 4000h to 7FFFh is used for swapping a 16K byte ROM page from the Flash ROM. This allows the TI-83 Plus system to extend beyond its 64K byte physical addressing capabilities.

- Z80 Physical RAM Structure

TI-83 Plus physical RAM consists of 32K bytes starting at address 8000h.

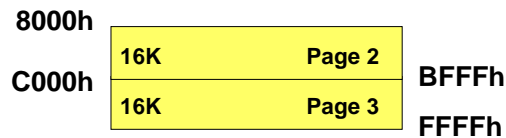


Fig. 2.2: TI-83 Plus RAM

Z80 RAM Structure

The TI-83 Plus has 32K bytes of RAM. The system code partitions the RAM into a number of areas, which it uses to maintain different types of information. Applications that need RAM must reuse some of the RAM not currently in use by the system code. They must request an allocation from the system code User RAM area. Fig. 2.4 shows how RAM is partitioned.

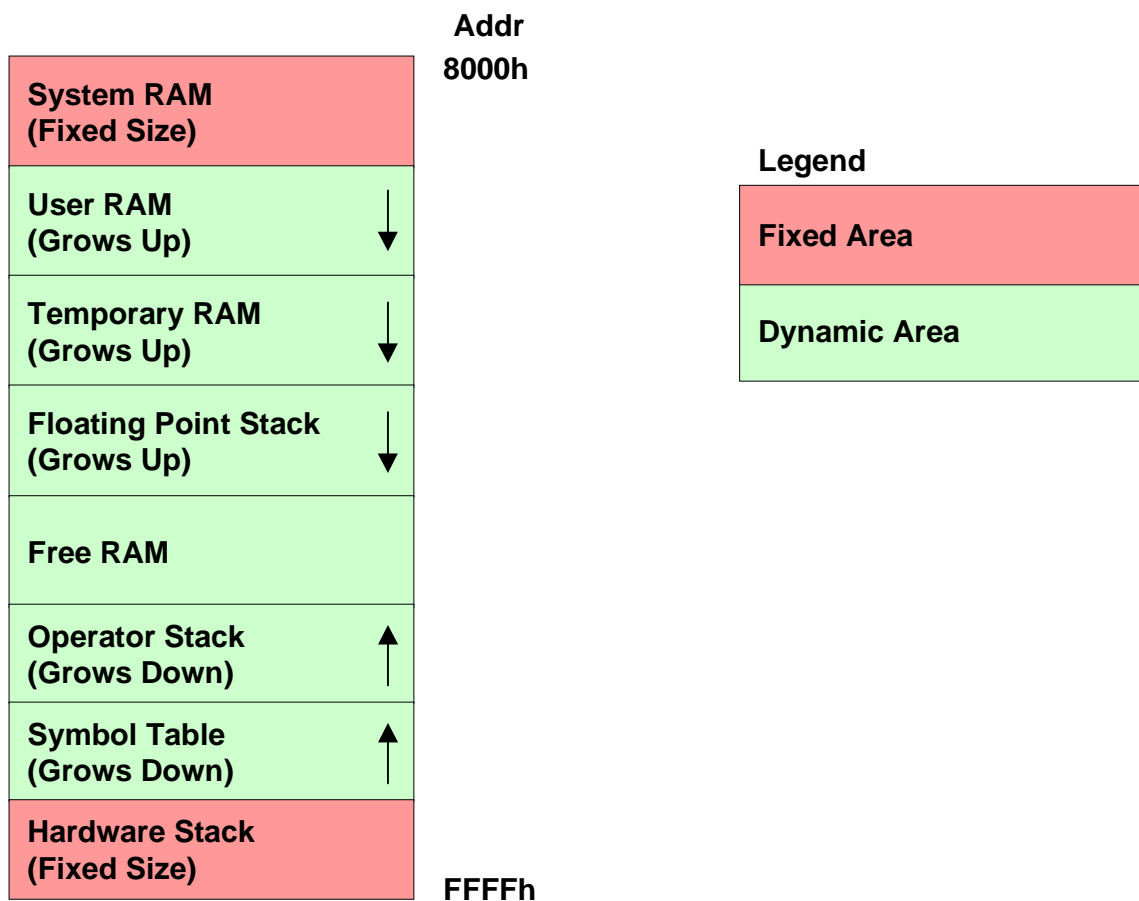


Fig. 2.4: TI-83 Plus RAM Structure

Fig. 2.4 shows the addresses of Z80 logical address space. RAM is always mapped into the 32K space beginning at logical address from 8000h to FFFFh. The areas (System RAM and Hardware Stack) at each end of RAM are fixed size. All other areas are dynamic. The positions of the areas in RAM with respect to each other never changes and never overlaps; however, their sizes grow and shrink and boundaries move as the calculator operates. The area labeled Free RAM is a leftover area. As the other areas grow, they push into the Free RAM area making it smaller. As the other areas shrink, the Free RAM area gets larger.

Following is a brief overview of each of these areas in RAM.

System RAM

This area contains system preallocated RAM structures.

- System Flags (Modes, Indicators)
- System Variables (for example, Xmin, Ymin...)
- OP1 through OP6 RAM Registers
- Memory Pointers
- Safe RAM Locations for Applications Use
- State Monitor Control RAM
- Graph Backup Screen — bit image
- Utility Backup Screens (two) — bit image
- Text Backup Screen

User RAM

Variables created by the calculator user are stored in User RAM. Each variable stored in User RAM has a Symbol Table entry associated with it.

Temporary RAM

This area is used during equation parsing and execution. It contains the data for the temporary variables that are created during parser execution. Some applications may need to perform *housekeeping* of this area if they invoke the equation parser and if temporary variables are returned as a result.

Floating Point Stack

This area is used during equation parsing and execution. It provides temporary storage outside the User RAM area.

Free RAM

This is the RAM that is currently not in use. The arrows in Fig. 2.4 show that the structures below and above Free RAM grow toward it.

Note: Applications should never use this area. Information about which RAM areas are available for applications will be provided, as well as how to create variables for long-term storage of data.

Operator Stack

This area of RAM is used by the system code for math expression evaluation and equation parsing (execution). No detailed description of this RAM area is provided since applications do not use the Operator Stack.

Symbol Table

This area of RAM is used to keep track of all of the variables, resident in both RAM and Flash ROM. The names, data types, pointers to the data, and where the variables reside in RAM or in Flash ROM (archived) are stored in the Symbol Table.

Hardware Stack

This is the area to which the Z80 Stack Pointers (SP) register points. This stack area is 400 bytes. The Hardware Stack starts at address FFFFh and it grows from high to low memory.

There are no safeguards against overflowing the stack and corrupting other RAM areas. The amount of space allocated for the stack should be sufficient for applications needs. Applications should avoid the use of recursive routines that can easily and quickly overflow the Hardware Stack. The Hardware Stack should not be used for saving large amounts of data. Using the Hardware Stack to save register values upon entry to routines should not cause problems.

None of the TI-83 Plus system routines use recursion that will overflow the Hardware Stack.

The TI-83 Plus Flash ROM is composed of 512K bytes divided into 32 pages, each of which is 16K bytes in size. Fig. 2.5 represents the Flash ROM structure.

Fig. 2.5: TI-83 Plus Flash ROM Structure

The explanations of some Flash ROM areas below are for informational purposes only.

Boot (Code) Area

This area contains the following unalterable items.

- Boot-strap code
- System initialization code
- Software validation routine
- Program download routine
- Software product ID
- Product code update loader

Certification Area

This area contains program authentication information.

- Calculator serial number
- Unit certificate public key
- Date-stamp public key
- Date-stamp certificate
- Unit certificate and license status
- Group certificates

Operating System (OS) Area

This area contains the operating system of the calculator — math, display, keyboard, I/O, etc. routines.

Certificate List Area

This area contains a list of unit certificates for the specific calculator.

User APPS (Calculator Software Applications)/Data Area

This area (160K bytes of available space) is shared by applications and variables archived by the user for long-term storage.

Swap Area/User APPS/Data Area

This area is dynamically allocated for use by the system as needed in the space indicated in Fig. 2.5.

System Development Environment

All TI-83 Plus applications are developed in Z80 assembly language. Chapter 3 contains more specific information and examples. This section provides in-depth information about the use of System RAM, User RAM, Floating Point Stack, etc. (see Fig. 2.4).

System Routines

Entry points for a set of TI-83 Plus system routines are provided in Appendix A. A list of entry point equated labels is provided in the file, TI83plus.inc. Later in this chapter, source code examples are included with detailed explanations of how to access system routines.

To access these system routines use the Z80 RST instruction. Two macro-instructions (macro) are provided for simplification. Each of these macros uses three bytes of code space.

If your assembler does not support macro calls, substitute:

```

B_CALL      label
with
RST         rBR_CALL
DW          label

B_JUMP      label
with
CALL        BRT_JUMP0
DW          label

```

The following section is a detailed explanation of the various RAM areas shown in Fig. 2.4.

RST Routines

The Z80 reset instruction, RST, can be used in place of B_CALL for some entry points. Using the RST instruction only takes one byte of ROM space as opposed to three bytes for a B_CALL. There are five routines set to use this method of access. These were chosen because of high-frequency use in the operating system.

- RST rMov9ToOP1 used instead of B_CALL Mov9ToOP1
- RST rFindSym used instead of B_CALL FindSym
- RST rPushRealO1 used instead of B_CALL PushRealO1
- RST rOP1ToOP2 used instead of B_CALL OP1ToOP2
- RST rFPAdd used instead of B_CALL FPAdd

Details on these routines can be found in this chapter or in Appendix A.

System RAM Areas

The details about system RAM follow.

System Flags

This area of RAM is used for bit flags. The TI-83 Plus accesses these flags through the Z80's IY register. The IY register is set to the start of this flag area and does not change, resulting in easy bit manipulation.

Example:

```
SET          trigDeg,(IY+trigFlags)    ; set to degree angle mode
```

trigFlags is the byte offset from the start of the flag area.

Some system flags that an application might use are listed in Table 2.1, along with information needed to support basic ASM programming on the TI-83 Plus.

The values for these symbols are located in the include file, TI83plus.inc.

Flag Name	IY Offset	Equate Description	Comments
trigDeg	trigFlags	0 = radian angle mode 1 = degree angle mode	
plotLoc	plotFlags	0 = write to display and buffer 1 = write to display only	Determines whether the graph line and point routines draw to the display or to the graph backup buffer, plotSScreen .
plotDisp	plotFlags	0 = graph screen not in display 1 = graph in display	
grfFuncM	grfModeFlags	1 = function graph mode	
grfPolarM	grfModeFlags	1 = polar graph mode	
grfParamM	grfModeFlags	1 = parametric graph mode	
grfRecurM	grfModeFlags	1 = sequence graph mode	
graphDraw	graphFlags	0 = graph is up to date 1 = graph needs to be updated	
grfDot	grfDBFlags	0 = graph connected draw mode 1 = graph dot draw mode	
grfSimul	grfDBFlags	0 = sequential graph draw mode 1 = simultaneous graph draw mode	
grfGrid	grfDBFlags	0 = graph mode grid off 1 = graph mode grid on	
grfPolar	grfDBFlags	0 = graph — rectangular coordinates 1 = graph — polar coordinates	
grfNoCoord	grfDBFlags	0 = graph coordinates off 1 = graph coordinates on	
grfNoAxis	grfDBFlags	0 = graph draw axis 1 = graph no axis	
grfLabel	grfDBFlags	0 = graph labels off 1 = graph labels on	
textEraseBelow	textFlags	1 = erase line below small font when writing small font	Deals with displaying small variable font characters, when set the pixels below the character displayed are cleared. See routines VPutMap and VPutS .
textInverse	textFlags	1 = write in reverse video	Affects both the normal 5×7 font and the small variable width font.

Table 2.1: System Flags

Flag Name	IY Offset	Equate Description	Comments
onInterrupt	onFlags	1 = [ON] key interrupt occurred	The [ON] key is interrupt driven, but it does not automatically stop execution. Flag is set by the interrupt handler when the [ON] key is pressed. An application must poll (test) this flag to implement the [ON] key press as a <i>break</i> .
statsValid	statFlags	1 = stat results are valid	
fmtExponent	fmtFlags	1 = scientific display mode	Resetting signifies NORMAL mode setting.
fmtEng	fmtFlags	1 = engineering display mode	Resetting signifies NORMAL mode setting.
fmtReal	fmtFlags	1 = real math mode	See Comment 1 below.
fmtRect	fmtFlags	1 = rect complex math mode	See Comment 1 below.
fmtPolar	fmtFlags	1 = polar complex math mode	See Comment 1 below.
curAble	curFlags	1 = cursor flash enabled	
curOn	curFlags	1 = cursor is showing	
curLock	curFlags	1 = cursor is locked off	
appTextSave	appFlags	1 = save characters written in textShadow	Places a copy of the character, normal font only, written to the display into the textShadow buffer.
appAutoScroll	appFlags	1 = auto-scroll text on last line	Causes the screen to automatically scroll when the normal font is written to the display and goes beyond the last row of the screen.
indicRun	indicFlags	1 = run indicator is enabled 0 = run indicator is disabled	Controls the run indicator that is displayed in the upper right corner of the display. See Run Indicator section.
comFailed	getSendFlg	1 = com failed 0 = com did not fail	
apdRunning	apdFlags	1 = APD™ is running 0 = APD™ is not running	

Table 2.1: System Flags (continued)

Comment 1: Controls the mode setting: REAL a + bi re^{∠θi} located on the mode screen.

Flag Name	IY Offset	Equate Description	Comments
indicOnly	indicFlags	1 = only update run indicator	Sets the interrupt handler to update the run indicator, but not to process APD, blink the cursor, or scan for keys. It is useful when executing I/O link port operations for speed.
shift2nd	shiftFlags	1 = second key pressed	
shiftAlpha	shiftFlags	1 = alpha mode	
shifLwrAlpha	shiftFlags	1 = lower case, shift alpha set also	
shiftALock	shiftFlags	1 = alpha lock, shift alpha set also	
grfSplit	sGrFlags	1 = horizontal graph split mode	
vertSplit	sGrFlags	1 = vertical graph split mode	
textWrite	sGrFlags	1 = small font writes to buffer 0 = small font writes to display	Use when writing small font characters. Determines if the character will be written to the display or to the corresponding location in the graph backup buffer, plotSScreen . Useful for building a screen in RAM and then displaying it in its entirety at once.
fullScrnDraw	apiFlag4	1 = allows draws to use column 95 and row 0	
bufferOnly	plotFlag3	1 = draw to graph buffer only	Causes all of the graph line and point routines (pixel coordinates as inputs) to be drawn to the graph backup buffer instead of to the display.
fracDrawLFont	fontFlags	1 = draw large font in UserPutMap	Enables the normal font to be drawn using the small font coordinate system. See section on Display in Appendix A.
customFont	fontFlags	1 = draw custom characters	Allows an application to have the small font routines display a font defined by an application. See section on Display in Appendix A.
lwrCaseActive	appLwrCaseFlag	1 = enable lower case in GetKey loop	Causes the GetKey routine to recognize lower case alpha key presses. When set, the key sequence [ALPHA] [ALPHA] causes lower case alpha mode to be set.

Table 2.1: System Flags (continued)

Flag Name	IY Offset	Equate Description	Comments
asm_Flag1_0	asm_Flag1	available for ASM programming	See Comment 2 below.
asm_Flag1_1	asm_Flag1	available for ASM programming	See Comment 2 below.
asm_Flag1_2	asm_Flag1	available for ASM programming	See Comment 2 below.
asm_Flag1_3	asm_Flag1	available for ASM programming	See Comment 2 below.
asm_Flag1_4	asm_Flag1	available for ASM programming	See Comment 2 below.
asm_Flag1_5	asm_Flag1	available for ASM programming	See Comment 2 below.
asm_Flag1_6	asm_Flag1	available for ASM programming	See Comment 2 below.
asm_Flag1_7	asm_Flag1	available for ASM programming	See Comment 2 below.
asm_Flag2_0	asm_Flag2	available for ASM programming	See Comment 2 below.
asm_Flag2_1	asm_Flag2	available for ASM programming	See Comment 2 below.
asm_Flag2_2	asm_Flag2	available for ASM programming	See Comment 2 below.
asm_Flag2_3	asm_Flag2	available for ASM programming	See Comment 2 below.
asm_Flag2_4	asm_Flag2	available for ASM programming	See Comment 2 below.
asm_Flag2_5	asm_Flag2	available for ASM programming	See Comment 2 below.
asm_Flag2_6	asm_Flag2	available for ASM programming	See Comment 2 below.
asm_Flag2_7	asm_Flag2	available for ASM programming	See Comment 2 below.
asm_Flag3_0	asm_Flag3	available for ASM programming	See Comment 2 below.
asm_Flag3_1	asm_Flag3	available for ASM programming	See Comment 2 below.
asm_Flag3_2	asm_Flag3	available for ASM programming	See Comment 2 below.
asm_Flag3_3	asm_Flag3	available for ASM programming	See Comment 2 below.
asm_Flag3_4	asm_Flag3	available for ASM programming	See Comment 2 below.
asm_Flag3_5	asm_Flag3	available for ASM programming	See Comment 2 below.
asm_Flag3_6	asm_Flag3	available for ASM programming	See Comment 2 below.
asm_Flag3_7	asm_Flag3	available for ASM programming	See Comment 2 below.

Table 2.1: System Flags (continued)

Comment 2: Used by applications to provide easy bit flag implementation. Once an application completes, flag will most likely be changed by another application. It will not hold its state.

OP1 through OP6 RAM Registers

This area of RAM is used extensively by the TI-83 Plus system routines for such things as:

- Executing floating-point math
- Passing arguments to and from system routines
- Extracting elements out of lists or matrices
- Executing the parser
- Formatting numbers for display

There are six OP registers allocated — OP1, OP2, OP3, OP4, OP5, and OP6. Each of these labels are equated in the include file, TI83plus.inc.

Each of these OP registers is 11 bytes in length; they are allocated in contiguous RAM.

OP1	11 bytes
OP2	11 bytes
OP3	11 bytes
OP4	11 bytes
OP5	11 bytes
OP6	11 bytes

Table 2.2: OP Registers

The size of these registers was determined by the size of the TI-83 Plus floating-point number format and by the maximum size (nine bytes) of a variable name. The 10th and 11th bytes in each register are used by the floating-point math routines for extra precision.

Below are the Utility routines that manipulate the OP registers. See Appendix A for details.

OP1ToOP2	OP2ToOP1	OP3ToOP1	OP4ToOP1	OP5ToOP1	OP6ToOP1
OP1ToOP3	OP2ToOP3	OP3ToOP2	OP4ToOP2	OP5ToOP2	OP6ToOP2
OP1ToOP4	OP2ToOP4	OP3ToOP4	OP4ToOP3	OP5ToOP3	OP6ToOP5
OP1ToOP5	OP2ToOP5	OP3ToOP5	OP4ToOP5	OP5ToOP4	
OP1ToOP6	OP2ToOP6		OP4ToOP6	OP5ToOP6	

Table 2.3: Transfer one OP register to another (11 byte operation)

OP1ExOP2	OP1ExOP3	OP1ExOP4	OP1ExOP5	OP1ExOP6
OP2ExOP4	OP2ExOP5	OP2ExOP6	OP5ExOP6	

Table 2.4: Exchange one OP register with another (11 byte operation)

OP1Set0	OP1Set4	OP2Set3	OP2Set8	OP3Set2
OP1Set1	OP2Set0	OP2Set4	OP2SetA	OP4Set0
OP1Set2	OP2Set1	OP2Set5	OP3Set0	OP4Set1
OP1Set3	OP2Set2	OP2Set60	OP3Set1	OP5Set0
SetXXOP1	SetXXOP2	SetXXXOP2		

Table 2.5: Load a floating-point value into an OP register (9 byte operation)

CkInt	CkOdd	CkOP1FPO	CkOP1Pos	CkOP1Real
CkOP2FPO	CkOP2Pos	CkOP2Real	ClrOP1S	ClrOP2S
InvOP1S	InvOP2S	CpOP1OP2	ConvOP1	

Table 2.6: Miscellaneous floating-point utility routines in OP registers

ZeroOP1	ZeroOP2	ZeroOP3	ZeroOP
---------	---------	---------	--------

Table 2.7: Set an OP register to all zeros (11 byte operation)

The OP registers are also used as inputs and outputs for floating-point and complex number math. See Floating Point and Complex Math sections.

Safe RAM Locations for Application Use

If the amount of RAM an application needs is not too great, use safe pieces of RAM that exist in the System RAM area. These are chunks of RAM that are not used by system routines except under rare circumstances. They are, therefore, available as scratch RAM for the application.

saveSScreen (86ECh)

This is 768 bytes used by the system code only if the calculator automatically powers down (APD). This RAM is safe to use as long as an APD cannot occur. See the Keyboard and Automatic Power Down™ (APD™) sections.

statVars (8A3Ah)	This is the start of 531 bytes of RAM used to store statistical results. If you use this area, do not compute statistics in your ASM program. Make this B_CALL to invalidate statistics, as well. B_CALL DelRes
appBackUpScreen (9872h)	This is the start of 768 bytes of RAM not used by the system. It is intended for ASM and applications. Its size is large enough to hold a bit image of the display, but it can be used for whatever you want.
tempSwapArea (82A5h)	This is the start of 323 bytes used only during Flash ROM loading. If this area is used, avoid archiving variables.

WARNING: The RAM is safe to use only until the application exits. Data in any of these areas of RAM may be destroyed between successive executions of an application. Therefore, any data that must remain between executions cannot be kept in these areas. This RAM is only for the variables that can be discarded when the application exits.

System Variables Area

This area of system RAM consists of preallocated variables needed by much of the TI-83 Plus built-in functionality. Because they are floating-point numbers these variables are all nine bytes. Because these variables are always needed, the system always keeps them around and never changes their addresses.

There are two classes of system variables — those that you can store to and recall from, and those that are referred to as output only variables because the system routines can store to them.

System Variables that are Both Input and Output

In general, these values should only be changed by system routines that applications can call. Modifying these variables directly, rather than modifying them through the appropriate system routine, could corrupt the state of the system. Most of these system variables have restrictions on what values are valid to store to them. Using the system routine to store to them guarantees that the proper checks are made on the values being stored to them.

System Variable Characteristics

- There are no Symbol Table entries for system variables.
- These variables can be changed by the user, but cannot be deleted or renamed. For example, you can change Xmax, but you cannot delete it.
- These variables are initialized to a predetermined value upon reset.
- These variables always reside in RAM. For example, it is not possible to archive Xmin.

Storing and Recalling System Variable Values

Since system variables are located at a fixed location in RAM, an application can access the contents of a system variable directly. This method is safe only when recalling a single system variable.

There is also a system routine that copies the contents of a system variable to OP1; the value in the accumulator determines what system variable is recalled. See **SysTok** values in Table 2.8.

RclSysTok Copies the contents of a system variable to OP1.

StoSysTok Stores the contents of OP1, if valid, to a system variable.

Note: An application should not modify the contents of a system variable directly; it should always use this system routine.

The system variable stored to is determined by the value in the accumulator.

Example: If you want to store -3 in Xmin:

```

B_CALL    OP1Set3      ; Reg OP1 = Floating point 3
B_CALL    InvOP1S      ; Negate FP number in OP1, OP1 = -3
LD         A,XMINt     ; ACC = Xmin variable token value
B_CALL    StoSysTok    ; store OP1 to Xmin,
```

Example: If you want to recall the contents of Xmin to OP1:

```

LD         A,XMINt
B_CALL    RclSysTok    ; OP1 = contents of Xmin, -3
```


Table 2.8 lists each system variable, its RAM address equate, and the token values used to access them with the routines above.

Variable Name	RAM Equate	SysTok Value
Xscl	Xscl	XSCLt
Yscl	Yscl	YSCLt
Xmin	Xmin	XMINt
Xmax	Xmax	XMAXt
Ymin	Ymin	YMINt
Ymax	Ymax	YMAXt
tMin	tMin	TMINt
tMax	tMax	TMAXt
θmin	ThetaMin	THETMINt
θmax	ThetaMax	THETMAXt
PlotStart	PlotStart	PLOTSTARTt
nMin	nMin	NMINt
nMax	nMax	NMAXt
deltaTbl	TblStep	TBLSTEPt
Tstep	Tstep	TSTEPt
θstep	ThetaStep	THETSTEPt
deltaX	deltaX	DELTAXt
deltaY	deltaY	DELTAYt
XFact	XFact	XFACTt
YFact	YFact	YFACTt
Xres	XresO	XRES t
PlotStep	PlotStep	PLOTSTEPt
N (TVM)	fin_N	FINNt
I%	fin_I	FINIt
PV	fin_PV	FINPVt
PMT	fin_PMT	FINPMTt
FV	fin_FV	FINFVt
C/Y	fin_CY	FINCYt
P/Y	fin_PY	FINPYt

Table 2.8: Variable Name, RAM Equate, and SysTok Value

System Variables that Are Output Only

These are the statistical output variables. They are stored to after executing either the 1-varstat, 2-varstat, or a regression command. The TI-83 Plus system considers these variables invalid if no statistical command was executed; therefore, values are not stored to them.

Recall these values using the following system routine.

Rcl_StatVar Recalls a statistical result into OP1, if statistics are valid. The accumulator contains a token value of the statistical variable to recall.

The token values are contained in the include file, TI83plus.inc.

User RAM

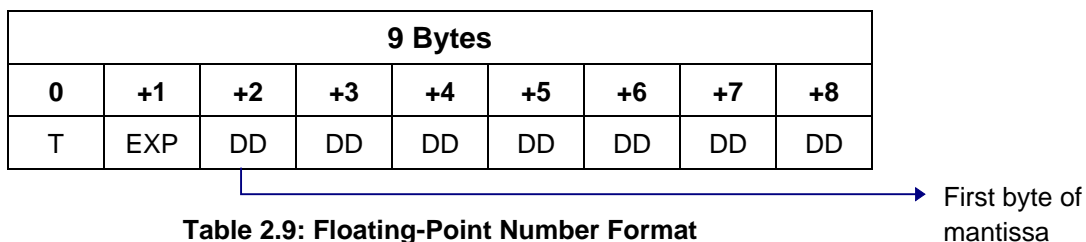
User RAM (see Fig. 2.4) is used to store the data structures of variables that are dynamically created. These variables are created by both users and the TI-83 Plus system.

The following sections contain an overall description of TI-83 Plus variable naming conventions, data structures, creation, and accessing.

Variable Data Structures

Numeric Based Data Types

This class of data types is built of floating-point numbers, and in some cases, a size field. These data types include Real, Complex, Real List, Complex List, and Matrix.



T = object type where:

<u>Bit</u>	<u>Description</u>
0 – 4	0 if a real variable's data, 0Ch if part of a complex variable's data
5 – 6	Future use
7	Mantissa sign — 0 = positive/1 = negative
EXP = 00h to FFh	80h to FFh = Exponent of (0) to (128) 7Fh to 00h = Exponent of (-1) to (-127)
DD	two digits of the mantissa, two per byte

A floating-point number has a left-justified mantissa (the most significant digit is always the first digit). If the MSD is 0, the TI-83 Plus system assumes it is floating-point 0. A floating-point number has a 14-digit mantissa and an exponent range of -128 to 127. For example:

T EXP Mantissa

80 82 23 45 00 00 00 00 00 = -234.5

Real Data Type Structure

This data type structure is simply a floating-point number with bits 0 – 4 of its sign byte = 0. For example:

80 82 23 45 00 00 00 00 00 = -234.5

Complex Data Type Structure

Complex numbers stored in a variable are two consecutive floating-point numbers, with the first value being the real part and the second value being the imaginary part. Each part of the complex number has bits 0 – 4 of its sign byte = 0Ch, the complex object value. For example:

8C 82 23 45 00 00 00 00 00

0C 7F 25 00 00 00 00 00 00 = -234.5 + 0.25i

Note: When complex numbers are handled in the OP1 to OP6 areas, the real and imaginary parts are not in consecutive RAM locations. They are, however, in consecutive OP registers.

Real List Data Type Structure

This data type consists of a two-byte size field with the number of elements in the list, followed by a real number for each element in the list. The maximum number of elements is 999. For example, a Real List with two elements, -234.5 and 230 would look like:

size		element number 1			element number 2
02 00		80 82 23 45 00 00 00 00 00			00 82 23 00 00 00 00 00 00

The size bytes are stored with the least significant byte first.

Complex List Data Type Structure

This data type consists of a two byte-size field with the number of elements in the list, followed by a complex number for each element in the list. The maximum number of elements is 999. For example, a complex list with two elements (1,2) and (4,5):

size		element number 1 — real part		element number 1 — imaginary part
02 00		0C 80 10 00 00 00 00 00 00 00		0C 80 20 00 00 00 00 00 00 00
		element number 2 — real part		element number 2 — imaginary part
		0C 80 40 00 00 00 00 00 00 00		0C 80 50 00 00 00 00 00 00 00

Matrix Data Type Structure

This data type consists of a two byte-size field with the number of columns and rows in the matrix, followed by a real number for each matrix element.

Matrices are stored in row major order, that is, each element of a given row is stored in contiguous RAM. For example, given the following structure:

size bytes		row 1
		.
		.
		.
		row 2
		.
		.
		.
		row 3
		.
		.
		.
matrix		
size		element
03 02		00 80 10 00 00 00 00 00 00 00 element (1,1)
		00 80 20 00 00 00 00 00 00 00 element (1,2)
		00 80 30 00 00 00 00 00 00 00 element (1,3)
		00 80 40 00 00 00 00 00 00 00 element (2,1)
		00 80 50 00 00 00 00 00 00 00 element (2,2)
		00 80 60 00 00 00 00 00 00 00 element (2,3)

A row or column dimension cannot be 0, and it cannot be greater than 99. If an application creates a matrix with either of these illegal dimensions, the TI-83 Plus system may lock up.

Token Based Data Types

This class of data types is made up of a size field and tokens that represent TI-83 Plus functions, commands, programming instructions, variable names — essentially anything that can be entered into an TI-83 Plus BASIC program.

TI-83 Plus Tokens

A token can be comprised of one or two bytes which represents system functions, commands, and variables. Instead of having to store the entire spelling of a function inside a program, the function can be stored as a token that uses only one or two bytes. For most applications, the tokens are only necessary when using variables. This will be explained in the section on Variable Naming.

A list of tokens and their values can be found in the include file, TI83plus.inc.

Program, Protected Program, Equation, New Equation, and String Data Type Structures

All of these data types have the same storage structure — a two-byte size field, the number of bytes for token storage (not the number of tokens), followed by the tokens themselves. For example, if graph equation $Y1 = \text{LCM}(X,5)$, it would be stored as:

Size byte	Two-byte token					
	LCM	(X	,	5)
07 00	BB 08	10	58	2B	35	11

Note: New Equation type should be treated like any other equation.

Screen Image Data Type Structure

There is only one data type for this class of data structures — the Pict data type.

This variable's data is a bit image of a graphic screen minus the bottom row of pixels. It is made up of a two-byte size field, which is always equal to 756d (2F4h) and followed by the 756 bytes. The first byte represents the first eight pixels of the display's top pixel row. Each successive byte represents the next eight pixels. When the end of a row is reached, the next byte is the first eight pixels of the following row.

Example:

size | First 12 bytes is the top row of pixels

F4 02 12 34 56 78 09 23 45 98 A3 CB DE 12

70 65 34 98 56 77 09 06 80 C5 4D 00

Second row of pixels

.
.
.

Graph Database Data Type Structure

There is only one data type for this class of data structures — the GDB data type.

The variable data is a collection of graph equations, window variables, and mode flags that have been saved.

Unformatted AppVar Data Type Structure

This data type was created solely for use by applications. It allows you to save and restore a *state* after an application is exited and then re-entered by users.

Since you can put almost anything into an AppVar, the system does not know the format of these variables. The system only shows the amount of memory taken by AppVars. It also allows them to be deleted and to be sent/received through the link port.

The system code does not modify or destroy this memory between one execution of an app and the next.

Users cannot access the contents of an AppVar, but they can delete, archive, and send the contents over the link port to another TI-83 Plus or the TI-83 Plus GRAPH LINK™.

Guidelines for AppVar Usage

- To avoid conflicts with other application's AppVars, use unique names that tie an AppVar to the application.
- To verify that an application is using an AppVar that is intended for that application, an expected value for the first four bytes of the AppVar should be written when it is created and checked before it is used.

For example, my application uses AppVars to save some information about different users who have run the application at sometime. When the application is started it will search for all of the AppVars that represent users of the application, and ask the user to choose their AppVar from a list. The application will know which AppVars to display by looking at the first four bytes of the AppVar for a certain set of values. The AppVars that contain the correct first four bytes are assumed to contain user information.

- Applications must make sure that an AppVar that it uses is Unarchived before attempting to modify it. See Archiving/Unarchiving.

Variable Naming Conventions

The OP registers are used to input variable names for many system routines. They are used here to illustrate variable naming conventions.

Every variable name is a nine-byte entry that is moved in and out of system routines. All of the utility routines that move floating-point numbers in RAM can be used to move variable names.

The general format of variable names is illustrated here using OP1.

OP1	+1	+2	+3	+4	+5	+6	+7	+8
T	Variable Name							

Table 2.10: Variable Name Format

T = object type where:

<u>Bit</u>	<u>Flag</u>
0	Object Type
1	Object Type
2	Object Type
3	Object Type
4	Object Type
5	?
6	?
7	?

Every variable name has associated with it an object (data) type, which is always stored in the first byte of the variable name format.

<u>Object Type Value</u>	<u>Object Type</u>	<u>Object Type Equate</u>
00	Real	RealObj
01	List	ListObj
02	Matrix	MatObj
03	Equation	EquObj
04	String	StrngObj
05	Program	ProgObj
06	Protected Program	ProtProgObj
07	Picture	PictObj
08	Graph Database	GDBObj
0B	New EQU Obj	NewEquObj
0C	Complex Obj	CplxObj
0D	Complex List Obj	CListObj
14	Application Obj	AppObj
15	AppVar Obj	AppVarObj
17	Group Obj	GroupObj

Note: To check the type of a variable name in OP1, use the system routine **CkOP1Real**, which places the type value from OP1 into the accumulator.

```
B_CALL      CkOP1Real      ; type of OP1 to ACC
CP          CListObj       ; see if complex list
```

Variable Name Spellings

There are two classes of variable names for the TI-83 Plus — predefined and user defined. All variables are comprised of TI-83 Plus tokens, which are part of the include file, TI83plus.inc.

Predefined Variable Names

These variable's names are fixed by the TI-83 Plus and can only have a predetermined data type.

Variables: A – Z and 0

These variables can only be of type RealObj or CplxObj.

They are all spelled with one token, tA to tTheta, followed by two zeros.

Example: Real Variable A

OP1	+1	+2	+3	+4	+5	+6	+7	+8
RealObj 00h	tA 41h	00	00	?	?	?	?	?

Example: Complex Variable 0

OP1	+1	+2	+3	+4	+5	+6	+7	+8
CplxObj 0Ch	tTheta 5Bh	00	00	?	?	?	?	?

List Variables: L1 – L6

These variables can be either ListObj or CListObj.

They are all spelled with two tokens followed by one zero.

The first token of the name is tVarLst, which labels it as a list variable name. The second token signifies which predefined list name it is, tL1 – tL6.

Example: Complex List Variable L3

OP1	+1	+2	+3	+4	+5	+6	+7	+8
CListObj 0Dh	tVarLst 5Dh	tL3 02h	00	?	?	?	?	?

Note: Lists can also be user-defined, see section entitled User-Defined Variable Names in this chapter.

Matrix Variables: [A] – [J]

These variables can only be type MatObj.

They are all spelled with two tokens followed by one zero.

The first token of the name is tVarMat, which labels it as a matrix variable name. The second token signifies which predefined matrix name it is, [A] – [J].

Example: Matrix Variable [J]

OP1	+1	+2	+3	+4	+5	+6	+7	+8
MatObj 02h	tVarMat 5Ch	tMatJ 09h	00	?	?	?	?	?

Equation Variables: Y1 – Y0, X1t – X6t, Y1t – X1t, r1 – r6, u(n), v(n), w(n)

These variables can be type EquObj or NewEquObj.

They are all spelled with two tokens followed by one zero.

The first token of the name is tVarEqu, which labels it as an equation variable name. The second token signifies which predefined equation name it is:

tY1 – tY0	for	Y1 – Y0
tX1T – tX6T	for	X1t – X6t
tY1T – tY6T	for	Y1t – Y6t
tR1 – tR6	for	r1 – r6
tun	for	u(n)
tvn	for	v(n)
twv	for	w(n)

Example: Function Equation Variable Y6

OP1	+1	+2	+3	+4	+5	+6	+7	+8
EquObj 03h	tVarEqu 5Eh	tY6 05h	00	?	?	?	?	?

Example: Parametric Equation Variable Y6t

OP1	+1	+2	+3	+4	+5	+6	+7	+8
EquObj 03h	tVarEqu 5Eh	tY6T 2Bh	00	?	?	?	?	?

Example: Polar Equation Variable r1

OP1	+1	+2	+3	+4	+5	+6	+7	+8
EquObj 03h	tVarEqu 5Eh	tR1 40h	00	?	?	?	?	?

Example: Sequence Equation Variable w(n)

OP1	+1	+2	+3	+4	+5	+6	+7	+8
EquObj 03h	tVarEqu 5Eh	tw 82h	00	?	?	?	?	?

String Variables: Str1 – Str0

These variables can only be type StrngObj.

They are all spelled with two tokens followed by one zero.

The first token of the name is tVarStrng, which labels it as a string variable name. The second token signifies which predefined string name it is, tStr1 – tStr0.

Example: String Variable Str5

OP1	+1	+2	+3	+4	+5	+6	+7	+8
StrngObj 04h	tVarStrng AAh	tStr5 04h	00	?	?	?	?	?

Picture Variables: Pic1 – Pic0

These variables can only be type PictObj.

They are all spelled with two tokens followed by one zero.

The first token of the name is tVarPict, which labels it as a picture variable name. The second token signifies which predefined picture name it is, tPic1 – tPic0.

Example: Picture Variable Pic0

OP1	+1	+2	+3	+4	+5	+6	+7	+8
PictObj 07h	tVarPict 60h	tPic0 09h	00	?	?	?	?	?

Graph Database Variables: GDB1 – GDB0

These variables can only be type GDBObj.

They are all spelled with two tokens followed by one zero.

The first token of the name is tVarGDB, which labels it as a graph database variable name. The second token signifies which predefined graph database name it is, tGDB1 – tGDB0.

Example: Graph Database Variable GDB0

OP1	+1	+2	+3	+4	+5	+6	+7	+8
GDBObj 08h	tVarGDB 60h	tGDB0 09h	00	?	?	?	?	?

Variable: Ans

This is a special variable that can be a string or any numeric data type. This variable should not be used for long-term storage since the system updates it automatically.

It is spelled with one token, tAns followed by two zeros.

Example: Matrix Variable Ans

OP1	+1	+2	+3	+4	+5	+6	+7	+8
MatObj 02h	tAns 72h	00	00	?	?	?	?	?

User-Defined Variable Names

The TI-83 Plus allows open naming for some data types. Listed below are the naming rules that these variables have in common. The restriction on the length of the name varies by data type and is detailed for each data type.

- All variable names must start with a token in the range tA – tTheta (A – Z or θ).
- All subsequent tokens can be a token in the range of tA – tTheta (A – Z or θ) or t0 – t9 (0 – 9).
- Do not use lowercase or international character tokens.

User-Named Lists

These variables can be either ListObj or CListObj.

They are all spelled with the token tVarLst followed by up to a five-token name for the list.

If the name of the list, following the tVarLst token, is less than five tokens, then the name must be zero (0) terminated.

Example: Real List Variable LST1

OP1	+1	+2	+3	+4	+5	+6	+7	+8
ListObj 01h	tVarLst 5Dh	tL 4Ch	tS 53h	tT 54h	t1 31h	00	?	?

Example: Complex List Variable LIST1

OP1	+1	+2	+3	+4	+5	+6	+7	+8
CListObj 0Dh	tVarLst 5Dh	tL 4Ch	tI 49h	tS 53h	tT 54h	t1 31h	?	?

Note: There are lists with predefined names also. See the section entitled Predefined Variable Names.

User-Named Programs

These variables can be either ProgObj or ProtProgObj.

Unlike other variable names detailed so far, these do not have a leading token to signify that they are a program name.

The sign byte of a program name must be set to one of the program types.

Program names can be up to eight tokens in length. If less than eight tokens, the name must be zero (0) terminated.

Example: Program Variable ABC

OP1	+1	+2	+3	+4	+5	+6	+7	+8
ProgObj 05h	tA 41h	tB 42h	tC 43h	00	?	?	?	?

User-Named AppVars

These variables must be type AppVarObj.

Like program names, these variables do not have leading tokens to signify that they are AppVar names.

The sign byte of AppVar names must be set correctly.

AppVar names can be up to eight tokens in length. If less than eight tokens, the name must be zero (0) terminated.

Example: AppVar Variable AppVar1

OP1	+1	+2	+3	+4	+5	+6	+7	+8
AppVarObj 15h	tA 41h	tP 50h	tP 50h	tV 56h	tA 41h	tR 52h	t1 31h	00

Accessing User Variables Stored In RAM — (Unarchived)

There are two ways to access variables.

- Use system routines that return pointers to them.
- Use system routines that recall the contents of variables.

This section addresses using system routines that return pointers.

Every variable that exists in the user data area has an entry in the variable Symbol Table structure. To access the data for a particular variable, the Symbol Table is searched for the variable's entry.

Applications can use system routines to search the Symbol Table.

There are two main search routines that are used to find variables in the Symbol Table. The routine you use depends on the type of variable being looked up. Program and AppVar variables have separate search routines from all other data types.

Accessing Variables that Are Not Programs or AppVars

All of these variables have a type designator (e.g., tVarLst) as the first token in their variable name. See the naming conventions section above.

The routine to search the Symbol Table for these variables is **FindSym**.

- Input: OP1 = name of variable to search for

The sign byte need not have the correct data type of the variable; the search is done on the name alone.

For example, if an application looks up variable A, the data type cannot be known before searching because A can be a real or a complex data type.

The same applies to lists, which can be either real or complex.

- Output: See Output from a variable search on the Symbol Table section below.

Accessing Programs and AppVar Variables

This type of variable does not have as part of its name a token that signifies its data type.

The routine to search the Symbol Table for these variables is **ChkFindSym**.

- Input: OP1 = name of variable to search for

For this routine, the input name must have the data type in the sign byte set correctly.

If the search is for a program variable having the data type in OP1 set to ProgObj, the search also finds variables of the ProtProgObj data type.

For example, if an application wants to look up program ABC but does not know whether it is a normal program, ProgObj, or a protected program, ProtProgObj, using OP1 as indicated below finds program ABC if it exists and is set to either program data type.

OP1	+1	+2	+3	+4	+5	+6	+7	+8
ProgObj 05h	tA 41h	tB 42h	tC 43h	00	?	?	?	?

- Output: Output from a variable search on the Symbol Table section below.

Output from a Variable Search on the Symbol Table

The output is the same for both search routines above.

- **Does the variable exist?**

The carry flag is set if the variable is not found.

The carry flag is reset if the variable is found.

Example:

```

B_CALL    FindSym      ; look up variable in OP1
JR        C,NotFound   ; jump if it is not created

```

- **What data type is the variable?**

When searching for some variables, the type is not always known.

ACC (accumulator) = data type of the variable

OP1 object type is also set to the variable data type.

Note: Only the lower five bits of both the ACC and OP1 are set. The remaining bits are random and must be masked off to get the correct data type when checking.

Example: Search for list L1 to determine if it is a real or complex list.

```

LD      HL,L1name
B_CALL  Mov9ToOP1      ; OP1 = list L1 name
;
B_CALL  FindSym        ; look up list variable in OP1
JR      C,NotFound     ; jump if it is not created
;
AND     1Fh            ; remove none data type bits
CP      CListObj
JR      Z,ComplexList  ; jump if the list was complex
.
.
.
L1name:
DB      ListObj, tVarLst, tL1, 0

```

- **Is the variable's data in RAM or archived in Flash ROM?**

This is important information since variables that are archived need to be unarchived for use by nearly all system routines and also for easier direct access by applications.

- B register = 0 if the variable resides in RAM.

DE register = address in RAM of the first byte of the variable data structure.

The address returned is valid as long as no memory is created or deleted by archiving, unarchiving, creating, or deleting variables. If any of these actions are taken, it is necessary to relook up the variable and get the new address of the data structure.

- B register does not = 0 if the variable resides in archive.

Note: An archived variable may need to be unarchived to be used in certain system routines.

Example: Look up program ABC. If it is archived, then unarchive it.

```

LD      HL,ProgABC
B_CALL  Mov9ToOP1      ; OP1 = program ABC name
;
B_CALL  ChkFindSym     ; look up program
JR      C,NotFound     ; jump if it is not created
;
LD      A,B            ; ACC = archived/unarchived info
OR      A              ; is it archived?
JR      Z,NotArchived  ; jump if not
;
B_CALL  Arc_Unarc      ; unarchive the var
NotArchived:
ProgABC:
DB      ProgObj, 'ABC', 0

```

Example: Search for list L1 and set DE = to the number of elements in the list. Assume it is not archived.

```

LD      HL,L1Name
B_CALL  Mov9ToOP1      ; OP1 = list L1 name
;
B_CALL  FindSym        ; look up list variable in OP1
JR      C,NotFound     ; jump if it is not created
;
EX      DE,HL          ; HL = pointer to data structure
LD      E,(HL)          ; get the LSB of the number elements
INC     HL              ; move to MSB
LD      D,(HL)          ; DE = number elements in L1
.
.
.
L1Name:
DB      ListObj, tVarLst, tL1, 0

```

- **A pointer to the variable's Symbol Table entry.**

The HL register = address of the variable's Symbol Table entry.

This is returned for both archived and unarchived variables. The Symbol Table entries for all variables reside in RAM.

Creating Variables

There are two ways that variables can be created.

- Use system routines that create them directly.
- Use system routines that store a value to a variable, creating that variable if it does not already exist.

This section addresses the first method, and the following section deals with the second method.

- Variables can only be created in RAM. Once created, they can be archived to the Flash ROM.
- A variable that already exists, even if archived, should not be recreated without first deleting the current one. See Deleting Variables section below.

Routines that create variables do not check to see if a variable currently exists before creating it. An application must check by searching the Symbol Table for the variable. See routines **FindSym** and **ChkFindSym**. If this is not done, multiple versions of the same variable exist leading to unpredictable side effects.

- Do not create variables with sizes outside of their specified limits. For example, do not create a list with 1000 elements. The system does not check for these types of errors when creating a variable.

Some system routines will fail and may cause a lock-up condition if bad data is input to them.

For more information see the Variable Data Structure section earlier in this chapter.

- If there is not enough free memory available to create a variable, a system memory error is generated, and the system's error context will take over execution.

This can be avoided in two ways.

- Use the routine **MemChk** to see if there is enough free memory available before attempting to create the variable.
- Use an error exception handler to trap the memory error (if one is generated).

To use option one, the size of the Symbol Table entry and the data structure must be computed by the application. Therefore, the easiest is option two.

See the Error Handlers section.

- **When a variable is created, its data structure is not initialized.** Only the two-byte size field, if one is part of the structure, is initialized to the size the variable was created at. For example, after creating a complex variable, the entire 18 bytes of the data structure has random values.

After creating a list with 23 elements, the first two bytes of the data structure are set to the number of elements, 17h 00h, the number of elements in hex, with the LSB followed by the MSB.

If created data structures are not initialized by applications before returning to normal system operation, the potential for a lock-up condition is very high.

- Routines for creating variables:

Create0Equ	CreateEqu	CreatePair	CreateStrng
CreateRList	CreateCList	CreateRMat	
CreateReal	CreateCplx	CreatePict	
CreateAppVar	CreateProg	CreateProtProg	

- Inputs:

OP1 = variable name to create.

HL = Number of bytes, number of elements or a dimension for some.

See Appendix A for exact inputs for each routine.

- Outputs:

Possible memory error, see above.

OP4 = variable name created with its sign byte set to the correct data type

OP1 = random

DE = pointer to data structure

HL = pointer to Symbol Table entry

For example, create a real list CAT with one element and initialize that element to a value of five. Return CA = 0 if the variable is created, else CA = 1 if there is not enough memory.

```

Create_CAT:
    LD        HL,catname
    B_CALL    Mov9ToOP1        ; OP1 = name
;
    AppOnErr   NoMem           ; install error handler
;
    LD        HL,1             ; 1 element list
    B_CALL    CreateRList      ; ret from call if no mem error
    INC       DE
    INC       DE               ; DE = pointer to start of element 1
    LD        HL,FP_5
    LD        BC,9
    LDIR                          ; set first element = 5
;
    AppOffErr                          ; remove error handler
;
    OR        A                ; CA = 0 if successful
    RET
CatName:
    DB        ListObj, tVarLst, 'CAT', 0
FP_5:
    DB        00h,80h,50h,00h,00,00,00,00,00
;
; control comes here if memory error during create
;
NoMem:
    SCF                          ; CA = 1 if not successful
    RET

```

Storing to Variables

There are system routines that can be used to store to the entire contents of a variable's data structure.

These routines store a real or complex variable to N, X, Y, R, T, θ .

StoN	StoX	StoY
StoR	StoT	StoTheta

StoAns stores any numeric, equation or string to Ans.

StoOther stores to any numeric, equation or string variable.

Attributes of these routines include:

- If the variable that is being stored to does not exist, it is created if enough free RAM is available.
- The current contents of the variable are not deleted if the new data being stored to the variable does not fit in memory.
- Error checking is done to make sure that the data type being stored to the variable is valid for that variable.

- If the variable being stored to is archived, a system error is generated.
- Since system errors can be generated by these routines, an error handler should be placed around calls to them. See the Error Handlers section.

The details on inputs and outputs for these routines can be found in Appendix A.

Note: The following example uses the routine **PushRealO1**. See the Floating Point Stack section for details.

Example: Store a value of 1.5 to variable Z

return CA = 0 if successful

CA = 1 if failed to store

```

Sto_Z:
        B_CALL    OP1Set1      ; OP1 = 1
        LD        A,15h
        LD        (OP1+2),A    ; OP1 = 1.5
;
        B_CALL    PushRealO1   ; 1.5 -> FPST
        B_CALL    ZeroOP1     ; OP1 = 00000000000
        LD        A,'Z'
        LD        (OP1+1),A    ; OP1 = Z VAR NAME
;
        AppOnErr   Fail        ; install error handler
;
        B_CALL    StoOther     ; attempt to store, RET if no error
;
        AppOffErr                ; remove error handler
        OR        A            ; CA = 0 for store is good
        RET
Fail:
        SCF                        ; CA = 1 for no store
        RET

```

Recalling Variables

There are system routines that can be used to recall the contents of real and complex variables to OP1/OP2.

RclVarSym	RclY	RclN	RclX	RclAns
------------------	-------------	-------------	-------------	---------------

Attributes of these routines include:

- If the variable does not exist or if it is archived, a system error is generated.
- If the variable is real, OP1 = the value.
- If the variable is cplx, OP1/OP2 = the value.

Note: Since system errors can be generated by these routines, an error handler should be placed around calls to them.

The details on inputs and outputs for these routines can be found in Appendix A.

Example: Recall the contents of variable C, assume it is created and not archived, and check if it is real.

```

      B_CALL      ZeroOP1      ; OP1 = 000000000000
      LD          A, 'C'
      LD          (OP1+1),A      ; OP1 = C var name
;
      B_CALL      RclVarSym      ; OP1/OP2 = value
      B_CALL      CkOP1Real      ; ACC = type, Z = 1 if real

```

Deleting Variables

- Any variable that has an entry in the Symbol Table can be deleted, even if the data is archived.
- Preallocated system variables located in system RAM, such as Xmin, cannot be deleted.
- There are some system variables that also reside in user RAM. They are created in the same way as user variables and have Symbol Table entries. All of these system variables are spelled with an illegal first character so that they are excluded from any menus that show the current variables that exist.

Some of these variables include # and ! which are two program variables used for home screen entry and the first level of last entry. None of these variables should be deleted.

- The graph equations should not be deleted. The TI-83 Plus system will crash if these equations are not created.

If an application wants to free the RAM used by a graph equation, it can delete the equation and **immediately** recreate the equation with a size of 0 bytes. See the **Create0Equ** routine for further information.

- When a variable is deleted, its Symbol Table entry and its data structure are removed from RAM. If the data was archived, only the Symbol Table entry is removed from RAM and the archive space made available. Deleting an archived variable will not free much RAM space for other uses.

There are no holes left in RAM when a variable is deleted. Both the user memory and Symbol Table are immediately compressed, and all of the freed RAM now becomes part of the free RAM area.

- There are three routines for deleting variables — **DelVar**, **DelVarArc**, and **DelVarNoArc**. The difference between them is how an archived variable is handled.

Common inputs:

HL = pointer to the variable's Symbol Table entry

DE = pointer to the variable's data structure

Note: These inputs are output from a successful Symbol Table search, such as **FindSym**.

DelVar	Error if the variable is archived. This routine checks the contents of the b register to be non-zero. If the contents is non-zero, it assumes the variable is archived and generates a system error. Otherwise, delete it from RAM. The b register is set to reflect whether or not a variable is archived by any of the Symbol Table search routines.
DelVarArc	Delete the variable if archived or unarchived. This routine checks the contents of the b register to be non-zero. If the content is non-zero, then it assumes the variable is archived and deletes it from the archive. Otherwise, it deletes it from RAM. The b register is set to reflect whether or not a variable is archived by any of the Symbol Table search routines.
DelVarNoArc	Assumes the variable is not archived and deletes it from RAM. This routine does not check the contents of the b register and assumes the pointers input are RAM pointers, not pointers into the archive space. Only use this routine if you are absolutely sure that the variable resides in RAM.

Note: OP1 through OP6 are kept intact.

For example, if matrix [A] exists and is not archived, delete it and recreate it with a dimension of five rows and three columns.

return CA = 0 if successful, or

CA = 1 if it was archived or there was not enough free RAM to create it.

```

Create_MatA:
    LD        HL,MatAname
    B_CALL    Mov9ToOP1        ; OP1 = name
    B_CALL    FindSym          ; look up
    JR        C,CreateIt       ; jump if it does not exist
;
    LD        A,B
    OR        A                ; archived?
    JR        NZ,Failed        ; jump if it is archived
;
    B_CALL    DelVarNoArc      ; delete it, it is not archived
CreateIt:
    AppOnErr   Failed          ; install error handler
;
    LD        HL,5*256+3       ; dim wanted 5x3
    B_CALL    CreateRMat       ; ret from call if no mem error
;
    AppOffErr
;
    OR        A                ; CA = 0 if successful
    RET
MatAName:
    DB        MatObj, tVarMat, tMatA, 0
;
; control comes here if memory error during create
;
Failed:
    SCF
    RET
; CA = 1 if not successful

```

Archiving and Unarchiving

Applications can use the Flash archive area in the same way as users do during normal system operation. Variables can be moved from RAM to the archive (archived) area, and also removed from the archive area and placed into RAM (unarchived). More information on the uses of archiving can be found in the TI-83 Plus Graphing Calculator Guidebook.

Note: Most system routines are not designed to work with variables stored in the Archive area, and many do not check for this error. Be sure to check where variables are located, RAM or Archive, before using them as inputs to system routines not expecting variables to be residing in the archive.

- **What can be archived?**

All user variables can be archived, **except** the following (listed by type):

RealObj / CplxObj:	X, Y, T, θ
ListObj / CListObj:	RESID, IDList
EquObj, NewEquObj:	Any

- **What cannot be unarchived?**

The following can not be unarchived:

GroupObj

AppObj

- **Entry Point**

Arc_Unarc If the variable in OP1 is archived, unarchive it, otherwise archive it. See Appendix A for further information.

System errors can be generated. See the Error Handlers section for further information.

A battery check should be done before attempting to archive a variable. There is a risk of corrupting the archive if the attempt fails due to low batteries. Applications should display a message informing users to replace the batteries if low batteries are detected.

As an Archive example, archive the variable whose name is in OP1.

```

      B_CALL      Chk_Batt_Low    ; check battery level
      RET         NZ              ; ret if low batteries
;
      B_CALL      ChkFindSym
      RET         C               ; return if variable does not exist
      LD          A,B             ; get archived status
      OR          A               ; if non zero then it is archived
                                ; already
      RET         NZ             ; ret if archived
      AppOnErr    errorHand      ; install error handler
;
      B_CALL      Arc_Unarc       ; archives the variable
;
      AppOffErr
errorHand:
      RET

```

Related Routines

ChkFindSym Searches the Symbol Table for a variable.

MemChk Returns the amount of free RAM available.

See Appendix A for further information.

Accessing Archived Variables without Unarchiving

Variable data residing in the archive can be accessed without unarchiving the data to RAM. This is a read-only operation, an application cannot write data directly to the archive.

- Locating archived variables

Archived variables will have an entry in the Symbol Table that contains information on where the data resides in the archive.

The Symbol Table search routines used to locate variables in RAM, **FindSym** and **ChkFindSym**, are also used to locate variables in the archive. See the Accessing User Variables Stored in RAM section for a detailed explanation of these routines.

If a variable is archived, the output from the Symbol Table search routine will return two key pieces of information.

B register = ROM page of the start of the archived data.

DE register = the offset on the ROM page to the start of the archived data.

- How is variable data stored in the archive?

The actual data for a variable has the same structure as when it resides in RAM. See Variable Data Structures section for further information.

In addition to the variable's data structure, a copy of the variable's Symbol Table entry is also stored in the archive. Fig. 2.11 below shows the format used for each variable stored in the archive.

Data valid	Size of symbol entry + Data		Size varies by the name size and data type	Size computed the same as variables in RAM
Flag	LSB	MSB	Symbol Table Entry	Variable Data Structure
Increasing addresses ----->				

Table 2.11: Format of Archive Stored Variables

Archived data for a single variable can cross ROM page boundaries. System routines to read from the archive are provided to make this cross boundary situation transparent to applications.

- Reading bytes from the archive

There are two methods provided for reading data from the archive — direct and cached.

- Direct

This method involves an application reading either one or two bytes at a time from the archive — supplying both the ROM page and offset to the data to be read.

Inputs: B register = ROM page of byte(s) to copy

HL register = offset on the ROM page to the byte(s) to copy

Routines:

- **LoadCIndPaged** Copies a byte from the archive to C
C = byte from archive
B, HL = intact
- **LoadDEIndPaged** Copies two bytes from the archive to DE
E = first byte read
D = second byte read
B, HL = location of the second byte, crossing a ROM page boundary is handled
- Recommended support routines that an application should include as part of the application.

```
LoadCIndPaged_inc:
    B_CALL      LoadCIndPaged    ; read byte from archive
;
; fall thru and INC pointer past byte read
;
inc_BHL:
    INC         HL               ; increment offset in page
    BIT         7,h             ; cross page boundary?
    RET         Z               ; no, B, HL = ROM page and
                                ; offset
;
    INC         B               ; increase ROM page number
    RES         7,H
set         6,H               ; adjust offset to be in
                                ; 4000h to 7FFFh
    RET
;
LoadDEIndPaged_inc:
    B_CALL      LoadDEIndPaged  ; read 2 bytes from
                                ; archive
    JR          inc_BHL         ; move pointer to byte
                                ; after 2 read
```

– Cached

This method provides management of the ROM page and offset of data in the archive while reading multiple bytes. These values are stored in predefined system RAM locations. A 16 byte RAM cache is used to queue up consecutive data from the archive. There are two routines used.

- **SetupPagedPtr** Sets the initial value of the system RAM used to track the current read location and the current amount of data in the cache. This must be called before any data is actually read.

Inputs: B register = ROM page of first byte to copy.

HL register = offset on the ROM page to the first byte(s) to copy.

- **PagedGet** This routine has two functions. First is to fill the 16 byte cache with mode data from the archive, whenever it has been completely read. Second, is to return the next byte from the cache to the caller. The first byte returned is at the location input to **SetupPagedPtr**, followed by each consecutive byte that follows.

Inputs: Initial inputs are set by **SetupPagedPtr**, and are updated after each subsequent call to **PagedGet**.

Outputs: ACC = byte read.

Cache pointers updated.

Cache reloaded with next 16 bytes of archive if exhausted.

Note: Both of these methods, direct and cached, will force an application to read data from the archive sequentially. This can be very inefficient if the eightieth byte of an archived equation needed to be read. An application would have to read through the first 79 bytes one at a time.

In Ram, the solution would be to add 80 to the address of the start of the equation and then do one read. In the archive, it is not as simple. An application has to be wary of ROM page boundaries and offsets into a ROM page.

Applications can use the following code to add a two byte value to a ROM page and offset archive address, so that page boundary crossing is adjusted for. This routine will work for adding values up to 4000h (16K) maximum.

```
;
; Add DE to ROM page and offset: B, HL
;
BHL_Plus_DE:
    ADD     HL,DE      ; add DE to the offset HL
    BIT     7,H        ; cross page boundary?
    RET     Z          ; no, B, HL = ROM page and offset
;
    INC     B          ; increase ROM page number
    RES     7,H
    SET     6,H        ; adjust offset to be in 4000h
                      ; to 7FFFh
    RET
```

For example, look up archived AppVar MYAPPVAR, and read past its Symbol Table entry in the archive to reach the data. Then read the two size bytes of the AppVar.

Data valid	Size of Symbol entry + Data		Size varies by the name size and data type	Size computed the same as variables in RAM
Flag	LSB	MSB	Symbol Table entry	Variable Data Structure
Increasing addresses ----->				

Table 2.12: Format of Archive Stored Variables

```

LD      HL,MyAppVar
RST
B_CALL  ChkFindSym      ; OP1 = AppVar name
                        ; find Symbol Table entry,
                        ; and get pointers
;
; B = ROM page and DE = offset, to start of data in the archive
;
EX      DE,HL           ; B, HL now points to the
                        ; data of the variable
CALL    LoadCIndPaged_inc ; skip data valid flag
CALL    LoadDEIndPaged_inc ; skip data length, B, HL
                        ; at symbol entry
;
; now the size of the Symbol Table entry needs to be computed so that
; it can be skipped over to get to the AppVar's data structure
;
LD      DE,5           ; DE = offset to name
                        ; length of AppVar
CALL    BHL_plus_DE    ; add DE to B, HL:
                        ; page, offset
;
CALL    LoadCIndPaged_inc ; C = name length, B, HL
                        ; advanced
LD      E,C            ; DE = offset to start of
                        ; AppVars data
;
CALL    BHL_plus_DE    ; add DE to B, HL: page,
                        ; offset
;
CALL    LoadDEIndPaged_inc ; DE = size bytes of
                        ; AppVar,
RET

MyAppVar: asciz AppVObj, 'MYAPPVAR'

BHL_Plus_DE:
ADD     HL,DE          ; add DE to the offset HL
BIT     7,H            ; cross page boundary?
RET     Z              ; no, B, HL = ROM page and
                        ; offset
;
INC     B              ; increase ROM page number
RES     7,H            ;
SET     6,H            ; adjust offset to be in
                        ; 4000h to 7FFFh
RET

```

Manipulation Routines

List Element Routines

These routines are used for storing and recalling list element values and for changing the dimension of a list.

AdrLEle	Returns the RAM address of a list element.
GetLToOP1	Recalls an element of a list to OP1 if Real or OP1/OP2 if Cplx.
PutToL	Stores OP1 if Real or OP1/OP2 if Cplx, to an element of a list.
IncLstSize	Increments the size of an existing list by adding an element to the end of the list.
InsertList	Inserts one or more elements into an existing list.
DelListEl	Deletes one or more elements from an existing list.

See Appendix A for details.

Matrix Element Routines

These routines are used for storing and recalling matrix element values and for changing the dimension of a matrix.

AdrMEle	Returns the RAM address of a matrix element.
GetMToOP1	Recalls an element of a matrix to OP1.
PutToMat	Stores OP1 to an element of a matrix.
RedimMat	Redimensions an existing matrix in RAM.

See Appendix A for details.

Resizing AppVar, Program, and Equation Variables

These data types can be resized in place without having to make an additional copy of the variable. Following are the two routines, with examples, used to increase the data size and to decrease the data size.

- Increasing the data size.

InsertMem Increases the size of an existing variable by inserting space at a given address.

For example, insert 10 bytes at the beginning of an existing AppVar. If there is not enough free RAM, the AppVar does not exist, or if the AppVar is archived, CA = 1 is returned.

```

Insert_10:
    LD      HL,10          ; number bytes to insert
    B_CALL  EnoughMem      ; check for free RAM
    RET     C              ; ret CA = 1 if not
;
    LD      HL,AppVarName
    B_CALL  Mov9ToOp1      ; Op1 = name of AppVar
    B_CALL  ChkFindSym     ; DE = pointer to data if exists
    RET     C              ; ret if not found
    LD      A,B            ; archived status
    ADD     0FFh           ; if archived then CA = 1
    RET     C              ; ret if archived
;
    PUSH    DE             ; save pointer to size bytes of
                           ; data
    INC     DE
    INC     DE             ; move DE past size bytes
;
    LD      HL,10          ; number bytes to insert
    B_CALL  InsertMem      ; insert the memory
    POP     HL             ; HL = pointer to size bytes
    PUSH    HL             ; save
;
    B_CALL  ldHLind        ; HL = old size of AppVar,
                           ; number bytes
    LD      BC,10
    ADD     HL,BC          ; increase by 10, amount inserted
    EX      DE,HL         ; DE = new size
    POP     HL             ; pointer to size bytes location
    LD      (HL),E
    INC     HL
    LD      (HL),D         ; write new size.
    OR      A              ; CA = 0
    RET
AppVarName: DB      AppVarObj,'AVAR',0

```

See Appendix A for details on **InsertMem**.

- Decreasing the data size

DelMem Decreases the size of an existing variable by removing data at a given address.

For example, delete 10 bytes at the beginning of an existing AppVar. If the AppVar does not exist or if it is archived, CA = 1 is returned.

```

Delete_10:
        LD          HL,AppVarName
        B_CALL      Mov9ToOP1      ; OP1 = name of AppVar
        B_CALL      ChkFindSym     ; DE = pointer to data if exists
        RET         C              ; ret if not found
;
        LD          A,B            ; archived status
        ADD         0FFh           ; if archived then CA = 1
        RET         C              ; ret if archived
;
        PUSH        DE             ; save pointer to size bytes of
                                   ; data
        INC         DE
        INC         DE             ; move DE past size bytes
;
        LD          HL,10          ; number bytes to insert
        EX          DE,HL          ; HL = pointer to start of delete,
                                   ; DE = number bytes
        B_CALL      DelMem         ; delete the memory
        POP         HL             ; HL = pointer to size bytes
        PUSH        HL             ; save
;
        B_CALL      ldHLind        ; HL = old size of AppVar,
                                   ; number bytes
        LD          BC,10
        OR          A
        SBC         HL,BC          ; decrease by 10, amount deleted
        EX          DE,HL          ; DE = new size
        POP         HL             ; pointer to size bytes location
        LD          (HL),E
        INC         HL
        LD          (HL),D          ; write new size.
        OR          A              ; CA = 0
        RET
AppVarName: DB      AppVarObj,'AVAR',0

```

See Appendix A for details on **DelMem**.

Symbol Table Structure

This structure contains an entry for each variable that is created. It contains information about a variable's type, name, and location in RAM or in the archive. The Symbol Table begins in high memory at the end of the hardware stack and grows towards low memory (backwards).

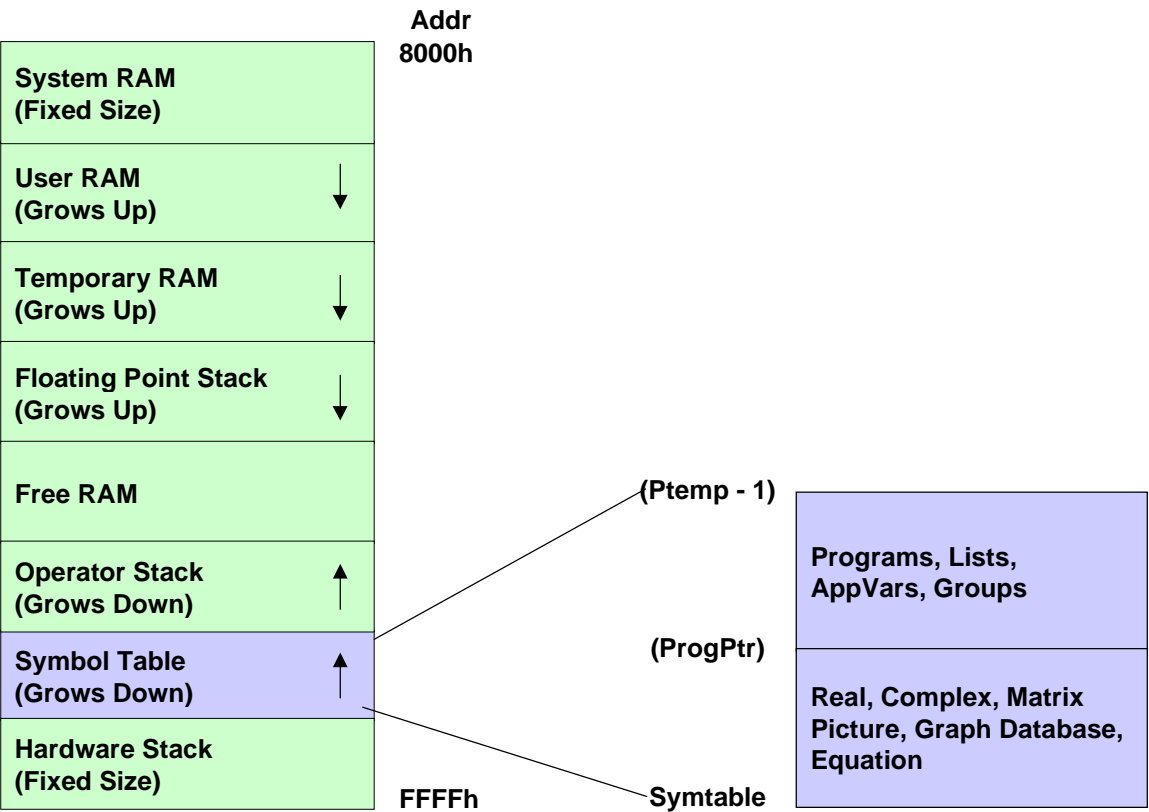


Fig. 2.6: Symbol Table Structure

The Symbol Table is divided into two sections by data type.

The first byte of the Symbol Table for Real, Cplx, Mat, Pict, GDB, and EQU is at address symTable and ends at address (progPtr-1).

The first byte of the Symbol Table for Prog's, List AppVar and Group is at address (progPtr) and ends at (pTemp-1).

symTable is a fixed address and never changes.

(progPtr) and (pTemp) are not fixed addresses.

For example, load the current start address of the Program/List/AppVar/Group Symbol Table into register HL.

```
LD      HL, (progPtr)
```

The Symbol Table is split by the structure of the entries.

Each entry is written from high memory to low memory (backwards).

Program, AppVar, Group

Start of
Entry
↓

-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0
Variable Name 8 characters max								NL	Page	DAH	DAL	Ver	T2	T

Table 2.13: Program, AppVar, Group

Lists

Start of
Entry
↓

-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0
F	Variable Name 5 characters max					tVarLst 5Dh	NL	Page	DAH	DAL	Ver	T2	T

Table 2.14: Lists

Real, Cplx, Mat, EQU, GDB, Pict

Start of
Entry
↓

-8	-7	-6	-5	-4	-3	-2	-1	0
00	Second token of name	First token of name	Page	DAH	DAL	Ver	T2	T

Table 2.15: Real, Cplx, Mat, EQU, GDB, Pict

- T = object type where:

<u>Bit</u>	<u>Flag</u>
0	Object Type
1	Object Type
2	Object Type
3	Object Type
4	Object Type
5	Graph equation selected
6	Variable used during graphing
7	Link transfer flag

<u>Object Type Value</u>	<u>Object Type</u>	<u>Object Type Equate</u>
00	Real	RealObj
01	List	ListObj
02	Matrix	MatObj
03	Equation	EquObj
04	String	StrngObj
05	Program	ProgObj
06	Protected Program	ProtProgObj
07	Picture	PictObj
08	Graph Database	GDBObj
0B	New EQU Obj	NewEquObj
0C	Complex Obj	CplxObj
0D	Complex List Obj	CListObj
14	Application Obj	AppObj
15	AppVar Obj	AppVarObj
17	Group Obj	GroupObj

- T2 = Reserved for future use.
- Ver = Version number.
 - Each variable's Symbol Table entry contains a byte field for its version.
 - The version of a variable determines its scope of compatibility with future upgrades of the TI-83 Plus.

- A future TI-83 Plus release may create a new data type that the earlier releases do not know how to handle. This variable's version number would be set higher than the version number of the previous code released.
- If a new variable type is sent to an TI-83 Plus running an earlier version of product code, the variable would not be accepted by the earlier product code since the variable's version number is higher than the products code.
- DAL = Data structure pointer's low (LSB) byte.
- DAH = Data structure pointer's high (MSB) byte.
- PAGE = ROM page the data structure resides on if archived, if it resides in RAM, unarchived, this byte is zero (0).
- NL = Name length of the variable.

Note: For lists include the byte tVarLst in the length.

- F = Formula number attached to a list.
 - Lists can have a formula attached to them that is executed every time the list is accessed. The result of the execution is stored into the lists data structure.
 - If this value is 0, there is no formula.
 - This value is used to generate a unique name for the formula attached to a particular list variable.
 - The Symbol Table entry for one of these formulas would be:

-8	-7	-6	-5	-4	-3	-2	-1	0
00	F #	? 3Fh	Page	DAH	DAL	Ver	T2	EquObj

Table 2.16: Formula Example

- Variable names — See Naming Conventions.

Example: A routine that traverses both sections of the Symbol Table.

```

Traverse_symTable:
    LD      HL,symTable      ; HL = pointer to first symbol entry
    LD      D,0
    LD      BC,(pTemp)      ; BC = pointer to byte after the end
                                ; of the Symbol Table

loop:
    OR      A
    SBC     HL,BC           ; current - end, if CA then done with
                                ; search
    RET     C               ; return if no more syms to check
    RET     Z               ; return if no more to check
;
    ADD     HL,BC           ; restore current search pointer
    LD      A,(HL)         ; get symbol entry type
    AND     1Fh            ; mask off variable type
;
    LD      E,6             ; DE = offset to NL or first byte of
                                ; name
    SBC     HL,DE           ; (HL) = NL or first byte of name
;
    LD      E,3             ; DE = offset to next entry if not a
                                ; program/list/group/AppVar
    CP      AppVarObj       ; current entry an AppVar
    JR      Z,movetnext    ; yes, get NL to find next entry
;
    CP      ProgObj        ; current entry a program
    JR      Z,movetnext    ; yes, get NL to find next entry
;
    CP      ProtProgObj    ; current entry a program
    JR      Z,movetnext    ; yes, get NL to find next entry
;
    CP      TempProjObj    ; current entry a program
    JR      Z,movetnext    ; yes, get NL to find next entry
;
    CP      groupprogobj   ; current entry a group var
    JR      Z,movetnext    ; yes, get NL to find next entry
;
    DEC     HL              ; (HL) = tVarLst if a list
    LD      A,(HL)
    INC     HL              ; fix
    CP      tVarLst        ; current entry a list
    JR      NZ,movetnext1  ; no

movetnext:
    LD      E,(HL)         ; DE = length of name
    INC     E               ; DE = length of name + 1
;
; move HL to next symbol table entry sign digit
;
movetnext1:
    OR      A
    SBC     HL,DE           ; HL = next symbol table entry address
    JR      loop

```

Floating Point Stack (FPS)

The Floating Point Stack (FPS) is a TI-83 Plus system RAM structure that begins at the end of the variable data storage area and grows toward the Symbol Table storage area.

The stack grows and shrinks in size in multiplies of nine bytes ONLY. This entry size is the size of a floating-point number.

This does not mean that all entries pushed onto the stack need be floating-point numbers. The content of the nine bytes, in most cases, can be random data or a Floating Point Stack. The only exception is when system routines are used to manipulate the Floating Point Stack expecting data type information to be stored in the entry to be placed on, removed from, copied to, or copied from the FPS.

Many of the TI-83 Plus system routines will use the FPS for argument passing and temporary storage during computations.

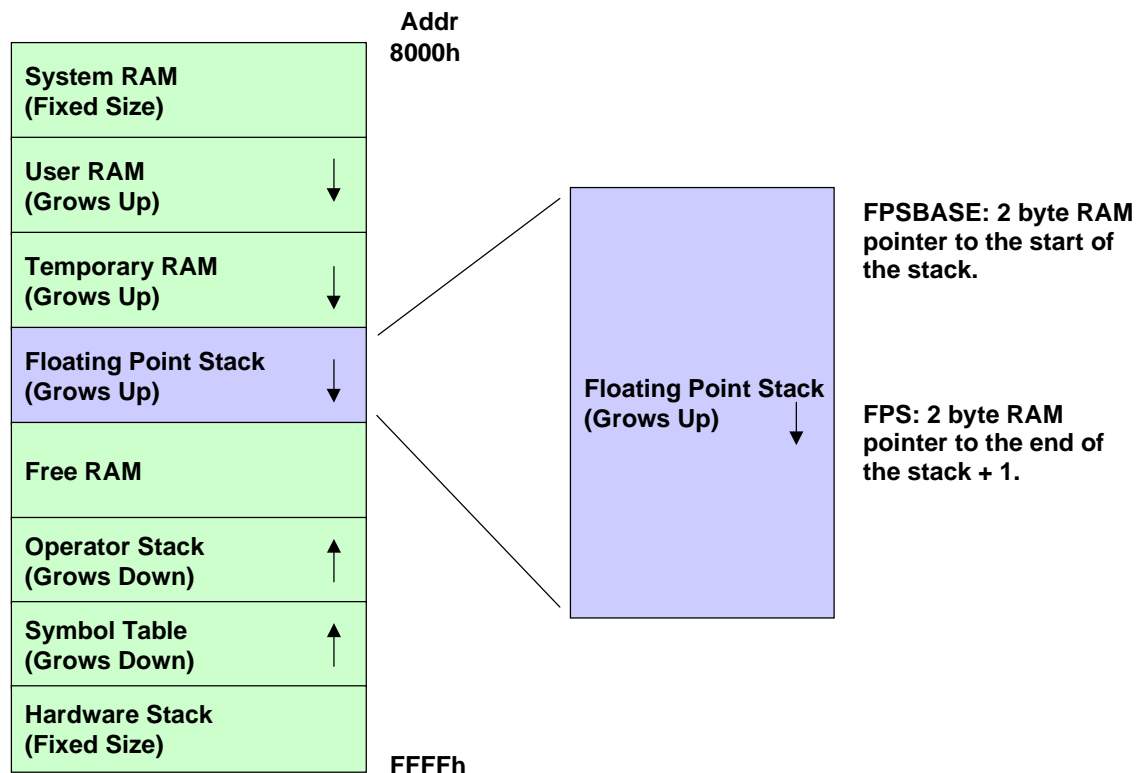


Fig. 2.7: TI-83 Plus System RAM

Naming Convention

The following abbreviations are used when dealing with the Floating Point Stack.

FPS = Floating Point Stack

FPST = Floating Point Stack Top. This is the last nine bytes of the FPS.

FPS1 = Floating Point Stack minus 1 entry. This is the second to last nine bytes of the FPS. Each previous nine bytes would continue this scheme FPS2, FPS3 ... FPSn.

For example, assume the FPS is empty, (FPS) = (FPSBASE) and OP1 = floating-point value 1, and OP2 = floating-point value 2.

```

      B_CALL      PushReal01      ; pushed 9 bytes of OP1 -> FPST
;
      B_CALL      PushReal02      ; OP2 -> FPST, FPST -> FPS1

```

RAM would look similar to this depending on fpBase value.

Addr

(fpBase)-----> 9C00 80h 10h 00 00 00 00 00 00 00 (1.00000000) FPS1

9C09 80h 20h 00 00 00 00 00 00 00 (2.00000000) FPST

(FPS)-----> 9C12

General Use Rules

The following are some general use rules when manipulating the FPS.

- The FPS can be used by applications at anytime.
- The only time that the FPS cannot be allocated or deallocated to is during a system edit input session.
- Any allocations (pushes) to the FPS are the responsibility of the routine that made the allocation. Some system routines will take arguments that have been put onto the FPS and will remove them.
- Not cleaning the FPS properly could cause system lockups during application execution or after the application is exited.
- If the system's error context is invoked, (e.g., ERR:DOMAIN), the FPS will be reset.
- If an attempt is made to allocate space on the FPS with insufficient free RAM available, a system error is generated.

These system errors can be avoided in the same manner as creating variables are, with the use of an error handler invoked before the allocation is attempted. See the section on Error Handlers later in Chapter 2.

FPS System Routines

The OP registers are used extensively by the system's FPS routines for input and output.

FPS Allocation Routines

These routines are separated by either the size of the allocation or by a Data Type of a value, Real/Complex.

- Pushes nine bytes onto the FPS. For these routines, the word Real implies nine bytes.

PushReal	Pushes nine bytes pointed to by HL onto the FPS.
PushRealO1	Allocates nine bytes on FPS then OP1 is copied to FPST.
PushRealO2	Allocates nine bytes on FPS then OP2 is copied to FPST.
PushRealO3	Allocates nine bytes on FPS then OP3 is copied to FPST.
PushRealO4	Allocates nine bytes on FPS then OP4 is copied to FPST.
PushRealO5	Allocates nine bytes on FPS then OP5 is copied to FPST.
PushRealO6	Allocates nine bytes on FPS then OP6 is copied to FPST.

- Pushes a complex number from two consecutive OP registers onto the FPS.

For these routines, the REAL part of the complex number is in the OP register specified and the IMAGINARY part is in the following OP register. Only nine bytes of each of the registers are pushed onto the FPS.

PushMCplxO1	Pushes OP1 onto FPS then pushes OP2 onto FPS. FPS1 = OP1, FPST = OP2.
PushMCplxO3	Pushes OP3 onto FPS then pushes OP4 onto FPS. FPS1 = OP3, FPST = OP4.

- Checks the data type of a value in an OP register for either Real or Cplx, and pushes the value onto the FPS.

These routines check the specified OP register's data type byte, and if CplxObj, then pushes a complex number from the OP registers in the same way as the **PushMCplx** routines above. Otherwise, pushes nine bytes from the register specified onto the FPS.

PushOP1	Pushes OP1 or OP1/OP2, checks OP1 = CplxObj.
PushOP3	Pushes OP3 or OP3/OP4, checks OP3 = CplxObj.
PushOP5	Pushes OP5 or OP5/OP6, checks OP5 = CplxObj.

- Block allocates space on the FPS with no data transfer. This is done to preallocate space needed on the FPS in one step. To set the values, the **CopyToFPS** routines need to be used. They are described later in this section.

AllocFPS Allocates HL number of nine-byte entries.

AllocFPS1 Allocates HL number of bytes, which must be a multiple of nine.

FPS Deallocation Routines

- Pops nine bytes off of the FPS. For these routines, the word Real implies nine bytes.

PopReal Removes nine bytes off of the FPS and writes to RAM pointed to by DE.

PopRealO1 Removes nine bytes from FPS then copies to OP1.

PopRealO2 Removes nine bytes from FPS then copies to OP2.

PopRealO3 Removes nine bytes from FPS then copies to OP3.

PopRealO4 Removes nine bytes from FPS then copies to OP4.

PopRealO5 Removes nine bytes from FPS then copies to OP5.

PopRealO6 Removes nine bytes from FPS then copies to OP6

- Pops a complex number, or two nine-byte entries, off of the FPS into two consecutive OP registers.

For this routine, the first nine-bytes removed from the FPS are written to the OP register following the one specified, and the preceding nine bytes are written to the OP register.

PopMCplxO1 Removes nine bytes from FPS then copies to OP2 and removes next nine bytes from FPS then copies to OP1.

- Checks the data type of a value in FPST for either Real or Cplx, and pops the value into one or two OP registers.

These routines check FPST entry's data type byte, and if CplxObj, then pops FPST and FPS1 entries into the specified OP registers. Otherwise pops nine bytes FPST into the specified OP register.

PopOP1 Removes nine or 18 bytes from the FPS placing them into OP1/OP2.

PopOP3 Removes nine or 18 bytes from the FPS placing them into OP3/OP4.

PopOP5 Removes nine or 18 bytes from the FPS placing them into OP5/OP6.

- Block deallocates entries from FPS with no data transfer.

These routines remove entries starting at FPST by modifying the value of the pointer FPS.

DeallocFPS Removes HL number of nine byte entries from the FPS.

DeallocFPS1 Removes DE number of bytes from the FPS, this must be a multiple of nine.

Copy Data To and From Existing FPS Entries

- Accesses entries on the FPS by using the RAM pointers FPS and FPSBASE, which define the boundaries of the FPS.
- Copies nine bytes from RAM to an FPS entry.

CpyToStack If this routine is to be used, it is recommended that you create this routine in your APP/ASM:

```

;
; input: C = offset from (FPS) to start of 9
;         byte entry to write to. max = 252
;
;         ex: C = 9  -> FPST
;              18 -> FPS1
;
;         DE = pointer to 9 bytes of RAM to copy to FPS
;
;
CpyToFPS:
                LD          HL,(FPS)
                B_CALL      CpyToStack

```

CpyToFPST Copies nine bytes at DE to FPST.

CpyToFPS1 Copies nine bytes at DE to FPS1.

CpyToFPS2 Copies nine bytes at DE to FPS2.

CpyToFPS3 Copies nine bytes at DE to FPS3.

CpyO1ToFPST Copies nine bytes in OP1 to FPST.

CpyO1ToFPS1 Copies nine bytes in OP1 to FPS1.

CpyO1ToFPS2 Copies nine bytes in OP1 to FPS2.

CpyO1ToFPS3 Copies nine bytes in OP1 to FPS3.

CpyO1ToFPS4 Copies nine bytes in OP1 to FPS4.

CpyO1ToFPS5 Copies nine bytes in OP1 to FPS5.

CpyO1ToFPS6 Copies nine bytes in OP1 to FPS6.

CpyO1ToFPS7 Copies nine bytes in OP1 to FPS7.

CpyO2ToFPST	Copies nine bytes in OP2 to FPST.
CpyO2ToFPS1	Copies nine bytes in OP2 to FPS1.
CpyO2ToFPS2	Copies nine bytes in OP2 to FPS2.
CpyO2ToFPS3	Copies nine bytes in OP2 to FPS3.
CpyO2ToFPS4	Copies nine bytes in OP2 to FPS4.
CpyO3ToFPST	Copies nine bytes in OP3 to FPST.
CpyO3ToFPS1	Copies nine bytes in OP3 to FPS1.
CpyO3ToFPS2	Copies nine bytes in OP3 to FPS2.
CpyO3ToFPS3	Copies nine bytes in OP3 to FPS3.
CpyO5ToFPS1	Copies nine bytes in OP5 to FPS1.
CpyO5ToFPS3	Copies nine bytes in OP5 to FPS3.
CpyO6ToFPST	Copies nine bytes in OP6 to FPST.
CpyO6ToFPS2	Copies nine bytes in OP6 to FPS2.

- Copies nine bytes from a FPS entry to RAM.

CpyStack If this routine is to be used, it is recommended that you create this routine in your APP/ASM.

```

;
; input: C = offset from (FPS) to start of 9
;         byte entry to copy. max = 252
;
;         ex: C = 9  -> FPST
;              18 -> FPS1
;
;         DE = pointer to 9 bytes of RAM to copy to
;
;
CpyfrFPS:
                LD          HL,(FPS)
                B_CALL      CpyStack

```

CpyFPST	Copies nine bytes from FPST to DE.
CpyFPS1	Copies nine bytes from FPS1 to DE.
CpyFPS2	Copies nine bytes from FPS2 to DE.
CpyFPS3	Copies nine bytes from FPS3 to DE.
CpyTo1FPST	Copies FPST to OP1.
CpyTo1FPS1	Copies FPS1 to OP1.

CpyTo1FPS2	Copies FPS2 to OP1.
CpyTo1FPS3	Copies FPS3 to OP1.
CpyTo1FPS4	Copies FPS4 to OP1.
CpyTo1FPS5	Copies FPS5 to OP1.
CpyTo1FPS6	Copies FPS6 to OP1.
CpyTo1FPS7	Copies FPS7 to OP1.
CpyTo1FPS8	Copies FPS8 to OP1.
CpyTo1FPS9	Copies FPS9 to OP1.
CpyTo1FPS10	Copies FPS10 to OP1.
CpyTo1FPS11	Copies FPS11 to OP1.
CpyTo2FPST	Copies FPST to OP2.
CpyTo2FPS1	Copies FPS1 to OP2.
CpyTo2FPS2	Copies FPS2 to OP2.
CpyTo2FPS3	Copies FPS3 to OP2.
CpyTo2FPS4	Copies FPS4 to OP2.
CpyTo2FPS5	Copies FPS5 to OP2.
CpyTo2FPS6	Copies FPS6 to OP2.
CpyTo2FPS7	Copies FPS7 to OP2.
CpyTo2FPS8	Copies FPS8 to OP2.
CpyTo3FPST	Copies FPST to OP3.
CpyTo3FPS1	Copies FPS1 to OP3.
CpyTo3FPS2	Copies FPS2 to OP3.
CpyTo4FPST	Copies FPST to OP4.
CpyTo5FPST	Copies FPST to OP5.
CpyTo6FPST	Copies FPST to OP6.
CpyTo6FPS2	Copies FPS2 to OP6.
CpyTo6FPS3	Copies FPS3 to OP6.

DRIVERS LAYER

The Drivers layer of the TI-83 Plus system includes such areas as the keyboard, the display, and the link port.

Keyboard

There are two ways to read key presses on the TI-83 Plus.

- Poll for scan codes directly.
- Use the system key read routine, **GetKey**.
- **Poll for scan codes**

This method is used in two different situations.

- When alpha or second functions located on the keyboard are not used in the application.
- When keys need to be recognized as fast as possible, this is usually used for game-type applications programming.
- See the Automatic Power Down™ (APD™) section.

This method will allow an application to know what physical key is pressed only.

- **This method will not support silent link activity.** Any link activity started by either another unit or a computer will not be detected by the system. Applications must poll for link activity on their own. **See the Link Port section later in this chapter.**

How it works:

- The system interrupt handler will look for key presses and when one is detected, write the scan code for that key to a RAM location. An application will then periodically check that RAM location for a scan code value.
- Interrupts must be enabled for the system to scan the keyboard in the background. This system flag must be reset:

indicOnly, (IY + indicFlags)

If this flag is set, then the interrupt handler will not scan the keyboard. This flag should only be set when the run indicator needs to be seen and no keyboard inputs are expected. Setting this flag will cause the interrupt service time to be shortened and overall execution faster.

- The **ON** key does not have a scan code assigned to it, the interrupt handler will set a flag if it is pressed. An application must check this flag to handle the **ON** key press.

Flag: **onInterrupt**, (IY + onFlags)

This flag should be reset by an application after detecting an **ON** key press. If it is not reset, an application will assume that the **ON** key had been pressed again. The interrupt handler does not reset this flag.

- The scan code values are equated in the include file named TI83plus.inc. Fig. 2.8 below shows the scan codes associated with their keys.

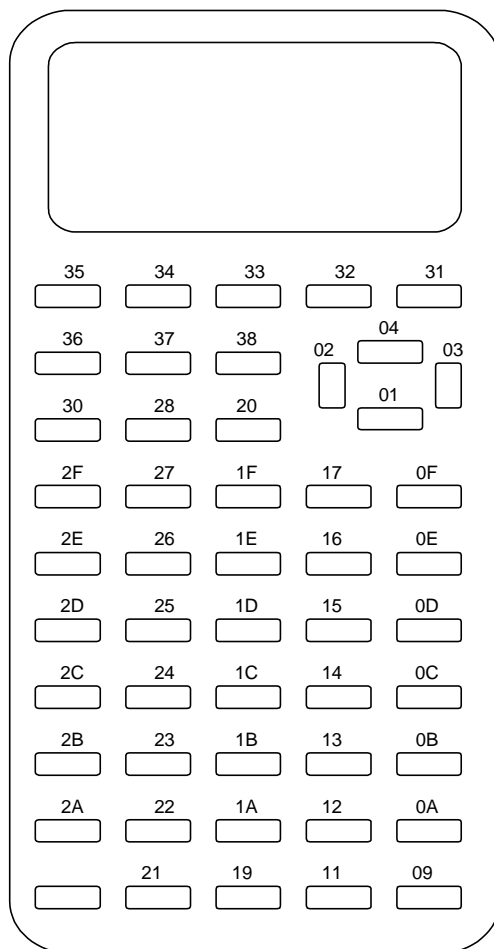


Fig. 2.8: Calculator Scan Code

Example one: This example will use the Z80 halt instruction to enter into low power mode, and upon waking up, will check:

- if a key had been pressed,
- check for the ON key being pressed,
- turn off the run indicator while waiting for a key, and
- disable APD while waiting and re-enable it after.

```

anykey:
    RES        indicOnly,(IY+indicFlags)    ; make sure keys are
                                           ; scanned
    B_CALL     RunIndicOff                  ; turn off run indicator
    RES        onInterrupt,(IY+onFlags)     ; reset On key flag
    RES        apdAble,(IY+apdFlags)        ; turn off APD
anykeylp:
    EI                                     ; turn on interrupts
    HALT                                              ; low power state
    BIT        onInterrupt,(IY+onFlags)     ; On key pressed
    JR         NZ,foundkey                  ; return if yes
;
    CALL       GetCSC                        ; local routine to look
                                           ; for scan code
    OR         A                               ; if non zero then have
                                           ; a scan code
    JR         Z,anykeylp                   ; jump if no scan code
                                           ; present
foundkey:
    SET        apdAble,(IY+apdFlags)        ; turn on APD
    RES        onInterrupt,(IY+onFlags)     ; reset On key flag
    RET
;
GetCSC:
    LD         HL,kbdScanCode
    DI                                     ; interrupts off
    LD         A,(HL)                        ; get possible scan code
    LD         (HL),0                        ; clear out for next
                                           ; scan
    RES        kbdSCR,(IY+kbdFlags)         ; needed for system
                                           ; key scan to work
    EI                                     ; interrupts on
    RET

```

Example two: This example will stay in a loop and make calls to read key, which will return:

- Z = 1 if no key found, Z = 0 if a key is detected,
- ACC = scan code of key, 0 = ON key
- run indicator will be running, and
- allow APD.

```

ex_2:
        B_CALL      RunIndicOn           ; turn on run indicator
        SET         apdAble,(IY+apdFlags) ; turn on APD
KeyLoop:
        RES         onInterrupt,(IY+onFlags) ; reset On key flag
;
; this part of the loop could be modifying the screen with
; animation of some kind, or doing other work while waiting for a key to
; be input.
;
        CALL        readKey              ; see if key pressed
        JR          Z,KeyLoop             ; jump if no key found
;
; here we have a key press, ACC = scan code, 0 = on key
;
        OR          A                    ; is it the on key ?
        JP          Z,Handle_On_Key       ; jump if yes
;
        CP          skEnter              ; enter key scan code ?
        JP          Z,Handle_Enter_key
;
; check for rest of keys that matter . . .
;
;
;
readkey:
        RES         indicOnly,(IY+indicFlags) ; make sure keys are
                                                ; scanned
        EI          ; turn on interrupts
        CALL        GetCSC               ; local routine to look
                                                ; for scan code
        BIT         onInterrupt,(IY+onFlags) ; On key pressed
        JR          Z,notOnkey
;
        LD          A,0                  ; scan code for on key,
                                                ; Z = 0 from test
        RET
notOnkey:
        OR          A                    ; any scan code found
        RET          ; Z = 1 if no key, else
                        ; Z = 0

```

- **Use the system key read routine, GetKey.**

This method is used when the alpha and second functions on the keyboard are valid inputs to the applications.

- Unlike polling for scan codes which returns only one value for each key on the keyboard, this routine could possibly return up to four different values for the same key. Depending what key modifiers, alpha and second, may have been activated.
- See the Automatic Power Down (APD) section.
- This method will support silent link activity. Any link activity started by either another unit or a computer will be detected by the system. If the TI-GGRAPH LINK attempts transfer a variable to/from the TI-83 Plus, the application will be shut down. See the following example.
- The pull down menu system is not controlled by this routine — the key value of the menu will be returned but the menu will not activate.

How it works:

- Interrupts must be enabled.
- The **ON** key flag should be reset before calling.

onInterrupt, (IY + onFlags)

- This system flag must be reset:

indicOnly, (IY + indicFlags)

If this flag is set, the interrupt handler will not scan the keyboard. This flag should only be set when the run indicator needs to be seen and no keyboard inputs are expected. Setting this flag will cause the interrupt service time to be shortened and overall execution faster.

- Make a B_CALL to **GetKey**.
- Control remains in **GetKey** until a returnable key entry is pressed, the unit is turned off, or link activity has caused the application to be put away.
- The key presses that are not returned are [ALPHA] and [2nd].
- The key code is returned in the ACC.

- The **ON** key has a key code of 0 and the flag indicating that it was pressed is also set.

onInterrupt, (IY + onFlags)

- The key code returned can be either one or two bytes. The ACC is checked to see if a one or two byte key code is returned.

There are two values returned that signal a two byte key code:

kExtendEcho and **kExtendEcho2**

There is a table for each of these keys that list the second byte key values associated with them which can be found in the include file, TI83plus.inc.

If either of the above values are returned, the second byte of the key code is located in the RAM location (**keyExtend**).

For example, the key code for **DrawF** are the two bytes **kExtendEcho** and **kDrawF**. **GetKey** would return the **ACC = kExtendEcho** and (**keyExtend**) = **kDrawF**.

- Lowercase Alpha keys

When the following flag is set, consecutive presses of the **ALPHA** key will become the mechanism for lowercase alpha key entry.

lwrCaseActive, (IY + appLwrCaseFlag)

This flag should be reset when lowercase is not needed. It should also be reset before exiting the application.

The lowercase alpha keys are two byte key codes with the first byte being **kExtendEcho2**.

For example, use the **GetKey** routine to input only keys A-Z until either **ENTER** or **ON** is pressed.

```

Enter_Alphas:
    B_CALL    RunIndicOff          ; no run indicator
    RES       indicOnly,(IY+indicFlags) ; make key reads are
                                           ; done
    B_CALL    DisableApd           ; no auto power down

keyLoop:
    RES       onInterrupt,(IY+onFlags) ; clear on pressed
    EI
    B_CALL    GetKey               ; wait for a key
;
    RES       onInterrupt,(IY+onFlags) ; clear on pressed
    OR        A                   ; on key ?
    JR        Z,Return             ; yes return
;
    CP        kEnter
    JR        Z,Return             ; jump if Enter key
;
    CP        kCapZ+1              ; possible A-Z
    JR        NC,keyLoop           ; no ignore
;
    CP        kCapA
    CALL      NC,StoreKey           ; store it if A-Z
    JR        keyLoop              ; look for more
;
Return:
    B_CALL    EnableApd            ; auto power down is
                                           ; enabled
    RET

```

Display

There are two methods to access the TI-83 Plus display.

- Using system routines for displaying characters and strings.
- Writing directly to the display driver that controls what is displayed (advanced).

Note: See the Graphing and Drawing section also.

Displaying Using System Routines

WARNING: Most of the TI-83 Plus system display routines will disable interrupts which results in no keyboard scans, run indicator updates, APD, or cursor updates. Applications must re-enable interrupts (**EI**), if needed.

Display Utility Routines

ClrLCD	Clears the display. The split screen setting is checked to determine how much of the display to clear.
ClrLCDFull	Clears the entire display while ignoring the split screen setting.
ClrScrn	Clears the display and the text shadow buffer. The split screen setting is checked to determine how much of the display and buffer to clear.
ClrScrnFull	Clears the display and the text shadow buffer while ignoring the split screen setting.
ClrTxtShd	Clears the entire text shadow buffer.
SaveScreen	Copies a bit image of the current display to RAM.
DisplayImage	Displays a bit map image.
RunIndicOff	Disables the run indicator located in the upper right corner of the display. See the Run Indicator section for further information.
RunIndicOn	Enables the run indicator located in the upper right corner of the display. See the Run Indicator section for further information.

Displaying Text

The display is made up of 64 rows of 96 pixels. The TI-83 Plus has two sets of routines that display text. The difference between the two sets of routines is how the text position in the display is specified. The following are two distinct mappings of the display, home screen and pen display.

- **Home Screen Display Mapping**

This mapping corresponds to the positioning of text that the home screen context uses. The display is mapped out to eight rows of 16 characters.

		penCol															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
penRow	0																
	1																
	2																
	3																
	4																
	5																
	6																
	7																

Fig. 2.9: Home Screen Display Mapping

- **Two bytes of RAM are used to position text written:**

- (curRow) = row coordinate (0 – 7)
- (curCol) = column coordinate (0 – 15d)

- **Font**

- 5 (width) x 7 (height) (pixels) large characters

- **Text formatting**

- **Reverse video:**

Display all text written in reverse video:

textInverse, (IY + textFlags); default = 0

- **Auto scroll:**

If the bottom of the screen is reached:

appAutoScroll, (IY + appFlags); default = 0

- **Echo characters to a RAM buffer:**

textShadow is a RAM buffer of 128 bytes, one byte for each character location. Character font codes will be written to this buffer as characters are sent to the display. The location in the buffer written to is determined by the location the character is placed in the display. This can be used to restore display contents quickly when using Home Screen Display Mapping text routines:

appTextSave, (IY + appFlags); default = 1

- **Preclear character space before writing a character:**

This option is used when text is written to the same location alternating between reverse/normal video:

preClrForMode, (IY + newDispF); default = 0

- **All of these settings remain until you change them.** Applications need to manage their state, if they are changed.

– **Entry Points**

PutMap	Displays a single character without updated cursor position.
PutC	Displays a single character and advances the cursor position.
PutS	Displays a zero (0) terminated string stored in RAM and updates the cursor position.
PutPS	Displays a string stored in RAM with its length being the first byte and updates the cursor position.
DispHL	Displays the value stored in HL.
ClrTxtShd	Clears the text shadow buffer.
EraseEOL	Writes spaces from (curCol) to end of the line.
OutputExpr	Positions the cursor and display a numeric value, a string, or an equation.
PutTokString	Displays a function token's string.

Note: The **VPutS** and **VPutSN** routines can be used without first copying strings to RAM by having a version of themselves place into the application. See Appendix A for the source code to these routines.

See the Display Utility Routines section.

See the Formatting Numeric Values for Display section.

See Appendix A for more details.

- **Pen Display Mapping**

This mapping is based on individual pixel locations. It is used mainly in the graph context for displaying text on a graph, but is also used in the statistics edit context to display list elements. The display is mapped out to 64 rows of 96 pixels.

		penCol												
		0	1	2	3	4	5	...	90	91	92	93	94	95
penRow	0							...						
	1							...						
	2							...						
	.							.						
	.							.						
	.							.						
	62							...						
63							...							

Fig. 2.10: Pen Display Mapping

- **Two bytes of RAM are used to position text written:**

- (penCol) = column coordinate (0 – 95d)
- (penRow) = row coordinate (0 – 63d)

The pen location specified represents the upper left most pixel of the character being displayed.

- **Fonts**

- 5 (width) x 7 (height) (pixels) large characters.
- 6/7 pixel high by variable-width small characters.
- Application defined custom characters.

- **Text formatting**

- **Reverse video:**

Display all text written in reverse video:

textInverse, (IY + textFlags); default = 0

- **Write to Graph backup buffer:**

The output can be directed to either the display, or the graph backup buffer, *plotSScreen*.

textWrite, (IY + sGrFlags) = 1 to write to buffer; default = 0

- **Use 5x7 large font:**

The default is to use the small variable width font. Set the below flag to use the large 5x7 font.

fracDrawLFont, (IY + fontFlags); default = 0

- **Erase the line below the character being displayed:**

This applies to the small variable width font only. Do not set this flag if the row of pixels below the character being displayed is off of the display.

textEraseBelow, (IY + textFlags); default = 0.

- **Display an application defined custom character:**

This option is only used with the **UserPutMap** routine.

customFont, (IY + fontFlags)

- **All of these settings remain until you change them.** Applications need to manage their state, if they are changed.

– **Entry Points**

VPutMap	Displays either a small variable width or large 5x7 character at the current pen location and updates penCol.
VPutS	Displays a zero (0) terminated string, using either small or large characters and updates penCol.
VPutSN	Displays a string whose length is the first byte using either small or large characters and updates penCol.
VPutBlank	Displays a space character at the current pen location using the small or large font and updates penCol.
DispOP1A	Rounds a floating-point number to the current fix setting and display it at the current pen location. Uses either the small or large characters and updates penCol.
SStringLength	Returns the width in pixels of a string using the small font.
SFont_Len	Returns the width in pixels of a character using the small font.
UserPutMap	Displays a character defined by an application at the current pen location and updates penCol.

Note: The **VPutS** and **VPutSN** routines can be used without first copying strings to RAM by having a version of themselves place into the application. See Appendix A for the source code to these routines.

Note: The space character for the small font is only one pixel wide. Applications may want to use two space characters to separate words, in strings to be displayed using the small font.

See the Display Utility Routines section.

See the Formatting Numeric Values for Display section.

See Appendix A for more details.

Formatting Numeric Values for Display

The following routines are used to convert RealObj (single floating-point) and CplxObj (pair of floating-points) values into displayable strings. These routines do not display the string.

Entry Points

- | | |
|------------------|--|
| FormReal | Converts a RealObj in OP1 into a displayable string and specify the maximum width allowed for the string. If the current mode setting is SCI or ENG, the output string will reflect the setting. The value will be Rounded based on the maximum width entered and the current FIX setting. |
| FormBase | Converts a RealObj in OP1 into a displayable string. Uses the current mode settings SCI, ENG, NORMAL, and FIX settings to format the string. The output can also be formatted as a fraction, or a degrees-minutes-seconds (DMS) number. If a value cannot be represented in the desired format, it defaults back to decimal. |
| FormEReal | Converts a RealObj in OP1 into a displayable string and specify the maximum width allowed for the string. All mode settings are ignored. |
| FormDCplx | Converts a CplxObj value in OP1/OP2 into a displayable string. Uses the current mode settings SCI, ENG, NORMAL, FIX setting, and complex output settings a + bi and re^θi to format the string. The output can also be formatted as a fraction or a degrees-minutes-seconds (DMS) number. If a value cannot be represented in the desired format, it defaults back to decimal. |

See Appendix A for further information.

Modifying Display Format Settings

Resetting the next two flags signifies NORMAL mode setting.

fmtExponent, (fmtFlags) = 1 for scientific display mode

fmtEng, (IY + fmtFlags) = 1 for engineering display mode

fmtRect, (IY + numMode) = 1 rectangular complex display mode

fmtPolar, (IY + numMode) = 1 polar complex display mode

Fix setting:

(fmtDigits) = 0FFh for FLOAT, no fix setting

= 0 – 9 if a fix setting is specified

Writing Directly to the Display Driver

The display driver is a device that controls the display. The driver contains RAM that represents what is currently being displayed. Commands are sent to the driver to modify, or access what is displayed. The following is a brief description of the commands that control the driver which is the Toshiba T6A04.

- Driver RAM

The RAM on the driver is mapped to a grid of 64 rows of 12 bytes. Each row represents a row of pixels in the display with each byte representing eight pixels.

The addressing of the RAM is done by setting a row and column value to address a particular byte. The addressing is built into the command used to set either a row or column value. The figure below shows the command values used to set either a row (X) or column (Y) value.

	20h	21h	Y Direction	2Bh
X Direction	80h			
	81h			
	BFh			

Fig. 2.11: Command Values

The first byte — row 80h and column 20h — represents the eight pixels in the first row of the display's left edge. The most significant bit of the byte is the left most pixel.

- Sending Commands

The following areas must be considered when sending commands.

- Interrupts should be disabled to send commands/data to the driver.

- A delay is necessary before each communication with the driver. The following routine should be added to an application and used.

```

lcd_busy:
        PUSH    AF
        INC     HL
        DEC     HL
        POP     AF
        RET

```

- Communication is done with the drive through two IO ports:

lcdinstport = 10h command port

lcddataport = 11h data port

- Addressing a byte of RAM

- **Row (X) addressing**

Commands 80h to BFh — sets the row address to 0 – 63 or top to bottom rows.

Top Row

```

        LD      A,80h                ; top row
        CALL    lcd_busy
        OUT     (lcdinstport),A

```

Bottom Row

```

        LD      A,0BFh              ; last row
        CALL    lcd_busy
        OUT     (lcdinstport),A

```

- **Column (Y) addressing**

Commands 20h to 2Bh — sets the column address to 0 – 0Ch.

First byte of row

```

        LD      A,20h                ; first byte of row
        CALL    lcd_busy
        OUT     (lcdinstport),A

```

Last byte of row

```

        LD      A,2Bh                ; last byte of row
        CALL    lcd_busy
        OUT     (lcdinstport),A

```

- Setting auto addressing modes. The driver can act in four different ways after a read or write.

Command 05h — X Direction auto increment

Command 07h — Y Direction auto increment

Command 04h — X Direction auto decrement

Command 06h — Y Direction auto decrement

The TI-83 Plus system expects the driver to be in X-increment mode and must be set to this mode before giving control to the system.

- Reading the Contents of the Display Driver RAM

```
CALL    lcd_busy
IN      A,(lcddataport)  ; read disp byte that X and Y
                        ; settings point to
```

Reading the Display Driver After Setting X or Y Coordinates

A dummy read needs to be done after setting either the x or y coordinate of the driver if one wants to read from the driver. For example, read nine bytes of data from the display starting in LCD row 5, column 1, to OP1.

```
LD      A,85h
CALL    lcd_Busy
OUT     (LcdInstPort),A      ; set X to row 5
;

LD      A,07h
CALL    lcd_Busy
OUT     (LcdInstPort),A      ; set Y auto increment mode
;

CALL    lcd_busy
LD      A,21h
OUT     (LcdInstPort),A      ; set Y to column 1
;

LD      B,9                  ; number of bytes to read
LD      HL,OP1
CALL    lcd_busy
IN      A,(lcdDataPort)      ; dummy read since we changed
                        ; X, Y position

Lp:
CALL    lcd_busy
IN      A,(lcddataPort)      ; read byte, auto increment Y
;

LD      (HL),A
INC     HL
DJNZ    ..Lp
;

LD      A,05h
CALL    lcd_Busy
OUT     (LcdInstPort),A      ; set X auto increment mode
```

- Writing to the display driver RAM

```
CALL    lcd_busy
OUT     (lcddataport),A      ; write byte to disp
```

For example, write the contents of the graph backup buffer, *plotSScreen*, to the display.

```

        DI
        LD      HL,plotSScreen
        LD      B,64
        LD      A,07h
        CALL    lcd_busy
        OUT     (lcdinstport),A      ; set to y INC mode
        LD      A,7fh                ; first row
;
; new row
;
loop1:
        PUSH    BC                  ; save number rows left to copy
        INC     A                   ; move to next row
        LD      (curXRow),A         ; save new row
        CALL    lcd_busy
        OUT     (lcdinstport),A     ; set new x
        LD      A,20h
        CALL    lcd_busy
        OUT     (lcdinstport),A     ; set to first column
        LD      B,12                ; 12 columns
..loop2:
        LD      A,(HL)              ; get source
        INC     HL
        CALL    lcd_busy
        OUT     (lcddataport),A     ; write to disp
        DJNZ    ..loop2            ;
;
; row done
;

        POP     BC                  ; get number rows left
        LD      A,(curXRow)
        DJNZ    ..loop1            ; decrease number left, jump if
;                                   ; not done

        LD      A,05h
        CALL    lcd_busy
        OUT     (lcdinstport),A     ; set to x INC mode
        EI
        RET

```

Contrast Control

Adjusting the contrast setting of the display from an application can be done in two ways.

- Executing the system **GetKey** routine will allow normal adjusting of the contrast by the user, using the **2nd** **▲** and **▼** keyboard keys.
- The display driver controls the contrast level of the display. Applications can send a new contrast setting to the display driver.

Below is an example of how to send a contrast setting command to the display driver.

```

;
; accumulator = valid contrast value 18h to 3Fh
;
; let us set the contrast to its darkest

LD      A,3Fh
OR      0C0h          ; or in LCD contrast command
CALL    lcd_busy      ; delay for LCD driver
OUT     (lcdinstport),A ; set contrast
RET

```

Note: Adjusting the contrast directly will not affect the systems contrast RAM value. The new contrast setting will only be in effect temporarily. In order to make the new setting permanent the systems contrast value must be updated. The system's contrast value ranges from 0 to 27h, and is stored in RAM location (contrast). Display driver setting minus 18h = (contrast).

Split Screen Modes

The TI-83 Plus has three mode settings that define the size of the display, Full screen, Horizontal split and Graph-Table (vertical split). All of the system display writing and graph utility routines adjust for the current split mode setting.

Applications need to be aware of the current split screen setting and take steps to ensure that the current setting will not alter the intended output to the display.

Applications that do not intend to take advantage of a split screen have two ways to avoid problems.

- Temporarily change the screen setting to full screen and then reset it. This option is chosen if an application wants to retain the current split screen setting after completion.

The current split screen settings are saved in some application defined RAM locations (six bytes in length). Then the setting is changed to full screen mode. The application must restore the original split screen settings back to the input state upon completion. The following routines will save the current split screen setting and restore it.

```

setToFull:
    LD      HL,YOffset          ; address of split
                                ; attributes
    LD      DE,savevals        ; app defined RAM
                                ; location to save
    LD      BC,5                ; save first 5 bytes
    LDIR                                ; save split
                                ; attributes
;
    LD      A,(IY+sGrFlags)     ; split flags ->
                                ; ACC
    LD      (DE),A              ; save split flags
                                ; in 6th byte
;
    RES     grfSplit,(IY+sGrFlags)
    RES     vertSplit,(IY+sGrFlags) ; set flags to
                                ; Full screen
;
    B_CALL  SetNorm_Vals        ; screen attributes
                                ; to full
    SET     grfSplitOverride,(IY+sGrFlags)
    RET
;

rstrYOffset:
    RES     grfSplitOverride,(IY+sGrFlags)
    LD      DE,YOffset
    LD      HL,savevals
    LD      BC,5
    LDIR                                ; restore input
                                ; screen attributes
    LD      A,(HL)              ; get input split
                                ; flags
    LD      (IY+sGrFlags),A     ; restore
    RET

```

- Change the split screen mode to full screen mode without restoring it back to the input setting.

```

B_CALL  ForceFullScreen

```

Note: The B_CALL routine was not used in the first option above so that the graph would not be marked dirty. If the split screen mode is not temporarily changed, the graph needs to be marked as dirty so it will reflect the new screen size. Example one restores the input setting, so no regraph is necessary. It is entirely up to the application if causing the regraph is a concern or not.

Graphing and Drawing — What's the difference?

Drawing

Routines include lines, circles, points, etc., which are defined by pixel coordinates. Drawing routines cannot be defined with points outside of the physical display area. Only pixel coordinates that exist can be used. The current WINDOW settings have no affect on the drawing routine's output. Inputs to routines are normally byte values.

Applications use drawing routines for general purpose drawing and animation. They are easier to use and are more efficient than graphing routines that can generate the same output. Drawing routines can also be used to annotate graphs generated by the systems grapher.

Graphing

These routines include system grapher, lines, circles, points etc., which are all drawn with respect to the current WINDOW settings, Xmin, Xmax, Ymin, and Ymax. These settings define the boundaries of the display. Unlike drawing routines, graphing routines can be defined with points that reside outside of the current WINDOW settings.

Graphing routines would be used by applications that want to annotate in a way that is determined by the current WINDOW settings.

Graphing and Drawing Utility Routines

These routines could be useful to applications in combination with some of the graphing and drawing routines. Detailed information for each of these routines can be found in Appendix A.

BufClr	Clears a RAM display buffer representing a bit image of the display. Similar to GrBufClr except the address of the RAM display buffer is input.
BufCpy	Displays a RAM display buffer representing a bit image of the display. Similar to GrBufCpy except the address of the RAM display buffer is input.
GrBufClr	Clears the graph backup buffer, <i>plotSScreen</i> . The portion of the buffer cleared is determined by the split mode setting.
GrBufCpy	Displays the graph backup buffer, <i>plotSScreen</i> . The portion of the buffer displayed is determined by the split mode setting.
ClrLCD	Clears the display and the split screen setting is checked to determine how much of the display to clear.
ClrLCDFull	Clears the entire display ignoring the split screen setting.

SaveScreen	Copies a bit image of the current display to RAM.
DisplayImage	Display a bit map image.
RunIndicOff	Disables the run indicator located in the upper right corner of the display. See the Run Indicator section for further information.
RunIndicOn	Enables the run indicator located in the upper right corner of the display. See the Run Indicator section for further information.
AllEq	Selects or deselects all graph equations in the current graph mode
SetAllPlots	Selects or deselects all statplots.
SetTblGraphDraw	Sets the graph to dirty, which causes a complete regraph the next time the graph is brought to the display.

Drawing Routine Specifics

The following sections cover drawing pixel coordinates, drawing to a split screen, and drawing routines.

- Drawing pixel coordinates

The display is 96 pixels wide by 64 pixels high.

Fig. 2.12 shows the layout of the pixels along with the X and Y coordinate scheme used by drawing routines.

		X Coordinate							
		0	1	2	...	92	93	94	95
Y Coordinate	63				...				
	62				...				
	61				...				
	.				.				
	.				.				
	.				.				
	2				...				
	1				...				
	0								

Fig. 2.12: Pixel Coordinates

Coordinates are input to drawing routines mainly in a register pair such as BC, where BC = (X,Y) drawing pixel coordinate.

For example, the upper top left pixel in the display is drawing pixel coordinates (0,63); (X,Y).

Note: The drawing routines, by default, DO NOT use the last row of pixels, Y = 0 and the last column of pixels, X = 95. This is done to allow for an odd number of pixels for both the X and Y axes. This restriction can be overridden thus allowing for the drawing routines to make use of the entire display.

- Drawing in a split screen

If either Horizontal or Vertical (G-T) split screen is the current mode, the output from the drawing routines will be affected. Listed below are the effects of each split mode.

Horizontal Valid Y pixel range = 1 – 31, where Y-pixel row 1 is moved up 32 rows from its normal position.

Vertical (G-T) Valid Y pixel range = 1 – 51, where Y-pixel row 1 is moved up 12 rows from its normal position.

Valid X pixel range = 0 – 31, with X-pixel column 0 in its original position.

If split screen modes are not required by an application, it is recommended that all drawing routines be performed with no split modes set. See the Split Screen section for further information.

- System flags associated with drawing routines

The following flags are input by most of the drawing routines. The table gives an overview of some the options available to applications. Appendix A contains further information.

fullScrnDraw, (IY + apiFlag4)	1 = allows draws to use column 95 and row 0.
plotLoc, (IY + plotFlags)	0 = draws affect both the display and the graph backup buffer plotSScreen . 1 = draws affect only the display.
bufferOnly, (IY + plotFlag3)	1 = draws affect the graph backup buffer plotSScreen only.

- Drawing routines

The descriptions given below refer to affecting a pixel coordinate location in the display, however the system flags above can be used to affect **plotSScreen**. Appendix A contains further information.

IPoint Performs one of the following operations to a pixel coordinate point: darken, lighten, reverse, test, or copy from **plotSScreen** to display.

PointOn Darkens a pixel coordinate point.

ILine Darkens or lightens a line between two pixel coordinate points.

DarkLine	Darkens a line between two pixel coordinate points.
PixelTest	Tests a pixel coordinate in <i>plotScreen</i> , to see if it is set.
GrphCirc	Draws a circle, given the pixel coordinates, of the center and a point on the circle.
IBounds	Tests if a pixel coordinate lies within the graph window defined by the current mode settings.
IBoundsFull	Tests if a pixel coordinate lies within the full pixel range of the display.
IOffset	<p>Given a pixel coordinate point, computes the offset to add to the start address of the graph buffer to the byte in the buffer containing that pixel.</p> <p>Also returns the bit number in that byte for that pixel.</p> <p>Additionally, computes the row and column commands to set the LCD driver to the display byte for that pixel.</p>

Graphing Routine Specifics

The following section covers graph WINDOW settings, graphing in a split screen, and graphing routines and system flags.

Graph WINDOW Settings

Fig. 2.13 below shows how the graph window is bounded by the current WINDOW settings.

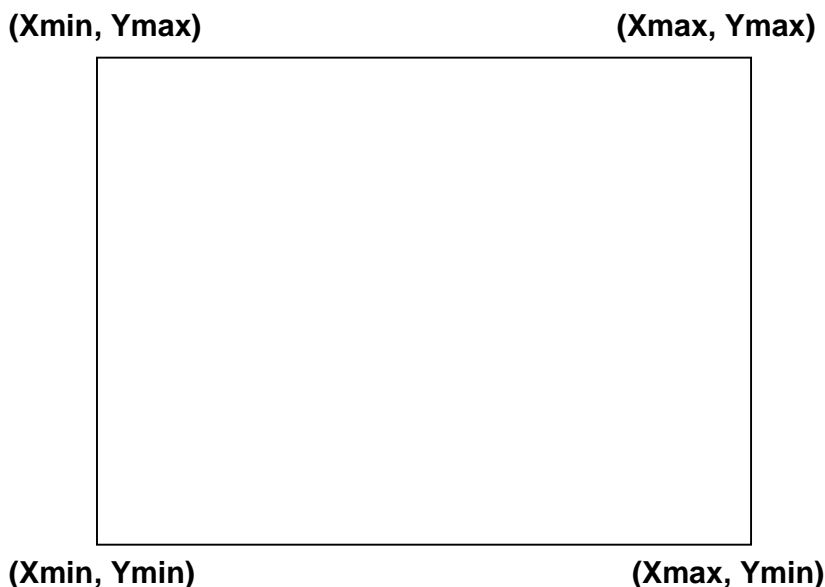


Fig. 2.13: Graph WINDOW Setting

Graphing routine parameters (points) can be defined outside of the WINDOW settings. Those settings only define what is currently viewed in the display.

Graphing in a Split Screen

If either Horizontal or Vertical (G-T) split screen is the current mode, the graphing routines will be limited to the section of the display designated for graphing by the mode setting.

For more information about disabling any split screen, see the Split Screen section of this document.

Graphing Routines and System Flags

The graphing routines are grouped by common attributes into four groups. See Appendix A for further information.

- Routines that do not automatically display or redraw the current graph screen. These routines will draw over the existing contents of the display.

- System flags

plotLoc, (IY + plotFlags) 0 = draws affect both the display and the Graph backup buffer, **plotSScreen**.

1 = draws affect the display only.

bufferOnly, (IY + plotFlag3) 1 = draws affect the graph backup buffer **plotSScreen** only.

- Entry Points

CPoint Darkens, lightens, or reverses a graph coordinate point defined in OP1/OP2.

CPointS Darkens, lightens, or reverses a graph coordinate point defined in FPS1/FPST.

CLine Darkens a line between two graph coordinate points defined in OP1/OP2 and OP3/OP4.

CLineS Darkens a line between two graph coordinate points defined in FPS3/FPS2 and FPS1/FPST.

UCLineS Erases a line between two graph coordinate points defined in FPS3/FPS2 and FPS1/FPST.

DarkPnt Darkens a graph coordinate point defined in OP1/OP2.

DrawCirc2 Draws a circle given the center, a graph coordinate point in FPS2/FPS1, and the radius in FPST.

- Routines that will automatically display or redraw the current graph screen before executing. If the graph does not need to be redrawn, the contents of the graph backup buffer, **plotSScreen**, are copied to the display.

- System flags

bufferOnly, (IY + plotFlag3) 1 = draws affect the graph backup buffer **plotSScreen** only.

- Entry Points

Regraph Graphs any select equations in the current graph mode, and also any selected statplots.

PDspGrph	Tests if the graph of the current mode needs to be redrawn. If so, call the Regraph routine, otherwise copies <i>plotSScreen</i> to the display.
PointCmd	Darkens, lightens, or reverses a graph coordinate point defined in (FPS2, FPS1).
LineCmd	Darkens a line between two graph coordinate points defined in (FPS3, FPS2) and (FPS1, FPST).
UnLineCmd	Erases a line between two graph coordinate points defined in (FPS3, FPS2) and (FPS1, FPST).
DrawCmd	Graphs an equation variable in FPST.
InvCmd	Graphs an equation variable in FPST along the Y-axis instead of the X-axis.
CircCmd	Draws a circle given the center, a graph coordinate point in (FPS2, FPS1), and the radius in FPST.
VertCmd	Draws a vertical line at the X value in FPST.
HorizCmd	Draws a horizontal line at the Y value in FPST.

- WINDOW zooming routines, which automatically display or redraw the current graph screen, will not redraw after changing the window settings.

– Entry Points

Change the WINDOW settings such that:

ZooDefault	The default settings are set, (-10,10) for both the X and Y ranges.
ZmFit	All selected functions are fully visible in the display.
ZmInt	ΔX and $\Delta Y = 1.0$ given a new center (OP1, OP5).
ZmPrev	The settings that were set before the latest zoom.
ZmSquare	$\Delta X = \Delta Y$, either the X ,or Y window settings are changed.
ZmStats	All selected statplots are fully visible in the display.
ZmTrig	Appropriate for graphing trig functions dependent upon the current trig mode.
ZmUsr	The settings that were saved by the last ZoomSto executed.
ZmDecml	(0,0) is in the center and ΔX and $\Delta Y = .1$.

- Routines that change the current graph mode.
 - Entry Points

SetFuncM	Switches to function mode.
SetParM	Switches to parametric mode.
SetPolM	Switches to polar mode.
SetSeqM	Switches to sequence mode.

Run (Busy) Indicator

The run indicator is used by the TI-83 Plus to indicate that the calculator is busy while computing. It is normally turned off while waiting for input from a user. When an application is first started, the run indicator will most likely be running.

Applications have the option of using the indicator or not.

The indicator is updated by the interrupt handler, so if it is to be used, interrupts need to be enabled.

RunIndicOff Disables the run indicator located in the upper right corner of the display.

RunIndicOn Enables the run indicator located in the upper right corner of the display.

There are two choices for the appearance of the run indicator:

- A short solid line that circles around from top to bottom — this is the default indicator.
- A long dashed line that circles around from top to bottom — this is the Pause indicator for the TI-83 Plus.

To use the Pause indicator, execute the following code before turning the run indicator on:

```
LD      A,busyPause
LD      (indicBusy),A
```

If the Pause indicator is used, an application needs to set the default indicator back:

```
LD      A,busyNormal
LD      (indicBusy),A
```

Example of common usage:

```
EI
B_CALL  RunIndicOn      ; indicator on
B_CALL  GetKey           ; wait for a key
B_CALL  RunIndicOff     ; indicator off
```

APD™ (Automatic Power Down™)

Applications have the choice of allowing the APD feature of the TI-83 Plus to be active or not. APD is implemented to preserve battery life by turning the calculator off after about four minutes of inactivity. Unless an application's functionality absolutely requires that APD be disabled, it should be left active.

How does APD work?

Under normal system operation, the APD counter is reset after each key press. If no key press is made in approximately four minutes, the calculator powers down.

Similar to the run indicator, the APD counter is updated by the interrupt handler; therefore, interrupts must be enabled. When the APD counter is exhausted, the calculator turns off. The interrupt handler routine is not exited.

The application is not notified that the calculator has been turned off. The contents of the screen are saved in the 768 bytes of RAM located at **saveSScreen**, which is a bit image representation of the screen.

When the calculator is turned back on, the screen is restored and the interrupt handler is exited. Execution resumes at the location of the last interrupt before the calculator is powered down. Applications should not be affected by this event in any way.

- **Resetting the APD counter**

This routine will reset the APD counter.

```
B_CALL      ApdSetup
```

The **GetKey** routine will make a call to this routine upon entry.

- **Disabling APD**

There are two ways to disable APD and each have a specific situation in which they should be used.

- Disable APD when calling the **GetKey** routine.

```
B_CALL      DisableApd
```

This method of disabling the APD is a global, and will stay in effect after an application exits. Applications need to re-enable the APD before exiting.

```
B_CALL      EnableApd
```

- Disable APD while executing outside of the **GetKey** routine.

```
RES          apdRunning, (IY+apdFlags)
```

APD will be disabled until this flag is set, or the **GetKey** routine is called.

Link Port

Communications to and from the TI-83 Plus calculator is possible through the I/O port using the unit-to-unit cable (included with the unit) or the graphic link cable (available as an option).

Applications can use the link port for transferring data on two different levels.

- Using system routines that send/receive TI-83 Plus variables using the systems link protocol. There are three system routines that are used:

AppGetCalc Retrieves a variable from a TI-83 Plus or TI-83 calculator.

AppGetCbl Retrieves a variable from a Calculator Based Laboratory™ (CBL™) or Calculator Based Ranger™ (CBR™) device.

SendVarCmd Sends a variable to a CBL or CBR device.

The **AppGetCalc** and **AppGetCbl** routines will automatically replace existing variable data if the variable received does exist already.

No error handler is needed to be placed around calls to these routines. If any error occurs, a flag is returned to indicate that the link operation failed. Nothing more specific about the error is known.

See Appendix A for more details.

For example, assume that L1 contains a list to set up the CBR to continuously poll for data using one of its probes, sends the list to the CBR, and polls it for data.

```

CALL      llname                ; L1
RES       onInterrupt,(IY+onFlags) ; clear break
B_CALL    SendVarCmd            ; send L1 to start up
                                ; CBR
BIT       comFailed,(IY+getSendFlg) ; fail ?
RET       NZ                    ; return if yes
;
; loop and read data into OP1
;
read_Loop:
CALL      GetNewValue           ; try to get another
                                ; value
RET       NZ                    ; ret if link failed
CALL      StoreData             ; store data somewhere
JR        Read_Loop
;
; get from CBL into var L1 and recall to OP1
;
GetNewValue:
CALL      llname                ; L1
B_CALL    AppGetCbl             ; get data
BIT       comFailed,(IY+getSendFlg) ; fail ?
RET       NZ                    ; yes
;
; RCL L1(1) -> OP1
; ACC = size of list, 1 = CBL, 2 = CBR
;
Rcl_new_val:
CALL      llname
RST       rFindSym              ; look up L1 in symbol
                                ; table
;
INC       DE
INC       DE                    ; move past size bytes
EX        DE,HL                 ; HL = pointer to
                                ; element 1
RST       rMov9ToOP1            ; OP1 = val
RET
;
Llname:
LD        HL,Lllname
RST       rMov9ToOP1            ; OP1 = L1 name
RET

```

- Send and receive bytes of data directly through the port.

This operation involves the application interpreting the data sent and received in a custom format. This type of communication is for applications that either interacts with another TI-83 Plus or computer without using the built-in messaging protocol, which is not documented in this developer's guide.

The TI-83 Plus link port uses two data lines, D0 and D1, for communicating. These data lines are accessed through the B-port of the Z80.

- Bits 0 and 1 are for writing/reading data, D0 = bit 0, D1 = bit 1.

For example, the following code shows all of the values that can be written to the B-port.

```
LD      A,D0LD1L
OUT     (bport),A    ; is used for setting d0 low, d1 low

LD      A,D0LD1H
OUT     (bport),A    ; is used for setting d0 low, d1 high

LD      A,D0HD1L
OUT     (bport),A    ; is used for setting d0 high, d1 low

LD      A,D0HD1H
OUT     (bport),A    ; is used for setting d0 high, d1 high
```

Note: Data lines are high when not in use.

For example, the code below will poll the B-port until it detects some activity and then examine which line has the activity.

```
IN      A,(bport)      ; poll the b-port
CP      D0D1_bits      ; any data line go low ?
JR      Z,no_activity  ; jump if no activity detected

;

CP      D0HD1L         ; is d0 high ?
JR      Z,d0_low       ; yes,

;
; else d1 is high
;
```

The following systems routines are used for polling the link and sending/receiving a byte of data.

- | | |
|---------------------|---|
| Rec1stByte | Polls the link port for activity until either a byte is received, the ON key is pressed, or an error occurs during communications. The cursor will be turned on by this routine. |
| Rec1stByteNC | Polls the link port for activity until either a byte is received, the ON key is pressed, or an error occurs during communications. The cursor is not activated by this routine. |
| RecAByteIO | Attempts to read a byte of data. If no activity is detected in about 1.1 seconds, an error occurs. |
| SendAByte | Attempts to send a byte of data. If no activity is detected in about 1.1 seconds, an error occurs. |

An error handler should be set when using these routines. Each of these routines will generate system errors.

See Appendix A for more details.

Example one:

The following routine is called to do a spot check of the link port for activity for a single byte of data being sent.

- If no activity is detected or any error occurs during communication, then Z = 0 is returned.
- If activity is detected, then the signal is debounced to make sure it is not random noise.
- The byte is then read and returned in the ACC with Z = 1.

```

haveiocmd:  IN      A,(bport)                ; poll the port
            AND      D0D1_bits
            CP       D0D1_bits
            JR       Z,..noio                ; jump if no activity
;
            DI                          ; for speed
            LD       HL,ioData
            LD       (HL),A                ; save code
            LD       BC,15                  ; debounce counter
dblpl1:    IN      A,(bport)                ; poll again
            AND      D0D1_bits
            CP       (HL)                  ; still the same data?
            JR       NZ,..noio              ; no, failed debounce
;
            DEC      BC                    ; dec counter
            LD       A,C
            OR       B
            JR       NZ,dblpl1              ; jump if debounce not done
;
            AppOnErr linkfail                ; set error handler
            SET      indicOnly,(IY+indicFlags) ; no key scan
            B_CALL   RecAByteIO              ; read the byte
endexio:    RES      indicOnly,(IY+indicFlags)
            LD       (ioData),A              ; save data
;
            AppOffErr                          ; remove error handler
            LD       A,D0HD1H
            OUT      (bport),A              ; reset B-port
            LD       A,(ioData)              ; get data byte
            CP       A                      ; Z = 1 for successful
            EI
            RET
linkfail:   LD       A,D0HD1H
            OUT      (bport),A              ; reset B-port
..noio:    OR       1                      ; Z = 0 for fail
            EI
            RET

```

Example two:

In the following example, the routine in the above example is used to create a loop that checks for key input and also for a one byte command to be sent over the link port.

```
IO_Key_Lp:
    RES      indicOnly,(IY+indicFlags)    ; key scan turned on
    EI
    HALT                                ; low power sleep mode
;
    B_CALL   GetCSC                      ; check for Scan Code on
                                           ; wake up
;
    CP       skEnter                     ; jump if enter key
    JR       Z,HaveEnterKey
;
    CALL     haveIOcmd                   ; check for link
    JR       NZ,..keylp1st               ; jump if no byte sent
;
    JP       LinkCmdSent                  ; link command received
;
GetCSC:
    LD       HL,kbdScanCode
    DI                                ; interrupts off
    LD       A,(HL)                      ; get possible scan code
    LD       (HL),0                      ; clear out for next scan
    RES      kbdSCR,(IY+kbdFlags)        ; needed for key scan to work
    EI                                ; interrupts on
    RET
```

Example three:

This sample routine will attempt to send the register pair HL over the link port. RET Z = 1 if successful, else Z = 0.

```
sendHl:
    LD       A,H                        ; send H first
    PUSH     HL                         ; save L
    CALL     sendbyte                   ; send to other side
    POP      HL
    RET      NZ                          ; return if failed
    LD       A,L                        ; time to send L
;
sendbyte:
    DI
    PUSH     AF
    LD       A,D0HD1H                  ; set both data lines to high,
                                           ; free
    OUT      (bport),A
    POP      AF
    SET      indicOnly,(IY+indicFlags)
;
    AppOnErr linkfail                   ; See Example 1
    B_CALL   SendAByte                  ; system routine to send byte
    JR       endexio                    ; See Example 1
```

TOOLS AND UTILITIES LAYER

Error Handlers

Error exception handlers can be set up to capture any system error that occurs while executing a block of code that an error handler is placed Around.

- A macro is used to install the error handler:

AppOnErr Label

If your assembler does not support macros, use the following code:

```
LD          HL,Label
CALL       APP_PUSH_ERRORH
```

- Label = Location that the Program Counter (PC) is set to if a system error occurs.
- All registers are destroyed, except the Accumulator.
- Six pushes are made onto the stack. Make sure all the information that is needed from the stack is removed before installing the error handler.

- A macro is also used to remove the error handler:

AppOffErr

If your assembler does not support macros, use the following code:

```
CALL       APP_POP_ERRORH
```

The above is used when the error handler is no longer needed and no system error has occurred.

The Stack Pointer (SP) must be at the level it was at immediately following the AppOnErr. Do not call a routine to set the error handler and then remove it outside of that routine.

- If an error occurs while the handler is place:
 - The system restores the SP, the Floating Point Stack, and the Operator Stack back to their levels when the handler was initiated.
 - The error handler is removed from the stack.
 - The PC is set to the Label specified when the handler was initiated and execution begins there. The Accumulator contains the error code for the error that tripped the handler.

- At this point, the Application can:
 - Ignore the error.
 - Display its own error message.
 - Do some clean up and let the system report the error.
 - Modify the error code to remove the GoTo option and have the system report the error with only a Quit option.

Example one:

Do not allow the error to be reported by the TI-83 Plus. Compute 1/X and return CA = 0 if no error, otherwise return CA = 1.

```

AppOnErr      My_Err_handle
;
      B_CALL      RclX              ; OP1 = (X)
      B_CALL      FPRecip          ; 1/OP1,
;
; If no error then returns from the call
;
      AppOffErr                      ; remove the error handler
      OR          A                ; CA = 0 for no error
      RET
;
; control comes here if X = 0 and generates an error
;
My_Err_handle:
      SCF                      ; CA = 1 for error
      RET

```

Example two:

Allow the error to be reported by the TI-83 Plus, but remove the **GoTo** option. Compute 1/X.

```

AppOnErr      My_Err_handle
;
      B_CALL      RclX              ; OP1 = (X)
      B_CALL      FPRecip          ; 1/OP1,
;
; If no error then returns from the call
;
      AppOffErr                      ; remove the error handler
      RET
;
; control comes here if X = 0 and generates an error, ACC = error code
;
My_Err_handle:
      RES          7,A              ; bit 7 of error code controls GoTo
                                   ; option
      B_JUMP      JError          ; trip the error with no GoTo option

```

Nested Error Handlers

Error handlers can be nested inside of each other. The last error handler initiated will be notified of any error that occurs. When the first handler is notified of the error, none of the previous handlers initiated are notified. If the handler ignores the error or handles it on its own, execution continues on with the other handlers still installed.

If that first error handler B_JUMPS back to the system error handler, (**JError** or **JErrorNo**), the error handler that was initiated before the one that was just tripped is now tripped itself.

Fig. 2.14 below shows the flow of the error with three nested error handlers initiated.

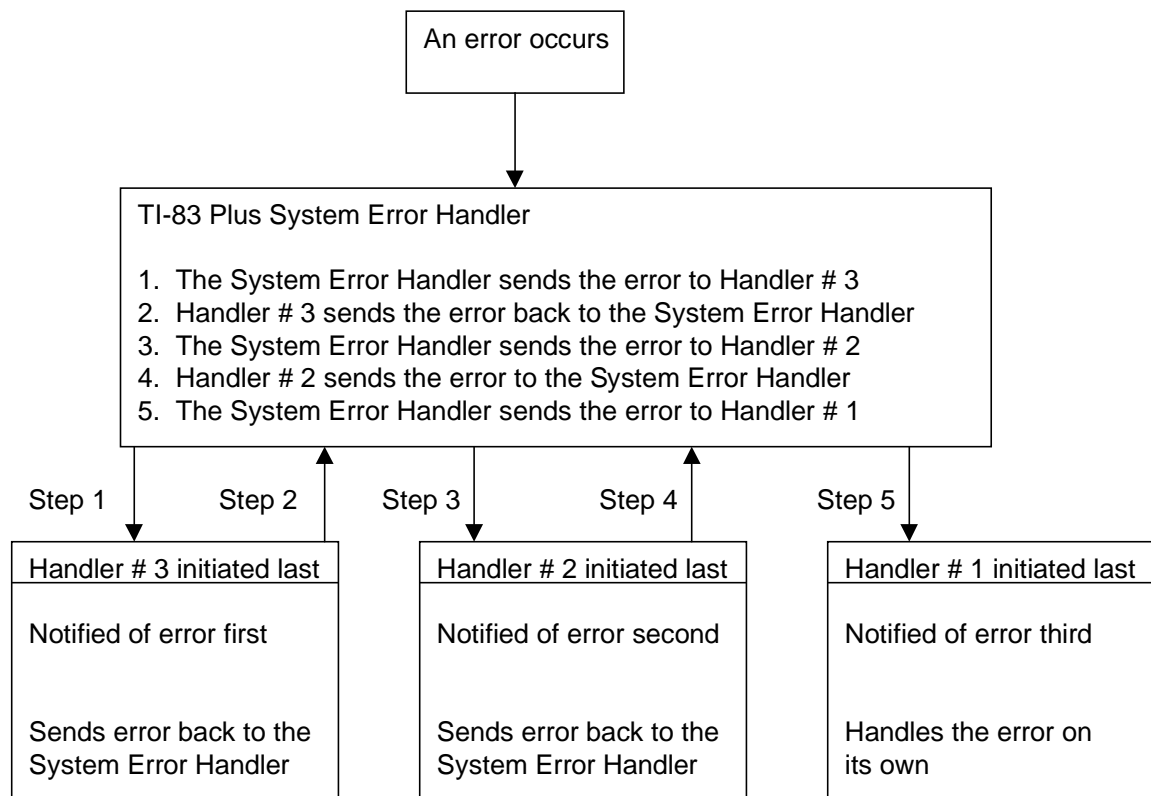


Fig. 2.14: Error Flow

See Appendix A for details on the JError and JErrorNo routines.

Utility Routines

The following is information on the floating-point, complex number, and other math routines.

Floating-Point Math

- All of the floating-point math routine arguments are input in OP1 or OP1/OP2, and output in OP1, unless noted below.
- Errors can be generated by the math routines. See the Error Handlers section.
- All of the inputs to these routines are floating-point numbers.
- See Appendix A, entry points **UnOPExec** and **BinOPExec** to access this functionality with arguments other than floating-point numbers.

Routine	Function
FPAdd	OP1 plus OP2
FPSub	OP1 minus OP2
FPRecip	1 divided by OP1
FPMult	OP1 times OP2
FPDiv	OP1 divided by OP2
FPSquare	OP1 times OP1
SqRoot	Square (OP1)
Plus1	OP1 plus 1
Minus1	OP1 minus 1
InvSub	OP2 minus OP1
Times2	OP1 plus OP1
TimesPt5	OP1 times .5
AbsO1PAbsO2	OP1 plus OP2
Factorial	(OP1)!

Table 2.17: Floating-Point Basic Math Functions

Routine	Function
Sin	Sin(OP1)
Cos	Cos(OP1)
Tan	Tan(OP1)
SinCosRad	OP1 = Sin(OP1) and OP2 = Cos(OP1) force radian mode on input
ASin	inv Sin(OP1)
ACos	inv Cos(OP1)
ATan	inv Tan(OP1)
ASinRad	inv Sin(OP1) force answer in radians
ATanRad	inv Tan(OP1) force answer in radians
DToR	OP1 degrees to radians
RToD	OP1 radians to degrees
SinH	SinH(OP1)
CosH	CosH(OP1)
TanH	TanH(OP1)
SinCosHRad	OP1 = SinH(OP1) and OP2 = CosH(OP1)
ASinH	inv SinH(OP1)
ACosH	inv CosH(OP1)
ATanH	inv TanH(OP1)

Table 2.18: Trigonometric and Hyperbolic Functions

Routine	Function
YToX	$OP1^{OP2}$
XRootY	$OP1^{(1 \text{ divided by } OP2)}$
Cube	$OP1^3$
EToX	e^{OP1}
TenX	10^{OP1}
LnX	$\ln(OP1)$
LogX	$\log(OP1)$

Table 2.19: Floating-Point Power and Logarithmic Math Functions

Routine	Function
Max	Max(OP1, OP2)
Min	Min(OP1, OP2)
Ceiling	Intgr(negative OP1)
Int	Int(OP1)
Intgr	Intgr(OP1)
Trunc	integer part(OP1)
Frac	fractional part(OP1)
CpOP1OP2	non-destructive compare OP1 and OP2
Round	generic Round(OP1)
RndGuard	Round(OP1) to 10 digits
RnFx	Round to current fix setting
Random	generate random floating-point number
RandInt	Generate a random integer between OP1 and OP2

Table 2.20: Floating-Point Miscellaneous Math Functions

Miscellaneous Math Functions

Floating-Point Math Functions that Output Complex Results

The TI-83 Plus has two complex math modes, $a + bi$ and $re^{\theta i}$, that allow complex numbers to be generated by functions that take RealObj data type (floating-point) as input. If neither of these modes is set, then these functions will generate an error when the arguments input would produce a complex result. These functions include **LnX**, **LogX**, **SqRoot**, **YToX** and **XRootY**.

To have these routines return complex results for real data type inputs:

- set one of the complex modes:
 - fmtRect, (IY + numMode) rectangular complex
 - fmtPolar, (IY + numMode) polar complex
- reset
 - fmtReal, (IY + numMode) real output only

- The floating-point math routines described in the previous sections will always return an error when the result is a complex number. To have floating-point math routines return the complex result, the routines described in Other Math Functions need to be used.

Note: You do not need to change the mode to complex in order to use the complex functions with complex inputs. This is only done to get complex results when inputs are of the RealObj type.

Complex Math

- Complex numbers are composed of pairs of floating-point numbers.
- Complex number math routine arguments are input in OP1/OP2 or OP1/OP2 and FPS1/FPST, and the results are returned in OP1/OP2 or OP1. See Floating Point Stack section.
- Errors can be generated by the math routines. See the Error Handlers section.
- See Appendix A, entry points **UnOPExec** and **BinOPExec**, to access this functionality with arguments other than complex numbers only.

Routine	Function
CAdd	FPS1/FPST plus OP1/OP2
CSub	FPS1/FPST minus OP1/OP2
CRecip	(OP1/OP2) ^{negative 1}
CMult	FPS1/FPST times OP1/OP2
CDiv	FPS1/FPST divided by OP1/OP2
CSquare	OP1/OP2 times OP1/OP2
CSqRoot	Square (OP1/OP2)
CMltByReal	OP1/OP2 times OP3
CDivByReal	OP1/OP2 divided by OP3

Table 2.21: Complex Math Basic Math Functions

Routine	Function
CYtoX	$FPS1/FPST^{OP1/OP2}$
CXrootY	$FPS1/FPST^{((OP1/OP2)^{\text{negative } 1})}$
CEtoX	$e^{(OP1/OP2)}$
CTenX	$10^{(OP1/OP2)}$
CLN	$LN(OP1/OP2)$
CLog	$\log(OP1/OP2)$

Table 2.22: Complex Math Power and Logarithmic Math Functions

Routine	Function
CAbs	$OP1 = \text{abs}(OP1/OP2)$
Conj	$\text{Conj}(OP1/OP2)$
Angle	$OP1 = \text{Angle}(OP1/OP2)$
CIntgr	$\text{Intgr}(OP1/OP2)$
CTrunc	$\text{integer part}(OP1/OP2)$
CFrac	$\text{fractional part}(OP1/OP2)$
RToP	$(OP1/OP2) \text{ rect to polar}$
PtoR	$(OP1/OP2) \text{ polar to rect}$
ATan2	$OP1 — \text{ATan2}(OP1/OP2) \text{ where } OP1 = \text{imaginary part}, OP2 = \text{real part of complex}$
ATan2Rad	Same as ATan2 except force results to radian mode

Table 2.23: Complex Math Miscellaneous Math Functions

Other Math Functions

This section covers math functions with data types other than RealObj and CplxObj. It also covers accessing math functions not listed in the above sections.

Many of the functions in the previous two sections can also be used with arguments other than RealObj and CplxObj. For example

<code>Sin(L1)</code>	Sine of list L1
<code>4 * [A]</code>	4 times matrix [A]
<code>(1,2) + L3</code>	complex number (1,2) + list L3

The problem is the entry points that execute the above functions only use RealObj and CplxObj arguments as inputs/outputs. There are two solutions to this problem:

- An application could use these entry points to produce results for arguments that are lists or matrices by doing the element-by-element operations on the input. This approach is not recommended.
- Execute these functions with mixed arguments using the system's executor context.

The systems executor is used during parsing (see the next section for details) to generate results. The executor is partitioned by the number of arguments that a function takes as inputs. The routines used include:

UnOPExec	Executes functions with one argument.
BinOPExec	Executes functions with two arguments.
ThreeExec	Executes functions with three arguments.
FourExec	Executes functions with four arguments.
FiveExec	Executes functions with five arguments.

Input to each of the above routines is a function to be executed along with the argument(s) to be input to the function.

See Appendix A for a complete list of what functions can be executed through the executor, and also for more details on the inputs/outputs requirements.

Results from these routines may be stored in Temporary Variables. See to the Temporary Variables Returned from the Parser section for additional details.

Function Evaluation

Applications may need to evaluate (parse in TI-83 Plus terminology) functions (equations). Using the TI-83 Plus, equations can only contain functions that return values. Programming commands and other commands that do not return a result to **Ans** are not valid in expressions, and therefore can only be executed from a program variable. See the TI-83 Plus Graphing Calculator Guidebook for more information.

Parsing an equation is done to return the value of the equation with the current value of the variables that are contained in it.

Equations can only be parsed if they are stored in an equation variable, an EquObj data type — for example Y1, Xt1, or a temporary equation variable.

Errors can be generated during parsing. If this occurs, the system error context will take over and in most cases, cause the application to be shut down. Applications should install error handlers before parsing equations in order to stop the system error context from activating.

See the Error Handling section in this chapter for further information.

Parse Routine

ParseInp — executes an equation or program stored in a variable.

- Inputs: OP1 equals the name of equation to parse
- Outputs: OP1 equals the result if no error was reported. The output can be any numeric data type including strings. If the result returned from the parser is:
 - RealObj then OP1 equals the result — a floating-point number.
 - CplxObj then OP1/OP2 equals the result — two floating-points numbers.
 - ListObj, CListObj, MatObj, or StrngObj then the name of a variable that contains the result data is returned in OP1, a temporary system variable. Use of temporary variables returned by the parser will be explained later in this section.
- The parser can create temporary variables even if a temporary variable is not returned as the result.

For example, parse the graph equation Y1 and store the answer in Y. Install an error handler around the parsing and the storing routine to catch any errors. RET CA = 0 if OK, else ret CA = 1.

```

                LD      HL,y1Name
                RST      rMov9ToOP1      ; OP1 = Y1 name
;
                AppOnErr  ErrorHan      ; error handler installed
;
                B_CALL    ParseInp      ; parse the equation
;
; returns if no error
;
                B_CALL    CkOP1Real     ; check if RealObj
                JR      Z,storit        ; if a RealObj, try to store to Y
;
                AppOffErr                ; remove the error handler
;
; come here if any error was detected
; error handler is removed when the error occurred
;
ErrorHan:
                B_CALL    CleanAll      ; remove temps if any
                SCF                ; set CA flag to signal failure
                RET
;
storit:
;
                B_CALL    StoY          ; store to Y, ret if no error, else
;                                     ; ErrorHan
;
                AppOffErr                ; remove error handler
;
                B_CALL    CleanAll      ; remove temps if any
                CP      A              ; CA = 0 for no error
                RET
;
YName:         DB      EquObj, tVarEqu, tY1, 0

```


Temporary Variables

The parser can return results that cannot be fully contained in the OP registers due to their size. In these cases, the parser needs to return the result stored in a temporary variable. Temporary variables can also be created by parsing and not be returned as results (see the **CleanAll** routine in the following section).

A temporary variable is like any other user variable that can be created. They reduce free memory available and have Symbol Table entries. Temporary variables exist for the following data types:

ListObj CListObj MatObj StrngObj EquObj

Temporary variables are assigned unique names at the time that they are created. The first character of a temporary variable name is the \$, followed by a two-byte counter, Least Significant Byte (LSB), Most Significant Byte (MSB). The counter is used to create the unique names. For example, if the fifth temporary variable is a list, it would be:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
ListObj 01h	\$ 24h	04h	00h	?	?	?	?	?

Table 2.24: Temporary Variables Example

(pTempCnt) is a two-byte counter in RAM that the system uses to generate the next temporary variable. This allows for up to 64K unique temporary variables.

The (pTempCnt) counter is initialized to 0000h and is incremented after each new temporary variable is created. This counter needs to be managed properly when using temporary variable. It needs to be completely or partially reset periodically in order to keep temporary variable usage available. The Managing Temporary Variables section provides additional details.

Fig. 2.15 illustrates the location in RAM the temporary information is stored.

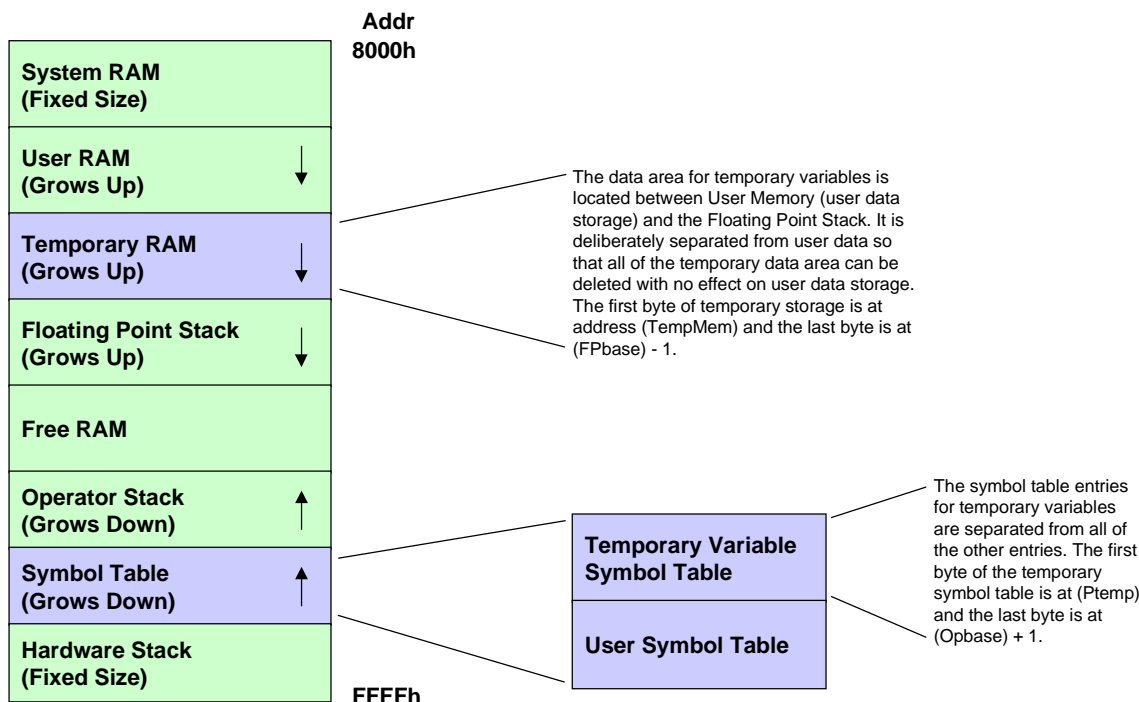


Fig. 2.15: TI-83 Plus System RAM

Using Temporary Variables

Temporary variables can be used in the same manner as any user variable. They can be modified, resized, used to store in to a user variable, and input to system routines.

These variables are called temporary as they are not intended for long term use. Their main purpose is to provide a way to hold onto intermediate results dynamically as the results are needed. Temporary variables should be freed up as soon as they are no longer needed. Some system routines will automatically free up temporary variables if they are used as inputs (this information is noted in the Appendix A entry point documentation).

Managing Temporary Variables

The life span of a temporary variable is determined by the application. Once a temporary variable is no longer needed, it can be marked dirty by the application. Marking a temporary variable dirty identifies it for deletion. Deleting the temporary variable frees the RAM space it occupied.

This marking scheme is used to save time while parsing an equation. The parser/executor does not use time deleting temporary variable — it only marks the temporary variable for deletion after the variable is no longer needed.

Every time a temporary variable is needed, a check is made for available RAM. If there is not enough free RAM, the temporary variables that are marked dirty are deleted one at a time until enough RAM has been freed. If enough RAM were free at the start of parsing, then in most cases, none of these deletions would take place.

A temporary variable is marked dirty by setting bit seven of the temporary variable's sign byte located in its Symbol Table entry. For example, if OP1 equals the name of a temporary variable to mark dirty:

```
MarkTemp:
            B_CALL      ChkFindSym      ; look up temp
;
; HL = pointer to Symbol Table entry
;
            SET         7,(HL)          ; mark dirty
            RET
```

Deleting Temps and Setting (pTempCnt)

There are five different ways that temporary variables are deleted.

- Quitting the application and returning to the screen — This will delete all temporary variables and reset (pTempCnt) equal to 0000h
- System error context is started — This will delete all temporary variables and reset (pTempCnt) equal to 0000h
- System routine **EnoughMem** — This routine is used to check if a certain amount of RAM is free. If the requested amount is not free, this routine will delete dirty temporary variables until either no more dirty temps exist, or the requested amount of RAM is available due to temporary variable deletions. (pTempCnt) is not affected.
- System Routine **FixTempCnt** — This routine is used to delete all temporary variables with a name that contains a counter value equal to DE.

The parser uses this routine in its handling of temporary variables when parsing a program or the home screen entry.

Before each line of the program is parsed, the current value of (pTempCnt) is saved. This value is used to create the next temporary variable needed.

After parsing each line of the program, the resulting value, if one, is stored into the **Ans** variable. Once the result is stored into **Ans**, there can be no other temporary variable that may have been created during the parsing of the line that are still needed.

Calling **FixTempCnt** with DE equal to save pTempCnt, will delete all temporary variables created by the last line parsed. The value (pTempCnt) is reset back to the value saved before the line was parser, DE.

- System Routine CleanAll — This routine is used when the error context is started, or control is returned to the home screen. This will delete all temporary variables and reset (pTempCnt) equal to 0000h.

What should applications do?

Most applications should be able to use the **CleanAll** routine to manage temporary variables. Applications should make a call to the **CleanAll** routine as soon as all temporary variables in use are no longer needed. This is especially important if temporary variables are going to be created in a looping environment. If the temporary variables are not cleaned before the loop is restarted, RAM will become full.

If some temporary variables are needed to be kept alive for extended periods of time, make sure that any other temporary variables that may be created by the application, or returned from the parser, are at least marked dirty when they are no longer needed. That way, the RAM they take up can be reused if needed.

It is also good a good practice to try and use the **Ans** variable instead of temporary variable. The **StoOther** routine can be used to store to the **Ans** variable.

Working with TI Language Localization Applications

TI has made available applications that change the language used for functions commands and strings, from English to an alternate language. Applications can take advantage of the language setting by being able to modify their output to match the current language setting, if desired. The language setting is stored in two bytes of RAM. The table below matches each language with their corresponding values.

The values are store in RAM locations localLanguage and localLanguage+1.

Language	Main language	Sub Language
English	LANG_ENGLISH	SUBLANG_ENGLISH
Danish	LANG_DANISH	SUBLANG_NEUTRAL
Dutch	LANG_DUTCH	SUBLANG_DUTCH
Finnish	LANG_FINNISH	SUBLANG_NEUTRAL
French	LANG_FRENCH	SUBLANG_FRENCH
German	LANG_GERMAN	SUBLANG_GERMAN
Hungarian	LANG_HUNGARIAN	SUBLANG_NEUTRAL
Italian	LANG_ITALIAN	SUBLANG_ITALIAN
Norwegian	LANG_NORWEGIAN	SUBLANG_NEUTRAL
Polish	LANG_POLISH	SUBLANG_NEUTRAL
Portuguese	LANG_PORTUGUESE	SUBLANG_PORTUGUESE
Spanish	LANG_SPANISH	SUBLANG_SPANISH
Swedish	LANG_SWEDISH	SUBLANG_NEUTRAL

Table 2.25: Language Table

For example, check if the current language is Spanish:

```

LD      HL,(localLanguage)      ; H = sublang,
                                ; L = main
LD      DE,LANG_SPANISH + 256*SUBLANG_SPANISH
;
B_CALL  CPHLDE                  ; compare, Z = 1
                                ; if Spanish

```

Entering and Exiting an Application Properly

The state monitor passes control to the TI-83 Plus application loader which sets the monitor's control vectors for key presses, partial put aways, full put aways, window resizing, redisplay, and error.

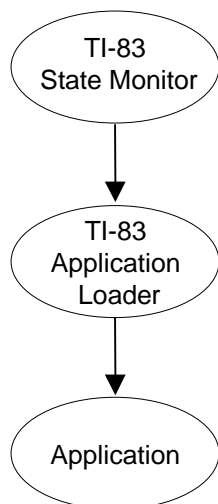


Fig. 2.16: Control Flow

The application now has three choices in which type of environment it will run in – Stand-alone, Stand-alone with Put Away notification, and Monitor driven (not covered in this release)

Stand-alone

The application handles all key inputs itself and does not need access to the TI-83 Plus menu system.

The application will also not be notified if the user turns the unit off. This means that no data, not already saved in a variable, will be lost when the unit turns off. The application is terminated with no notice.

Note: Turning off can occur only if the **GetKey** routine is used directly by an application, or if a system routine called by the application uses **GetKey**.

The application terminates without notice if link activity is detected while waiting for a key.

Start-up Code

No special code is necessary at the start of execution.

Exit Code

The application wants to terminate and return to normal TI-83 Plus operations. Some of the calls in this sequence are not always needed — see the comments.

The following sequence exits the application cleanly even if the hardware stack is not at the same level upon entry to the application. The stack is reset by the system.

```
ExitCode:
          LD          (IY+textFlags),0      ; reset text flags
;
; This next call is done only if application used the Graph Backup Buffer
;
          B_CALL      SetTblGraphDraw
;
          B_CALL      ReloadAppEntryVecs    ; make sure Application Loader set
;
          B_JUMP      JForceCmdNoChar      ; force to home screen
```

Fig. 2.27 shows the sequence of events once the application executes the B_JUMP to **JForceCmdNoChar** instruction.

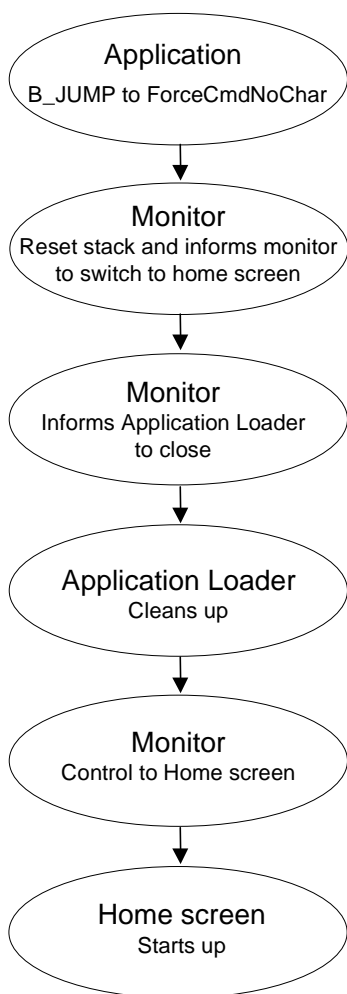


Fig. 2.17: Event Sequence

Stand-alone with Put Away Notification

An application can be notified when the monitor wants the application to terminate. Terminating events include: turning off, a system error was generated and the user chose the quit option, and silent link was activated and closed the application. All of these events are detected while waiting for a key press in the **GetKey** routine.

An application would want to be notified for a variety of reasons.

- An application needs to save its state before being closed down so that the next time it is run it can restore the state it was last in.
- An application may want to delete some variables it has created for temporary use while executing.
- An application may have an edit open that it needs to take care of.
- An application may want to inform the user of some options that are available when being shut down.
- An application may have modified some system flags that need to be set back to their normal state such as disabling APD or enabling lower case alpha entry.

<p>Note: The Put Away cannot be stopped by the application. Once notified by the monitor, the application must terminate.</p>
--

How is the application notified?

If an application needs to be notified when it is being closed down by the system, it must change the system monitor vectors.

Only applications that are extensively integrated with the TI-83 Plus system need to use the monitor. These types of applications are currently not fully supported by this document. However, the level of support provided allows the application to receive notification of the application being shut down.

The monitor vectors control the flow of information to the context that is in control at a given time. A context loads the monitor vectors with pointers to its handling routines. Information that is sent out by the system monitor include key presses, partial put aways, full put aways, window size changes, and error recovery. Normally there is a separate handler for each of these events.

When an application is executing, the current context in control is the Application Loader as noted in the figure below.

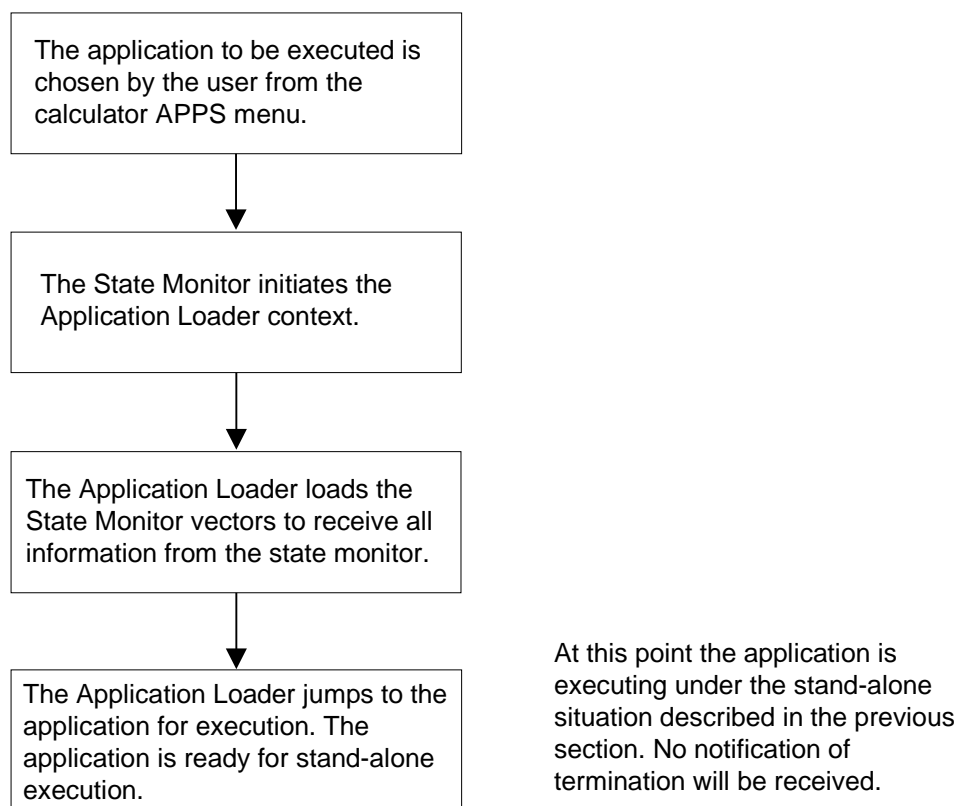


Fig. 2.18: Application Loader Process

An application must change the monitor vectors so that any information sent by the monitor, is sent directly to the application.

Start-up Code

These lines of code must be at the beginning of the application.

```

;
;          LD          HL,AppVectors
;          B_CALL      AppInit          ; Apps monitor control vectors written
;
; all of the vectors are set to a 'RET' instruction in the App except
; for the 'Put Away' vector which is set to the routine to handle the
; Put Away in the App.
;
;
;

```

This is the rest of the application code.

```
Dummy:
    RET
;
; Table of vectors loaded into monitor control vectors
;
AppVectors:
    DW      Dummy      ; set this vector to a 'RET' instruction
    DW      Dummy      ; set this vector to a 'RET' instruction
    DW      AppPutaway  ; set this vector to Apps Put Away
                        ; routine
    DW      Dummy      ; set this vector to a 'RET' instruction
    DW      Dummy      ; set this vector to a 'RET' instruction
    DW      Dummy      ; set this vector to a 'RET' instruction
    DB      appTextSaveF ; system flag, this is a normal setting
```

Now the application is connected to the system monitor through the system monitor vectors. If the monitor were allowed to be in control then all of the information it sends to the system would come to the application.

Since the monitor is not in control, information will be sent to the application under three circumstances.

- While **GetKey** is executing the TI-83 Plus is turned off.
- While **GetKey** link activity is detected.
- If a system error is generated and allowed to be displayed, the Quit option is chosen by the user.

In all three circumstances, the system monitor will jump to the application at the label AppPutAway, or whatever label is used in the AppVectors table.

Sample code to handle the apps termination is given. The turning off situation is handled differently than the other two.

Put Away Code

This code should not be used when the application terminates on its own. An application should follow the Stand-alone example to exit without the monitor initiating the termination.

```
AppPutAway:
;
;
; Application gets itself ready for terminating by cleaning any system flags
; or saving any information it needs to.
;
;
; This next call resets the monitor control vectors back to the App Loader
;
          B_CALL      ReloadAppEntryVecs          ; App Loader in control of
                                                  ; monitor
;
          LD          (IY+textFlags),0          ; reset text flags
;
; This next call is done only if application used the Graph Backup Buffer
;
          B_CALL      SetTblGraphDraw
;
; Need to check if turning off or not, the following flag is set when
; turning off:
;
          BIT          monAbandon,(IY+monFlags)   ; turning off ?
          JR          NZ,..TurnOff              ; jump if yes
;
; if not turning off then force control back to the home screen
;
; note: this will terminate the link activity that caused the application
; to be terminated.
;
          B_JUMP      JForceCmdNoChar           ; force to home screen
;
..TurnOff:
          B_JUMP      Putaway                   ; force App loader to do its
                                                  ; put away
```


PROGRAMMING LAYER

Chapter 2 covered the Hardware layer, the Driver layer, and the Tools and Utilities layer. The final layer in the TI-83 Plus architecture is the Programming layer.

There are three kinds of programs that can be created for the TI-83 Plus: TI-BASIC programs, ASM programs, and Applications. This chapter is primarily concerned with applications. In the following discussion, Z80 refers to the type of microprocessor used by the TI-83 and TI-83 Plus.

TI-BASIC Programs

These programs were available on the TI-83 and may be known as scripts or keystroke programs. These programs are created using the PC program TI-GRAPH LINK™ for TI-83 Plus or directly on the calculator using the [PRGM] New [1:Create New] options. The details for creating this kind of program are provided in the *TI-83 Plus Guidebook*. These programs consist of commands that mimic the calculator keystroke commands, plus some additional keywords for control-flow logic. These programs are loaded into, and run from, the calculator RAM. There must be sufficient free RAM available in order to be able to load a TI-BASIC program. This language is interpreted, so these programs do not have to be assembled or compiled before you run them on the calculator. Interpreting the programs, however, causes them to be relatively slow. When these programs execute, if they contain an illegal statement or perform an illegal operation, the interpreter stops the program and displays an error message. The calculator functions normally after such an error.

ASM Programs

ASM programs were available on the TI-83 and may be known as assembly programs or ASAPs. These programs are written in Z80 assembly language and then adapted to use the calculator's pre-existing ability to run TI-BASIC programs. After the ASM program is assembled, it is converted to a readable text format that can then be downloaded to the calculator in the same way as a TI-BASIC program. A special keyword at the start of the program tells the calculator interpreter that it is an ASM program instead of a normal TI-BASIC program. The interpreter then converts the program into Z80 machine language and gives it control of the processor. Since these programs have total control over the calculator, they are fast, but any programming errors can be serious, causing the calculator to become unusable until reset. These programs are able to call built-in calculator routines. They run in RAM and are limited in size to 8K.

Applications

Applications, or apps, are assembly language programs. These programs are different from ASM programs primarily in that they are stored in and run from the Flash ROM, where they are not likely to be erased, and they take no RAM space. Applications only need RAM for any variables they might create. Apps have access to all the same system routines as ASM programs and they can be much larger than ASM programs. Apps must be created on a PC. They have special requirements on content and linking. They must be digitally signed by TI if they are to be distributed. Additionally, a user calculator must have an internal digital certificate in order for the app to run. This is not true if the app is freeware or shareware.

ASM versus Applications

Assembly programs written to be ASM programs must be modified in order to function correctly as Applications. The major difference is that ASM programs run from RAM, but Applications run from Flash ROM. Therefore, applications cannot be self-modifying, whereas ASMs can. Applications also need additional identification code at the start of the program. They need additional code to handle errors and exceptional events. And, they must be digitally signed by TI if they are to be distributed.

DEVELOPMENT SYSTEM

The simulator is for general development use and the steps for setting it up, getting started, and creating a sample application are presented in the following sections.

Using the Simulator System — Requirements for Getting Started

The following are the requirements to be able to develop TI-83 Plus applications using TI's simulator development system. The Zilog Developer Studio and TI-83 Plus Simulator/Debugger installation and operations are covered in Chapter 4.

- IBM® PC compatible computer.
- Windows® 95 operating system
- The Zilog Developer Studio
- The TI Simulator/Debugger

With the above environment up and running, let us look at creating a sample application.

Creating an Application for Debugging — One-Page and Multi-Page Apps

In the section that discusses memory maps, you saw that there are up to ten 16K Flash ROM pages available for storing applications. This storage area is also used for archived calculator variables, so as the archive grows fewer pages are actually available for apps. In theory it is possible to create an app that takes up all 10 pages and is 160K in size. However, most apps will surely be smaller and this is desirable to conserve memory and download time.

Apps are always allocated in whole pages. It is not possible for an app to share a page with another app or archived variables. If an app only uses 40 bytes it is still allocated the whole 16K Flash ROM page. And if an app requires 16K+1 bytes, it is allocated exactly two 16K Flash ROM pages. For this reason we say that apps are a 1-Page App or a Multi-Page App. Creating multi-page is a little more complicated than 1-page apps, so we will begin with 1-page apps.

A Brief Overview of Certificates and Application Signing

In normal calculator usage, an application is installed in a calculator by downloading it from a PC or another calculator via the link cable. When the app is received it is examined by the operating system loader for a valid TI digital signature. All Flash apps to be distributed must be digitally signed by TI before they will be accepted by the operating system. Since all apps must be signed, an app build must go through TI before it can be loaded in the normal way. Since signing is 1) an external process for developers, 2) is limited, and 3) has turnaround time associated with it, a single calculator debugging technique is available to facilitate code development. The debugging technique that follows avoids having to sign each iterative build.

Creating Applications that Fit On One Page

Applications are written in Z80 Assembly language. While there are C to Z80 cross compilers, TI recommends the use of assembly language for efficiency and memory space reasons. The format of the source code depends on the assembler/linker package that you use. With the package TI recommends (ZDS), App source code is plain ASCII text. There is no special editor required. You can use any editor (such as Notepad) that can save the file as plain ASCII. The required source code syntax also varies by assembler. The examples and discussions provided by TI conform to the requirements of the Zilog Developer Studio (ZDS) assembler and linker.

ZDS uses a file naming convention of *.asm for all source files containing executable statements and *.inc for all include files.

The Hello Application

TI has provided a sample application called Hello. The source for this application is in the file `hello.asm`. Open this file in a text editor and look at it to get a general idea of the main structural elements. The following sections address these elements.

Accessing System Resources

The program begins by including the `TI83plus.inc` file. This file is provided by TI. This file includes constant definitions, macros, and system routine entry point definition needed to use system resources.

Application Headers

The most unique thing about the TI-83 Plus application source code is the long set of data that begins the file. This data is known as the application header. The application header contains information used by the calculator operating system when the user tries to run the application. The operating system uses this information to determine the app name and whether a user is permitted to use it. A valid header must be present as the first data in the source file, prior to any executable statement, in order for the app to run properly.

Header Creation

The header in the `hello.asm` file can be used for any single page application.

Calling System Routines

On the TI-83 Plus there are a number of built-in system routines available for an application to use. These routines can not be called directly using the standard Z80 call instruction. In order to call a system routine, you must use a statement of the form:

```
B_CALL    routine
```

In this example, `routine` is the name of any system routine. `B_CALL` is a macro defined in the system include file.

Accessing System Variables

Certain fixed locations in RAM are defined for system code usage. The contents of these locations typically affect some standard system behavior. System routines sometimes use the variables, so they are in effect parameters to the system calls. To access one of these variables, you use its symbolic name (e.g., `curRow`). The variable names are defined in the system include file, `TI83plus.inc`.

Defining a String

Many system routines operate on null-terminated strings, which are a series of characters followed by the byte 00h. The assembler supports null-terminated string creation through use of the directive `.asciz`. This permits you to type the string in readable text instead of defining each byte separately. Each character of the string is translated to its ASCII code and stored at the current location and a null character is then appended. In our example, we define a label that points to the first character of the string so that we can point to the string in our system calls.

Erasing the Screen

To erase the screen, the example does the system call.

```
B_CALL      ClrLCDFull      ; Clear the screen
```

Printing Text to the Screen

To print text to the screen, the example uses the system call.

```
B_CALL      PutS              ; Print the hello string from RAM
```

This routine prints a null-terminated string in large text to the screen. It expects you to have already set up the screen row and column where it should start printing the string. The screen rows range from 0 (Top) to 7 (Bottom), and the columns range from 0 (Left) to 15 (Right). You set these values in the system variable `curRow` and `curCol` prior to the call. The **PutS** routine expects Z80 register HL to contain the address of the first character of the string. It requires that this string be in RAM.

Copying the String

To copy a string from Flash ROM, where it is defined in your program, into RAM, where the system routine **PutS** can use it, you can use the system routine **StrCopy**. This routine expects the address of the source string to be in HL and the address of the first RAM destination character to be in DE. It expects a null-terminated string. The example copies the string Hello into the OP1 area in RAM (see next paragraph).

System RAM Registers

The calculator system code performs many operations on floating-point values. It uses a floating-point format that requires up to 11 bytes in certain situations. Since floating-point operations are so common, it defines six 11-byte areas that it uses frequently for storing such numbers. It gives these RAM areas the name OP1, OP2, OP3, OP4, OP5, and OP6. In our example, the system routines **StrCopy** and **PutS** do not use or modify these areas, so we use six of the eleven OP1 RAM bytes to temporarily store our string in RAM. In this case, we are just using OP1, since changing those locations is harmless; the fact that OP1 may be used at some later time to pass floating-point data does not matter.

Reading a Key Press

The system routine **GetKey** waits for a user to press a key on the calculator keypad. The example (found in the `hello.asm` file) uses this fact to implement a pause so that you can read the string it printed.

Exiting an Application

When an application is ready to quit and return control back to the calculator operating system so that normal calculator features will again be available, it must perform the following system call:

```
B_JUMP      JForceCmdNoChar  ; Exit the application
```

Creating a Multiple Page Application

The fundamental change in moving from a one-page application to a multi-page application is the addition of the branch table. The branch table is used by system code to perform the correct paging of physical Flash ROM pages into the logical address space when a call or jump is made to a routine that exists on a page that is not currently mapped.

Branch Table Entries

The branch table exists only on the first application page, immediately after the header. It is a table of three-byte entries. Each entry is a pointer to a routine that is either called or jumped to from a page of the application other than the page where it exists. A routine that is called or jumped to only from locations on the same page does not need an entry in the table. Each entry has the form:

DW	Address
DB	Relative App Page

The Address is the address of the routine on its page. To obtain the value, where the routine is defined, make the label public. You will need to refer to your assembler for instructions on how to make and reference a public routine.

The Relative Application Page is the page of the application where the routine resides. In this case, page numbers are relative to the first application page: the first application page is 0, the second is 1, and so on.

Branch Table Placement

Application execution begins at the address immediately following the header. The branch table is not part of the header, but must be placed immediately after the header. To resolve this conflict, a jump instruction to the start of the application needs to be placed between the end of the header and the start of the table.

Also, the first entry in the branch table must be located at an address which is a multiple of three bytes from the beginning of the page. You may need to add padding bytes before the branch table to ensure this.

Branch Table Equate File

Whenever a branch table exists, an include file must also be generated that contains equates for the branch table entries. Each equate in the file is the name of the routine in the branch table with an underscore character prefixed to it. The associated value is the byte offset where the routine's table entry begins.

For example, the routine `showGoodByeP2` exists on the second application page but must be called from the first application page, so it needs an entry in the branch table. The branch table entry for this routine happened to be located at a position 41 times three-bytes from the start of the first application page.

```
; Byte offset 41 * 3
          DW          showGoodByeP2      ; Address
          DB          1                  ; Second app page
```

So in the include file the following equate is created.

```
_showGoodByeP2 equ 41*3
```

This include file must be included in any source code that calls or jumps to a routine on another page.

Making Off-Page Calls and Jumps

When code calls or jumps to a routine on an application page different from the point of the call, this is known as an off-page call or jump. The `B_CALL` and `B_JUMP` macros must be used when making off-page calls and jumps. For example, when the routine `showHelloP2`, which is on the second page, is called from the first page, the call must be made as follow:

```
B_CALL      showHelloP2
```

A call of the form

```
CALL        showHelloP2
```

will not work at all.

When an on-page call, a call to a routine that exists on the same application page as the point of the call, is made, the normal call opcode should be used. B_CALL and B_JUMP should not be used in this case.

CREATING A ZILOG DEVELOPER STUDIO PROJECT

Let us go through the use of the Zilog Developer Studio software to build the Hello application presented earlier in this chapter.

Creating the Project

1. Copy the files from <install directory>\Demo to C:\mydemo directory
2. Start Zilog Developer Studio
3. Select File, and then New Project
4. In the New Project dialog box, set the following fields to the specified values:

Selection by = Family

Master = Z180

Project Target = Z80180

Project Name = C:\mydemo\mydemo.zws

In the Initializations dialog box, set the following fields to the specified values:

Program Counter = 0

Stack Pointer = 0

Adding Files to the Project

1. Select Project, then Add to project, and then Files...
2. In the Insert files into project dialog box double click on hello.asm.

Project Settings

1. Select Project, then Settings, and then Linker.
2. In the Linker Options dialog box select the Ranges tab.
3. Click on the New... button.

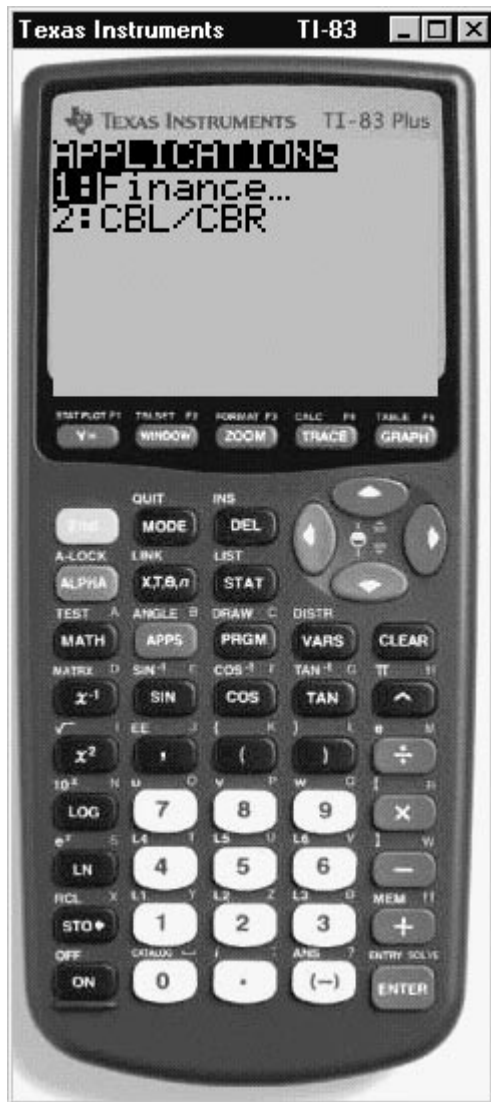
4. In the New Section Range dialog box set the following fields to the specified values:
Bounds = Length
Radix = Hexadecimal
Section Name = .text
Start Address = 4000
Length = 4000
5. Click OK then click Apply then click OK.

Building the Application

1. Select Build, and then Rebuild All.
2. The following text should appear in the output window:
Building...
hello.asm
hello.o — 0 error(s), 0 warning(s)
Linking...
mydemo.ld — 0 error(s), 0 warning(s)

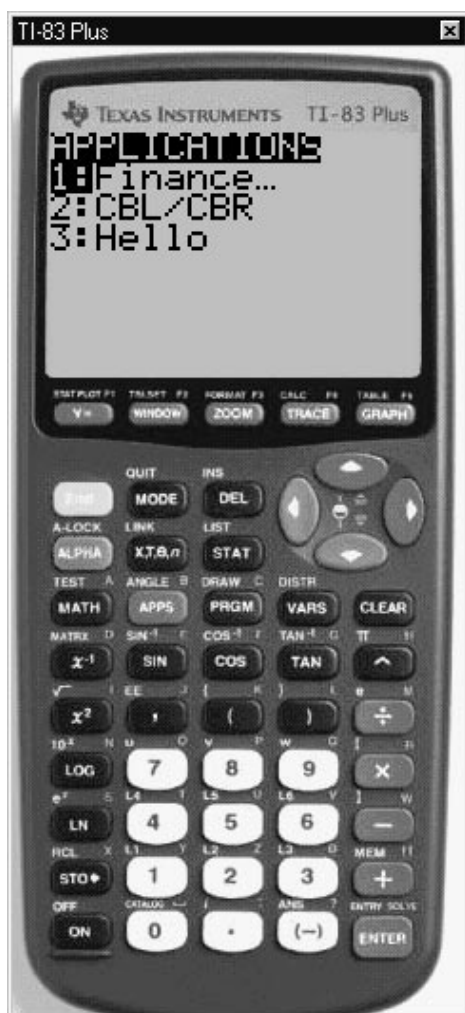
Loading the Application into the Simulator

1. Start the TI Flash Debugger.
2. Select File, and then New.
3. Select Debug, and then Go. The TI-83 Plus calculator will be displayed.
4. Click on the **APPS** key of the calculator.



Next:

1. Click the **CLEAR** button on the calculator.
2. On the Debugger menu select Debug, and then Stop.
3. Select Load, and then Application.
4. In the Load Application dialog box, double click on the file C:\mydemo\mydemo.hex.
5. Select Debug, and then Go.
6. Click on the **APPS** key on the calculator. Application three will be titled **Hello**.



Next:

1. Click the **3** key on the calculator to run the Hello application. Hello will appear on the screen.
2. Click on any key of the calculator to quit the Hello application.

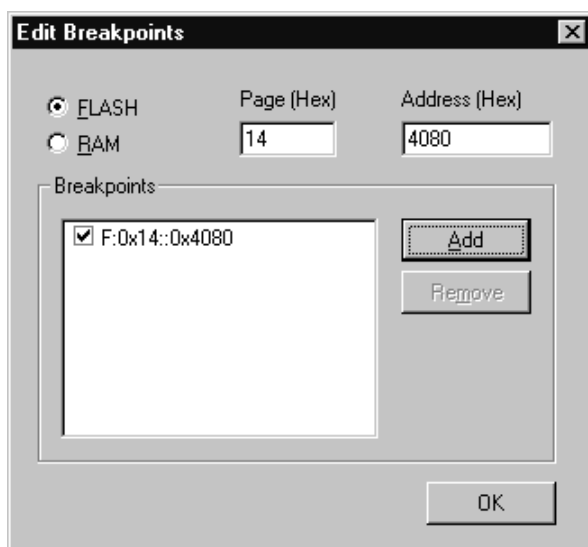
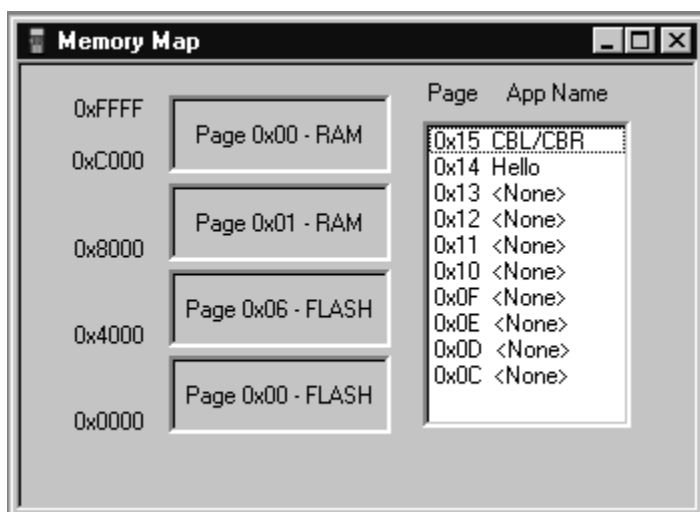
Debugging the Application

In the following steps we will demonstrate some of the debug capabilities. We will set a breakpoint at the start of our application and after the Hello string is copied to RAM. We will then modify the RAM copy of the string to HOWDY.

1. Select Debug, and then Stop.
2. Select View, and then Memory Map.

This view shows us that the Hello application is on page 0x14 of Flash.

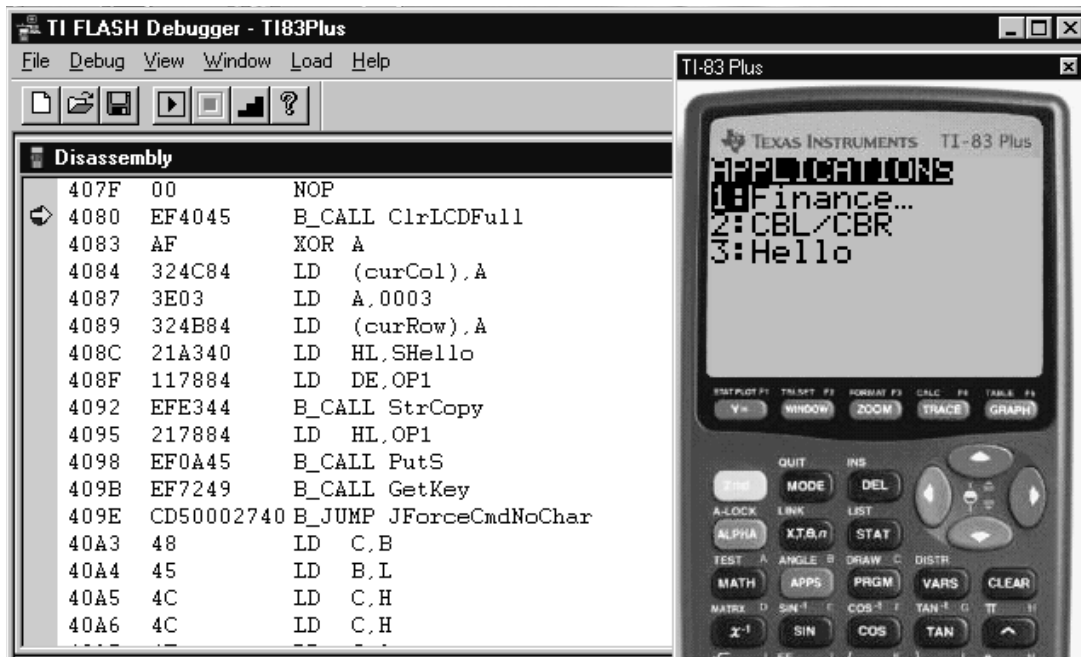
3. Select Debug, and then Breakpoints.
4. Update the Edit Breakpoints dialog box so that it looks like the following:



Note: If we look at the hello.lst file we will see that **StartApp:** is located 0x80 bytes from the start of the page (at x4080).

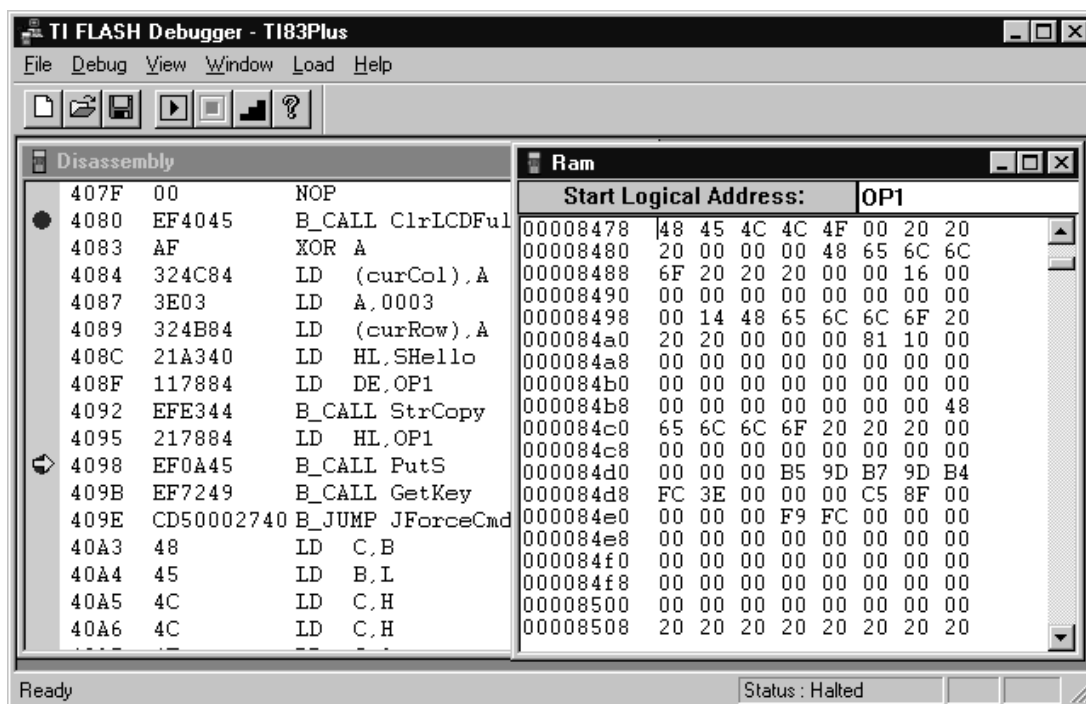
Next:

1. Click OK to exit the Edit Breakpoints dialog.
2. Select Debug, and then Go.
3. Click on the [APPS] key of the calculator. Note that the Status of the Debugger is Running.
4. Click on the 3 key of the calculator. The status of the Debugger will change to Halted when the breakpoint is reached.



Now:

1. Right Click on address line 4098 to bring up the breakpoints pop-up menu.
2. Select Set Breakpoint.
3. Select Debug, and then Go. The calculator display will be cleared and the disassembly view will be updated to indicate that it is stopped at address 4098.
4. Select View, and then RAM to bring up the RAM view. In the Start Logical Address field enter OP1.



Finally:

1. Change byte 8479 from 45 E to 4F O, 8480 from 4C L to 57 W, 8481 from 4C L to 44 D and 8482 from 4F O to 59 Y.
2. Select Debug, and then Go. The calculator will display HOWDY.
3. Click any key on the calculator to quit the application.
4. Select Debug, and then Stop.
5. Select Debug, and then Breakpoints to bring up the Edit Breakpoints dialog box. Disable the breakpoints by clicking on each of the check boxes in the breakpoint list.
6. Select Debug, and then Go.
7. Click the **[APPS]** key on the calculator.
8. Click the **3** key on the calculator. The Hello application will run and display Hello again.
9. Click any key on the calculator to quit the application.

Now we will modify Flash to change the original Hello string so that the change will persist between each execution of the application.

10. Select Debug, and then Stop.
11. Select View, and then Flash.
12. In the Start Physical Address field enter 500A3. If we look at the hello.lst file, we will see that the Hello string begins at offset 0 x A3. Since the application is on page 0x14 we get $0x14 * 0x4000 + 0xA3 = 0x500A3$.
13. Change the byte at address 0x500A3 to 0 x 53, 0x500A4 to 0x54, 0x500A5 to 0x41, 0x500A6 to 0x52 and 0x500A7 to 0x53.
14. Select Debug, and then Go.
15. Click the **APPS** key on the calculator.
16. Click the **3** key on the calculator.
17. The calculator will display STARS (as in the Dallas Stars, the 1999 Stanley Cup Champions) each time the application runs.
18. Select File, and then Close to close the debug session. A dialog box will appear asking if you want to save changes.
19. Click the Yes button.
20. The Save As dialog box will appear. Save debug session to C:\Mydemo\mydemo.83d.
21. Select File, and then Exit to exit the Debugger.

Preparing an Application for Site Testing

As mentioned in the section introducing certificates, applications must be digitally signed by TI and a user calculator must have a license before the application will run on that calculator. The debugging technique used with the simulator circumvents this restriction, but it only works on the simulator, not a real calculator.

Once an application has become well developed, some developers may need to perform testing at their development site beyond one person using an emulator. Typically, the development team may own 3 to 20 TI-83 Plus calculators on which they wish to test the app. To support this need, TI may issue certificates for the group of testing calculators and set the developer up so that they can sign the app instead of TI. In this situation, the app will only run on the selected calculator group.

To sign an app, the developer needs four things.

1. Another set of files provided in a TI Zip file.
2. A Developer ID.
3. A set of digital keys to use in signing.
4. Unit certificates for each calculator to be used in testing.

The developer will need to provide TI with the ID of each calculator to be used. This information is viewable on the calculator by selecting [2ND][Mem][1>About]. Each ID will have the form ID:00000-00000-0000.

On receiving this list, TI will assign the developer a Developer ID, generate a set of digital keys, and create unit certificates for the calculators. The keys are contained in one file: xxxx.key. The xxxx in the filename will be the developers Developer ID (a hex number). Each unit certificate file will be named U12345.cer, where the 12345 corresponds to the middle 5 digits of the calculator ID for which it is intended. Typically all of this information can be e-mailed to the developer.

The TI Zip file should be unzipped to the developer directory where the other files reside. The key and certificates should also be placed in the developer directory.

The unit certificates should be loaded to each calculator using the PC utility flashd.exe provided in the TI Zip file. This process requires a GRAPH LINK cable.

In the applications header, the App ID needs to be changed to the Developers ID given by TI. For example, the App ID used in demo application would be changed from 0104 to xxxx — where xxxx is the Developer ID. The developer should then rebuild the application.

Signing the Application

To sign the app, use the following command: `appsign xxxx.key demo.hex`.

The standard file naming convention for downloadable TI-83 Plus applications is *.app. We rename the final result to match this convention. The app should now download to and run on the test calculators.

Downloading the App

You can use either the flashd.exe program provided by TI or the TI-83 Plus GRAPH LINK program to download the app to the calculators.

Preparing for Public Release

When applications are ready to be distributed, they must go through a signing process at Texas Instruments. The current steps and requirements will be available on the TI Web site at www.ti.com/calc.

4

Development Tools

DEVELOPMENT ARCHITECTURE

The TI development architecture is based on the TI simulator/debugger using the Zilog Developer Studio software. In the following sections, we will address the TI simulator/debugger and the related tools used to develop applications for the TI-83 Plus calculator.

Z80 DEVELOPMENT SYSTEM

Zilog Developer Studio is a programming suite made by Zilog to compile assembly code for its microprocessors, including the Z80 used in many Texas Instruments graphing calculators. ZDS may have several advantages in that it is graphical, has a built-in editor, and most importantly, it is free. You may wish to consult Zilog's web site at <http://www.zilog.com> for more information. This documentation is currently written for version 2.12 of ZDS.

INSTALLATION

ZDS is easily obtained for free from Zilog's web site. A link to download the current version is present on their software downloads page at <http://www.zilog.com/support/sd.html> and is approximately 6.9 MB in size. Download the installer (currently named zds212.exe) and run it. Follow the instructions to install the ZDS suite. This will install the software on your computer and place a link to it in your Start menu. Now lets look at the simulator/debugger.

TI SOFTWARE SIMULATOR AND DEBUGGER

Introduction

The TI-83 Plus simulator provides the capability to simulate the TI-83 Plus calculator to allow debugging of applications. The following is a detailed description of the various menu options, screens, and operations.

Installation

To install the TI Flash Debugger, run the installation file that has been furnished with the SDK package.

Getting Started

Invoke the flashsim.exe file. The simulator/debugger application presents the following screen.



This window is the home screen for the application. Various other windows with selected views are presented which are explained below. The menu selections available from the home screen include:

File

New Ctrl + N

Open Ctrl + O

Open Selection Dialog box

Recent File (grayed out)

Exit

View**Tool Bar** (selected)**Status Bar** (selected)**Help****About TI Flash Debugger**

The tool bar icons, which are defined by hovering the cursor over the applicable icon, has selections for New (File), Open (File), Save (File), the debugger controls — Go, Stop, and Step (grayed out), — and ? (About).

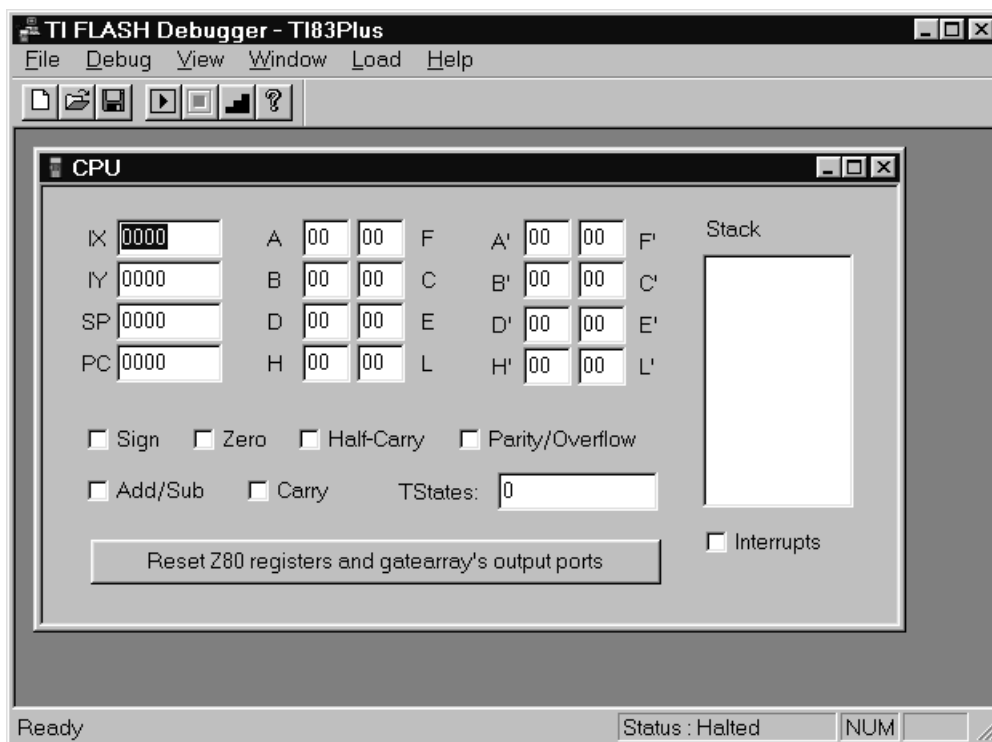
The status bar at the bottom of the window indicates the status of the debugger and simulator. The left side of the status bar indicates the status of the debugger (i.e., Ready). The first box on the right side of the status bar indicates the status of the simulator. In this case, the status of the simulator is halted.

The simulator/debugger uses two files:

<xyz>.83d which contains debug information (breakpoints).

<xyz>.clc which contains the calculator memory contents, where <xyz> is the file name.

The next step is either to create a new debug file or open an existing one. For example purposes, we will create a new debug file. Upon selecting File/New, the following CPU view is presented with additional selections on the menu bar and tool bar as noted below.

**File****New** Ctrl + N**Open** Ctrl + O

Open Selection Dialog Box

Close**Save** Ctrl + S**Save As...**

Save As Selection Dialog Box

Recent File (grayed out)**Exit****Debug****Go** F5 Starts the debugger**Stop** (grayed out) Stops the debugger**Step** F10 Allows single instruction stepping**Step Over** F11 Steps over CALL and B_CALL instructions.**Breakpoints...**

Edit Breakpoints Dialog Box

Trace Option...

Trace Option Dialog Box

View**CPU****Disassembly****Flash****RAM****Memory Map****Calculator****Trace Log****Toolbar (selected)****Status bar (selected)****Window****Cascade****Tile****1 CPU****Load****Application...**

Load Application (Hex) File Dialog Box

Base Code...

Load Base Code (Hex) File Dialog Box

RAM File...

Load RAM File Dialog Box

Options...

Loading Option Dialog Box

Help**About TI Flash Debugger**

First, we will look at the Edit Breakpoint and Trace Options dialog boxes.

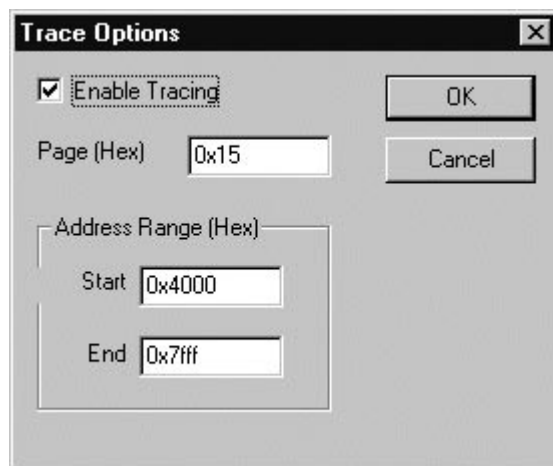
Breakpoints

Setting breakpoints is available via the manual setup dialog box from the (Debug/Breakpoint drop down menu). To remove breakpoints, select the breakpoint and press the Remove button.



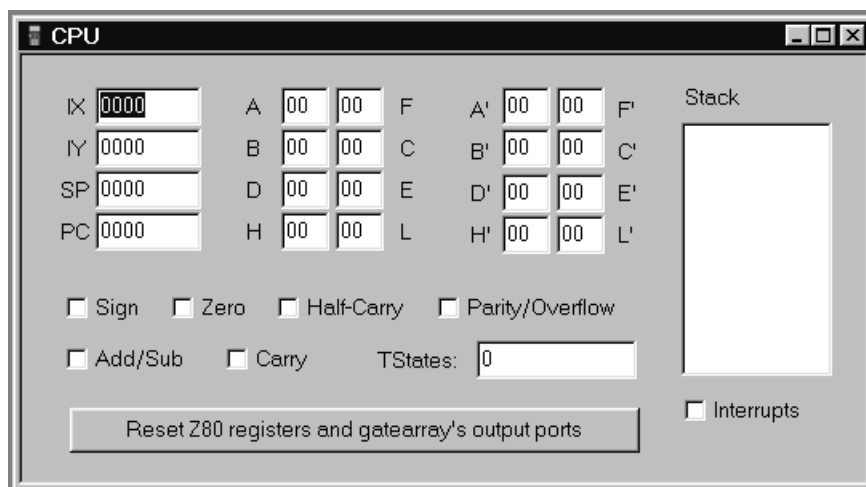
Trace Options

This dialog box presents options to be considered in performing a trace such as page, and address ranges.



Let us now look at the CPU View first, then we will present each of other views with details of each.

CPU View Window



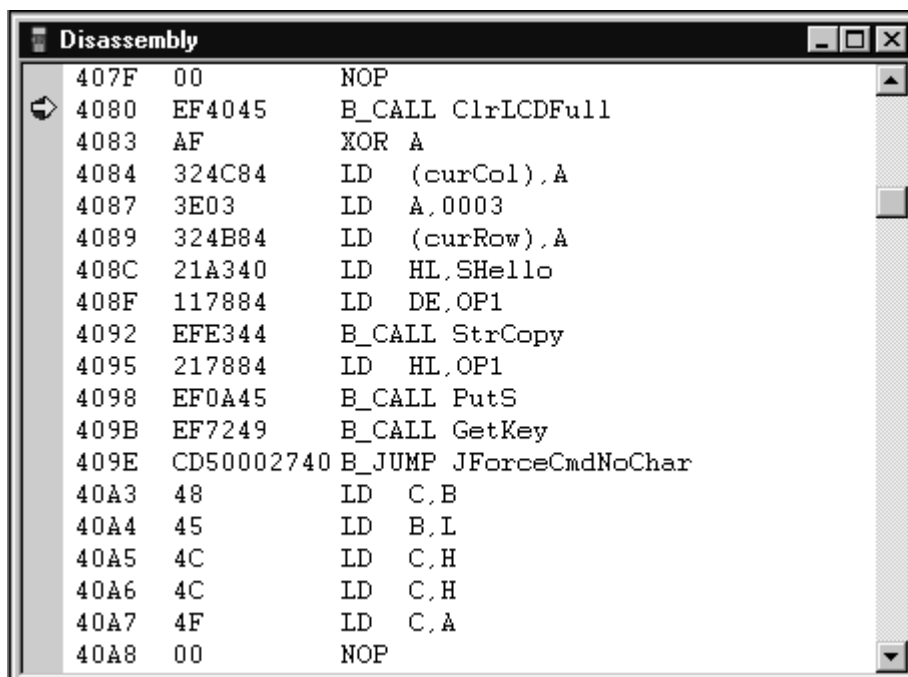
The CPU View displays several items of processor information.

IX	index register
IY	index register
SP	stack pointer
PC	program counter
AF	accumulator/Flag register
BC	register
DE	register
HL	register
A'F'	alternative register
B'C'	alternative register
D'E'	alternative register
H'L'	alternative register
Sign	Sign — flags
Zero	Zero — flags
Parity/Overflow	Parity/Overflow flag
Half Carry	Half Carry
Carry	Carry
Add/Sub	Flag set if a subtraction operation occurred, otherwise is reset for any other operation.

Tstate	Time State — counts the number of time periods.
Reset	Reset Z80 registers and gate array output ports.
Stack	List the values currently pushed onto the stack.
Interrupts	Indicates if interrupts are enabled.

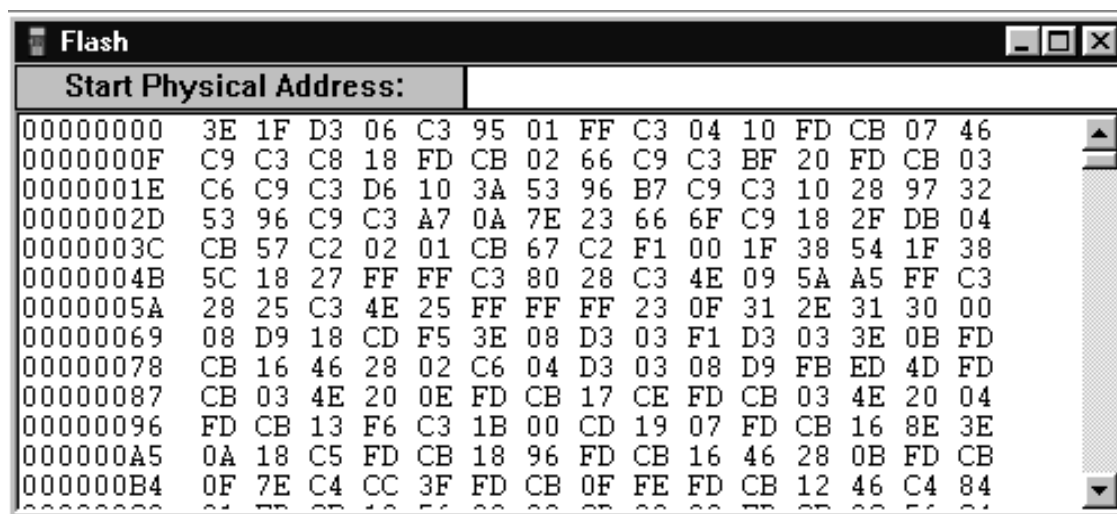
Disassembly View Window

Contains the address, byte code, and instructions of the disassembled code. Breakpoints can be set and cleared from this screen by use of the right mouse click. This window is automatically invoked when the Debugger STOP key is pressed.



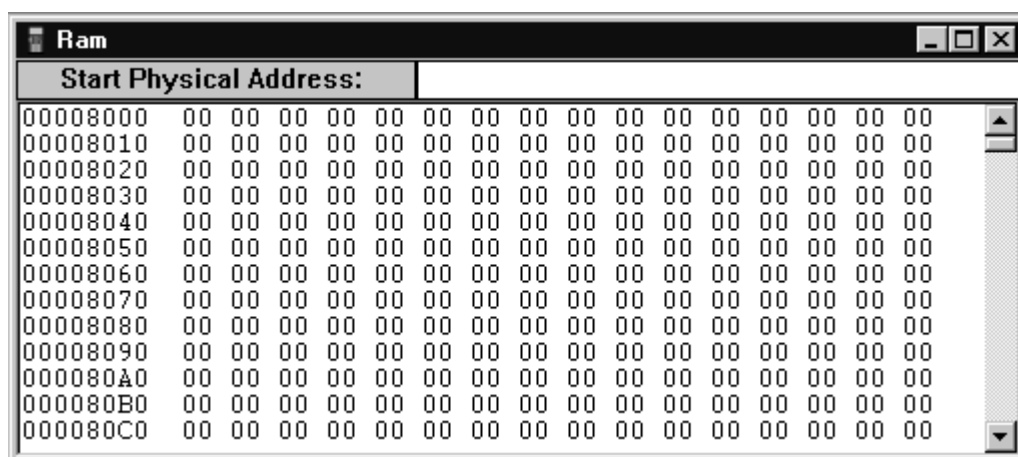
Flash View Window

Displays the entire contents of Flash memory. This is the Edit/View screen. The Start Physical Address edit box is used to view addresses by entering the desired address and hitting enter.



RAM View Window

Displays the entire contents of RAM. This is the Edit/View screen. The Start Physical Address edit box is used to view addresses by entering the desired address and hitting enter.



The input to the TI-83 Plus calculator screen can be done in two ways:

- Pressing the simulated keys with the mouse cursor and seeing the results on the screen.
- Using the computer keyboard keys and seeing the results on the screen. This method is provided via three configuration files that are included in the SDK — 83pkeymap.cfg, 83pkeys.cfg, and pckey.cfg.

The 83pkeymap.cfg file contains the mappings from the PC keys to the TI-83 Plus keys.

The 83pkeys.cfg file contains the TI-83 Plus keyboard keys with their values.

The pckey.cfg file contains the PC keyboard keys with their hex values.

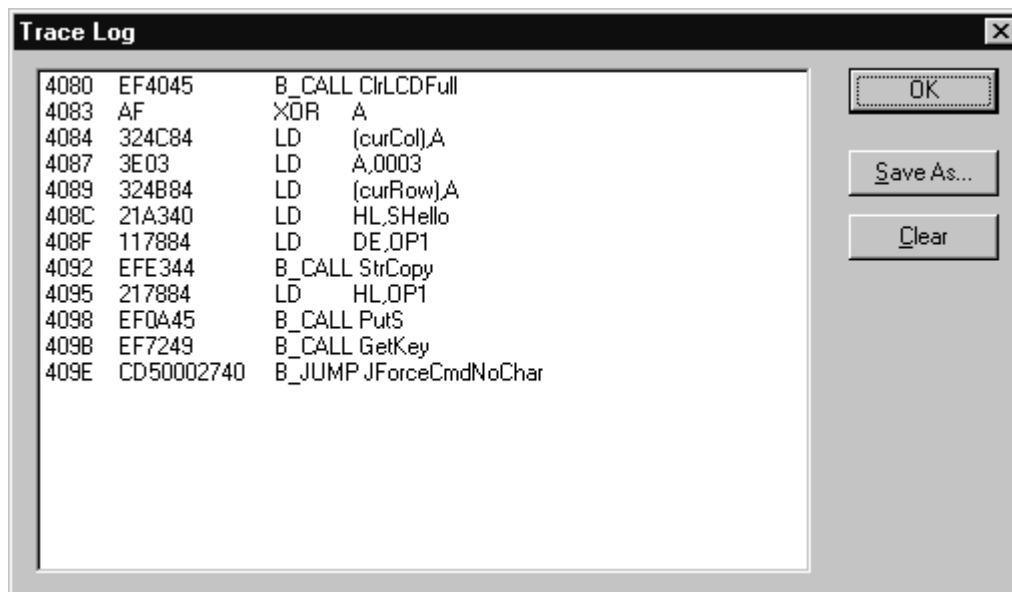
While all three files are viewable and editable in various editors including Notepad, the only file that should be edited by the developer is the 83pkeymap.cfg file.

Note: Shift key mapping is not supported.

Let us now look at other available views.

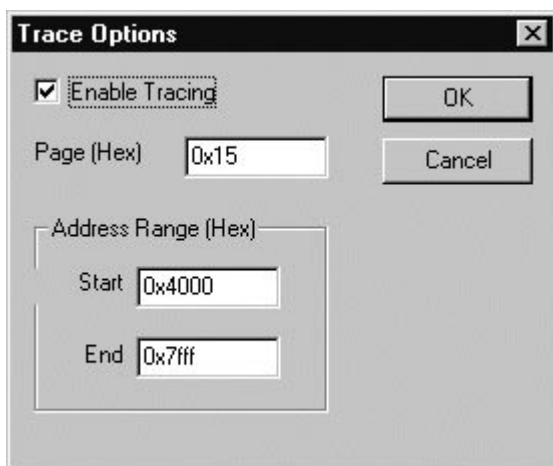
Trace Log Window

Displays the output of a trace — the execution of instructions within a developer definable address space.



The Trace Options dialog box is used to define this address space as indicated earlier:

Enable Tracing	If checked, tracing is enabled.
Page	The page of Flash or RAM that should be traced
Address Range Start	The start of the address space to trace.
End	The end of the address space to trace.



Here is how it works:

If tracing is enabled, the value of the PC is between the Start and End address and the current page equals the Page specified, the current instruction is added to the trace log buffer.

The developer can view the contents of the trace buffer by bringing up the Trace Log dialog box. The trace log buffer is a circular buffer and can hold up to 4K of instructions. From the Trace Log dialog box, the developer can save [Save As..] the contents of the trace buffer. Using the [Clear] button, the contents of the buffer is cleared

With the trace option settings of:

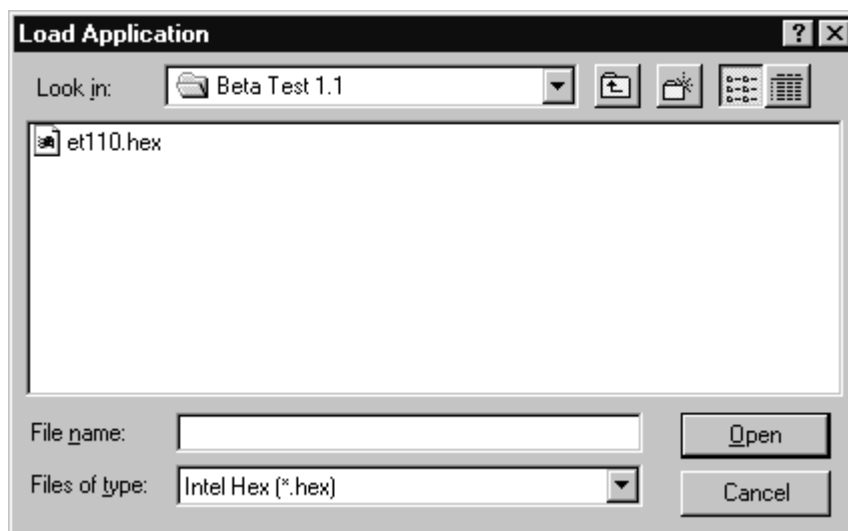
Page = 0x15
 Start = 0x4080
 End = 0x7FFF

the following output is produced:

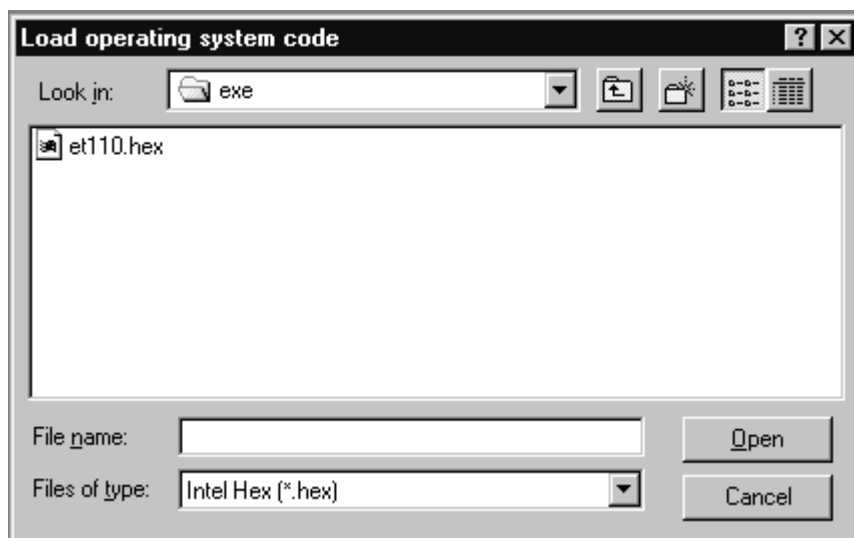
4080	EF4045	B_CALL	ClrLCDFull
4083	AF	XOR	A
4084	324C84	LD	(curCol),A
4087	3E03	LD	A,0003
4089	324B84	LD	(curRow),A
408C	21A340	LD	HL,SHello
408F	117884	LD	DE,OP1
4092	EFE344	B_CALL	StrCopy
4095	217884	LD	HL,OP1
4098	EF0A45	B_CALL	PutS
409B	EF7249	B_CALL	GetKey
409E	CD50002740	B_JUMP	JForceCmdNoChar

Loading Applications, Operating System, and RAM Files

Selecting the Load/Application... menu item allows you to load an Application.



Selecting the Load/Operating System menu item allows you to load a new version of Operating System.



The latest operating system is included during the installation of the simulator. Selecting **Go** from the Debug menu activates the calculator simulator with the operating system operational. When a new release of the operating system is produced, it will be available from the TI web site for download and installation. By invoking the Load menu and then selecting the Operating System item, the developer will be able to load the new version of the operating system.

Selecting the Load/RAM File... menu item allows you to load a RAM file.



Terminating a Session

Selecting Close from the File menu allows you to save the current debugging session.

Note: The default extension is .83d. This action also saves the <xyz>.clc file.



Support in Writing Applications

There are various sources for help in writing TI-83 Plus applications. A few of these resources include:

TI-83 Plus Developer's Guide (this book).

TI-83 Plus Graphing Calculator Guidebook

TI-83 Plus Tutorials @ <http://www.ti.com/calc>

G

GLOSSARY

ACC	ACC stands for accumulator.																
Address	A number given to a location in memory. You can access the location by using that number, like accessing a variable by using its name.																
APD™	Automatic Power Down™.																
API	Application Programmer's Interface —the set of software services available to an application and the interface for using them.																
Applet	A stand-alone application, usually in Flash ROM, with the associated security mechanisms in place. See ASAP.																
Archive memory	Part of Flash ROM. You can store data, programs, or other variables to the user data archive, which cannot be edited or deleted inadvertently.																
ASAP	Assembly Application Program —a RAM-resident application.																
ASCII	American Standard Code for Information Interchange —a convention for encoding characters, numerals in a seven or eight-bit binary number. ASCII stands for.																
Assembler	A program that converts source code into machine language that the processor can understand, similar to compilers used with high-level languages.																
Assembly language	A low-level language used to program microprocessors directly. Z80 assembly language can be used on the TI-83 Plus to write programs that execute faster than programs written in TI-BASIC. See Chapter 3 for advantages and disadvantages.																
Binary	<p>A system of counting using 0's and 1's. The first seven digits and the decimal equivalents are:</p> <table><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr><tr><td>10</td><td>2</td></tr><tr><td>11</td><td>3</td></tr><tr><td>100</td><td>4</td></tr><tr><td>101</td><td>5</td></tr><tr><td>110</td><td>6</td></tr><tr><td>111</td><td>7</td></tr></table> <p>See also Hexadecimal.</p>	0	0	1	1	10	2	11	3	100	4	101	5	110	6	111	7
0	0																
1	1																
10	2																
11	3																
100	4																
101	5																
110	6																
111	7																

Bit	Short for binary digit — either 1 or 0. In computer processing and storage, a bit is the smallest unit of information handled by a computer and is represented physically by an element such as a single pulse sent through a circuit or a small spot on a magnetic disk capable of storing either a 1 or a 0. Considered singly, bits convey little information a human would consider meaningful. In groups of eight, however, bits become the familiar bytes used to represent all types of information, including the letters of the alphabet and the digits 0 through 9. (Microsoft Encarta '97)
Boot (code)	A small amount of software that resides in ROM; therefore, it cannot be overwritten or erased. Boot code is required for the calculator to manage the installation of new base code.
Byte	A unit of information consisting of 8 bits, the equivalent of a single character, such as a letter. 8 bits equal {0-255} and there are 256 letters in the extended ASCII character set. Standard ASCII uses a 7-bit value (0-127), thus there are 128 characters.
Calculator serial number	An electronic serial number that resides in a calculator's Flash memory. It is used to uniquely identify that calculator.
Character	A single letter, digit, or symbol. Q is a character. 4 is a character. % is a character. 123 and yo are not characters.
Compiled language	A language that must be compiled before you can run the program. Examples include C/C++ and Pascal.
Compiler	A compiler translates high-level language source code into machine code.
D-Bus	A proprietary communication bus used between calculators, the Calculator-Based Laboratory™ (CBL™) System, the Calculator-Based Ranger™ (CBR™) and personal computers.
Decimal	The standard (base 10) system of counting, as opposed to binary (base 2) or hexadecimal (base 16).
E-Bus	Enhanced D-Bus.
Entry points	Callable locations in the base code corresponding to pieces of code that exhibit some coherent functionality.
Execute	To run a program or carry out a command.
Flash-D	A PC program that is the integration of a PC downloader application with a calculator application. When the Flash-D program is executed on the PC, the calculator application is transferred to the calculator via a TI-GRAPH LINK™ cable.
Freeware	Programs or databases that an individual may use without payment of money to the author. Commonly, the author will copyright the work as a way of legally insisting that no one change it prior to getting approval. Commonly, the author will issue a license defining the terms under which the copyrighted program may be used. With freeware, there is no charge for the license.

Garbage collection	A procedure that automatically determines what memory a program is no longer using and recycles it for other use. This is also known as automatic storage (or memory) reclamation .
TI-GRAPH LINK™	An optional accessory that links a calculator to a personal computer to enable communication.
Group certificate	Used to identify several calculators as a single unit . This allows the group of calculators, or unit , to be assigned a new program license using only one certificate (instead of requiring a new unique unit certificate for each calculator in the group). The group certificate must be used in conjunction with the unit certificate.
Hexadecimal	Base 16 system, which is often used in computing. Counting is as follows: {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.
High-level language	Any programming language that resembles English. This makes it easier for humans to understand. Unfortunately, a computer cannot understand it unless it is compiled into machine language. See also low-level language. Examples of high-level languages are C/C++, Pascal, FORTRAN, COBOL, Ada, etc.
IDE	Integrated D evelopment E nvironment.
Immediate	An addressing mode where the data value is contained within the instruction instead of being loaded from somewhere else. For example, in LD A, 17, 17 is an immediate value. In LD A, B, the value in B is not immediate, because it is not written into the code.
Interpreted language	A language that is changed from source code to machine language in real-time. Examples are BASIC (for the PC and the TI version, TI-BASIC) and JavaScript. Interpreted languages are often much simpler, which helps beginners get started and allows experienced programmers to write code quickly. Interpreted languages, however, are restricted in their capability, and they run slower.
Instruction	A command that tells the processor to do something, for example, add two numbers or get some data from the memory .
I/O port	An input/output interface from the calculator to the external world. It allows communication with other units, CBL™ and CBR™, and personal computers.
LCD port	An output port that drives LCD display device for use on overhead projectors. Available on the teacher's ViewScreen™ calculator only.
Low-level language	Any programming language that does not look like English but is still to be understandable by people. It uses words like add to replace machine language instructions like 110100 . See also high-level language.
Machine language	Any programming language that consists of 1's and 0's (called binary), which represents instructions. A typical machine instruction could be 110100, which means add two numbers together .
Mac Link	MacIntosh resident link software that can communicate with the calculator.

Marked Dirty	The graph is marked as needing to be updated. The next system routine that will affect the graph contents will cause the system to regraph all of the equations selected thereby making the graph clean.
Memory	Memory is where data is stored. On the TI-83 Plus, the main memory is the built-in 32K of RAM. This memory is composed of one-byte sections, each with a unique address.
Microprocessor	See processor.
Operating System (OS)	The software included with every new calculator. OS contains the features that are of interest to customers, as well as behind-the-scenes functionality that allows the calculator to operate and communicate. In our newer calculators, the OS is in Flash ROM, so the user can electronically upgrade it with OS.
Processor	A large computer chip that does most of the work in a computer or calculator. The processor in the TI-83 Plus is the Zilog Z80 chip.
Program	A program is a list of instructions written in sequential order for the processor to execute.
Program ID number	An ID number assigned to a particular software program. It is used during the program authentication process to match the program licenses in a unit/group certificate to the program being downloaded into the calculator.
Program license	A digital license purchased by a customer allowing the customer to authorize the download/execution of a particular software program to a specific calculator. The program licenses are assigned to and listed in the calculator unit/group certificates.
Register	A register is high-speed memory typically located directly on the processor. It is used to store data while the processor manipulates it. On the TI-83 Plus there are 14 registers.
Register pair	Two registers being used as if they were one, creating a 16-bit register. Larger numbers can be used in registered pairs than in single registers. The register pairs are AF, BC, DE, and HL. Register pairs are often used to hold addresses.
Run (Busy) Indicator	When the TI-83 Plus is calculating or graphing, a vertical moving line is displayed as a busy indicator in the top-right corner of the screen. When you pause a graph or a program, the busy indicator becomes a vertical moving dotted line.
SDK	Software Development Kit—a set of tools that allow developers to write software for specific platforms.
Shareware	Sometimes called User Supported or Try Before You Buy software. Shareware is not a particular kind of software, it is a way of marketing software. Users are permitted to try the software on their own computer systems (generally for a limited period of time) without any cost of obligation. Payment is required if the user has found the software to be useful or if the user wishes to continue using the software beyond the evaluation (trial) period.

Payment of the registration fee to the author will bring the user a license to continue using the software. Most authors will include other materials in return for the registration fee—like printed manuals, technical support, bonus or additional software, or upgrades.

Shareware is commercial software, fully protected by copyright laws. Like other business owners, shareware authors expect to earn money from making their software available. In addition, by paying, the user may then be entitled to additional functions, removal of time limiting or limits on use, removal of so-called **nag** screens, and other things as defined in the documentation provided by the program's author.

Signed application	An application that has been digitally signed by TI.
Silent link	Computer-initiated request—protocol version of communications between the computer and the calculator.
Software owner's account	An account set-up in the TI database listing all of the program licenses owned by a particular customer or group. The account also allows the software owner to assign a particular program to a specific calculator.
Source code	A text file containing the code, usually in a high-level or low-level programming language.
TASM	Table Assembler—a PC program that assembles source code for the Z80 and other processors. This has been one of the more popular tools for developing calculator ASM programs.
TI-BASIC	The programming language commonly used on the TI-83 Plus. It is the language that is used for PROGRAM variables. Its main drawback is that these programs run slower, since it is an interpreted language, rather than a compiled language.
TI signature	A digital signature placed on secured documents/files such as unit and group certificates, as well as software program images.
User Data Archive	Storage for user data in the Flash ROM. In some cases, the user can choose between the amount of Flash for applets versus user data.
Unique owner ID	An alphanumeric ID assigned to the owner of a software owner's account as a way of authorizing access to this account. Examples of the ID are mother's maiden name, social security number, birth date, etc.
Unit certificate	A digital certificate signed by TI that lists all of the program and group licenses issued to a specific calculator. The unit certificate also includes owner ID information and the calculator serial number.
Z80	This processor is used in the TI-83 Plus. Z80 assembler is the language used to program the Z80 chip.
ZDS	Zilog Development Studio—a tool used by developers to write software for Zilog products. This tool can be used to develop TI-83 Plus calculator applications and ASM programs.

Appendix A

System Routines

The following is the format in which each of the entry points will appear. The entry points are listed alphabetically by category.

Entry point name: Name used to identify the routine.

Category: Each entry point is identified by function into a category.

Description: Brief description of usage/purpose. How the routine works and additional information about the input.

Inputs:

Registers: Setup values in processor registers.

Flags: Setup values in processor flags (F register).

Others: OPX, stack or RAM locations initial conditions affecting results.

Outputs:

Registers: Return information in processor registers.

Flags: Return information in process flags.

Others: Return information in OPX, stack, or RAM.

Registers destroyed: Processor registers whose initial values may be modified, so caller is responsible for preserving.

RAM used: RAM space needed, where applicable.

Remarks: Description of appropriate usage context, limitations, and any other useful information, side effects, assumptions, etc.

Example: An example of how to set up initial conditions and use the routine.

NOTE () indicate indirection

A

System Routines — Display

Bit_VertSplit.....	159
CheckSplitFlag	160
ClearRow.....	161
ClrLCD.....	162
ClrLCDFull.....	163
ClrOP2S	164
ClrScrn	165
ClrScrnFull.....	166
ClrTxtShd	167
DispDone.....	168
DispHL.....	169
DisplayImage.....	170
DispOP1A.....	172
EraseEOL.....	173
FormBase.....	174
FormDCplx	176
FormEReal	178
FormReal.....	179
LoadPattern.....	180
Load_SFont.....	181
OutputExpr	182
PutC	183
PutMap.....	184
PutPS	185
PutS	187
PutTokString.....	189
RestoreDisp.....	190
RunIndicOff	191
RunIndicOn	192
SaveDisp	193
SetNorm_Vals	194

(continued)

Contents *(continued)*

SFont_Len.....	195
SStringLength.....	196
VPutMap.....	197
VPutS	198
VPutSN.....	200

Bit_VertSplit

Category: Display

Description: Tests if the TI-83 Plus is set to G-T (graph-table) display mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: Z = 1 if G-T mode is set

Others: None

Registers destroyed: None

Remarks: Applications may want to reset the 83+ to full screen mode if graphing functionality is used. In G-T mode the screen is split vertically with 1/2 being the graph screen and the other the table display.

Example:

B_CALL	Bit_VertSplit	; test for G-T mode
JR	NZ,Screen_is_Split	; jump if G-T mode

CheckSplitFlag

Category: Display

Description: Checks if either horizontal or G-T split screen modes are active.

Inputs:

Registers: None

Flags: grfSplitOverride, (IY + sGrFlags) = 1 to ignore split mode settings
This flag is set to make system routines draw to the full screen even when in a split screen mode.

Others: None

Outputs:

Registers: None

Flags: Z = 1 if no split screen mode is active
= 0 if a split screen mode is active

Others: None

**Registers
destroyed:** None

Remarks:

Example: `B_CALL CheckSplitFlag`

ClearRow

Category: Display

Description: Clears eight consecutive LCD display drive rows.

Inputs:

Registers: A = LCD display driver row coordinate (0x80 – 0xBF)

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Eight pixel rows cleared
Driver left in X increment mode

Registers destroyed: A, B, DE

Remarks: This routine requires A to be in LCD display driver row (X) coordinates, which have a valid range between 0x80 – 0xBF, with the top pixel row equal to 0x80 and the bottom pixel row equal to 0xBF. Passing in a value for A outside this range will cause unpredictable results and probably a lockup. This routine erases eight consecutive rows, so if you pass in A = 0x88, the 9th – 16th pixel rows from the top of the display are erased. If you pass in a value between 0xB9 – 0xBF, the erased rows wrap back to the top of the display. In normal usage, if you are erasing a line of large text, the A value should be a multiple of 0x08.

Example:

ClrLCD

Category: Display

Description: Clears the display.

Inputs:

Registers: None

Flags: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set
vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** All

Remarks: This routine only acts on the display, not the *textShadow*.

Example: Clear the display using the current split settings:

```
B_CALL    ClrLCD
```

ClrLCDFull

Category: Display

Description: Clears the display ignoring any split screen settings.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Entire display is cleared.

**Registers
destroyed:** All

Remarks:

Example: B_CALL ClrLCDFull

ClrOP2S

Category: Display

Description: Sets the floating-point number in OP2 to be positive.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_CALL ClrOP2S

ClrScrn

Category: Display

Description: Clears the display. If **textShadow** is in use clears it also.

Inputs:

Registers: None

Flags: appTextSave, (IY + appFlags) = 1 if the **textShadow** is to be cleared also
G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this turn off the split screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set

vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set

grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes

Others: None

Outputs:

Registers: None

Flags: None

Others: Display and possibly **textShadow** cleared.

Registers destroyed: All

Remarks:

Example: B_CALL ClrScrn

ClrScrnFull

Category: Display

Description: Clears the display entirely ignoring split screen settings. If *textShadow* is in use clears it also.

Inputs:

Registers: None

Flags: appTextSave, (IY + appFlags) = 1 if the *textShadow* is to be cleared also

Others: None

Outputs:

Registers: None

Flags: None

Others: Display and possibly *textShadow* cleared.

Registers destroyed: All

Remarks:

Example: B_CALL ClrScrnFull

ClrTxtShd

Category: Display

Description: Clears the *textShadow* buffer.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: *textShadow* set to spaces.

Flags: None

Others: None

**Registers
destroyed:** BC, DE, HL

Remarks: ClrScrn falls into this routine which zeros out 128 bytes starting at *textShadow* (one byte for each 5 * 7 screen position (8 rows * 16 columns)).

Example:

DispDone

Category: Display

Description: Displays Done on text screen.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** HL

Remarks:

Example: B_CALL DispDone

DispHL

Category: Display

Description: Converts the contents of HL to a decimal and writes it to the screen at current cursor position. The string displayed is always 5 characters and right justified. The large 5×7 font is used.

Inputs:

Registers: HL = two-byte value to convert

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: String displayed. (OP1) = start of five character decimal number string, right justified.

Registers destroyed: AF, DE, HL

Remarks: If the string does not fit on the current display row then it is truncated at the screen's edge.

Example: Set HL = 357 and display it starting in row 0 column 0.

```
LD      HL,0
LD      (curRow),HL      ; set cursor position
;
LD      HL,357
B_CALL  DispHL
RET
```

what will be displayed is " 357", which has two leading spaces.

DisplayImage

Category: Display

Description: Displays a bitmap image stored in RAM.

Inputs:

Registers: HL = pointer to image structure
 Height of image in pixels — one-byte
 Width of image in pixels — one-byte
 Image data by rows

The first byte contains the data for the first eight-pixels of the first row. Bit 7 is the left-most pixel of the first row.

Each new row starts on a byte boundary.

There may be unused bits in the last byte of each row if the image is not a multiple of eight in width.

DE = location on screen to place the upper left corner of the image.
 (row, column)

(0,0) = upper left corner of the screen.

The image can be oriented off of the screen: ffh = -1. The only restriction is that the image cannot be entirely off screen.

Flags: plotLoc, (IY + plotFlags) = 1 if image drawn to display only.
 = 0 if image drawn to display and graph buffer.
 bufferOnly, (IY + plotFlags) = 1 if image drawn to graph buffer only.
 This flag overrides the plotLoc flag.

Others: None

Outputs:

Registers: None

Flags: None

Others: Screen, graph buffer
 RAM locations @ ioPrompt – ioPrompt + 7

Registers destroyed: All

Remarks:

(continued)

DisplayImage *(continued)*

Example: Display an image three-pixels high by 17 pixels wide at position (0,0) to the display only.

```

ImageData:
    DB      3,17                ; height, width
;
    DB      80h,3eh,10h        ; row 1, only bit 7
                                ; of the last byte
                                ; is used
    DB      11h,35h,0h        ; row 2
    DB      0ffh,01h,10h      ; row 3
;
    LD      HL,ImageData       ; pointer to bitmap
    LD      DE,OP1
    LD      BC,11
    LDIR                                ; copy image data to
                                ; RAM
;
    LD      HL,OP1             ; pointer to image
    LD      DE,0               ; position of upper
                                ; left corner
;
    SET     plotLoc,(IY+plotFlags)
;
    B_CALL  DisplayImage

```


DispOP1A

Category: Display

Description: Displays a floating-point number using either small variable width or large 5x7 font. The value is rounded to the current “fix” setting (on the mode screen) before it is displayed.

Inputs:

Registers: ACC = maximum number of digits to format for displaying

Flags: textInverse, (IY + textFlags) = 1 for reverse video
textEraseBelow, (IY + textFlags) = 1 to erase line below character
textWrite, (IY + sGrFlags) = 1 to write to graph buffer not display
fracDrawLFont, (IY + fontFlags) = 1 to use large font, not small font

Others: (penCol) = pen column to display at
(penRow) = pen row to display at

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4

Remarks: Displaying stops if the right edge of the screen is reached.

Example:

EraseEOL

Category: Display

Description: Erases screen to end of line.

Inputs:

Registers: None

Flags: None

Others: curRow, curCol point to screen position.

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None, saves registers beforehand.

Remarks: curRow, curCol are also saved and restored.
If the sEditRunning, (IY + apiFlg3) flag is set (sfont running).

Example:

```

LD      HL,0801h      ; curRow = 1, curCol = 8
LD      (curRow),HL
LD      A,'H'
B_CALL  PutC
LD      A,'I'
B_CALL  PutC
B_CALL  EraseEOL      ; clear to end of line
;

```

FormBase

Category: Display

Description: Converts a RealObj (single floating-point number) in OP1 into a displayable string.

Use the current mode settings SCI, ENG, NORMAL and FIX setting to format the string.

The output can also be formatted as a fraction or a Degrees, Minutes, Seconds (DMS) number.

Inputs:

Registers: None

Flags: To use the current format settings:
(Flags + fmtFlags) copies to (Flags + fmtOverride)

To override the current settings, modify the following flags:
Resetting the next two flags sets NORMAL display mode.
fmtExponent, (fmtOverride) = 1 for scientific display mode
fmtEng, (fmtOverride) = 1 for engineering display mode

Setting the next three flags will signify DMS formatting.
fmtBin, (fmtOverride)
fmtHex, (fmtOverride)
fmtOct, (fmtOverride)

Setting the next two flags will signify Fraction formatting.
fmtHex, (fmtOverride)
fmtOct, (fmtOverride)

Others: (fmtDigits) = 0FFh for FLOAT, no fix setting
= 0 – 9 if fix setting is specified
OP1 = value to format.

Outputs:

Registers: BC = length of string

Flags: None

Others: String returned in RAM starting in OP3, and is 0 terminated

Registers destroyed: All

Ram Used: OP1 – OP6

Remarks: If the current display mode settings are SCI or ENG, the output string will reflect the setting. The value is rounded based on the maximum width entered and the current fix setting.

(continued)

FormBase *(continued)*

Example: Generate a random number and display it at the current cursor position. Use all the current format settings except force SCI formatting.

```

;          B_CALL      Random          ; OP1 = random number
;
;          LD          A,(flags+fmtFlags) ; get current format
;                               ; settings
;          RES          fmtEng,A
;          SET          fmtexponential    ; override current and
;                               ; set SCI formatting
;          LD          (flags+fmtOverride),A ; set override flags
;
;          B_CALL      FormBase          ; generate the string
;
;          LD          HL,OP3             ; start of string
;          B_CALL      PutS              ; display string

```

FormDCplx

Category: Display

Description: Converts a CplxObj (pair of floating-point numbers) in OP1/OP2 into a displayable string.

Use the current mode settings SCI, ENG, NORMAL, FIX setting, and complex number display format to format the string.

The output can also be formatted as a fraction or a Degrees, Minutes, Seconds (DMS) number.

Inputs:

Registers: None

Flags: To use the current format settings:
(Flags + fmtFlags) copies to (Flags + fmtOverride)

To override the current settings, modify the following flags:

Resetting the next two flags sets the NORMAL display mode.

fmtExponent, (fmtOverride) = 1 for scientific display mode

fmtEng, (fmtOverride) = 1 for engineering display mode

These flags control the formatting of complex numbers.

rectMode, (fmtOverride) = 1 for rectangular complex display

fmtEng, (fmtOverride) = 1 for polar complex display

Setting the next three flags will signify DMS formatting.

fmtBin, (fmtOverride)

fmtHex, (fmtOverride)

fmtOct, (fmtOverride)

Setting the next two flags will signify Fraction formatting.

fmtHex, (fmtOverride)

fmtOct, (fmtOverride)

Others: (fmtDigits) = 0FFh for FLOAT, no fix setting
= 0 – 9 if fix setting is specified

OP1 = value to format

Outputs:

Registers: BC = length of string

Flags: None

Others: String returned in RAM starting in OP3, and is 0 terminated.

Registers All

destroyed:

RAM used: OP1 – OP6

(continued)

FormDCplx *(continued)*

Remarks: If the current display mode settings are SCI or ENG, the output string will reflect the setting. The value is rounded based on the maximum width entered and the current fix setting.

Example: Generate a random complex number and display it at the current cursor position. Use all the current format settings except force SCI formatting.

```

        B_CALL    Random                ; OP1 = random number
        RST       rPushRealO1          ; save
;
        B_CALL    Random                ; OP1 = random number
        B_CALL    PopRealO2             ; OP2 = 2nd part of
;                                     ; floating-point number
;
        LD        A,(flags+fmtFlags)    ; get current format
;                                     ; settings
        RES       fmtEng,A
        SET       fmtexponential        ; override current and
;                                     ; set SCI formatting
        LD        (flags+fmtOverride),A ; set override flags
;
        B_CALL    FormDCplx             ; generate the string
;
        LD        HL,OP3                ; start of string
        B_CALL    PutS                  ; display string

```

FormEReal

Category: Display

Description: Converts a RealObj (single floating-point number) in OP1 into a displayable string.

This routine will ignore all format settings.

Specify the maximum width allowed for the string generated.

Inputs:

Registers: ACC = maximum width of output, minimum of six

Flags: None

Others: OP1 = value to format

Outputs:

Registers: BC = length of string

Flags: None

Others: String returned in RAM starting in OP3, and is 0 terminated.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: If the current display mode settings are SCI or ENG, the output string will reflect the setting. The value is rounded based on the maximum width entered and the current fix setting.

Example: Generate a random number and display it with a maximum of six characters at the current cursor position. Ignore all format settings when generating the string to display.

```

B_CALL Random      ; OP1 = random number
LD      A,6         ; max width to format value with
B_CALL FormEReal    ; generate the string
;
LD      HL,OP3       ; start of string
B_CALL PutS         ; display string

```

FormReal

Category: Display

Description: Converts a RealObj (single floating-point number) in OP1 into a displayable string.

Specify the maximum width allowed for the string generated.

Inputs:

Registers: ACC = maximum width of output, minimum of six

Flags: fmtExponent, (fmtFlags) = 1 for scientific display mode
 fmtEng, (fmtFlags) = 1 for engineering display mode
 If both of the above flags are reset, then NORMAL display mode.

Others: (fmtDigits) = 0FFh for FLOAT, no fix setting
 = 0 – 9 if fix setting is specified

OP1 = value to format

Outputs:

Registers: BC = length of string

Flags: None

Others: String returned in RAM starting in OP3, and is 0 terminated.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: If the current display mode settings are SCI or ENG, the output string will reflect the setting. The value is rounded based on the maximum width entered and the current fix setting.

Example: Generate a random number and display it with a maximum of six characters at the current cursor position.

```

      B_CALL    Random      ; OP1 = random number
      LD        A,6         ; max width to format value with
      B_CALL    FormReal    ; generate the string
;
      LD        HL,OP3      ; start of string
      B_CALL    PutS        ; display string

```


LoadPattern

Category: Display

Description: Loads the font pattern for a character to RAM. Also includes the characters width in pixels. This will work for both variable width and 5x7 fonts.

Inputs:

Registers: ACC = character equate

Flags: fracDrawLFont, (IY + fontFlags) = 1 to use Large 5x7 font
= 0 to use variable width font

Others: None

Outputs:

Registers: None

Flags: None

Others: For large 5x7 font: RAM @ IFont_record = width of character, seven-byte font
For variable width font: RAM @ sFont_record = width of character, seven-byte font

The first byte of the font is the pixel mapping for the top row and each subsequent byte is the next row.

The LSB of each byte represents the right most pixel of a row.

Registers destroyed: All

RAM used:

Remarks: If fracDrawLFont is set, it must be reset.

Example:

Load_SFont

Category: Display

Description: Copies small font attributes to RAM for a particular display character.

Inputs:

Registers: HL = offset into small font table

Flags: None

Others: None

Outputs:

Registers: HL = pointer to sFont_record RAM

Flags: None

Others: sFont_record...sFont_record + 7 = font

Registers destroyed: DE, HL

Remarks: This might be useful, if you wish to write your own **LoadPattern** or **VPutMap** routine for displaying small display characters. The system character fonts (large and small) use eight-bytes per character.

To convert a character number to a table offset, multiply the number by eight.

Example: Find the width of the small display character f:

```
LD      A, 'F'
LD      L, A
LD      H, 0
ADD     HL, HL          ; * 2 turn character into an
                        ; offset.
ADD     HL, HL          ; * 4
ADD     HL, HL          ; * 8 multiply by 8 to get
                        ; table offset.
B_CALL  Load_SFont     ; sFont_record =
                        ; 03,00,02,04,06,04,04,00
LD      A, (HL)        ; 1st byte is width
```

OutputExpr

Category: Display

Description: Converts a numeric value, string or equation, into a string and displays it using the large 5x7 font. This routine should be used with the split screen setting to set to FullScreen.

Inputs:

Registers: H = column number to display at: e.g., 0...15

L = row number to display at: e.g., 0...7

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video
appTextSave, (IY + appFlags) = 1 to write character to **textShadow** also

Others: OP1/OP2 = what to display:
Floating-point number in OP1
Complex number in OP1/OP2
A variable name in OP1 of type: complex, list (real/complex), matrix, string, equation.

Outputs:

Registers: None

Flags: None

Others: System errors can be generated, See the Error Handlers section in Chapter 2.
String output to display.

Registers destroyed: All

Remarks: Previous cursor setting is restored to curRow and curCol. Output will wrap to next line if complete string does not fit on a single line. Output will stop at bottom of screen.

Example: Output the contents of matrix variable [A] at cursor location row 2, column 3.

```

                LD      HL,matAname
                RST      rMov9ToOP1      ; OP1 = matrix [A] name
            ;
                AppOnErr  Catch_Error    ; install error handler
            ;
                LD      HL,3*256+2      ; row 2 column 3
                B_CALL   OutputExpr
            ;
                AppOffErr
            ;
Catch_Error:
                RET

```

PutC

Category: Display

Description: Displays a character and advance cursor.

Inputs:

Registers: A = character to display

Flags: textInverse: = 0, normal character 1, invert character

Others: curRow, curCol = display row and column values

Outputs:

Registers: None

Flags: None

Others: curRow, curCol Updated

**Registers
destroyed:** None

Remarks: This routine calls **PutMap** to do the character display.
This may cause a screen scroll if on the bottom line.

Example:

```
LD      HL,0801h      ; curRow = 1, curCol = 8
LD      (curRow),HL
LD      A,"H"
B_CALL  PutC
LD      A,"I"
B_CALL  PutC
;(PutS might be more useful for multiple characters)
```

PutMap

Category: Display

Description: Displays a character in the large font without affecting cursor position.

Inputs:

Registers: ACC = character to display, see TI83plus.inc

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video
 appTextSave, (IY + appFlags) = 1 to write char to **textShadow** also
 preClrForMode, (IY + newDispF) = 1 to preclear the character space before
 writing
 This is done when toggling between inverted and uninverted.

Others: (curRow) = home screen row to display in, 0–7
 (curCol) = home screen column to display in, 0–15

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: See: **PutC**.

Example: Display char C in row 3 column 4:

```

                                LD      HL,4*256+3
                                LD      (curRow),HL      ; set curRow & curCol
;
                                LD      A,'C'
                                B_CALL  PutMap
;

```

PutPS

Category: Display

Description: Displays a string with a leading length byte residing in RAM, at the current cursor position, and stops at the bottom of the display. This routine uses the large 5x7 font.

Inputs:

Registers: HL = pointer to length byte of string followed by the string

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video
appAutoScroll, (IY + appFlags) = 1 to scroll if need to display past the bottom of the display.
appTextSave, (IY + appFlags) = 1 to write character to **textShadow** also.
preClrForMode, (IY + newDispF) = 1 to preclear the character space before writing. This is done when toggling between inverted and noninverted.

Others: (curRow) = cursor row position, (0 – 7)
(curCol) = cursor column position, (0 – 15)

Outputs:

Registers: None

Flags: Carry = 1 if entire string was displayed
Carry = 0 if string did not fit in the display

Others: curRow and curCol are updated to the position after the last character displayed.

Registers destroyed: All but DE

Remarks: It is recommended that this routine be placed in-line so that strings can be displayed from an application without copying them to RAM first. See the Display Routines section in Chapter 2 for further information.

(continued)

PutPS *(continued)*

Example:

```

PutPS:
    LD      B,(HL)      ; B = length of string
    INC     HL
    LD      A,B
    OR      A
    RET     Z            ; IF LENGTH IS 0 RET

PutPS10:
    LD      A,(HL)      ; get a character of string name
    INC     HL

PutPS20:
    B_CALL  PutC        ; display one character of string

PutPS30:
    LD      A,(curRow)
    LD      C,A
    LD      A,(winBtm)
    CP      C            ; IS CURSOR OFF SCREEN ?
    RET     Z            ; RET IF YES

;
    DJNZ    PutPS10     ; display rest of string
    RET

```

PutS

Category: Display

Description: Displays a zero (0) terminated string residing in RAM at the current cursor position. This routine uses the large 5x7 font.

Inputs:

Registers: HL = pointer to start of string

Flags: textInverse, (IY + textFlags) = 1 to display in reverse video
appAutoScroll, (IY + appFlags) = 1 to scroll if need to display past the bottom of the display.
appTextSave, (IY + appFlags) = 1 to write character to **textShadow** also.
preClrForMode, (IY + newDispF) = 1 to preclear the character space before writing. This is done when toggling between inverted and noninverted.

Others: (curRow) = cursor row position, (0 – 7)
(curCol) = cursor column position, (0 – 15)

Outputs:

Registers: None

Flags: Carry = 1 if entire string was displayed
Carry = 0 if string did not fit in the display

Others: curRow and curCol are updated to the position after the last character displayed.

Registers destroyed: HL

Remarks: To avoid having to copy strings from an application to RAM before using this routine, it is much more efficient to place this routine inside of the application. By doing so, the application can display strings without first having to copy to RAM.

(continued)

PutS *(continued)*

```

Example:      PutS:
                  PUSH      BC
                  PUSH      AF
                  LD         A,(winBtm)
                  LD         B,A          ; B = bottom line of window
..10:
                  LD         A,(HL)      ; get a character of string name
                  INC        HL
                  OR         A           ; end of string?
                  SCF          ; indicate entire string was
                                ; displayed
                  JR         Z,..20      ; yes --->
                  B_CALL     PutC        ; display one character of string
;
                  LD         A,(curRow)  ; check cursor position
                  CP         B           ; off end of window?
                  JR         C,..10      ; no, display rest of string
..20:
                  POP        BC          ; restore A (but not F)
                  LD         A,B
                  POP        BC          ; restore BC
                  RET

```

PutTokString

Category: Display

Description: Displays the string for a token at the current cursor location.

Inputs:

Registers: DE = token value. If a one-byte token then D = 0, E = token.

Flags: None

Others: (curRow) = home screen row to display in, 0–7
(curCol) = home screen column to display in, 0–15

Outputs:

Registers: None

Flags: None

Others: String displayed with wrapping.

**Registers
destroyed:** All

Remarks:

Example: Display the string for the Sin(token at the current cursor location:

```
LD      D,0
LD      E,tSin      ; DE = token
;
B_CALL  PutTokString ; get its string and display
;                  ; it.
```

RestoreDisp

Category: Display

Description: Displays one to 64 rows of the display starting with the top row.

Inputs:

Registers: HL = pointer to ROM/RAM of the data for the first row to display, from left to right. This is followed by the remaining row's data. Each row is stored in 12-bytes, the first column is bit seven of the first byte for each row.

B = number of rows to be displayed

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Data written to the display.

Interrupts are disabled, turn them back on if needed.

Registers destroyed: All

RAM used: curXRow — 1 byte

Remarks:

Example: Copy the first 10 lines of the graph buffer to the display.

```

                LD        HL,plotSScreen    ; start of buffer
                LD        B,10              ; 10 rows to display
;
                B_CALL    RestoreDisp
;
                EI                          ; re-enable interrupts
;
```

RunIndicOff

Category: Display

Description: Turns off run indicator.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_CALL RunIndicOff

RunIndicOn

Category: Display

Description: Turns on run indicator.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: `B_CALL` `RunIndicOn`

SaveDisp

Category: Display

Description: Copies a bit image of the current display to RAM.

Inputs:

Registers: HL = pointer to RAM location to save the image — the bit image of the display is 768 bytes in size.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Contents of display written to RAM. Interrupts are disabled.

Registers destroyed: All

RAM used: curXRow

Remarks: Split screen modes are ignored, the entire display is copied.

Example: Copy the current display to the graph backup buffer, *plotSScreen*.

```
LD      HL,plotSScreen
B_CALL  SaveDisp
RET
```

SetNorm_Vals

Category: Display

Description: Sets display attributes to full screen mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Display attributes set to full screen. Allows for full screen drawing and text displaying.

Registers destroyed: All

Remarks: This routine should only be used in combination with the setting of appropriate system flags that control the screen split settings. See the Display and Split Screen Modes sections in Chapter 2 for further information.

Example:

SFont_Len

Category: Display

Description: Returns the width, in pixels, a character would use if displayed using the small variable width font.

Inputs:

Registers: HL = offset into the font look-up table. This is generated by multiplying the character equate of a character by eight.

Flags: None

Others: None

Outputs:

Registers: ACC = number of pixels needed to display the character using the small font.

Flags: None

Others: None

Registers destroyed: All B

Remarks:

Example: Return the width in pixels of the small font character:

```
LD      HL,Scolon*8      ; compute offset
B_CALL  SFont_Len
```


SStringLength

Category: Display

Description: Returns the width in pixels a string would use if displayed using the small variable width font.

Inputs:

Registers: HL = pointer to the string, with the first byte being the number of characters in the string. The string must reside in RAM.

Flags: None

Others: None

Outputs:

Registers: ACC and B = number of pixels needed to display the string using the small font.

Flags: None

Others: None

Registers destroyed: All but HL

Remarks:

Example:

VPutS

Category: Display

Description: Displays a zero (0) terminated string at the current pen location. Uses either the variable width font or the large 5x7 font.

The advantage to displaying the large font with this routine instead of the **PutS** routine is the string can be placed at any location on the screen. With the **PutS** routine, the string can only be displayed in the 8 row by 16 column grid specified by (curRow) and (curCol).

Inputs:

Registers: HL = pointer to 0 terminated string in RAM.

Flags:

textInverse, (IY + textFlags)	= 1 for reverse video
textEraseBelow, (IY + textFlags)	= 1 to erase line below character
textWrite, (IY + sGrFlags)	= 1 to write to graph buffer not display
fracDrawLFont, (IY + fontFlags)	= 1 use 5x7 font
	= 0 use variable width font (default)

Others: (penCol) = pen column to display at
(penRow) = pen row to display at

Outputs:

Registers: None

Flags: None

Others: CA = 1 if could not fit on the row of the screen entirely

Registers destroyed: HL

Remarks: Pen location (0,0) is the upper left corner of the display. If fracDrawLFont is set, it must be reset. It is recommended that the following routine be placed in-line so that strings can be displayed from an application without copying them to RAM first. See the Display Routines section in Chapter 2 for further information.

(continued)

VPutS *(continued)*

```

VPutS:
        PUSH        AF
        PUSH        DE
        PUSH        IX

    ..10:
        LD          A,(HL)      ; get a character of string name
        INC         HL
        OR          A           ; end of string?
        JR          Z,..20      ; yes --->
        B_CALL      VPutMap     ; display one character of string
        JR          NC,..10      ; display rest of string IF FITS

    ..20:
        POP         IX
        POP         DE
        POP         AF
        RET

```

Example: Display Hello world in variable width font at the current pen location.

```

        LD          HL,Hellostr
        LD          DE,OP1
        LD          BC,14
        LDIR                               ; copy string to RAM
;
        LD          HL,OP1
        B_CALL      VPutS
;
        RET
;
Hellostr:
        DB          "Hello World",0

```

VPutSN

Category: Display

Description: Displays a string of known length at the current pen location. Uses either the variable width font or the large 5x7 font.

The advantage to displaying the large font with this routine instead of the **PutS** routine, is the string can be placed at any location on the screen. With the **PutS** routine, the string can only be displayed in the 8 row by 16 column grid specified by (curRow) and (curCol).

Inputs:

Registers: HL = pointer to first character of string in RAM
B = number of characters to display

Flags: textInverse, (IY + textFlags) = 1 for reverse video
textEraseBelow, (IY + textFlags) = 1 to erase line below character
textWrite, (IY + sGrFlags) = 1 to write to graph buffer not display
fracDrawLFont, (IY + fontFlags) = 1 use 5x7 font
= 0 use variable width font (default)

Others: (penCol) = pen column to display at
(penRow) = pen row to display at

Outputs:

Registers: None

Flags: None

Others: CA = 1 if could not fit on the row of the screen entirely

Registers destroyed: HL

Remarks: Pen location (0,0) is the upper left corner of the display. If fracDrawLFont is set, it must be reset. It is recommended that the following routine be placed in-line so that strings can be displayed from an application without copying them to RAM first. See the Display Routines section in Chapter 2 for further information.

(continued)

VPutSN *(continued)*

```

VPutSN:
        PUSH        AF
        PUSH        DE
        PUSH        IX

..10:
        LD          A,(HL)      ; get a character of string name
        INC         HL
        B_CALL      VPutMap     ; display one character of string
        JR          C,PP11      ; JUMP IF NO ROOM ON LINE
        DJNZ        ..10       ; display rest of string

PP11:
        POP         IX
        POP         DE
        POP         AF
        RET

```

Example: Display Hello world in variable width font at the current pen location.

```

        LD          HL,Hellostr
        LD          DE,OP1
        LD          BC,14
        LDIR                               ; copy string to RAM
;
        LD          HL,OP1
        LD          B,11                  ; length of string
        B_CALL      VPutSN
;
        RET
;
Hellostr:
        DB          "Hello World"

```

A

System Routines — Edit

CloseEditBufNoR.....	203
CursorOff.....	204
CursorOn.....	205
DispEOL.....	206
KeyToString.....	207

CloseEditBufNoR

Category: Edit

Description: Closes edit buffer, but does not delete it.

Inputs:

Registers: None

Flags: editOpen, (IY + editFlags) set if open

Others: None

Outputs:

Registers: None

Flags: None

Others: Adjusts free RAM pointers

Registers destroyed: All

Remarks: An edit session allocates all available RAM, but generally only a portion of that RAM is actually used.

This routine is used to free up any extra RAM after an edit is finished and before the parser is invoked to evaluate the input.

Same as:

```

B_CALL    CanAlphaIns      ; cancel alpha and insert
           ; mode
B_CALL    CloseEditEqu     ; return edit buffer to
           ; user memory
RET

```

Example: ;

```

B_CALL    isEditEmpty      ; is edit buffer empty?
JR        NZ,..08          ; no
B_CALL    CloseEditBuf     ; close & delete buffer
           ; without parsing
RET

```

08:

```

B_CALL    CloseEditBufNoR  ; close but do not delete
CALL      AtName          ; Name of edit buffer
B_CALL    ParseInp         ; parse. result -> OP1
           ; store result
B_CALL    ReleaseBuffer    ; throw away edit buffer.
RET

```


CursorOff

Category: Edit

Description: Turns off the cursor if it is turned on and disable blinking.

Inputs:

Registers: None

Flags: curOn, (IY + curFlags) = 1 if cursor is currently on.
appCurGraphic, (IY + appFlags) = 1 if the graphic cursor
This mode should not be set by an application.
appCurWord, (IY + appFlags) = 1 if a full word cursor
This mode should not be set by an application.

Others: If a normal edit cursor:
(curRow), (curCol) = cursor location
(curUnder) = character the cursor is covering
If a graphic cursor:
(curGX), (curGY) = center pixel location of cursor
(curGStyle) = which graph cursor is active
If a full word cursor:
These are specific to the current context and entries are made in-line in the
cursor blink routine.

Outputs:

Registers: None

Flags: curOn, (IY + curFlags) = is reset
curAble, (IY + curFlags) = is reset to disable future blinking

Others: None

**Registers
destroyed:** All

Remarks:

Example:

CursorOn

Category: Edit

Description: Enables cursor blinking and show the cursor.

Inputs:

Registers: None

Flags: curLock, (IY + curFlags) = 1 if cursor is locked disabled, the cursor cannot be turned on to blink.

appCurGraphic, (IY + appFlags) = 1 if the graphic cursor
This mode should not be set by an application.

appCurWord, (IY + appFlags) = 1 if a full word cursor
This mode should not be set by an application.

Others: If a normal edit cursor:
(curRow), (curCol) = cursor location

If a graphic cursor:
(curGX), (curGY) = center pixel location of cursor
(curGStyle) = which graph cursor is active

If a full word cursor:
These are specific to the current context and entries are made in-line in the cursor blink routine.

Outputs:

Registers: None

Flags: curOn, (IY + curFlags) = is set
curAble, (IY + curFlags) = is set to enable future blinking

Others: (curUnder) = character the cursor is covering

Registers destroyed: All

Remarks:

Example:

DispEOL

Category: Edit

Description: Displays edit buffer to End of Line.

Inputs:

Registers: None

Flags: None

Others: editBuffer pointers

Outputs:

Registers: Display modified

Flags: None

Others: None

Registers destroyed: AF, BC, DE, HL

Remarks: Displays buffer from editTail to editBtm or until the end of the line is reached. If the buffer is finished before reaching the end of line, then **EraseEOL** is called to erase to the end of the line. Current curCol value is saved and restored by this routine; it is not modified. Since this routine only displays to the end of the current line, curRow is not modified.

Example:

KeyToString

Category: Edit

Description: Converts key to a string value.

Inputs:

Registers: DE = key
D = 0 if a one-byte key

Flags: None

Others: None

Outputs:

Registers: HL = keyToStrRam (keyForStr + 1)

Flags: None

Others: keyForStr initialized to string

Registers destroyed: AF, BC, DE, HL

Remarks: Keys are converted to tokens (if possible) and the token string copied to the keyForStr RAM area (18 bytes).

HL points to the length byte of the string (in keyToStrRam).
See TI83plus.inc for key and token values.

Example: To display the string for the Continue key:

```
LD      D,0      ; "Continue" is a one byte key,
                ; so set to 0.
LD      E,kCont  ; "Continue"
B_CALL  KeyToString ; convert to string: HL points
                ; to keyToStrRam.
B_CALL  PutPSB   ; display string preceded by a
                ; length byte...
B_CALL  EraseEOL ; erase the rest of the line if
                ; need be.
```

keyToStrRam would appear as follows:

08h, 43h, 6Fh, 6Eh, 74h, 69h, 6Eh, 75h, 65h
(Length of string is eight bytes, followed by the ASCII characters Continue.)

See TI83plus.inc or Appendix B for the TI-83 Plus character set values.

A

System Routines — Error

ErrArgument	209
ErrBadGuess	210
ErrBreak	211
ErrD_OP1_0	212
ErrD_OP1_LE_0	213
ErrD_OP1Not_R	214
ErrD_OP1NotPos	215
ErrD_OP1NotPosInt	216
ErrDataType	217
ErrDimension	218
ErrDimMismatch	219
ErrDivBy0	220
ErrDomain	221
ErrIncrement	222
ErrInvalid	223
ErrIterations	224
ErrLinkXmit	225
ErrMemory	226
ErrNon_Real	227
ErrNonReal	228
ErrNotEnoughMem	229
ErrOverflow	230
ErrSignChange	231
ErrSingularMat	232
ErrStat	233
ErrStatPlot	234
ErrSyntax	235
ErrTolTooSmall	236
ErrUndefined	237
JError	238
JErrorNo	239

ErrArgument

Category: Error

Description: Jumps to system error handler routine with the message ERR: ARGUMENT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrArgument

ErrBadGuess

Category: Error

Description: Jumps to system error handler routine with the message ERR: BAD GUESS.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrBadGuess

ErrBreak

Category: Error

Description: Jumps to system error handler routine with the message ERR: BREAK.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrBreak

ErrD_OP1_0

Category: Error

Description: If OP1 = 0.0, domain error system will take over with message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** A

Remarks:

Example: B_CALL ErrD_OP1_0

ErrD_OP1_LE_0

Category: Error

Description: If $OP1 \leq 0$ (not positive), domain error system will take over with message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** A

Remarks:

Example: B_CALL ErrD_OP1_LE_0

ErrD_OP1Not_R

Category: Error

Description: If OP1 is not real, domain error system will take over with message
ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** A

Remarks:

Example: B_CALL ErrD_OP1Not_R

ErrD_OP1NotPos

Category: Error

Description: If OP1 is not positive, domain error system will take over with message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: A

Remarks:

Example: B_CALL ErrD_OP1NotPos

ErrD_OP1NotPosInt

Category: Error

Description: If OP1 is not positive integer, domain error system will take over with message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** A

Remarks:

Example: B_CALL ErrD_OP1NotPosInt

ErrDataType

Category: Error

Description: Jumps to system error handler routine with the message ERR: DATA TYPE.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrDataType

ErrDimension

Category: Error

Description: Jumps to system error handler routine with the message ERR: INVALID DIM.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrDimension

ErrDimMismatch

Category: Error

Description: Jumps to system error handler routine with the message ERR: DIM MISMATCH.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrDimMismatch

ErrDivBy0

Category: Error

Description: Jumps to system error handler routine with the message ERR: DIVIDE BY 0.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrDivBy0

ErrDomain

Category: Error

Description: Jumps to system error handler routine with the message ERR: DOMAIN.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrDomain

ErrIncrement

Category: Error

Description: Jumps to system error handler routine with the message ERR: INCREMENT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrIncrement

ErrInvalid

Category: Error

Description: Jumps to system error handler routine with the message ERR: INVALID.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrInvalid

ErrIterations

Category: Error

Description: Jumps to system error handler routine with the message ERR: ITERATIONS.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrIterations

ErrLinkXmit

Category: Error

Description: Jumps to system error handler routine with the message ERR: IN XMIT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrLinkXmit

ErrMemory

Category: Error

Description: Jumps to system error handler routine with the message ERR: MEMORY.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrMemory

ErrNon_Real

Category: Error

Description: In Real mode, the result of a calculation yielded a complex result. This error is not returned during graphing. The TI-83 Plus allows for undefined values on a graph.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks: The error system will take over and report the error to the screen. Any application that was executing at that time will be aborted.

Example: `B_JUMP ErrNon_Real`

ErrNonReal

Category: Error

Description: Errors if nonreal input to command error. System will take over with message ERR: DATA TYPE.

Inputs:

Registers: B = number of arguments to check.

Flags: None

Others: Arguments on Floating Point Stack.

Outputs:

Registers: None

Flags: None

Others: Error if nonreal input to command.
Screen will have data type error menu.

**Registers
destroyed:** A, B

Remarks:

Example: `B_CALL ErrNonReal`

ErrNotEnoughMem

Category: Error

Description: If not enough memory, memory error system will take over with message ERR: MEMORY.

Inputs:

Registers: HL = number of bytes needed.

Flags: None

Others: None

Outputs:

Registers: DE = Amount of memory requested.

Flags: CA = 1 if not enough room.

Others: None

**Registers
destroyed:**

Remarks:

Example: B_CALL ErrNotEnoughMem

ErrOverflow

Category: Error

Description: Jumps to system error handler routine with the message ERR: OVERFLOW.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrOverflow

ErrSignChange

Category: Error

Description: Jumps to system error handler routine with the message ERR: NO SIGN CHANGE.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrSignChange

ErrSingularMat

Category: Error

Description: Jumps to system error handler routine with the message ERR: SINGULARITY.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrSingularMat

ErrStat

Category: Error

Description: Jumps to system error handler routine with the message ERR: STAT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrStat

ErrStatPlot

Category: Error

Description: Jumps to system error handler routine with the message ERR: STATPLOT.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrStatPlot

ErrSyntax

Category: Error

Description: Jumps to system error handler routine with the message ERR: SYNTAX.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrSyntax

ErrTolTooSmall

Category: Error

Description: Jumps to system error handler routine with message ERR: TOL NOT MET.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrTolTooSmall

ErrUndefined

Category: Error

Description: Jumps to system error handler routine with the message ERR: UNDEFINED.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example: B_JUMP ErrUndefined

JError

Category: Error

Description: Entry point into system error routine. This entry is almost always used in conjunction with an error exception handler.

After an error exception handler is tripped and control is returned to an application, the application may choose to modify the error by changing the error to another or most likely removing the GoTo option. This entry point is where the application would B_JUMP to continue on with the error after modifying it.

See the Error Handlers section in Chapter 2.

Inputs:

Registers: ACC bits (0 – 6) = error code
ACC bit (7) = 0 for no GoTo option
ACC bit (7) = 1 for allowing a GoTo option

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: (errNo) = error code (one-byte)
System error is displayed or another error.
Exception handler is tripped and the error is suppressed.

Registers destroyed: All

Remarks:

Example:

JErrorNo

Category: Error

Description: Same as JError except the error code is stored in the byte (errNo).

Remarks: See JError.

A

System Routines — Floating Point Stack

AllocFPS.....	241
AllocFPS1.....	242
CpyStack.....	243
CpyO1ToFPST, CpyO1ToFPS1, CpyO1ToFPS2, CpyO1ToFPS3, CpyO1ToFPS4, CpyO1ToFPS5, CpyO1ToFPS6, CpyO1ToFPS7, CpyO2ToFPST, CpyO2ToFPS1, CpyO2ToFPS2, CpyO2ToFPS3, CpyO2ToFPS4, CpyO3ToFPST, CpyO3ToFPS1, CpyO3ToFPS2, CpyO5ToFPS1, CpyO5ToFPS3, CpyO6ToFPST, CpyO6ToFPS2	244
CpyTo1FPST, CpyTo1FPS1, CpyTo1FPS2, CpyTo1FPS3, CpyTo1FPS4, CpyTo1FPS5, CpyTo1FPS6, CpyTo1FPS7, CpyTo1FPS8, CpyTo1FPS9, CpyTo1FPS10, CpyTo1FPS11, CpyTo2FPST, CpyTo2FPS1, CpyTo2FPS2, CpyTo2FPS3, CpyTo2FPS4, CpyTo2FPS5, CpyTo2FPS6, CpyTo2FPS7, CpyTo2FPS8, CpyTo3FPST, CpyTo3FPS1, CpyTo3FPS2, CpyTo4FPST, CpyTo5FPST, CpyTo6FPST,	245
CpyTo6FPS2, CpyTo6FPS3.....	245
CpyToFPST.....	246
CpyToFPS1	247
CpyToFPS2.....	248
CpyToFPS3.....	249
CpyToStack.....	250
PopOP1, PopOP3, PopOP5	251
PopReal.....	252
PopRealO1, PopRealO2, PopRealO3, PopRealO4, PopRealO5, PopRealO6.....	253
PushOP1, PushOP3, PushOP5.....	254
PushReal.....	255
PushRealO1, PushRealO2, PushRealO3, PushRealO4, PushRealO5, PushRealO6	256

AllocFPS

Category: Floating Point Stack

Description: Allocates space on the Floating Point Stack by specifying a number of nine-byte entries.

Inputs:

Registers: HL = number of entries to allocate

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If no memory error, the new entries are allocated on the end of the FPS.
FPST = last new entry allocated.

**Registers
destroyed:** All

Remarks: No initialization of the allocated entries is done. See section on Floating Point Stack.

Example:

AllocFPS1

Category: Floating Point Stack

Description: Allocates space on the Floating Point Stack by specifying a number of bytes, THIS MUST BE A MULTIPLE OF NINE.

Inputs:

Registers: HL = number of bytes to allocate — a multiple of nine.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If no memory error, the new entries are allocated on the end of the FPS.
FPST = last new entry allocated.

**Registers
destroyed:** All

Remarks: No check is made for the number of bytes being a multiple of nine. No initialization of the allocated entries is done. See section on Floating Point Stack.

Example:

CpyStack

Category: Floating Point Stack

Description: Copies nine-bytes from one of the systems nine-byte stacks, FPS and ES. Only the FPS (Floating Point Stack) is documented for application use. This routine should be used in the manner described in the example.

Input:

Registers: C = number of bytes from the next free byte in the stack back to the entry copying from. This will always be a multiple of nine.
 HL = address of next free byte for the stack, for the FPS the address is stored in the bytes (FPS).
 DE = pointer to the nine-bytes of RAM to copy the entry to.

Flags: None

Others: None

Output:

Registers: HL = pointer to byte after the entry just copied from.
 DE = DE + 9

Flags: None

Others: Nine bytes copied to the RAM from the stack entry.

Registers destroyed: All

Remarks: See Floating Point Stack documentation.

Example: Copy from FPS10 to OP2.

```

                LD      HL,(FPS)      ; copy to FPS
                LD      DE,(OP2)      ; start of 9 bytes to copy to
                                      ; FPS10
;
                LD      C,(10+1)*9    ; C = offset back to FPS10,
                                      ; 11*9 bytes
;
                B_CALL  CpyStack      ; copy to OP2 from FPS10
;
```


CpyO1ToFPST, CpyO1ToFPS1, CpyO1ToFPS2, CpyO1ToFPS3, CpyO1ToFPS4, CpyO1ToFPS5, CpyO1ToFPS6, CpyO1ToFPS7, CpyO2ToFPST, CpyO2ToFPS1, CpyO2ToFPS2, CpyO2ToFPS3, CpyO2ToFPS4, CpyO3ToFPST, CpyO3ToFPS1, CpyO3ToFPS2, CpyO5ToFPS1, CpyO5ToFPS3, CpyO6ToFPST, CpyO6ToFPS2

Category: Floating Point Stack

Description: This description covers a group of routines that copies a single nine-byte OP register (OP1 – OP6), to an entry on the Floating Point Stack (FPS). For example, CpyO1ToFPS2: OP1 is copied to (FPS2).

Inputs:

Registers: None

Flags: None

Others: OP register = 9 bytes to copy to FPS entry
For example, CpyO1ToFPS2: OP1 = nine-bytes to copy

Outputs:

Registers: DE = FPS entry following the one copied to
For example, CpyO1ToFPS2: DE = address of FPS1

HL = OP register + 9
For example, **CpyO1ToFPS2**: HL = OP1 + 9

Flags: None

Others: OP register = copy of the nine-byte FPS entry
For example, **CpyTo1FPS2**: OP1 = FPS2 entry

Registers All

destroyed: The OP register is written to.

Remarks: These routines do not allocate or deallocate entries. See entry point **CpyToStack**. See entry point **CpyTo1FPST**. See Floating Point Stack section of Chapter 2.

Example:

**CpyTo1FPST, CpyTo1FPS1, CpyTo1FPS2,
 CpyTo1FPS3, CpyTo1FPS4, CpyTo1FPS5,
 CpyTo1FPS6, CpyTo1FPS7, CpyTo1FPS8,
 CpyTo1FPS9, CpyTo1FPS10, CpyTo1FPS11,
 CpyTo2FPST, CpyTo2FPS1, CpyTo2FPS2,
 CpyTo2FPS3, CpyTo2FPS4, CpyTo2FPS5,
 CpyTo2FPS6, CpyTo2FPS7, CpyTo2FPS8,
 CpyTo3FPST, CpyTo3FPS1, CpyTo3FPS2,
 CpyTo4FPST, CpyTo5FPST, CpyTo6FPST,
 CpyTo6FPS2, CpyTo6FPS3**

Category: Floating Point Stack

Description: This description covers a group of routines that copies a single nine-byte entry from the Floating Point Stack (FPS), to one of the OP registers (OP1 – OP6). For example, CpyTo1FPS2: (FPS2) is copied to OP1.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: HL = FPS entry following one copied
 For example, CpyTo1FPS2: HL = address of FPS1

DE = OP register + 9

For example, **CpyTo1FPS2**: DE = OP1 + 9

Flags: None

Others: OP register = copy of the nine-byte FPS entry
 For example, **CpyTo1FPS2**: OP1 = FPS2 entry

Registers destroyed: All
 The OP register is written to.

Remarks: These routines do not allocate or deallocate entries. See entry point **CpyStack**. See entry point **CpyO1ToFPST**. See Floating Point Stack section of Chapter 2.

Example:

CpyToFPST

Category: Floating Point Stack

Description: Copies nine-bytes from RAM/ROM to FPST, Floating Point Stack top entry.

Input:

Registers: DE = address of nine-bytes to copy to FPST

Flags: None

Others: None

Output:

Registers: HL = input DE + 9
DE = (FPS), next free byte on the stack

Flags: None

Others: None

**Registers
destroyed:** All

Remarks: See Floating Point Stack documentation.

Example:

CpyToFPS1

Category: Floating Point Stack

Description: Copies nine-bytes from RAM/ROM to FPS1, Floating Point Stack top entry -1.

Input:

Registers: DE = address of nine-bytes to copy to FPS1

Flags: None

Others: None

Output:

Registers: HL = input DE + 9
DE = pointer to FPST entry

Flags: None

Others: None

**Registers
destroyed:** All

Remarks: See Floating Point Stack documentation.

Example:

CpyToFPS2

Category: Floating Point Stack

Description: Copies nine-bytes from RAM/ROM to FPS2, Floating Point Stack top entry -2.

Input:

Registers: DE = address of nine-bytes to copy to FPS2

Flags: None

Others: None

Output:

Registers: HL = input DE + 9
DE = pointer to FPS1 entry

Flags: None

Others: None

**Registers
destroyed:** All

Remarks: See Floating Point Stack documentation.

Example:

CpyToFPS3

Category: Floating Point Stack

Description: Copies nine-bytes from RAM/ROM to FPS3, Floating Point Stack top entry -3.

Input:

Registers: DE = address of nine-bytes to copy to FPS3

Flags: None

Others: None

Output:

Registers: HL = input DE + 9
DE = pointer to FPS2 entry

Flags: None

Others: None

**Registers
destroyed:** All

Remarks: See Floating Point Stack documentation.

Example:

CpyToStack

Category: Floating Point Stack

Description: Copies nine-bytes to one of the systems nine-byte stacks, FPS and ES. Only the FPS (Floating Point Stack) is documented for application use. This routine should be used in the manner described in the example.

Input:

Registers: C = number of bytes from the next free byte in the stack back to the entry copying to. This will always be a multiple of nine.

HL = address of next free byte for the stack, for the FPS the address is stored in the bytes (FPS).

DE = pointer to the nine-bytes to copy to the stack.

Flags: None

Others: None

Output:

Registers: HL = pointer to byte after the entry just copied to.
DE = DE + 9

Flags: None

Others: Nine-bytes copied to the stack entry.

Registers destroyed: All

Remarks: See Floating Point Stack documentation.

Example: Copy from OP2 to FPS10.

```

                LD      HL,(FPS)      ; copy to FPS
                LD      DE,(OP2)      ; start of 9 bytes to copy to
                                      ; FPS10
;
                LD      C,(10+1)*9    ; C = offset back to FPS10,
                                      ; 11*9 bytes
;
                B_CALL  CpyToStack    ; copy to FPS10
;
```

PopOP1, PopOP3, PopOP5

Category: Floating Point Stack

Description: This description covers three entry points that are similar. The description is given for PopOP1. The inputs/outputs are the same for the other two routines replacing OP1/OP2 with either OP3/OP4 or OP5/OP6.

These routines will pop either one or two floating-point numbers off of the top of the FPS. They are used to either pop a real or a complex value off of the top of the FPS without knowing in advance whether a real or a complex value is on the top of the stack.

The top entry (FPST) is popped into OP1. The sign byte of the popped value in OP1 is checked for CplxObj. If it is complex, OP1 is moved to OP2 and the new FPST is popped into OP1. If it is not complex, the floating-point number popped into OP1 is left there.

Input:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If the data type of FPST = RealObj then OP1 = FPST
If the data type of FPST = CplxObj then OP1 = FPS1,
the real part of the complex value
OP2 = FPST, the imaginary part of the complex value.

Registers destroyed: All

RAM used: OP1/OP2 or OP3/OP4 or OP5/OP6 depending on which of the routines is used.

Remarks: When using this routine make sure that the FPST entry is not a complex variable name. If it is, it will be interpreted as a complex value causing two floating-point numbers to be popped from the FPS. See **PopRealO1** and **PopMcplxO1**. See Floating Point Stack section.

Example:

PopReal

Category: Floating Point Stack

Description: Pops the last entry FPST, off of the FPS to an input RAM location. No matter what the data in FPST is only nine (9) bytes are popped off of the stack.

Inputs:

Registers: DE = pointer to RAM location to pop FPST into

Flags: None

Others: None

Outputs:

Registers: DE = DE + 9

Flags: None

Others: The nine-byte entry FPST is removed from the FPS and copied to the nine-bytes starting at address DE.

Registers destroyed: All but the ACC

Remarks: The entry is removed from the FPS shrinking the size of the FPS by nine-bytes. See the Floating Point Stack section.

Example:

PopRealO1, PopRealO2, PopRealO3, PopRealO4, PopRealO5, PopRealO6

Category: Floating Point Stack

Description: This description covers six entry points that are similar. The description is given for **PopRealO1**. The inputs/outputs are the same for the other five routines replacing OP1 with either OP2, OP3, OP4, OP5 or OP6.

Pops the last entry FPST, off of the FPS to OP1. No matter what the data in FPST is, only nine (9) bytes are popped off of the stack.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: The nine-byte entry FPST is removed from the FPS and copied to the nine-bytes starting at address OP1.

Registers destroyed:

Remarks: The entry is removed from the FPS shrinking the size of the FPS by nine-bytes. See **PopOP1**. See the Floating Point Stack section.

Example:

PushOP1, PushOP3, PushOP5

Category: Floating Point Stack

Description: This description covers three entry points that are similar. The description is given for PushOP1. The inputs/outputs are the same for the other two routines replacing OP1/OP2 with either OP3/OP4 or OP5/OP6.

These routines will push either one or two floating-point numbers onto the FPS. It is used to either push a real or a complex value onto the FPS without knowing in advance whether a real or a complex value is being pushed onto the stack.

The sign byte of OP1 is checked for CplxObj. If it is Complex, OP1 is pushed on to the stack and the OP2 is pushed onto the stack. If it is not complex, the floating-point number in OP1 is only pushed onto the stack.

Input:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If the data type of OP1 = RealObj then FPST = OP1
 If the data type of OP1 = CplxObj then FPS1 = OP1,
 the real part of the complex value
 FPST = OP2, the imaginary part of the complex value.

Registers destroyed: All

RAM used: None

Remarks: When using this routine make sure that the OP1 is not a complex variable name. If it is it will be interpreted as a complex value causing two floating-point numbers to be pushed onto the FPS. See **PushRealO1**, **PushMcplxO1**. See the Floating Point Stack section.

Example:

PushReal

Category: Floating Point Stack

Description: Pushes a new entry onto the FPS and copy the nine-bytes at address HL into the new entry. No checks are made on the data that is put onto the stack.

Inputs:

Registers: HL = pointer to nine-bytes to push onto the FPS

Flags: None

Others: None

Outputs:

Registers: HL = HL + 9

Flags: None

Others: FPST = nine-bytes at HL pushed onto the stack

**Registers
destroyed:** All

Remarks: The previous FPST is now entry FPS1. See **PushRealO1**, **PushOP1**. See the Floating Point Stack section.

Example:

PushRealO1, PushRealO2, PushRealO3, PushRealO4, PushRealO5, PushRealO6

Category: Floating Point Stack

Description: This description covers six entry points that are similar. The description is given for PushRealO1. The inputs/outputs are the same for the other five routines replacing OP1 with either OP2, OP3, OP4, OP5 or OP6.

Pushes a new entry onto the FPS and copy the nine-bytes at OP1 into the new entry. No checks are made on the data that is put onto the stack.

Inputs:

Registers: None

Flags: None

Others: OP1 = nine-bytes to push onto the FPS

Outputs:

Registers: None

Flags: None

Others: FPST = nine-bytes at OP1 pushed onto the stack

**Registers
destroyed:**

Remarks: The previous FPST is now entry FPS1. See **PushReal**, **PushOP1**. See the Floating Point Stack section.

Example:

A

System Routines — Graphing and Drawing

AlIEq	259
BufClr	260
BufCpy	261
CircCmd	262
ClearRect	264
CLine	265
CLineS.....	267
ClrGraphRef	269
CPoint	269
CPointS	272
DarkLine	274
DarkPnt	276
Disp	278
DrawCirc2.....	279
DrawCmd	281
DrawRectBorder	282
DrawRectBorderClear.....	283
EraseRectBorder	284
FillRect	285
FillRectPattern	287
GrBufClr	289
GrBufCpy.....	290
GrphCirc.....	291
HorizCmd	292
IBounds	293
IBoundsFull	294
ILine	295
InvCmd.....	297
InvertRect.....	298
IOffset.....	299
IPoint	300

(continued)

Contents *(continued)*

LineCmd.....	302
PDspGrph.....	304
PixelTest.....	305
PointCmd.....	306
PointOn	308
Regraph.....	309
SetAllPlots	310
SetFuncM.....	311
SetParM	312
SetPolM.....	313
SetSeqM.....	314
SetTblGraphDraw	315
TanLnF	316
UCLineS	317
UnLineCmd	318
VertCmd	319
VtoWHLDE	320
Xftol	321
Xitof	322
Yftol	323
ZmDecml.....	324
ZmFit	325
ZmInt	326
ZmPrev.....	327
ZmSquare.....	328
ZmStats	329
ZmTrig.....	330
ZmUsr	331
ZooDefault.....	332

AlIEq

Category: Graphing and Drawing

Description: Select or deselect all graph equations in the current graph mode.

Inputs:

Registers: ACC = 3 to select all equations in the current graph mode
= 4 to deselect all equations in the current graph mode

Flags: Current graph mode: IY + grfModeFlags = flag byte

Others: None

Outputs:

Registers: None

Flags: None

Others: All graph equations for the current mode are deselected.

**Registers
destroyed:** All

RAM used: OP1, OP2

Remarks:

Example:

BufClr

Category: Graphing and Drawing

Description: Executes the routine **GrBufClr** on a bitmap of the graph screen other than *plotSScreen*, the system graph backup buffer.

Inputs:

Registers: HL = pointer to start of graph buffer to clear, 768 bytes

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: RAM cleared.

Registers destroyed: All

Remarks: G-T and Horizontal modes will affect how much of the buffer is cleared. In order to have the entire buffer cleared set to full screen mode.

There are two additional bit image display buffers allocated other than *plotSScreen*, they start at addresses *appBackUpScreen* and *saveSScreen*.

Example:

```
LD      HL, appBackUpScreen
B_CALL  BufClr      ; clear backup
```

BufCpy

Category: Graphing and Drawing

Description: Executes the routine **GrBufCpy** on a bitmap of the graph screen other than **plotSScreen**, the system graph backup buffer. The contents of the buffer are displayed.

Inputs:

Registers: HL = pointer to start of graph buffer to display, 768 bytes

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: G-T and Horizontal modes will affect how much of the buffer is displayed. In order to have the entire buffer displayed, set to full screen mode.

There are two additional bit image display buffers allocated other than **plotSScreen**, they start at addresses **appBackUpScreen** and **saveSScreen**.

Example:

```
LD      HL, appBackUpScreen
B_CALL  BufCpy          ; display backup buffer
```

CircCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a circle on the graph screen given the center and the radius, relative to the current window settings.

Inputs:

Registers: None

Flags: useFastCirc, (IY + plotFlag3) = 1 for fast circle routine that draws the circle in sections simultaneously
 useFastCirc, (IY + plotFlag3) = 0 for normal circle routine that draws in a circular direction
 bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer **plotSScreen**, not to the display

Others: FPST = radius, a floating-point number
 FPS1 = Y value of center, a floating-point number
 FPS2 = X value of center, a floating-point number

The center specified is with respect to the current window settings.

Outputs:

Registers: None

Flags: None

Others: Current graph, and point operation are drawn to the screen and the graph backup buffer, **plotSScreen**.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

Remarks: If a zoom square is not done before using this routine the output circle will most likely not look circular but skewed in either the X or Y axis direction.

If useFastCirc is used, the flag must be reset by the caller.

(continued)

CircCmd *(continued)*

Example: Execute a zoom standard and then draw a circle at (0,0) with radius 3 using the alternate fast circle draw.

```

        B_CALL    ZooDefault          ; standard window
        B_CALL    OP1Set0             ; OP1 = 0
        RST       rPushReal01
        RST       rPushReal01        ; (0,0) pushed
                                        ; onto FPS
;
        B_CALL    OP1Set3             ; radius is 3
        RST       rPushReal01        ; 3 pushed onto
                                        ; FPS
;
        SET       useFastCirc,(IY+plotFlag3) ; fast circle
                                        ; routine
;
        AppOnErr   ClrFlag            ; set up error
                                        ; handler to clear
                                        ; fast circle flag
;
        B_CALL    CircCmd             ;
;
        AppOffErr                      ; remove no error

        RES       useFastCirc,(IY+plotFlag3) ; reset flag
        RET

;
;   come here if error
;
ClrFlag:
        RES       useFastCirc,(IY+plotFlag3) ; reset flag
;
        B_JUMP    JErrorNo           ; continue on with
                                        ; system error
                                        ; handle
;

```

ClearRect

Category: Graphing and Drawing

Description: Clears a rectangular area on the screen (to Pixel off).

Inputs:

Registers: H = upper left corner pixel row
 L = upper left corner pixel column
 D = lower right corner pixel row
 E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
 0: update display and graph buffer
 1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is upper left corner of screen and row = 63, column = 95 is lower right corner of screen.

Area includes row and column of both coordinates.

Inputs must satisfy conditions: $D \geq H$, $E \geq L$.

Modifies **saveSScreen** RAM area.

Example:

```

LD      HL,0000h
LD      DE,3F5Fh
B_CALL  FillRect      ; Make the whole screen
                        ; black

LD      H,0
LD      L,48
LD      D,31
LD      E,95
B_CALL  ClearRect     ; Clear the screen's top
                        ; right quarter

B_CALL  GetKey         ; Get key press
B_JUMP  JForceCmdNoChar ; Exit app
  
```

CLine

Category: Graphing and Drawing

Description: Draws a line between two points specified by graph coordinates. The line is plotted according to the current window settings Xmin, Xmax, Ymin, Ymax.

The points do not need to lie within the current window settings this routine will clip the line to the screen edges if any portion of the line goes through the current window settings.

This routine should only be used to draw lines in reference to the window settings.

ILine can be used to draw lines by defining points with pixel coordinates, which will be a faster draw.

Inputs:

Registers: OP4 — Y1-coordinate
OP3 — X1-coordinate
OP2 — Y2-coordinate
OP1 — X2-coordinate

Flags: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set
vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes
plotLoc, (IY + plotFlags) = 1 to draw to the display only
 = 0 to draw to display and ***plotSScreen*** buffer.
bufferOnly, (IY + plotFlag3) = 1 to draw to ***plotSScreen*** buffer only.

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

CLine *(continued)*

Example:

```

; Draw a line between
; the points (1.5,3)
; & (4,6):
; point to (1.5,3) in
; ROM
LD      HL,Point_1

LD      DE,OP3
LD      BC,18
LDIR

LD      HL,Point_2      ; point to (4,6) in
                        ; ROM
B_CALL  Mov9OP1OP2      ; OP1 = 4 OP2 = 6
B_CALL  PushMCplx01

;
B_CALL  CLine           ; draw the line
RET

Point_1:
DB      0,80h,15h,0,0,0,0,0,0 ; 1.5
DB      0,80h,30h,0,0,0,0,0,0 ; 3

Point_2:
DB      0,80h,40h,0,0,0,0,0,0 ; 4
DB      0,80h,60h,0,0,0,0,0,0 ; 6

```

CLineS

Category: Graphing and Drawing

Description: Draws a line between two points specified by graph coordinates. The line is plotted according to the current window settings Xmin, Xmax, Ymin, Ymax.

The points do not need to lie within the current window settings this routine will clip the line to the screen edges if any portion of the line goes through the current window settings.

This routine should only be used to draw lines in reference to the window settings.

ILine can be used to draw lines by defining points with pixel coordinates, which will be a faster draw.

Inputs:

Registers: FPS2 — Y1-coordinate
 FPS3 — X1-coordinate
 FPS1 — Y2-coordinate
 FPST — X2-coordinate

Flags: plotLoc, (IY + plotFlags) = 1 to draw to the display only
 = 0 to draw to display and **plotSScreen** buffer
 bufferOnly, (IY + plotFlag3) = 1 to draw to **plotSScreen** buffer only

G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this turn off the split screen modes.
 See **ForceFullScreen**.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set
 vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set
 grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

CLineS *(continued)*

Example:

```

; Draw a line between
; the points (1.5,3)
; & (4,6):
LD      HL,Point_1      ; point to (1.5,3) in
                        ; ROM
B_CALL  Mov9OP1OP2      ; OP1 = 1.5 OP2 = 3
B_CALL  PushMCplx01     ; push OP1 and then
                        ; OP2 onto the FPS
;
LD      HL,Point_2      ; point to (4,6) in
                        ; ROM
B_CALL  Mov9OP1OP2      ; OP1 = 4 OP2 = 6
B_CALL  PushMCplx01     ; push OP1 and then
                        ; OP2 onto the FPS
;
B_CALL  CLineS          ; draw the line
RET
Point_1:
DB      0,80h,15h,0,0,0,0,0 ; 1.5
DB      0,80h,30h,0,0,0,0,0 ; 3
Point_2:
DB      0,80h,40h,0,0,0,0,0 ; 4
DB      0,80h,60h,0,0,0,0,0 ; 6

```

ClrGraphRef

Category: Graphing and Drawing

Description: Clears all graph reference flags in the symtable and the temporary symtable.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Graph reference reset

Registers destroyed: HL, DE, BC

Remarks:

Example: B_CALL ClrGraphRef

CPoint

Category: Graphing and Drawing

Description: Turns on, turns off, or inverts a point in the display specified by graph coordinates. The point is plotted according to the current window settings: Xmin, Xmax, Ymin, Ymax.

This routine should only be used to draw points in reference to the window settings.

IPoint can be used to draw points by defining points with pixel coordinates, which causes a faster draw.

Inputs:

Registers: ACC = what to do

0: turn point off

1: turn point on

2: invert point

Flags: G-T and HORIZ split-screen modes affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set

vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set

grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes

plotLoc, (IY + plotFlags) = 1 to draw to the display only

= 0 to draw to display and **plotSScreen** buffer

bufferOnly, (IY + plotFlag3) = 1 to draw to **plotSScreen** buffer only

Others: OP1 — X Coordinate of point

OP2 — Y Coordinate of point

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

CPoint *(continued)*

Example: Draw a point in the graph window at coordinates (1.5,3):

```

                                LD      HL,Point_1      ; point to (1.5,3)
                                B_CALL  Mov9OP1OP2      ; OP1 = 1.5 OP2 = 3
;
                                LD      A,1             ; turn on
                                B_CALL  CPoint          ; draw the point
                                RET
Point_1:
                                DB      0,80H,15H,0,0,0,0,0 ; 1.5
                                DB      0,80H,30H,0,0,0,0,0 ; 3

```

CPointS

Category: Graphing and Drawing

Description: Turns on, turns off or inverts a point in the display specified by graph coordinates. The point is plotted according to the current window settings: Xmin, Xmax, Ymin, Ymax.

This routine should only be used to draw points in reference to the window settings.

IPoint can be used to draw points by defining points with pixel coordinates, which causes a faster draw.

Inputs:

Registers: ACC = what to do

0: turn point off

1: turn point on

2: invert point

Flags: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set

vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set

grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes

plotLoc, (IY + plotFlags) = 1 to draw to the display only

= 0 to draw to display and **plotSScreen** buffer

bufferOnly, (IY + plotFlag3) = 1 to draw to **plotSScreen** buffer only

Others: FPS1 — X Coordinate of point

FPST — Y Coordinate of point

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

CPointS *(continued)*

Example: Draw a point in the graph window at coordinates (1.5,3)

```

;
LD      HL,Point_1      ; point to (1.5,3)
; in ROM
B_CALL  Mov9OP1OP2      ; OP1 = 1.5 OP2 = 3
B_CALL  PushMCplx01     ; push OP1 and then
; OP2 onto the FPS
;
LD      A,1             ; turn on
B_CALL  CPointS         ; draw the point
RET
Point_1:
DB      0,80H,15H,0,0,0,0,0 ; 1.5
DB      0,80H,30H,0,0,0,0,0 ; 3

```

DarkLine

Category: Graphing and Drawing

Description: Draws a line between two pixel points defined by their pixel coordinates.

Inputs: The graph window is defined with the lower left corner of the display to be pixel coordinate (0,0).
The system graphing routines do not normally draw in the last column and the bottom row of the screen, column 95 and row 0.
This routine can be made to use column 95 and row 0 by setting the flag:
fullScrnDraw, (IY + apiFlg4)

Registers: X = column
Y = row

B = X-coordinate of first point — 0...94 (95) see above

C = Y-coordinate of first point — 1(0)...63

D = X-coordinate of second point — 0...94 (95)

E = Y-coordinate of second point — 1(0)...63

Flags: fullScrnDraw, (IY + apiFlg4) = 1 to use column 95 and row 0
plotLoc, (IY + plotFlags) = 1 to draw to the display only
= 0 to draw to display and **plotSScreen** buffer
bufferOnly, (IY + plotFlag3) = 1 to draw to **plotSScreen** buffer only

Others: None

Outputs:

Registers: None

Flags: None

Others: Line drawn where specified.

Registers destroyed: All registers are preserved.

Remarks: If the draw is going to the buffer then the contents of the buffer are used to draw the line and copied to the screen.
No clipping, X, Y points assumed to be defined on the screen.
G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes.
See **ForceFullScreen**.

(continued)

DarkLine *(continued)*

Example:

```
; Clear the screen.
; Draw a line in the display only, between pixel coordinates (25,30)
; and (62,50):
      B_CALL    ClrLCD                ; clear the screen;
      LD        BC,25*256+30          ; 1st point, B = 25,
                                      ; C = 30
      LD        DE,62*256+50          ; 2nd point, D = 62,
                                      ; E = 50
;
      SET        plotLoc,(IY+plotFlags) ; display only
;
      B_CALL    DarkLine              ; draw the line
```


DarkPnt

Category: Graphing and Drawing

Description: Turns on a point in the display specified by graph coordinates. The point is plotted according to the current window settings: Xmin, Xmax, Ymin, Ymax.

This routine should only be used to draw points in reference to the window settings.

IPoint can be used to draw points by defining points with pixel coordinates, which causes a faster draw.

Inputs:

Registers: None

Flags: G-T and HORIZ split screen modes affect how this routine maps the coordinates specified. To avoid this, turn off the split-screen modes. See **ForceFullScreen**.

grfSplit, (IY + sGrFlags)	= 1 if horizontal split mode set
vertSplit, (IY + sGrFlags)	= 1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags)	= 1 to ignore split modes
plotLoc, (IY + plotFlags)	= 1 to draw to the display only
	= 0 to draw to display and plotSScreen buffer.
bufferOnly, (IY + plotFlag3)	= 1 to draw to plotSScreen buffer only

Others: OP1 — X Coordinate of point
OP2 — Y Coordinate of point

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed:

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph, if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

(continued)

DarkPnt *(continued)*

Example: Draw a point in the graph window at coordinates (1.5,3):

```
LD      HL,Point_1      ; point to (1.5, 3)
                        ; in ROM
B_CALL  Mov9OP1OP2      ; OP1 = 1.5 OP2 = 3
;
B_CALL  DarkPnt         ; draw the point
RET
Point_1:
DB      0,80h,15h,0,0,0,0,0 ; 1.5
DB      0,80h,30h,0,0,0,0,0 ; 3
```

Disp

Category: Graphing and Drawing

Description: Checks if graph screen is in the display. If it is, restores the text shadow to the screen.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: shiftFlags, textFlags

Others: curRow, curCol, winTop

**Registers
destroyed:** All

Remarks: This is intended to be used when an application uses both the home screen and the graph screen.

Using this routine allows the application to switch between the home screen and the graph screen without having to rebuild the home screen.

When switching to the graph screen, all of the text previously written to the home screen should have been also written to the text shadow.

The plotLoc flag should be set when switching to the graph screen.

Example:

DrawCirc2

Category: Graphing and Drawing

Description: Draws a circle given the center and the radius, relative to the current window settings.

The current graph screen is not put into the display by this routine.

This icircle routine is one of two available, and is the faster of the two.

Inputs:

Registers: None

Flags: plotLoc, (IY + plotFlags) = 1 to draw to the display only
plotLoc, (IY + plotFlags) = 0 to draw to display and buffer
bufferOnly, (IY + plotFlag3) = 1 to draw to buffer only

Others: FPST = radius, a floating-point number
FPS1 = Y value of center, a floating-point number
FPS2 = X value of center, a floating-point number

The center specified is with respect to the current window settings.

Outputs:

Registers: None

Flags: None

Others: Circle is drawn either to the display, the buffer, or both.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

Remarks: If a zoom square is not done before using this routine the output circle will most likely not look circular but skewed in either the X or Y axis direction. See **CircCmd**. See Floating Point Stack section.

(continued)

DrawCirc2 *(continued)*

Example: Execute a zoom standard and then draw a circle at (0,0) with radius 3.

```

        B_CALL    ZooDefault    ; standard window
        B_CALL    PDspGrph      ; get current graph to the
                                ; display
;
        B_CALL    OP1Set0       ; OP1 = 0
        RST       rPushRealO1
        RST       rPushRealO1   ; (0,0) pushed onto FPS
;
        B_CALL    OP1Set3       ; radius is 3
        RST       rPushRealO1   ; 3 pushed onto FPS
;
        AppOnErr   circerr      ; set up error handler
;
        B_CALL    DrawCirc2     ;
;
        AppOffErr                ; remove no error

        RET

;
;   come here if error
;
Circerr:
;

```

DrawCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a function on it. Same as TI-83 Plus instruction DrawF.

Inputs:**Registers:** None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen
bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: FPST = name of equation to evaluate and draw, with X being the independent variable.

Outputs:

Registers: None

Flags: None

Others: Current graph and function are drawn to the screen and the graph backup buffer, *plotSScreen*.

FPST = name of equation drawn, this must be cleaned by the calling routine.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw, see Error Handlers section.
See section on Floating Point Stack

Example: Draw Y1 on the graph screen.

```

LD      HL,Y1name
B_CALL  Mov9ToOp1      ; OP1 = Y1
B_CALL  PushRealO1     ; push Y1 into FPST
;
B_CALL  DrawCmd        ; draw
;
B_CALL  PopRealO1      ; clean Y1 off of FPS
;
```

DrawRectBorder

Category: Graphing and Drawing

Description: Draws a rectangular outline on the screen.

Inputs:

Registers: H = upper left corner pixel row
 L = upper left corner pixel column
 D = lower right corner pixel row
 E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
 0: update display and graph buffer
 1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.

Inputs must satisfy conditions: $D \geq H$, $E \geq L$

Modifies **saveSScreen** RAM area.

Example: ;

```
LD      HL,0000h
LD      DE,3F5Fh
B_CALL  DrawRectBorder      ; Draw an outline around
                             ; the screen
B_CALL  GetKey               ; Get key press
B_JUMP  JForceCmdNoChar     ; Exit app
```

DrawRectBorderClear

Category: Graphing and Drawing

Description: Draws a rectangular outline on the screen and clears the area inside the outline.

Inputs:

Registers: H = upper left corner pixel row
 L = upper left corner pixel column
 D = lower right corner pixel row
 E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
 0: update display and graph buffer
 1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.

Inputs must satisfy conditions: $D \geq H$, $E \geq L$.

Modifies **saveSScreen** RAM area.

Example: ;

```

B_CALL    ClrLCDFull
LD        H,32
LD        L,48
LD        D,63
LD        E,95
B_CALL    FillRect                ; Blacken the screen's
                                   ; lower right quarter
B_CALL    GetKey                  ; Get key press
LD        HL,0000h
LD        DE,3F5Fh
B_CALL    DrawRectBorderClear    ; Draw an outline
                                   ; around the screen and
                                   ; clear inside
B_CALL    GetKey                  ; Get key press
B_JUMP    JForceCmdNoChar        ; Exit app

```


EraseRectBorder

Category: Graphing and Drawing

Description: Erases a rectangular outline on the screen (to white).

Inputs:

Registers: H = upper left corner pixel row
 L = upper left corner pixel column
 D = lower right corner pixel row
 E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
 0: update display and graph buffer
 1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.

Inputs must satisfy conditions: $D \geq H$, $E \geq L$

Modifies **saveSScreen** RAM area.

Example: ;

```
LD      HL,0000h
LD      DE,3F5Fh
B_CALL  DrawRectBorder    ; Draw an outline around the
                           ; screen
B_CALL  GetKey             ; Get key press
B_CALL  EraseRectBorder    ; Erase an outline around
                           ; the screen
B_CALL  GetKey             ; Get key press
B_JUMP  JForceCmdNoChar    ; Exit app
```

FillRect

Category: Graphing and Drawing

Description: Fills a rectangular area on the screen (to black).

Inputs:

Registers: H = upper left corner pixel row
L = upper left corner pixel column
D = lower right corner pixel row
E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
0: update display and graph buffer
1: update display only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.

Inputs must satisfy conditions: $D \geq H$, $E \geq L$

Modifies **saveSScreen** RAM area.

(continued)

FillRect *(continued)*

Example: ;

```

B_CALL  ClrLCDFull      ; Clear the whole screen
LD      HL,1C2Ch
LD      DE,2232h
B_CALL  FillRect        ; Put black square in
                        ; screen center
B_CALL  GetKey          ; Get key press
LD      H,0
LD      L,0
LD      D,63
LD      E,95
B_CALL  InvertRect      ; Turn to white square on
                        ; black background
B_CALL  GetKey          ; Get key press
LD      H,0000h
LD      D,3F5Fh
B_CALL  InvertRect      ; Return to black square on
                        ; white background
B_CALL  GetKey          ; Get key press
B_JUMP  JForceCmdNoChar ; Exit app

```

FillRectPattern

Category: Graphing and Drawing

Description: Fills a rectangular area on the screen with a pattern.

Inputs:

Registers: H = upper left corner pixel row
 L = upper left corner pixel column
 D = lower right corner pixel row
 E = lower right corner pixel column

Flags: plotLoc, (IY + plotFlags):
 0: update display and graph buffer
 1: update display only

Others: RectFillPHeight = pattern's height in pixel rows (byte, 1 – 8)
 RectFillPWidth = pattern's width in pixel columns (byte, 1 – 8)
 RectFillPattern = one-byte for each pattern pixel row
 Pattern is right justified — bit 0 is right-most pixel in pattern row. First byte is the top row of the pattern.

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is upper left corner of screen and row = 63, column = 95 is lower right corner of screen.

Area includes row and column of both coordinates.

Inputs must satisfy conditions: $D \geq H$, $E \geq L$.

You should not use the right-most column (95). This routine fails if you try to use it.

Modifies **saveSScreen** RAM area.

The pattern is written across the screen and is truncated at the right edge of the specified rectangle. The pattern will also be truncated at the bottom of the rectangle if needed.

(continued)

FillRectPattern *(continued)*

Example:

```

B_CALL  ClrLCDFull      ; Clear the whole screen
LD      A,6             ; Pattern is 6 pixels
                        ; high

LD      (RectFillPHeight),
A
LD      A,4             ; Pattern is 4 pixels
                        ; wide

LD      (RectFillPWidth),A
LD      HL,MyPattern    ; Copy source is the
                        ; pattern in this code

LD      DE,RectFillPattern ; Copy destination is the
                        ; pattern buffer

LD      BC,6            ; Copy 6 bytes
LDIR                      ; Copy pattern to pattern
                        ; buffer

LD      HL,1F2Fh
LD      DE,3F5Eh        ; Coordinates of the full
                        ; screen except last
                        ; column

B_CALL  FillRectPattern  ; Fill it with the
                        ; pattern

B_CALL  GetKey           ; Get key press
B_JUMP  JForceCmdNoChar  ; Exit app

MyPattern: DB 0Fh, 07h, 03h, 01h, 03h, 07h

```

GrBufClr

Category: Graphing and Drawing

Description: Clears out the graph backup buffer *plotSScreen*.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: All 768 bytes of *plotSScreen* set to 0.

**Registers
destroyed:** All

Remarks:

Example:

GrBufCpy

Category: Graphing and Drawing

Description: Copies the graph backup buffer *plotSScreen* to the display.

Inputs:

Registers: None

Flags: None

Others: (winBtm) should be = 8

Outputs:

Registers: None

Flags: None

Others: Graph buffer sent to display.

**Registers
destroyed:** All

Remarks: Both vertical and horizontal split setting will affect what is copied to the screen.
See **ForceFullScreen**. See **RestoreDisp**.

Example:

GrphCirc

Category: Graphing and Drawing

Description: Draws a circle on the screen given the pixel coordinates of the center and a point on the circle.

Inputs:

Registers: None

Flags: useFastCirc, (IY + plotFlag3) = 1 for fast circle routine that draws the circle in sections simultaneously
useFastCirc, (IY + plotFlag3) = 0 for normal circle routine that draws in a circular direction
plotLoc, (IY + plotFlags) = 1 to draw to the display only
plotLoc, (IY + plotFlags) = 0 to draw to display and buffer
bufferOnly, (IY + plotFlag3) = 1 to draw to buffer only.

Others: Pixel coordinates for the center and a point on the circle. Coordinate (0,0) is the pixel in the lower left corner of the display, (x,y).

(curGX2) = x coordinate of center

(curGY2) = y coordinate of center

(curGX) = x coordinate of point on the circle

(curGY) = y coordinate of point on the circle

Outputs:

Registers: None

Flags: None

Others: Circle drawn on the display.

Registers destroyed: All

Remarks: The graph screen does not have to be displayed. The current window settings have no affect. If useFastCirc is used, the flag must be reset by the caller. See **CircCmd** and **DrawCirc2** routines.

Example:

HorizCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a horizontal line at $X = OP1$.
Same as TI-83 Plus instruction Horizontal.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty, and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen.

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer **plotSScreen**, not to the display

Others: OP1 = X value to draw the horizontal line at.

Outputs:

Registers: None

Flags: None

Others: Current graph and the line are drawn to the screen and the graph backup buffer, **plotSScreen**.
FPST = name of equation drawn, this must be cleaned by the calling routine.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example: Draw a horizontal line at $X = 3$ on the graph screen.

```

                B_CALL    OP1Set3      ; OP1 = 3
;
                B_CALL    HorizCmd     ; draw the line
;
```

IBounds

Category: Graphing and Drawing

Description: Tests if a pixel coordinate lies within the graph window defined by the current split mode settings.

Inputs:

Registers: B = X pixel coordinate
C = Y pixel coordinate

Flags: The current split screen setting.

Others: None

Outputs:

Registers: None

Flags: CA = 1 if out of graph window
= 0 if in graph window

Others: Line drawn where specified.

Registers destroyed: All registers are preserved.

Remarks: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See **ForceFullScreen** and **IBoundsFull** routines for further information.

Example:

IBoundsFull

Category: Graphing and Drawing

Description: Tests if a pixel coordinate lies within the full pixel range of the display. Full screen mode should be active when using this routine. Valid values will include all 64 rows and 96 columns of the display. Normally only 63 rows and 95 columns are valid.

Inputs:

Registers: B = X pixel coordinate
C = Y pixel coordinate

Flags: The current split screen setting.

Others: None

Outputs:

Registers: None

Flags: CA = 1 if out of graph window
= 0 if in graph window

Others: Line drawn where specified.

Registers destroyed: All registers are preserved.

Remarks: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See the **ForceFullScreen** and **IBounds** routines for further information.

ILine

Category: Graphing and Drawing

Description: Draws a line between two-pixel points defined by their pixel coordinates. The line drawn can be on, off, or inverted.

Inputs: The graph window is defined with the lower left corner of the display to be pixel coordinates (0,0).
The system graphing routines do not normally draw in the last column and the bottom row of the screen, column 95 and row 0.
This routine can be made to use column 95 and row 0 by setting the flag: fullScrnDraw, (IY + apiFlg4)

Registers: X = column
Y = row

B — X Coordinate of first point — 0...94 (95) see above

C — Y Coordinate of first point — 1(0)...63

D — X Coordinate of second point — 0...94 (95)

E — Y Coordinate of second point — 1(0)...63

H — Type of line to draw

0 — Set points to light, on-line

1 — Set points to dark

2 — Invert points (XOR operation)

Flags: fullScrnDraw, (IY + apiFlg4) = 1 to use column 95 and row 0
plotLoc, (IY + plotFlags) = 1 to draw to the display only
= 0 to draw to display and **plotSScreen** buffer
bufferOnly, (IY + plotFlag3) = 1 to draw to **plotSScreen** buffer only

Others: None

Outputs:

Registers: None

Flags: None

Others: Line drawn where specified.

Registers destroyed: All registers are preserved.

(continued)

ILine *(continued)*

Remarks: If the draw is going to the buffer, then the contents of the buffer are used to draw the line and copied to the screen.

G-T and HORIZ split-screen modes affect how this routine maps the coordinates specified. To avoid this, turn off the split-screen modes. See **ForceFullScreen**.

No clipping, X, Y points assumed to be defined on the screen.

Example: Erase a line in the display only, between pixel coordinates (25,30) and (62,50).

```

;
;          LD          BC,25*256+30          ; 1st point, B=25,
;                                          ; C=30
;          LD          DE,62*256+50          ; 2nd point, D=62,
;                                          ; E=50
;
;          SET         plotLoc,(IY+plotFlags) ; display only
;
;          LD          H,0                   ; signal turn pixels
;                                          ; off
;          B_CALL      ILine                 ; draw the line
;

```

InvCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a function along the Y-axis.

The equation is evaluated with respect to X, but the value of X will range between Ymin and Ymax, and the result of each evaluation will be the X coordinate, and the Y coordinate will be the value of X. It is the same as switching X and Y, and having Y be the independent variable. But it is important to write the expression in terms of X.

Same as TI-83 Plus instruction DrawInv.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer **plotSScreen**, not to the display

Others: FPST = name of equation to evaluate and draw

Outputs:

Registers: None

Flags: None

Others: Current graph and function are drawn to the screen and the graph backup buffer, **plotSScreen**.
FPST = name of equation drawn, this must be cleaned by the calling routine.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw — see Error Handlers section.
See section on Floating Point Stack.

Example: Draw Y1 on the graph screen along the Y-axis.

```
LD          HL,Y1name
B_CALL     Mov9ToOP1      ; OP1 = Y1
B_CALL     PushRealO1     ; push Y1 into FPST
;
B_CALL     InvCmd         ; draw
;
B_CALL     PopRealO1      ; clean Y1 off of FPS
;
```

InvertRect

Category: Graphing and Drawing

Description: Inverts a rectangular area on the screen (black pixels to white; white pixels to black).

Inputs:

Registers: H = upper left corner pixel row
 L = upper left corner pixel column
 D = lower right corner pixel row
 E = lower right corner pixel column

Flags: None

Others: plotLoc, (IY + plotFlags):
 0: update display and graph buffer
 1: update display only

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: Rectangle is defined by pixel coordinates, where row = 0, column = 0 is the upper left corner of screen and row = 63, column = 95 is the lower right corner of screen.

Area includes row and column of both coordinates.

Inputs must satisfy conditions: $D \geq H$, $E \geq L$.

Modifies **saveSScreen** RAM area.

Example:

B_CALL	ClrLCDFull	; Clear the screen
LD	HL,0	; HL = upper left corner
LD	DE,3F5Fh	; DE = lower right corner
B_CALL	InvertRect	; Blacken entire screen
LD	HL,2030h	; HL = middle of screen
LD	DE,3F5Fh	; DE = lower right corner
B_CALL	InvertRect	; Whiten lower right quadrant
B_CALL	GetKey	; Get key press

IOffset

Category: Graphing and Drawing

Description: Given a pixel location, computes the offset to add to the start address of the graph buffer to the byte in the buffer containing that pixel.

Also returns the bit number in that byte for that pixel.

Also computes the row and column commands to set the LCD driver to the display byte for that pixel.

Inputs:

Registers: Pixel's row and column coordinate, (0,0) = lower left pixel of the display.
 B — Column coordinate value, (0 – 95)
 C — Row coordinate value, (0 – 63)

Flags: None

Others: None

Outputs:

Registers: ACC = bit that corresponds to the pixel's location in the byte it resides in is set.
 For example, pixel (0,0) would return with ACC = 80h, bit 7 is set.

HL = byte offset to add to the start address of the display buffer to the byte that contains the pixel's bit.

(curXRow) = row command to send to the LCD driver for that pixel.

(curY) = column command to send to the LCD driver for that pixel.

Flags: None

Others: None

Registers destroyed: All but DE

Remarks:

Example: Test if pixel (23,14) is set in the graph buffer *plotSScreen*.

```

LD      BC,23*256+14      ; BC = 23,14
B_CALL  IOffset
;

LD      DE,plotSScreen    ; start of graph buffer
ADD     HL,DE              ; add offset to byte with
                          ; pixel
AND     (HL)              ; and pixels bit with byte
                          ; in buffer
JR      Z,Pixel_is_Off    ; jump if pixel is not set
                          ; in buffer

```


IPoint

Category: Graphing and Drawing

Description: Executes one of the following pixel operations without displaying the current graph screen:

- Turn Off
- Turn On
- Change (invert)
- Test
- Copy

Inputs: The pixels are addressed with the lower left corner of the display being pixel (0,0), (row,col)

The system does not normally draw in the last column, and the bottom row of the screen, column 95 and row 0.

This routine can be made to use column 95 and row 0 by setting the flag: fullScrnDraw, (IY + apiFlg4)

Registers: B = pixel row address — 0...94 (95 if full screen) see above
 C = Y Coordinate of first point — 1(0)...63 (64 if full screen)
 D = Function to perform

- 0 — Turn point off
- 1 — Turn point on
- 2 — Invert point (XOR operation)
- 3 — Test point
- 4 — Copy a point from buffer to the display

Flags: fullScrnDraw, (IY + apiFlg4) = 1 to use column 95 and row 0

plotLoc, (IY + plotFlags) = 1 to draw to the display only
 plotLoc, (IY + plotFlags) = 0 to draw to display and buffer

bufferOnly, (IY + plotFlag3) = 1 to draw to buffer only

Others: None

Outputs:

Registers: None

Flags: For option 3 (test)

- Z = 1 for point off
- Z = 0 for point on

Others: None

Registers destroyed: None, except for option 3 (test) then all.

(continued)

IPoint *(continued)*

Remarks: The test option always tests the buffer not the display. This means that in order to use the test option the pixel tested must have been written to the graph buffer.

If the buffer is specified then the contents of the buffer are used to draw/copy, not what is in the screen.

G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this turn off the split screen modes. See **ForceFullScreen**.

If G-T mode is set then this routine will turn on pixels if the display byte containing the center column of pixels is accessed. This is done to keep the center line in G-T drawn.

Example: Turn on the point specified by pixel coordinates at (5,10).

```
LD      BC,5*256+10
LD      D,1          ; point on cmd
;
B_CALL  IPoint      ; turn on the point
;
```

LineCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a line defined by two points. These points are graph coordinates with respect to the current range settings. They do not have to be points on the screen. If they are not on the screen the line will still be drawn if it passes through the screen with the current range settings.

Same as TI-83 Plus instruction Line(.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer **plotSScreen**, not to the display

Others: points (X1, Y1) (X2, Y2), all are floating-point numbers
FPST = Y2 COORDINATE
FPS1 = X2 COORDINATE
FPS2 = Y1 COORDINATE
FPS3 = X1 COORDINATE

See Floating Point Stack section.

Outputs:

Registers: None

Flags: None

Others: Current graph and line are drawn to the screen and the graph backup buffer, **plotSScreen**.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw. See Error Handlers section. See **CLine** and **ILine** to draw lines without graphing. See section on Floating Point Stack.

(continued)

LineCmd *(continued)*

Example: Draw a line on the current graph screen between (1,2) and (3,4)

```

        B_CALL    OP1Set1      ; OP1 = X1
        B_CALL    PushRealO1   ; to FPS
;
        B_CALL    Plus1        ; OP1 = OP1 + 1, = Y1
        B_CALL    PushRealO1   ; to FPS
;
        B_CALL    Plus1        ; OP1 = OP1 + 1, = X2
        B_CALL    PushRealO1   ; to FPS
;
        B_CALL    Plus1        ; OP1 = OP1 + 1, = Y2
        B_CALL    PushRealO1   ; to FPS
;
        B_CALL    LineCmd      ; copy graph to screen and
                                ; draw line
;
```

PDspGrph

Category: Graphing and Drawing

Description: Tests if the graph of the current mode needs to be regraphed. If so, the graph is regraphed, otherwise copies ***plotSScreen*** to the display.

Inputs:

Registers: None

Flags: bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer ***plotSScreen***, not to the display

Others: Current graph window settings and equations

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this situation, turn off the split screen modes. See the **ForceFullScreen** routine for further information.

Example: Generate the current graph screen in the display.

B_CALL PDspGrph

PixelTest

Category: Graphing and Drawing

Description: Tests a pixel in the graph buffer specified by pixel coordinates without copying the graph to the display.

Inputs: Pixel coordinate (0,0), (row,col), is the upper left most pixel.
 FPST = Pixel coordinate's column value, a floating-point number
 (0 – 94) in full screen and horizontal split
 (0 – 46) in vertical split
 FPS1 = Pixel coordinate's row value, a floating-point number
 (0 – 62) in full screen
 (0 – 30) in horizontal split
 (0 – 50) in vertical split
 See Floating Point Stack section.

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: Z = 1 for point off
 Z = 0 for point on

Others: None

Registers destroyed: All

Remarks:

Example: Test on the point specified by pixel coordinates at (5,10).

```

                LD          BC,5*256+10
;
                B_CALL      PixelTest      ; test the point
;
```

PointCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and executes one of the following point operations:

Turn Off
Turn On
Change (invert)

The point is defined by graph coordinates with respect to the current range settings. The point does not need to be on the screen, and if it is not, then nothing will be drawn.

Same as TI-83 Plus instructions Pt-On(, Pt-Off(, Pt-Change(.

Inputs:

Registers: ACC = point command
0 = On
1 = Off
2 = Change

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer **plotSScreen**, not to the display

Others: Bit 5 of RAM location (OP1 + 2) MUST = 0
FPST = Y coordinate of the point, a floating-point number
FPS1 = X coordinate of the point, a floating-point number

Outputs:

Registers: None

Flags: None

Others: Current graph and point operation are drawn to the screen and the graph backup buffer **plotSScreen**.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw. See Error Handlers section. See **CPoint**, **CPointS**, and **IPoint** for point commands without graphing.

(continued)

PointCmd *(continued)*

Example: Invert point at coordinate (1.5,2)

```
LD      HL,fp_1p5      ;
B_CALL  Mov9ToOP1      ; OP1 = X coordinate, 1.5
B_CALL  PushRealO1     ; to FPS
;
B_CALL  OP1Set2        ; OP1 = Y coordinate, 2, resets
                        ; bit 5 (OP1 + 2)
B_CALL  PushRealO1     ; to FPS
;
LD      A,2            ; command to invert
B_CALL  PointCmd       ; copy graph to screen and
                        ; invert point
;
```


PointOn

Category: Graphing and Drawing

Description: Turns on a point specified by its pixel coordinates.

Inputs: The graph window is defined with the lower left corner of the display to be pixel coordinates (0,0).
The system graphing routines do not normally draw in the last column and the bottom row of the screen, column 95 and row 0.
This routine can be made to use column 95 and row 0 by setting the flag: fullScrnDraw, (IY + apiFlg4)

Registers: X = column
Y = row

B — X Coordinate of first point — 0...94 (95) see above
C — Y Coordinate of first point — 1(0)...63

Flags: fullScrnDraw, (IY + apiFlg4) = 1 to use column 95 and row 0
plotLoc, (IY + plotFlags) = 1 to draw to the display only
= 0 to draw to display and **plotSScreen** buffer
bufferOnly, (IY + plotFlag3) = 1 to draw to **plotSScreen** buffer only

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: None

Remarks: If the buffer is specified, then the contents of the buffer are used to draw the point.

G-T and HORIZ split-screen modes affect how this routine maps the coordinates specified. To avoid this, turn off the split-screen modes.

See **ForceFullScreen**.

Example: Turn on the point specified by pixel coordinates at (5,10):

```
LD          BC, 5*256+10
;
B_CALL     PointOn      ; turn on the point
```

Regraph

Category: Graphing and Drawing

Description: Graphs any selected equations in the current graph mode along with any selected statplots.

Inputs:

Registers: None

Flags: smartGraph_inv, (IY + smartFlags) = 1 to defeat smart regraphing feature and force all equations to be regraphed, not just new ones.

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer **plotSScreen**, not to the display.

Others: Current graph equations
Current window settings

Outputs:

Registers: None

Flags: None

Others: Graph redrawn to the display and backup buffer **plotSScreen**, or the **plotSScreen** only.

Registers destroyed: All but AF

Remarks: G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. . To avoid this situation, turn off the split screen modes. See the **ForceFullScreen** routine for further information. Also, see the Smart Regraphing section.

Example: B_CALL Regraph

SetAllPlots

Category: Graphing and Drawing

Description: Selects or deselects all statplots.

Inputs:

Registers: B = 0 to unselect
B = 1 to select

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: If any plot's selection stat changes then the graph is marked dirty.

**Registers
destroyed:** All

Remarks:

Example: Turn off all stat plots.

```
LD      B,0
B_CALL  SetAllPlots
```

SetFuncM

Category: Graphing and Drawing

Description: Changes from current graph mode to function mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Current flags saved with current mode, function mode flags and pointers set up.

Registers destroyed: A, BC, DE, HL

Remarks:

Example: B_CALL SetFuncM

SetParM

Category: Graphing and Drawing

Description: Changes from current graph mode to parametric mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Current flags saved with current mode. Parametric mode flags and pointer set up.

Registers destroyed: A, BC, DE, HL

Remarks:

Example: B_CALL SetParM

SetPolM

Category: Graphing and Drawing

Description: Changes from current graph mode to polar mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Current flags saved with current mode, polar mode flags and pointers set up.

Registers destroyed: A, BC, DE, HL

Remarks:

Example: B_CALL SetPolM

SetSeqM

Category: Graphing and Drawing

Description: Changes from current graph mode to sequence mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Current flags saved with current mode, sequence mode flags and pointers set up.

Registers destroyed: A, BC, DE, HL

Remarks:

Example: B_CALL SetSeqM

SetTblGraphDraw

Category: Graphing and Drawing

Description: Sets the current graph to dirty to cause a complete regraph the next time the graph needs to be displayed. Also marks the table of values as dirty, unless a graph is currently being graphed.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: smartGraph_inv, (IY + smartFlags) is set to invalidate smart graph
reTable, (IY + tblFlags) is set to dirty the table, if not graphing
graphDraw, (IY + graphFlags) is set to dirty the graph

Others: None

**Registers
destroyed:** None

Remarks:

Example:

TanLnF

Category: Graphing and Drawing

Description: Draws the tangent line for given equation at a given point.

The equation itself is not drawn only the tangent line.

The graph screen is not displayed — it is assumed to be displayed already.

Inputs:

Registers: None

Flags: None

Others: FPST = equation name, X is the independent variable
Variable X = X coordinate of point
OP1 = Y coordinate of point, a floating-point number
Window settings for the current graph are used

Outputs:

Registers: None

Flags: None

Others: Tangent line drawn to the display.
Equation name removed from the FPS.

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks: See section on the Floating Point Stack in Chapter 2.

Example:

UCLineS

Category: Graphing and Drawing

Description: Draws a WHITE line between two points specified by graph coordinates.

The line is plotted according to the current window settings Xmin, Xmax, Ymin, Ymax.

The points do not need to lie within the current window settings. This routine will clip the line to the screen edges if any portion of the line goes through the current window settings.

This routine should only be used to draw lines in reference to the window settings.

ILine can be used to draw lines by defining points with pixel coordinates, which will be a faster draw.

Inputs:

Registers: FPS2 — Y1 Coordinate
FPS3 — X1 Coordinate
FPS1 — Y2 Coordinate
FPST — X2 Coordinate

Flags: plotLoc, (IY + plotFlags) = 1 to draw to the display only
= 0 to draw to the display and the **plotSScreen** buffer

bufferOnly, (IY + plotFlag3) = 1 to draw to the **plotSScreen** buffer only

G-T and HORIZ split screen modes will affect how this routine maps the coordinates specified. To avoid this, turn off the split screen modes. See the **ForceFullScreen** routine.

grfSplit, (IY + sGrFlags) = 1 if horizontal split mode set
vertSplit, (IY + sGrFlags) = 1 if graph-table split mode set
grfSplitOverride, (IY + sGrFlags) = 1 to ignore split modes

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: This routine does not copy the graph buffer to the screen or invoke a regraph if needed. Use **PDspGrph** to make sure the graph in the screen is valid.

Example: See the **CLineS** routine.

UnLineCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and erases a line defined by two points. These points are graph coordinates with respect to the current range settings. They do not have to be points on the screen. If they are not on the screen, the line will still be drawn if it passes through the screen with the current range settings.

Same as the TI-83 Plus instruction Line(with the last argument = 0 for unline.

Inputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer **plotSScreen**, not to the display

Others: Points (X1,Y1) (X2,Y2), all are floating-point numbers
FPST = Y2 COORDINATE
FPS1 = X2 COORDINATE
FPS2 = Y1 COORDINATE
FPS3 = X1 COORDINATE

See the Floating Point Stack section.

Outputs:

Registers: None

Flags: None

Others: Current graph and line are drawn to the screen and the graph backup buffer, **plotSScreen**.

Inputs are removed from the Floating Point Stack.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: Errors can be generated during the draw — see the Error Handlers section. See **UCLineS** to draw lines without graphing. See the Floating Point Stack section.

Example: See **LineCmd**.

VertCmd

Category: Graphing and Drawing

Description: Displays the current graph screen and draws a vertical line at $Y = OP1$. Same as TI-83 Plus instruction Vertical.

Inputs:**Registers:** None

Flags: graphDraw, (1Y + graphFlags) = 1 if current graph is dirty and needs to be redrawn
= 0 if graph buffer is up to date and is copied to the screen

bufferOnly, (IY + plotFlag3) = 1 if draw to the backup buffer **plotSScreen**, not to the display

Others: OP1 = Y value to draw the vertical line at

Outputs:

Registers: None

Flags: None

Others: Current graph and the line are drawn to the screen and the graph backup buffer, ***plotSScreen***.

FPST = name of equation drawn, this must be cleaned by the calling routine.

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example: Draw a vertical line at $Y = 3$ on the graph screen.

```

        B_CALL    OP1Set3        ; OP1 = 3
;
        B_CALL    VertCmd        ; draw the line
;

```

VtoWHLDE

Category: Graphing and Drawing

Description: In the current graph window converts a pixel point to its corresponding X and Y values, floating-point numbers.

The graph must be up to date for this routine to return correct values.

Inputs:

Registers: B = X pixel value, 0 – 94, 0 = left most pixel column
C = Y pixel value, 1 – 62, 1 = next to last row of pixels from bottom

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point value representing X pixel coordinate
OP4 = floating-point value representing Y pixel coordinate

Registers destroyed: All

RAM used: OP1, OP2, OP3, OP4

Remarks: The bottom row of pixels is not used. Graph is up to date.

Example:

Xftol

Category: Graphing and Drawing

Description: In the current graph window, converts a floating-point value to an X pixel coordinate.

This is used by the graphing routines to plot points in the current graph.

The graph must be up to date for this routine to return correct values.

Inputs:

Registers: HL = pointer to floating-point number representing the X coordinate

Flags: None

Others: None

Outputs:

Registers: ACC = X pixel value, 0 – 94, 0 = left most pixel column

Flags: None

Others: None

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks: The right most column is not used. Graph is up to date.

Example:

Xitof

Category: Graphing and Drawing

Description: In the current graph window converts an X pixel coordinate to the floating-point value of X for that pixel.

The graph must be up to date for this routine to return correct values.

Inputs:

Registers: ACC = X pixel value, 0 – 94, 0 = left most pixel column
HL = pointer to location to return floating-point value

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Floating-point value representing X pixel coordinate returned at input HL to HL + 8.

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks: The bottom row of pixels is not used. Graph is up to date.

Example:

Yftol

Category: Graphing and Drawing

Description: In the current graph window, converts a floating-point value to an Y pixel coordinate.

This is used by the graphing routines to plot points in the current graph.

The graph must be up to date for this routine to return correct values.

Inputs:

Registers: HL = pointer to floating-point number representing the Y coordinate

Flags: None

Others: None

Outputs:

Registers: ACC = Y pixel value, 1 – 62, 1 = next to last row of pixels from bottom

Flags: None

Others: None

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3

Remarks: The bottom row of pixels is not used. Graph is up to date.

Example:

ZmDecml

Category: Graphing and Drawing

Description: Changes the window settings such that (0,0) is in the center of the display and ΔX and $\Delta Y = 0.1$. See the ZDecimal selection in the TI-83 Plus ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: Current window settings.

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious. New windows settings set to X: -4.7 to 4.7, Y: -3.1 to 3.1

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmFit

Category: Graphing and Drawing

Description: Changes the window settings such that the minimum and maximum Y value for all selected functions fit in the graph window.
The same ZoomFit under the ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: Current window settings

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.

New windows settings set so that all selected functions Y values fit in the display when regraphed.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmInt

Category: Graphing and Drawing

Description: Changes the window settings such that ΔX and $\Delta Y = 1.0$, given the coordinates in the center of the screen. The coordinates of the center of the screen are rounded to the closest integer before the window range is set. See the ZInteger selection in the TI-83 Plus ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: OP1 = X coordinate of new center of the screen, floating-point number
OP5 = Y coordinate of new center of the screen, floating-point number
Current window settings.

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.
New windows settings set.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmPrev

Category: Graphing and Drawing

Description: Changes the window settings back to the settings before the last zoom command was executed, if one was. See the ZPrevious selection in TI-83 Plus ZOOM/MEMORY menu.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: If ZPrevious values exist they are copied to the current window settings.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmSquare

Category: Graphing and Drawing

Description: Changes the window settings in either the X or Y direction such that $\Delta X = \Delta Y$. Doing this operation will make a circle drawn have the shape of a circle instead of an ellipse. See the ZSquare selection in the TI-83 Plus ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.
New windows settings set.

**Registers
destroyed:** All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmStats

Category: Graphing and Drawing

Description: Changes the window settings such that all selected Statplots will be visible in the graph window. See the ZoomStat in the TI-83 Plus ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.
New windows settings set.

**Registers
destroyed:** All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmTrig

Category: Graphing and Drawing

Description: Changes the window settings to preset values that are appropriate for trigonometrical function graphs. See the ZTrig selection in the TI-83 Plus ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: Current window settings

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.
New windows settings set to X: $-(47/24) * \pi$, Y: $(47/24) * \pi$

If the current angle mode setting is radians, then those values are used. If the current angle mode setting is degrees, then those values are converted from radians to degrees.

Registers destroyed: All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZmUsr

Category: Graphing and Drawing

Description: Recalls the window settings stored by the last ZoomSto command. See the ZoomRcl selection in the TI-83 Plus ZOOM/MEMORY menu.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: Current window settings are moved to ZPrevious.
New windows settings set.

**Registers
destroyed:** All

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

ZooDefault

Category: Graphing and Drawing

Description: Changes the window settings back to the default settings of (-10,10) for both X and Y ranges.

The same ZStandard under the ZOOM menu.

Inputs:

Registers: None

Flags: None

Others: Current window settings

Outputs:

Registers: None

Flags: graphDraw, (IY + graphFlags) = 1, dirty the graph

Others: New windows settings set to X: -10 to 10, Y: -10 to 10

Registers All

destroyed:

Remarks: The graph is marked dirty for redrawing, but the graph is not redrawn.

Example:

A

System Routines — Interrupt

DivHLBy10	334
DivHLByA	335

DivHLBy10

Category: Interrupt

Description: Divides HL by 10.

Inputs:

Registers: HL = dividend

Flags: None

Others: None

Outputs:

Registers: HL = $\text{Int}(\text{HL}/10)$
A = $\text{mod}(\text{HL}/10)$

Flags: None

Others: None

**Registers
destroyed:** None

Remarks: None

Example:

DivHLByA

Category: Interrupt

Description: Divides HL by accumulator.

Inputs:

Registers: HL = dividend
A = divisor

Flags: None

Others: None

Outputs:

Registers: HL = $\text{Int}(\text{HL}/\text{A})$
A = $\text{mod}(\text{HL}/\text{A})$ (remainder)

Flags: None

Others: None

Registers destroyed: None

Remarks: None

Example:

A

System Routines — IO

AppGetCalc	337
AppGetCbl.....	338
Rec1stByte	339
Rec1stByteNC	340
RecAByteIO.....	341
SendAByte	342

AppGetCalc

Category: IO

Description: Executes the basic **GetCalc** command to retrieve a variable from another TI-83 Plus or a TI-83.

Inputs:

Registers: OP1 = name of variable to attempt to retrieve

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: comFailed, (IY + getSendFlg) = 0 if variable received
comFailed, (IY + getSendFlg) = 1 if variable not received
Variable updated or created if received

Registers destroyed: All

Remarks: Variables can be received from both an TI-83 Plus and a TI-83.

Example:

```

;
      B_CALL      AnsName              ; OP1 = Ans
                                           ; variable name
      B_CALL      AppGetCalc           ; attempt to get
                                           ; Ans
      BIT         comFailed,(IY+getSendFlg) ; did it work?
      JP          NZ,GetFailed          ; jump if no

```

AppGetCbl

Category: IO

Description: Executes the basic **GetCbl** command to retrieve data from a CBL or CBR device.

Inputs:

Registers: OP1 = name of variable to attempt to retrieve

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: comFailed, (IY + getSendFlg) = 0 if variable received
comFailed, (IY + getSendFlg) = 1 if variable not received
Variable updated or created if received

Registers destroyed: All

Remarks:

Example:	LD	HL,Llname	
	RST	rMov9ToOP1	; OP1 = L1 variable
			; name
	B_CALL	AppGetCbl	; attempt to get
			; data
	BIT	comFailed,(IY+getSendFlg)	; did it work?
	JP	NZ,GetFailed	; jump if no
Llname:	DB	RListObj,tVarLst,tL1,0,0	

Rec1stByte

Category: IO

Description: Polls the link port for activity until either a byte is received, the [ON] key is pressed, or an error occurred during communications. The cursor is turned on for updates.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: ACC = byte received if one

Flags: None

Others: Error will be generated if communications fail.
An error is also generated if the [ON] key is pressed.

**Registers
destroyed:** All

RAM used:

Remarks: APD can occur while waiting for link activity. See Chapter 2 for Error Handlers and Link Port. See entry points **Rec1stByteNC**, **RecAByte**, and **SendAByte**.

Example: See Chapter 2.

Rec1stByteNC

Category: IO

Description: Polls the link port for activity until either a byte is received, the [ON] key is pressed, or an error occurred during communications. The cursor is not turned on for updates.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: ACC = byte received if one

Flags: None

Others: Error will be generated if communications fail. An error is also generated if the [ON] key is pressed.

Registers destroyed: All

RAM used:

Remarks: APD can occur while waiting for link activity. See Chapter 2 for Error Handlers and Link Port. See entry points **Rec1stByte**, **RecAByte**, and **SendAByte**.

Example: See Chapter 2.

RecAByteIO

Category: IO

Description: Attempts to read a byte of data over the link port.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: ACC = byte if successful

Flags: None

Others: None

**Registers
destroyed:** All

Remarks: If no link activity is detected within about 1.1 seconds, a system error is generated. See entry points **Rec1stByte**, **Rec1stByteNC**, and **SendAByte**.

Example: See Chapter 2.

SendAByte

Category: IO

Description: Attempts to send a byte of data over the link port.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** All

Remarks: If no link activity is detected within about 1.1 seconds, a system error is generated. See entry points **Rec1stByte**, **Rec1stByteNC**, and **RecAByte**.

Example: See Chapter 2.

A

System Routines — Keyboard

ApdSetup.....	344
CanAlphIns.....	345
GetCSC	346
GetKey	349

ApdSetup

Category: Keyboard

Description: Resets the Automatic Power Down timer.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: (apdTimer)

Flags: None

Others: None

**Registers
destroyed:** HL

Remarks:

Example:

CanAlphIns

Category: Keyboard

Description: Cancels alpha, alpha lock, and insert mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: textInsMode (In textFlags) and shiftALock (In shiftFlags) cleared
shiftAlpha (In shiftFlags) and shiftLwrAlph (In shiftFlags) may also be cleared
depends on flag shiftKeepAlph (In shiftFlags)

Others: None

**Registers
destroyed:** None

Remarks:

Example: B_CALL CanAlphIns

GetCSC

Category: Keyboard

Description: Gets and clears keyboard scan code. This routine should be used to read the keyboard only when an app does not care about second keys or alpha keys or pull down menus.

This routine only returns to the application which physical key on the keyboard was last pressed.

Inputs:

Registers: None

Flags: None

Others: None

Outputs: This routine does not wait for a key press to return back to the app. Key presses are detected in the interrupt handler, this routine returns that value. A 0 value is returned if no key has been pressed since the previous call to **GetCSC**.

Registers: A = (kbdScanCode) value

Flags: None

Others: (kbdScanCode) set to 0. kbdSCR flag reset.

Registers destroyed: AF, HL

(continued)

GetCSC *(continued)*

Remarks: No silent link activity will be detected if this routine is used to poll for keys. Below are the scan code equates.

;

;

skDown	equ	01h	skCos	equ	1Eh
skLeft	equ	02h	skPrgm	equ	1Fh
skRight	equ	03h	skStat	equ	20h
skUp	equ	04h	sk0	equ	21h
skEnter	equ	09h	sk1	equ	22h
skAdd	equ	0Ah	sk4	equ	23h
skSub	equ	0Bh	sk7	equ	24h
skMul	equ	0Ch	skComma	equ	25h
skDiv	equ	0Dh	skSin	equ	26h
skPower	equ	0Eh	skMatrix	equ	27h
skClear	equ	0Fh	skGraphvar	equ	28h
skChs	equ	11h	skStore	equ	2Ah
sk3	equ	12h	skLn	equ	2Bh
sk6	equ	13h	skLog	equ	2Ch
sk9	equ	14h	skSquare	equ	2Dh
skRParen	equ	15h	skRecip	equ	2Eh
skTan	equ	16h	skMath	equ	2Fh
skVars	equ	17h	skAlpha	equ	30h
skDecPnt	equ	19h	skGraph	equ	31h
sk2	equ	1Ah	skTrace	equ	32h
sk5	equ	1Bh	skZoom	equ	33h
sk8	equ	1Ch	skWindow	equ	34h
skLParen	equ	1Dh	skYEqu	equ	35h
			sk2nd	equ	36h
			skMode	equ	37h
			skDel	equ	38h

(continued)

GetCSC *(continued)*

Example: Poll for the 2nd key.

```

                                EI                                ; enable interrupts
;
                                ; the halt is optional, this
                                ; will help save battery life.
;
                                ; you can still use GetCSC at
                                ; anytime without the halt.
;
..sleep:
                                HALT                            ; sleep in low power for a
                                                                ; little
;
                                B_CALL    GetCSC                ; check for a scan code
                                CP        ksk2nd                ; 2nd key ?
                                JR        NZ,..sleep            ; jump if no
;

```

GetKey

Category: Keyboard

Description: Keyboard entry routine that will return second keys, alpha keys — both capital and lower case, the on key, APD, and link communication. Contrast adjustment is also handled by this routine.

When called, this routine scans for keys until one is pressed, or an APD occurs, or the unit is turned off, or link activity is detected.

Inputs:

Registers: None

Flags:

- indicOnly, (IY + indicFlags) = MUST BE RESET, otherwise no key presses will be detected.
- indicRun, (IY + indicFlags) = 1 to show the run indicator while waiting for a key press.
- apdAble, (IY + apdFlags) = 1 if APD is enabled
= 0 if APD is disabled
- lwrCaseActive, (IY + appLwrCaseFlag) = 1 for the key sequence [alpha] [alpha] to access lower case alpha key presses
= 0 for normal alpha key operation

Others: None

Outputs:

Registers: ACC = key code, 0 = ON key
See TI83plus.inc file.

Flags: onInterrupt, (IY + onFlags) = 1 if ON key, this should be reset

Others: APD: If the auto power down occurs the application will not be notified. Once the unit is turned back on control is returned to the GetKey routine.

OFF: If the unit is turned off the application is put away. When the unit is turned back on the home screen will be in control.

Link Activity: When link activity is initiated, control is given to the silent link handler. If the communication is from the GRAPH LINK, the application will be shut down in most cases. The only exception is getting screen snap shots, in that case the application is not shut down. After the screen is sent control returns to GetKey.

Registers destroyed: DE, HL

Remarks: If APD is disabled, it should be re-enabled before exiting the application. If lower case is enabled, it should be disabled upon exiting the application.

Example:

A

System Routines — List

AdrLEle.....	351
AdrMEle.....	352
ConvDim.....	353
ConvLcToLr.....	354
ConvLrToLc.....	355
DelListEl	356
Find_Parse_Formula	357
GetLToOP1	358
IncLstSize.....	359
InsertList.....	361
PutToL.....	363

AdrLEle

Category: List

Description: Computes the RAM address of an element of a list.

Inputs:

Registers: DE = pointer to start of list's data storage, output of **FindSym**
 HL = element number in list to compute address of. List element number one is checked for real or complex data type to determine if the list is real or complex.

Flags: None

Others: None

Outputs:

Registers: HL = pointer in RAM to the start of the desired element

Flags: None

Others: None

Registers destroyed:

Remarks: This routine does not check to see if the element's address requested is within the current size of the list.
 Do not use this routine on a list that does not have element number 1 initialized.

Example: Compute the address of element number 23 of list L1.

```

LD      HL,L1Name
RST     rMov9ToOP1      ; OP1 = L1 name
B_CALL  FindSym         ; look it up
JP      C,UndefinedL1   ; jump out if L1 is not
                        ; defined;
LD      A,B              ; if b<>0 then L1 is archived
                        ; in Flash ROM
OR      A
JP      NZ,ArchivedL1    ; jump if not in RAM
; DE = pointer to start of list data storage;
LD      HL,23d           ; element's address desired
B_CALL  AdrLEle          ; RET HL = pointer to 23rd
                        ; element
RET
L1Name:
DB      ListObj,tVarLst,tL1,0,0

```

AdrMEle

Category: List

Description: Computes the RAM address of an element of a matrix.

Inputs:

Registers: DE = pointer to start of matrix's data storage, output of **FindSym**
 BC = element's (row, column) to compute address of Matrix Element (1,1) is checked for real or complex data type to determine if the matrix is real or complex.

Flags: None

Others: None

Outputs:

Registers: HL = pointer in RAM to start of desired element

Flags: None

Other: None

Registers destroyed:

Remarks: This routine does not check to see if the element's address requested is within the current dimension of the matrix.
 Do not use this routine on a matrix that does not have element (1,1) initialized.

Example: Compute the address of element (5,6) of matrix [A].

```

LD          HL,MatAName
RST         rMov9ToOP1      ; OP1 = [A] name
B_CALL      FindSym         ; look it up
JP          C,Undefined_A   ; jump out if [A] is not
                                ; defined;
LD          A,B              ; if b<>0 then [A] is
                                ; archived in Flash ROM
OR          A
JP          NZ,Archived_A    ; jump if not in RAM;
                                ; DE = pointer to start of
                                ; matrix data storage;
LD          BC,5*256+6       ; element's address
                                ; desired
B_CALL      AdrMEle         ; RET HL = pointer to
                                ; element (5,6)
RET
MatAName:
DB          MatObj,tVarMat,tMatA,0,0

```

ConvDim

Category: List

Description: Converts floating-point value in OP1 to a two-byte hex value — make sure valid matrix or vec dimension. Less than 100 is valid dimension

Inputs:

Registers: None

Flags: None

Others: OP1 = FP number

Outputs:

Registers: A = LSB HEX VALUE, DE = ENTIRE HEX VALUE

Flags: None

Others: None

Registers destroyed: A, BC, DE, HL, OP1

Remarks: Error if negative, non-integer, or greater than 99.

Example: B_CALL ConvDim

ConvLcToLr

Category: List

Description: Converts an existing complex list variable to a real list variable.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of complex list variable to convert

Outputs:

Registers: None

Flags: None

Others: Error if the list was undefined.
OP1 = name of list with type set to ListObj. The imaginary part of each element is deleted and the data storage area is compressed. All symbol table pointers are updated.

Registers destroyed: All

Remarks: Do not use this routine if the input list is already a real list.

Example:

ConvLrToLc

Category: List

Description: Converts an existing real list variable to a complex list variable.

Inputs:

Registers: DE = pointer to data storage for list, output of **ChkFindSym**

Flags: None

Others: FPST = name of variable converted, see Floating Point Stack

Outputs:

Registers: DE = pointer to data storage of converted list

Flags: None

Others: Error if not enough free RAM to convert to complex.
Each element of the list is converted to a complex number with a 0 imaginary part.
FPST = name of variable converted, see Floating Point Stack.
All symbol table pointers are updated.

Registers destroyed: All

Remarks: Do not use this routine if the input list is already a complex list.

Example: Convert real list L1 to a complex list.

```

LD      HL,L1Name
RST     rMov9ToOP1      ; OP1 = L1 name
B_CALL  PushReal01      ; FPST = name of list
;
B_CALL  FindSym         ; look it up, DE = pointer
                        ; to data storage
JP      C,convertError  ; jump out if L1 is not
                        ; defined
;
AppOnErr convertError   ; install error handler in
                        ; case not enough RAM
;
B_CALL  ConvLrToLc      ; attempt to convert to
                        ; complex
;
AppOffErr                                ; remove error handler,
                                        ; successful
;
convertError:
B_CALL  PopReal01       ; remove name of list from
                        ; FPST
;
RET
;
L1Name:
DB      ListObj,tVarLst,tL1,0,0

```


DelListEl

Category: List

Description: Deletes one or more elements from an existing list, residing in RAM.

Input:

Registers: A = ListObj if the list has real elements
 = CListObj if the list has complex elements
 DE = pointer to start of list's data storage, output of **FindSym**
 HL = number of elements to delete
 BC = element number to start deleting at

Flags: None

Others: None

Output:

Registers: HL = pointer to start of list's data storage, output of **FindSym**
 DE = new dimension of the list.

Flags: None

Others: (insDelPtr) = pointer to start of the list

Registers destroyed: All

Remarks: DO NOT ATTEMPT ON AN ARCHIVED LIST. The size bytes of the list are adjusted. All pointers in the symbol table are updated

Example: Delete three elements from list L1 starting with element number two.

```

                                LD      HL,L1Name
                                RST      rMov9ToOP1      ; OP1 = L1 name
                                B_CALL   FindSym          ; look it up, DE = pointer
                                                ; to data storage
                                JP      C,UndefinedL1     ; jump out if L1 is not
                                                ; defined
                                ;
                                LD      C,A              ; save type
                                LD      A,B              ; get archived status
                                OR      A                ; in RAM or archived
                                JP      NZ,errArchived    ; cannot insert if archived
                                ;
                                LD      A,C              ; get type back
                                AND     1Fh             ; mask type of list in ACC
                                LD      HL,3              ; want to delete 3 elements
                                LD      BC,2              ; delete 2nd element on
                                ;
                                B_CALL   DelListEl        ; delete elements
                                ;
                                L1Name:
                                DB      ListObj,tVarLst,tL1,0,0

```

Find_Parse_Formula

Category: List

Description: Checks if a list variable has a formula attached to it and parses the formula and stores it back into the list data.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of list

Outputs:

Registers: None

Flags: None

Others: If no error, then the list values are updated.

Registers destroyed: All

Remarks: If no formula is attached, nothing is done to the existing list data.

Any error that occurs during the parsing of the formula will cause an error screen to be displayed if no error handler is invoked.

If the resulting type from the formula parsing is not a list, this will also generate an error.

See Error Handlers.

Example:

GetLToOP1

Category: List

Description: Copies a list element to OP1 or OP1/OP2.

Inputs:

Registers: HL = element number to copy
DE = pointer to start of list's data storage

Flags: None

Others: None

Outputs:

Registers: HL = pointer to next element in the list

Flags: None

Others: OP1 = list element if a real list
OP1/OP2 = list element if a complex list

**Registers
destroyed:** All

Remarks:

Example:

IncLstSize

Category: List

Description: Increments the size of an existing list in RAM by adding one element at the end of the list. No value is stored in the new element.

Input:

Registers: A = ListObj if the list has real elements
 = CListObj if the list has complex elements
 DE = pointer to start of list's data storage, output of **FindSym**

Flags: None

Others: None

Output:

Registers: DE = intact
 HL = new dimension of the list

Flags: None

Others: (insDelPtr) = pointer to start of the list

**Registers
destroyed:** All

Remarks: DO NOT ATTEMPT ON AN ARCHIVED LIST. A memory error will be generated if insufficient RAM. The size bytes of the list are adjusted. All pointers in the symbol table are updated.

(continued)

IncLstSize *(continued)*

Example: Increment real list L1 and store a 3 in the new element.

```

LD      HL,L1Name
RST     rMov9ToOP1      ; OP1 = L1 name
B_CALL FindSym          ; look it up, DE = pointer to
                        ; data storage
JP      C,UndefinedL1   ; jump out if L1 is not
                        ; defined
;
LD      A,B             ; get archived status
OR      A               ; in RAM or archived
JP      NZ,errArchived  ; cannot insert if archived
;
LD      A,ListObj       ; type of list in ACC
;
B_CALL IncLstSize       ; insert element at end
;
PUSH    DE              ; save pointer to list
PUSH    HL              ; save last element #, just
                        ; inserted
;
B_CALL  OP1Set3         ; OP1 = 3
;
POP     HL              ; restore
POP     DE
;
B_CALL  PutToL          ; store OP1 to inserted
                        ; element
;
L1Name:
DB      ListObj,tVarLst,tL1,0,0

```

InsertList

Category: List

Description: Inserts one or more elements into an existing list, residing in RAM.

Inputs:

Registers: A = ListObj if the list has real elements
A = CListObj if the list has complex elements
DE = pointer to start of list's data storage, output of **FindSym**
HL = number of elements to insert
BC = List element number to insert after

Flags: CA = 0 to set new elements to 0
CA = 1 to set new elements to 1

Others: None

Outputs:

Registers: DE = intact
HL = new dimension of the list.

Flags: None

Others: (insDelPtr) = pointer to start of the list

Registers destroyed: All

Remarks: DO NOT ATTEMPT ON AN ARCHIVED LIST. A memory error will be generated if insufficient RAM. The size bytes of the list are adjusted. All pointers in the symbol table are updated

(continued)

InsertList *(continued)*

Example: Insert three new elements in list L1 after its second element, set the new elements to 0's.

```

LD      HL,L1Name
RST     rMov9ToOP1      ; OP1 = L1 name
B_CALL FindSym          ; look it up, DE = pointer to
                        ; data storage
JP      C,UndefinedL1   ; jump out if L1 is not
                        ; defined
;
LD      C,A              ; save type
LD      A,B              ; get archived status
OR      A                ; in RAM or archived
JP      NZ,errArchived   ; cannot insert if archived
;
LD      A,C              ; get type back
AND     1Fh              ; mask type of list in ACC
LD      HL,3              ; want to insert 3 elements
LD      BC,2              ; insert after 2nd element
OR      A                ; CA = 0, to set new elements
                        ; to 0
;
B_CALL  InsertList       ; insert elements
;
L1Name:
DB      ListObj,tVarLst,tL1,0,0

```

PutToL

Category: List

Description: Stores either a floating-point number or a complex pair to an existing element of a list.

Inputs:

Registers: HL = element number to store to
There is no check to see if this element is valid for the list.
DE = pointer to the start of the list's data area, output of **FindSym**

Flags: None

Others: None
OP1 = floating-point number set to RealObj to store to a real list
OP1/OP2 = floating-point numbers representing a complex number to store to a complex list

There are no checks made that the correct data type is being stored to the correct type of list (real/complex).

Outputs:

Registers: DE = pointer to next element in the list

Flags: None

Others: OP1/OP2 = intact

Registers destroyed: All

Remarks:

Example:

```

; Look up L1 and store 1 to element 30.
      LD      HL,L1name
      B_CALL  Mov9ToOP1    ; OP1 = name
;
      B_CALL  FindSym      ; look up
      RET     C            ; return if undefined
;
                                ; DE = pointer to data area of list
;
      PUSH   DE            ; save pointer
      B_CALL  OP1Set1      ; OP1 = 1
;
      POP    DE
      LD     HL,30d        ; element to store to
      B_CALL  PutToL       ; store 1 to element 30
      RET
;
L1name:
      DB      ListObj,tVarLst,tL1,0

```

A

System Routines — Math

AbsO1O2Cp	368
AbsO1PAbsO2	369
ACos	370
ACosH	371
ACosRad	372
Angle	373
ASin	374
ASinH	375
ASinRad	376
ATan	377
ATan2	378
ATan2Rad	379
ATanH	380
ATanRad	381
CAbs	382
CAdd	383
CDiv	384
CDivByReal	385
CEtoX	386
CFrac	387
CIntgr	388
CkInt	389
CkOdd	390
CkOP1C0	391
CkOP1Cplx	392
CkOP1FP0	393
CkOP1Pos	394
CkOP1Real	395
CkOP2FP0	396
CkOP2Pos	397
CkOP2Real	398

(continued)

Contents *(continued)*

CkPosInt.....	399
CkValidNum	400
CLN	401
CLog.....	402
ClrLp.....	403
ClrOP1S	404
CMltByReal	405
CMult	406
Conj.....	407
COP1Set0	408
Cos.....	409
CosH	410
CpOP1OP2	411
CpOP4OP3	412
CRecip.....	413
CSqRoot.....	414
CSquare	415
CSub	416
CTenX	417
CTrunc	418
Cube.....	419
CXrootY	420
CYtoX.....	421
DecO1Exp	422
DToR.....	423
EToX	424
ExpToHex.....	425
Factorial.....	426
FPAdd	427
FpDiv	428
FPMult.....	429
FPRecip.....	430
FPSquare	431
FPSub	432
Frac	433
HLTimes9.....	434
HTimesL.....	435

(continued)

Contents *(continued)*

Int	436
Intgr	437
InvOP1S	438
InvOP1SC	439
InvOP2S	440
InvSub	441
LnX	442
LogX	443
Max	444
Min	445
Minus1	446
OP1ExpToDec	447
OP1Set0, OP1Set1, OP1Set2, OP1Set3, OP1Set4, OP2Set0, OP2Set1, OP2Set2, OP2Set3, OP2Set4, OP2Set5, OP2Set60, OP3Set0, OP3Set1, OP3Set2, OP4Set0, OP4Set1, OP5Set0	448
OP2Set8	449
OP2SetA	450
Plus1	451
PtoR	452
RandInit	453
Random	454
RName	455
RndGuard	456
RnFx	457
Round	458
RToD	459
RToP	460
Sin	461
SinCosRad	462
SinH	463
SinHCosH	464
SqRoot	465
Tan	466
TanH	467
TenX	468
ThetaName	469
Times2	470

(continued)

Contents *(continued)*

TimesPt5	471
TName	472
ToFrac	473
Trunc	474
XName	475
XRootY	476
YName	477
YToX	478
Zero16D	479
ZeroOP	480
ZeroOP1, ZeroOP2, ZeroOP3	481

AbsO1O2Cp

Category: Math

Description: Compares Abs(OP1) to Abs(OP2).

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point
OP2 = floating point

Outputs:

Registers: None

Flags: Z = 1: Abs(OP1) = Abs(OP2)
Z = 0, CA = 1: Abs(OP1) < Abs(OP2)
Z = 0, CA = 0: Abs(OP1) ≥ Abs(OP2)

Others: OP1 = Abs(OP1)
OP2 = Abs(OP2)

Registers destroyed: A, BC, DE, HL

Remarks: None

Example:

AbsO1PAbsO2

Category: Math

Description: Calculates the sum of the absolute values of the floating point in OP1 plus the floating point in OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point
OP2 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point with value $(\text{Abs}(\text{OP1}) + \text{Abs}(\text{OP2}))$

Registers destroyed: A, BC, DE, HL

Remarks: None

Example:

ACos

Category: Math

Description: Computes the inverse cosine of a floating point. The answer will not go complex.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse cosine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks: Domain error if answer is complex.

Example:

ACosH

Category: Math

Description: Computes inverse hyperbolic cosine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse hyperbolic cosine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks: Domain error if OP1 is negative.

Example:

ACosRad

Category: Math

Description: Computes the inverse cosine of a floating point and force radian mode.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse cosine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

Angle

Category: Math

Description: Calculates a polar complex angle from a rectangular complex.

Input:

Registers: None

Flags: None

Others: OP1 = real representing complex X
OP2 = real representing complex Y

Outputs:

Registers: None

Flags: None

Others: OP1 = real representing complex angle

**Registers
destroyed:** TBD

Remarks: OP1 is not modified.

Example:

ASin

Category: Math

Description: Computes the inverse sine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse sine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

ASinH

Category: Math

Description: Computes the inverse hyperbolic sine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse sine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

ASinRad

Category: Math

Description: Computes the inverse sine of a floating point and force radian mode.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse sine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

ATan

Category: Math

Description: Computes the inverse tangent of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse tangent (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

ATan2

Category: Math

Description: Returns the angle portion of a complex number in rectangular form.

Inputs:

Registers: None

Flags: trigDeg, (IY + trigFlags) = 1 to return angle in degrees
= 0 to return angle in radians

Others: OP1 = imaginary part of complex number, floating-point number
OP2 = real part of complex number, floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = the angle portion of the polar form of the input rectangular complex number.

Registers destroyed: All

RAM used: OP1 – OP5

Remarks:

Example:

ATan2Rad

Category: Math

Description: Returns the angle portion of a complex number in rectangular form — forced to return the angle in radians no matter what the current system angle settings are.

Inputs:

Registers: None

Flags: None

Others: OP1 = imaginary part of complex number, floating-point number
OP2 = real part of complex number, floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = the angle portion of the polar form of the input rectangular complex number.

Registers destroyed: All

RAM used: OP1 – OP5

Remarks:

Example:

ATanH

Category: Math

Description: Computes the inverse hyperbolic tangent of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse hyperbolic tangent (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks: Initial input rules:

- If floating point = 0, then output = 0.
- If the absolute value of input is greater than 1 then domain error.
- FOR |OP1| <.7 Use Cordic; otherwise, use Logs.

Example:

ATanRad

Category: Math

Description: Computes the inverse tangent of a floating point and forces radian mode.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = inverse tangent (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

CAbs

Category: Math

Description: Computes the magnitude of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result, a real number

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks: $\text{SqRoot}(\text{OP1}^2 + \text{OP2}^2)$.

Example: B_CALL CAbs

CAdd

Category: Math

Description: Addition of two complex numbers.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument
FPS1/FPST = first argument

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result (first argument) + (second Argument)

**Registers
destroyed:** All

RAM used: OP1 – OP2

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

CDiv

Category: Math

Description: Division of two complex numbers.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument
FPS1/FPST = first argument

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result (first argument) / (second Argument)

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

CDivByReal

Category: Math

Description: Divides a complex number by a real number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number
OP3 = floating point real number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result, OP1/OP2 / OP3
OP3 = intact

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks:

Example: B_CALL CDivByReal

CEtoX

Category: Math

Description: Returns e^X where X is a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks:

Example: B_CALL CEtoX

CFrac

Category: Math

Description: Returns the fractional part of both the real and imaginary components of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1, OP2

Remarks:

Example: B_CALL Cfrac

CIntgr

Category: Math

Description: Executes the Intgr function on a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1, OP2

Remarks: Return the next integer less than or equal to, for both the real and imaginary parts of the complex number.

See **Intgr**.

Example: B_CALL CIntgr

CkInt

Category: Math

Description: Tests floating-point number to be an integer.

Inputs:

Registers: HL = pointer to the exponent of the number to check

Flags: None

Others: None

Outputs:

Registers: Z = 1 if integer, Z = 0 if noninteger

Flags: None

Others: None

**Registers
destroyed:** All

RAM used: OP1 – OP5

Remarks: If exponent of OP1 > 13 then it is considered to be an integer.

Example:

CkOdd

Category: Math

Description: Tests if a floating-point number is odd or even.

Inputs:

Registers: HL = pointer to exponent of number to check

Flags: None

Others: None

Outputs:

Registers: None

Flags: If even, then Z = 1. If odd, then Z = 0.

Others: None

**Registers
destroyed:** All

RAM used: None

Remarks: If exponent of OP1 > 13, then it is considered to be an even.
If $0 < \text{Abs}(\text{OP1}) < 1$, then it is considered odd, negative exponent.

Example: Test a floating-point number in OP1 for odd/even.

```
LD      HL,OP1+1
B_CALL  CkOdd
JP      Z,Is_Even
```

CkOP1C0

Category: Math

Description: Tests a complex number in OP1/OP2 to be (0,0).

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: If (0,0), then Z = 1; otherwise, Z = 0.

Others: None

**Registers
destroyed:** A

Remarks:

Example: B_CALL CkOP1C0

CkOP1Cplx

Category: Math

Description: Tests value in OP1 for complex data type.

Inputs:

Registers: None

Flags: None

Others: (OP1) = objects data type byte

Outputs:

Registers: None

Flags: If OP1 contains a complex number, then $Z = 1$; otherwise, $Z = 0$.

Others: None

**Registers
destroyed:** A

RAM used: None

Remarks:

Example: B_CALL CkOP1Cplx

CkOP1FP0

Category: Math

Description: Tests floating-point number in OP1 to be 0.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: Z = 1: OP1 = 0
Z = 0: OP1 <> 0

Others: None

**Registers
destroyed:** A

RAM used: None

Remarks:

Example: B_CALL CkOP1FP0

CkOP1Pos

Category: Math

Description: Tests floating-point number in OP1 to be positive.

Inputs:

Registers: None

Flags: None

Others: (OP1) = sign byte of floating-point number in OP1

Outputs:

Registers: ACC bit 7 = sign bit

Flags: If $OP1 > 0$, $Z = 1$; otherwise, $Z = 0$.

Others: None

**Registers
destroyed:** A

RAM used: None

Remarks:

Example: B_CALL CkOP1Pos

CkOP1Real

Category: Math

Description: Tests object in OP1 to be a real data type.

Inputs:

Registers: None

Flags: None

Others: (OP1) = objects data type byte

Outputs:

Registers: ACC = data type of object in OP1

Flags: If OP1 contains a real number, then Z = 1; otherwise, Z = 0.

Others: None

**Registers
destroyed:** A

RAM used: None

Remarks:

Example: B_CALL CkOP1Real

CkOP2FP0

Category: Math

Description: Tests floating-point number in OP2 to be 0.

Inputs:

Registers: None

Flags: None

Others: OP2 = floating-point number

Outputs:

Registers: None

Flags: If OP2 = 0, then Z = 1; otherwise, Z = 0.

Others: None

**Registers
destroyed:** A

RAM used: None

Remarks:

Example: B_CALL CkOP2FP0

CkOP2Pos

Category: Math

Description: Tests floating-point number in OP2 to be positive.

Inputs:

Registers: None

Flags: None

Others: (OP2) = sign byte of floating-point number in OP2

Outputs:

Registers: ACC bit 7 = sign bit

Flags: If $OP2 > 0$, then $Z = 1$; otherwise, $Z = 0$.

Others: None

**Registers
destroyed:** A

RAM used: None

Remarks:

Example: B_CALL CkOP2Pos

CkOP2Real

Category: Math

Description: Tests object in OP2 to be a real data type.

Inputs:

Registers: None

Flags: None

Others: (OP1) = objects data type byte

Outputs:

Registers: ACC = data type of object in OP2

Flags: If OP2 contains a real number, then Z = 1; otherwise, Z = 0.

Others: None

**Registers
destroyed:** A

RAM used: None

Remarks:

Example: B_CALL CkOP2Real

CkPosInt

Category: Math

Description: Tests floating-point number in OP1 to be a positive integer.

Inputs:

Registers: OP1 = floating-point number

Flags: None

Others: None

Outputs:

Registers: If OP1 is a positive integer, then Z = 1.

Flags: None

Others: None

**Registers
destroyed:** All

RAM used: None

Remarks:

Example:

B_CALL	CkPosInt	; check OP1 = positive integer
JR	Z,PosInt	; jump if positive integer

CkValidNum

Category: Math

Description: Checks for a valid number for a real or complex number in OP1/OP2.

Inputs:

Registers: OP1, if real
OP1 and OP2, if complex

Flags: None

Others: None

Outputs:

Registers: Err: Overflow if exponent > 100
Value set to 0 if exponent < -99

Flags: None

Others: None

Registers destroyed: AF, HL

Remarks: This should be used before storing a real or complex to a user variable or a system variable.

Intermediate results from the math operations can generate values outside of the valid exponent range for the TI-83 Plus. This routine will catch those cases.

If this is not done, then problems can occur when trying to display the invalid numbers.

This does not need to be done after every floating-point operation. The core math routines can handle exponents in the range of +/- 127.

Example: After a floating-point multiply, check the result for validity before stringing to variable X. Assume OP1 and OP2 have values already.

```

;          B_CALL    FPMult      ; generate value to store to 'X'
;
;          B_CALL    CkValidNum   ; make sure valid exponent
;
;          B_CALL    StoX         ; store to 'X'
```

CLN

Category: Math

Description: Computes the natural log of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks:

Example: B_CALL CLN

CLog

Category: Math

Description: Computes the base 10 log of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks:

Example: B_CALL CLog

ClrLp

Category: Math

Description: Clears a memory block (to 00h's).

Inputs:

Registers: HL = address of start of memory block
B = number of bytes to clear

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Memory block cleared

Registers destroyed: B, HL

Remarks: None

Example:

ClrOP1S

Category: Math

Description: Clears the mantissa sign bit in OP1.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks: This routine only acts on the display, not the *textShadow*.

Example:

CMltByReal

Category: Math

Description: Multiplies a complex number by a real number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number
OP3 = floating point real number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result, $OP1/OP2 * OP3$
OP3 = intact

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks:

Example: B_CALL CMltByReal

CMult

Category: Math

Description: Multiplication of two complex numbers.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument
FPS1/FPST = first argument

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result (first argument) * (second argument)

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

Conj

Category: Math

Description: Computes the complex conjugate of a real complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = real complex number

Outputs:

Registers: None

Flags: None

Others: OP2 = -OP2, negate imaginary
Set OP1/OP2 = current complex mode

**Registers
destroyed:** All

Remarks: No error checking. Sets Ans to the current complex mode.

Example:

COP1Set0

Category: Math

Description: Puts a complex (0,0) in OP1/OP2.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex (0,0)

**Registers
destroyed:** A, HL

Remarks: OP1 is not modified.

Example:

Cos

Category: Math

Description: Computes the cosine of a floating point.

Inputs:

Registers: None

Flags: None

Others: Current angle mode
OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = cosine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

Cosh

Category: Math

Description: Computes the hyperbolic cosine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = hyperbolic cosine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example:

CpOP1OP2

Category: Math

Description: Compares floating-point values in OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point value
OP2 = floating-point value

Outputs:

Registers: None

Flags: Z = 1: OP1 = OP2
Z = 0, CA = 1: OP1 < OP2
Z = 0, CA = 0: OP1 ≥ OP2

Others: None

Registers destroyed: A, BC, DE, HL

Remarks: OP1 and OP2 are preserved.

Example:

CpOP4OP3

Category: Math

Description: Compares floating-point values in OP4 and OP3.

Inputs:

Registers: None

Flags: None

Others: OP4 = floating-point value
OP3 = floating-point value

Outputs:

Registers: None

Flags: Z = 1: OP4 = OP3
Z = 0, CA = 1: OP4 < OP3
Z = 0, CA = 0: OP4 ≥ OP3

Others: None

Registers destroyed: A, BC, DE, HL

RAM used: OP1, OP2

Remarks: OP4 and OP3 are preserved.

Example:

CRecip

Category: Math

Description: Computes the reciprocal of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = input complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = resulting complex number

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks:

Example: B_CALL CRecip

CSqRoot

Category: Math

Description: Computes the square root of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks:

Example: B_CALL CSqRoot

CSquare

Category: Math

Description: Computes the square of a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks:

Example: B_CALL CSquare

CSub

Category: Math

Description: Subtracts two complex numbers.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument
FPS1/FPST = first argument

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result (first argument) – (second argument)

Registers destroyed: All

RAM used: OP1 – OP3

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: Assume that variable X and Y both have complex values.

Recall the contents and subtract Y from X, such that $OP1/OP2 = X - Y$

```

                B_CALL      RclX          ; OP1/OP2 = complex value of X
;
; This next call pushes OP1 the real part of the complex #, onto FPST;
; then pushes OP2, the imaginary part, onto the FPST which pushes the
; real part to FPS1 position.
;
; FPS1 = 1st argument real part
; FPST = 1st argument imaginary part
;
                B_CALL      PushMCplx01   ; push 1st argument on FPS, X
;
                B_CALL      RclY          ; OP1/OP2 = complex value of Y
;
                B_CALL      CSub          ; OP1/OP2 = result X - Y, FPS
; is cleaned

```

CTenX

Category: Math

Description: Returns 10^X where X is a complex number.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks:

Example: B_CALL CTenX

CTrunc

Category: Math

Description: Returns the integer part of both the real and imaginary components of a complex number; no rounding is done.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result

**Registers
destroyed:** All

RAM used: OP1, OP2

Remarks: No rounding is done; for example, Trunc ($1.5 + 3i$) returns $1 + 3i$.

Example: B_CALL CTrunc

Cube

Category: Math

Description: Computes the cube of a floating-point number.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = OP1³

Registers destroyed: A, BC, DE, HL

RAM used: OP1 – OP3

Remarks:

Example: B_CALL Cube

CXrootY

Category: Math

Description: Returns the complex root of a complex number, $y^{(1/x)}$.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument (y)
FPS1/FPST = first argument (x)

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result $\text{second_argument}^{(1/(\text{first_argument}))}$

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

CYtoX

Category: Math

Description: Raises a complex number to a complex power, y^x .

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = second argument (x)
FPS1/FPST = first argument (y)

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = complex result $\text{first_argument}^{\text{second_argument}}$

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: First argument is removed from the FPS (Floating Point Stack).

Example: See **CSub**.

DecO1Exp

Category: Math

Description: Decrements OP1 exponent.

Inputs:

Registers: None

Flags: None

Others: OP1

Outputs:

Registers: None

Flags: None

Others: Decrement OP1 exponent by one.

**Registers
destroyed:** A

Remarks:

Example: B_CALL DecO1Exp

DToR

Category: Math

Description: Converts the floating-point number in OP1 from a degrees angle to a radian angle.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to convert

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point number representing the radian angle of the input value

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks:

Example:

EToX

Category: Math

Description: Computes $e^{OP1} = 10^{(OP1 * \text{LOG}(e))}$.

Inputs:

Registers: None

Flags: None

Others: OP1 = value e is raised to

Outputs:

Registers: None

Flags: None

Others: OP1 = result

Registers destroyed: All, OP2, OP3, OP4

Remarks:

Example:

ExpToHex

Category: Math

Description: Converts absolute value of one-byte.
Exponent (in HL) to hexadecimal.

Inputs:

Registers: (HL) = exponent to convert

Flags: None

Others: None

Outputs:

Registers: (HL) = absolute value of exponent

Flags: None

Others: None

**Registers
destroyed:** A

Remarks: This converts the floating point exponent value from the offset type (e.g., 7Fh = 10^{-1} , 80h = 10^0 , 81h = 10^1 ,...) to a value of 0...n. It treats positive and negative exponents the same:

e.g., 80h = 0
81h = 1
82h = 2
7Fh = -1
7Eh = -2

See **OP1ExpToDec** for another exponent conversion routine.

Example:

```
LD      HL, Exponent
LD      (HL), 7Eh
B_CALL  ExpToHex      ; change (HL) from FEh -> 02h.
```

Factorial

Category: Math

Description: Computes the factorial of an integer or a multiple of .5.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, must be an integer or a multiple of .5 in the range of -.5 to 69

Outputs:

Registers: None

Flags: None

Others: OP1 = factorial of input, floating-point number. Else, error if input is out of range.

Registers destroyed: All

RAM used: OP1 – OP3

Remarks:

Example:

FPAdd

Category: Math

Description: Floating point addition of OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, argument one
OP2 = floating-point number, argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result $OP1 + OP2$

**Registers
destroyed:** All

RAM used: OP1, OP2

Remarks:

Example: `B_CALL` `FPAdd`

FPDiv

Category: Math

Description: Floating point division of OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, argument one
OP2 = floating-point number, argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result OP1 / OP2
OP2 = intact

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3

Remarks:

Example: `B_CALL` `FPDiv`

FPMult

Category: Math

Description: Floating point multiplication of OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, argument one
OP2 = floating-point number, argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result $OP1 * OP2$
OP2 = intact

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3

Remarks:

Example: `B_CALL FPMult`

FPRecip

Category: Math

Description: Floating point reciprocal of OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result $1 / OP1$
OP2 = input OP1

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3

Remarks:

Example: B_CALL FPRecip

FPSquare

Category: Math

Description: Floating point square of OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result $OP1 * OP1$
OP2 = input OP1

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3

Remarks:

Example: B_CALL FPSquare

FPSub

Category: Math

Description: Floating point subtraction of OP1 and OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, argument one
OP2 = floating-point number, argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point result $OP1 - OP2$

**Registers
destroyed:** All

RAM used: OP1, OP2

Remarks:

Example: B_CALL FPSub

Frac

Category: Math

Description: Returns the fractional part of a floating-point number.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result

**Registers
destroyed:** All

RAM used: OP1

Remarks: No rounding; for example, $\text{Frac}(1.5) = .5$

Example: `B_CALL Frac`

HLTimes9

Category: Math

Description: Multiplies HL by nine.

Inputs:

Registers: HL = multiplicand

Flags: None

Others: None

Outputs:

Registers: HL = $HL * 9$ module 65536

Flags: CA = 1: answer larger than 65535
CA = 0: answer less than 65535

Others: None

**Registers
destroyed:** BC

Remarks: None

Example:

HTimesL

Category: Math

Description: Multiplies H (register) * L (register).

Inputs:

Registers: H, L

Flags: None

Others: None

Outputs:

Registers: HL = product of (original H) * (original L)

Flags: None

Others: None

Registers destroyed: B, DE

Remarks: Restriction: H cannot be 0; If H is 0, performs 256 * L.
Cannot overflow if $H > 0$.

Example:

Int

Category: Math

Description: Rounds a floating-point number to an integer.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to round

Outputs:

Registers: None

Flags: None

Others: OP1 = Int (OP1)

**Registers
destroyed:** All

RAM used: OP1

Remarks: The mantissa sign of the input has no affect on the result.

Example: B_CALL Int

Intgr

Category: Math

Description: Returns the integer.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result

Registers destroyed: A, BC, DE, HL

Remarks: If OP1 is an integer, then result = OP1. Otherwise, for positive numbers, returns the same as Trunc (OP1); for negative numbers, returns the Trunc (OP1 - 1).

Example:

InvOP1S

Category: Math

Description: Negates a floating-point number OP1, if OP1 = 0 then set OP1 = positive.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number. No check is made for a valid floating-point number.

Outputs:

Registers: None

Flags: None

Others: OP1 = -(OP1), unless 0 then it is set to positive.

Registers destroyed: A

Remarks:

Example: Set OP1 = -1

```
B_CALL    OP1Set1    ; OP1 = floating point 1
B_CALL    InvOP1S    ; OP1 = -1
```

InvOP1SC

Category: Math

Description: Used to negate a complex number in OP1/OP2 by negating both OP1 and OP2. If OP1 or OP2 = 0, then that OP register is set positive.

Inputs:

Registers: None

Flags: None

Others: OP1/OP2 = two floating-point numbers that make up a complex number

Outputs:

Registers: None

Flags: None

Others: OP1 = -(OP1), unless 0 then it is set to positive
OP2 = -(OP2), unless 0 then it is set to positive

Registers destroyed: A

Remarks:

Example:

InvOP2S

Category: Math

Description: Negates a floating-point number OP2, if OP2 = 0 then set OP2 = positive.

Inputs:

Registers: None

Flags: None

Others: OP2 = floating-point number, no check is made for a valid floating-point number.

Outputs:

Registers: None

Flags: None

Others: OP2 = -(OP2), unless 0 then it is set to positive

**Registers
destroyed:** A

Remarks:

Example: Set OP2 = -1

```
B_CALL    OP2Set1      ; OP2 = floating point 1
B_CALL    InvOP2S      ; OP2 = -1
```

InvSub

Category: Math

Description: Negates OP1 and add to OP2.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point
OP2 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point with value $(-OP1) + OP2$

Registers destroyed: A, BC, DE, HL

RAM used: OP1, OP2

Remarks: None

Example:

LnX

Category: Math

Description: Returns natural log of a floating-point number in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, must be positive

Outputs:

Registers: None

Flags: None

Others: Error if OP1 is negative
Else OP1 = Ln(OP1)

Registers destroyed: All

RAM used: OP1 – OP5

Remarks: A system error can be generated. See section on Error Handlers.

Example: Compute the Ln(OP1), install an error handler to avoid the system reporting the error.

```

;               AppOnErr    CatchError    ; install error handler
;
;               B_CALL      LnX           ; compute Ln(OP1)
;
;               AppOffErr                                ; remove error handler, no
;                                                       ; error occurred
;
;               RET
;
; come here if LnX generated an error
;
CatchError:
```

LogX

Category: Math

Description: Returns log base 10 of a floating-point number in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, must be positive

Outputs:

Registers: None

Flags: None

Others: Error if OP1 is negative
Else OP1 = Log(OP1)

**Registers
destroyed:** All

RAM used: OP1 – OP5

Remarks: A system error can be generated. See section on Error Handlers.

Example: See LnX.

Max

Category: Math

Description: Returns the maximum (OP1, OP2), two floating-point numbers.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number
OP2 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = maximum (OP1, OP2)
OP2 = intact

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks: See **CpOP1OP2**, for non destructive compare.

Example:

Min

Category: Math

Description: Computes the minimum of two floating-point numbers.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number argument one
OP2 = floating-point number argument two

Outputs:

Registers: None

Flags: None

Others: OP1 = minimum (OP1, OP2)
OP2 = intact
OP3 = argument one
OP4 = argument two

Registers destroyed: A, BC, DE, HL

RAM used: OP1 – OP4

Remarks:

Example:

Minus1

Category: Math

Description: Floating point subtraction of one from OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result $OP1 - 1$

**Registers
destroyed:** All

RAM used: OP1, OP2

Remarks:

Example: B_CALL Minus1

OP1ExpToDec

Category: Math

Description: Converts absolute value of exponent to a bcd number.

Inputs:

Registers: None

Flags: None

Others: OP1 + 1 = exponent to convert

Outputs:

Registers: (HL) = OP1 + 1 = |Exp| as hex
A = |Exp| as bcd

Flags: None

Others: OP1 + 1 = |Exp| as hex

**Registers
destroyed:** A, BC

Remarks: Overflow Error if |Exp| > 99

Example:

; Input OP1 + 1 value ->		Output OP1 + 1 and A register	
81h (10 ¹)	->	01h	& 01h
7Fh (10 ⁻¹)	->	01h	& 01h
8Dh (10 ¹³)	->	0Dh	& 13h
73h (10 ⁻¹³)	->	0Dh	& 13h

OP1Set0, OP1Set1, OP1Set2, OP1Set3, OP1Set4, OP2Set0, OP2Set1, OP2Set2, OP2Set3, OP2Set4, OP2Set5, OP2Set60, OP3Set0, OP3Set1, OP3Set2, OP4Set0, OP4Set1, OP5Set0

Category: Math Utility

Description: Sets value of OP(x) to floating point (value).

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP(x) = floating-point value

Registers destroyed: A, HL

Remarks:

Combinations Available:							
Value	0	1	2	3	4	5	60
Register							
OP1	X	X	X	X	X		
OP2	X	X	X	X	X	X	X
OP3	X	X	X				
OP4	X	X					
OP5	X						

Example: B_CALL OP2Set5

OP2Set8

Category: Math

Description: Sets OP2 = floating point 8.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP2 = floating point 8

**Registers
destroyed:** A, HL

Remarks:

Example:

OP2SetA

Category: Math

Description: Sets OP2 = floating-point value between 0 and 9.9.

Inputs:

Registers: ACC = two digits of mantissa to set OP2 to

Flags: None

Others: OP2 set to floating-point value

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: A, HL

Remarks:

Example:

```
    ; Set OP2 = 7.6
                LD      A,76h      ; mantissa digits
    B_CALL      OP2SetA      ; OP2 = 7.6
```

Plus1

Category: Math

Description: Floating point subtraction of one from OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point result $OP1 - 1$

**Registers
destroyed:** All

RAM used: OP1, OP2

Remarks:

Example: `B_CALL Plus1`

PtoR

Category: Math

Description: Converts complex number in OP1/OP2 from a polar complex number to a rectangular complex number.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number radius part of complex number
OP2 = floating-point number angle part of complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = rectangular representation of input polar complex number

Registers destroyed: All

RAM used: OP1 – OP6

Remarks:

Example:

RandInit

Category: Math

Description: Initializes random number seeds to default value.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: HL, DE, BC

Remarks: Seeds initialized.

Example:

Random

Category: Math

Description: Returns a random floating-point number, $0 < \text{number} < 1$.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point random number

**Registers
destroyed:** All

RAM used: OP1 – OP3

Remarks: See **RnFx** and **Round** routines.

Example:

RName

Category: Math

Description: Constructs a name for real variable R in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for R in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

RndGuard

Category: Math

Description: Rounds a floating-point number to 10 mantissa digits. The exponent value has no effect on this routine.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to round to 10 mantissa digits
(fmtDigits) = current fix value
Offh = floating, no rounding will be done
Otherwise, the value is the number of decimal
Digits to round to, 0 – 9

Outputs:

Registers: None

Flags: None

Others: OP1 = input floating point rounded to 10 mantissas digits

**Registers
destroyed:** All

RAM used: OP1

Remarks: See the **RnFx** and **Round** routines.

Example:

RnFx

Category: Math

Description: Rounds a floating-point number to the current FIX setting for the calculator. This will round the digits following the decimal point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to round

Outputs:

Registers: None

Flags: None

Others: OP1 = input rounded to at maximum of 10 mantissa digits
(fmtDigits) = current fix value
0ffh = floating, no rounding will be done
Otherwise, the value is the number of decimal
Digits to round to, 0 – 9

**Registers
destroyed:** All

RAM used: OP1

Remarks: See **Round** and **RndGuard** routines.

Example:

Round

Category: Math

Description: Rounds a floating-point number to a specified number of decimal places. This will round the digits following the decimal point.

Inputs:

Registers: D = number of decimal places to round to, 0 – 9

Flags: None

Others: OP1 = floating-point number to round
(fmtDigits) = current fix value
Offh = floating, no rounding will be done
Otherwise, the value is the number of decimal digits to round to, 0 – 9

Outputs:

Registers: None

Flags: None

Others: OP1 = input rounded to at maximum of 10 mantissa digits

Registers destroyed: All

RAM used: OP1

Remarks: See **RnFx** and **RndGuard** routines.

Example:

RToD

Category: Math

Description: Converts the floating-point number in OP1 from a radian angle to a degree angle.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number to convert

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point number representing the degree angle of the input value.

Registers destroyed: All

RAM used: OP1, OP2, OP3

Remarks: See **DToR** routine.

Example:

RTOP

Category: Math

Description: Converts complex number in OP1/OP2 from a rectangular complex number to a polar complex number.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number X part of complex number
OP2 = floating-point number Y part of complex number

Outputs:

Registers: None

Flags: None

Others: OP1/OP2 = polar representation of input rectangular complex number

Registers destroyed: All

RAM used: OP1 – OP6

Remarks: See **RTOP** routine.

Example:

Sin

Category: Math

Description: Computes the sine and cosine of a floating point.

Inputs:

Registers: Current angle mode
OP1 = floating point

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = sine (floating point)
OP2 = cosine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example: B_CALL Sin

SinCosRad

Category: Math

Description: Computes the sine and cosine of a floating point and radian mode is forced.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = sine (floating point)
OP2 = cosine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example: `B_CALL SinCosRad`

SinH

Category: Math

Description: Computes hyperbolic sine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = hyperbolic sine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example: B_CALL SinH

SinHCosH

Category: Math

Description: Computes the hyperbolic sine and cosine of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = hyperbolic sine (floating point)
OP2 = hyperbolic cosine (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example: B_CALL SinHCosH

SqRoot

Category: Math

Description: Returns the square root of OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number, must be positive

Outputs:

Registers: None

Flags: None

Others: Error if OP1 is negative, else OP1 = Sqrt(OP1)

**Registers
destroyed:** All

RAM used: OP1 – OP3

Remarks: See section on Error Handlers.

Example:

Tan

Category: Math

Description: Computes the tangent of a floating point.

Inputs:

Registers: None

Flags: None

Others: Current angle mode
OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = tangent (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example: B_CALL Tan

TanH

Category: Math

Description: Computes the hyperbolic tangent of a floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = hyperbolic tangent (floating point)

**Registers
destroyed:** All

RAM used: OP1, OP2, OP3, OP4, OP5

Remarks:

Example: B_CALL TanH

TenX

Category: Math

Description: Returns $10^{(OP1)}$.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = $10^{(OP1)}$

**Registers
destroyed:** All

RAM used: OP1 – OP4

Remarks:

Example:

ThetaName

Category: Math

Description: Constructs a name for real variable Theta in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for Theta in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

Times2

Category: Math

Description: Calculates OP1 times two.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point with value $OP1 * 2.0$
OP2 = floating point 2

Registers destroyed: A, BC, DE, HL

RAM used: OP1, OP2

Remarks: None

Example:

TimesPt5

Category: Math

Description: Calculates OP1 times 0.5.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = floating point with value $OP1 * 0.5$
OP2 = floating point 0.5

Registers destroyed: A, BC, DE, HL

RAM used: OP1, OP2

Remarks:

Example:

TName

Category: Math

Description: Constructs a name for real variable T in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for T in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

ToFrac

Category: Math

Description: Converts a floating-point number to the integer numerator and integer denominator of the equivalent fraction.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: Carry = 0: Success
= 1: Failure.

Others: OP1:
On Failure — unchanged.
On Success — Numerator (floating-point integer)
OP2:
On Failure — unchanged.
On Success — Denominator (floating-point integer)

Registers destroyed: All

Remarks: Also modifies OP3, OP4, OP5, OP6.
Smallest possible denominator is created.
Fails if denominator must be > 999.

Example:

```

LD      HL,ExampleNum
RST     rMov9ToOP1
; OP1 = 1.25
      B_CALL      ToFrac
; Convert to fraction form

; Carry is now 0 (success)

; OP1 now contains: 00h 80h 50h 00h 00h 00h 00h 00h 00h = 5

; OP2 now contains: 00h 80h 40h 00h 00h 00h 00h 00h 00h = 4
LD      HL,ExampleNum2
RST     rMov9ToOP1
; OP1 = 1.2345678901234
      B_CALL      ToFrac
; Convert to fraction form
; Carry is now 1 (failure)
; ExampleNum = 1.25
ExampleNum:  DB 00h, 80h, 12h, 50h, 00h, 00h, 00h, 00h, 00h
; ExampleNum2 = 1.2345678901234
ExampleNum:  DB 00h, 80h, 12h, 34h, 56h, 78h, 90h, 12h, 34h
```

Trunc

Category: Math

Description: Truncates the fractional portion of a floating-point number returning the integer portion with no rounding.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: None

Others: OP1 = Trunc (OP1)

**Registers
destroyed:** All

RAM used: OP1 – OP2

Remarks:

Example: `Trunc(1.5)` = 1

XName

Category: Math

Description: Constructs a name for real variable X in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for X in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

XRootY

Category: Math

Description: Inverses power function and returns $OP1^{(1/OP2)}$.

Inputs:

Registers: None

Flags: None

Others: OP1 = number to find root of, floating point
OP2 = root to find, floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = result if no error, floating point

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks:

Example:

YName

Category: Math

Description: Constructs a name for real variable Y in the format required by routine **FindSym**.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = contains variable name for Y in format required by routine **FindSym**

Registers destroyed: A, HL

Remarks: This routine is used to prepare for a call to routine **FindSym**.

Example:

YToX

Category: Math

Description: Power function, returns $OP1^{OP2}$.

Inputs:

Registers: None

Flags: None

Others: OP1 = number to raise to a power, floating point
OP2 = power, floating point

Outputs:

Registers: None

Flags: None

Others: OP1 = result if no error, floating point

**Registers
destroyed:** All

RAM used: OP1 – OP6

Remarks:

Example:

Zero16D

Category: Math

Description: Sets eight-byte memory block to all 00h's.

Inputs:

Registers: HL = start of target block in memory

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Memory block starting at original HL is all 00h's

Registers destroyed: A, HL

Remarks:

Example:

ZeroOP

Category: Math

Description: Sets 11 bytes in OP(x) to 00h.
Note that this does not set the value to floating point 0.0.

Inputs:

Registers: HL = pointer to OP(x), x = 1...6

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP(x) = all 11 bytes 00h

Registers destroyed: A (= 0), HL

Remarks:

Example:

```

; Set OP2 contents to all 00h:
; OP2+0 OP2+1 OP2+3 OP2+4 OP2+5 OP2+6 OP2+7 OP2+8 OP2+9 OP2+10
; 00h   00h   00h   00h   00h   00h   00h   00h   00h   00h
                LD      HL,OP2
                B_CALL  ZeroOP

```

ZeroOP1, ZeroOP2, ZeroOP3

Category: Math

Description: Sets 11 bytes in OP(x) to 00h.
Note that this does not set the value to floating point 0.0.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP(x) = all 11 bytes 00h

Registers destroyed: A(= 0), HL

Remarks: Combinations Available:
(x) = 1, 2, 3

Example:

```

; Set OP2 contents to all 00h:
; OP2+0 OP2+1 OP2+3 OP2+4 OP2+5 OP2+6 OP2+7 OP2+8 OP2+9 OP2+10
; 00h   00h   00h   00h   00h   00h   00h   00h   00h   00h
      B_CALL      ZeroOP2

```

A

System Routines — Matrix

AdrMRow.....	483
GetMToOP1	484
PutToMat.....	485

AdrMRow

Category: Matrix

Description: Computes the RAM address of the start of a row of a matrix.

Input:

Registers: DE = pointer to start of matrix's data storage, output of **FindSym**
B = row to compute address of

Matrix Element (1,1) is checked for real or complex data type to determine if the matrix is real or complex.

Do not use this routine on a matrix that does not have element (1,1) initialized.

Flags: None

Others: None

Output:

Registers: HL = pointer in RAM to start of desired element

Flags: None

Others: None

**Registers
destroyed:**

Remarks: This routine does not check to see if the row address requested is within the current dimension of the matrix. See **AdrMEle** routine.

Example:

GetMToOP1

Category: Matrix

Description: Copies an element from a matrix to OP1.

Input:

Registers: BC = element to get, row,col
DE = pointer to start of matrix's data storage

Flags: None

Others: None

Output:

Registers: HL = pointer to next element in the same row, or the start of the next row of the matrix.

Flags: None

Other: OP1 = matrix element, floating-point number

Registers destroyed: All

Remarks:

Example:

PutToMat

Category: Matrix

Description: Stores a floating-point number to an existing element of a matrix.

Inputs:

Registers: BC = (row, column) to store to
There is no check to see if this element is valid for the matrix.
DE = pointer to the start of the matrix's data area, output of **FindSym**

Flags: None

Others: None
OP1 = floating-point number

Outputs:

Registers: DE = pointer to next element in the matrix. This will be the next element in the same row or the start of the next row.

Flags: None

Others: OP1 = intact

Registers destroyed: All

Remarks:

Example: Look up MatA and store 1 to element (5,7).

```

                                LD          HL,MatAname
                                B_CALL      Mov9ToOP1      ; OP1 = name
                                ;
                                B_CALL      FindSym         ; look up
                                RET          C              ; return if undefined
                                ;
                                ; DE = pointer to data area of
                                ; matrix
                                ;
                                PUSH        DE              ; save pointer
                                B_CALL      OP1Set1         ; OP1 = 1
                                ;
                                POP         DE
                                LD          BC,5*257+7      ; element to store to (5,7)
                                B_CALL      PutToMat         ; store 1 to element (5,7)
                                RET
MatAname:
                                DB          MatObj,tVarMat,tMatA,0

```

A

System Routines — Memory

Arc_Unarc	488
ChkFindSym	489
CleanAll	491
CloseProg	492
CmpSyms	493
Create0Equ	494
CreateAppVar	495
CreateCList	496
CreateCplx	497
CreateEqu	498
CreatePair	499
CreatePict	500
CreateProg	501
CreateProtProg	502
CreateReal	503
CreateRList	504
CreateRMat	505
CreateStrng	506
DataSize	507
DataSizeA	508
DeallocFPS	509
DeallocFPS1	510
DelMem	511
DelVar	513
DelVarArc	514
DelVarNoArc	515
EditProg	516
EnoughMem	517
Exch9	518
ExLp	519
FindAlphaDn	520

(continued)

Contents *(continued)*

FindAlphaUp.....	522
FindApp.....	524
FindAppNumPages	525
FindAppDn	526
FindAppUp	527
FindSym	528
FixTempCnt.....	530
FlashToRam.....	531
InsertMem	532
LoadCIndPaged.....	534
LoadDEIndPaged	535
MemChk.....	536
PagedGet.....	537
RclGDB2	538
RclN	539
RclVarSym	540
RclX.....	541
RclY.....	542
RedimMat.....	543
SetupPagedPtr	544
SrchVLstDn, SrchVLstUp	545
StMatEl.....	546
StoAns.....	547
StoGDB2	548
StoN	549
StoOther	550
StoR	552
StoSysTok.....	553
StoT.....	554
StoTheta.....	555
StoX	556
StoY	557

Arc_Unarc

Category: Memory

Description: Swaps a variable between RAM and archive.

Inputs:

Registers: None

Flags: None

Others: OP1 contains variable name

Outputs:

Registers: None

Flags: None

Others: Symbol table and data area (RAM and Flash) modified.

Registers destroyed: All

Remarks: Destroys OP3 as well.
Will unarchive a variable already archived and will archive a variable that is currently unarchived.

Gives an Err: Variable for any name that is not archivable or unarchivable (e.g., Groups cannot be unarchived and X cannot be archived).
Gives an Err: Undefined for any name that does not already exist.

Does memory checking to make sure there is enough space (in RAM or in Archive) to store the variable. Generates a memory error if not.

Example:

```

                                ; unarchive variable A (real
                                ; or complex) if it is
                                ; archived:
B_CALL    ZeroOP1              ; set OP1 to all 0s
LD        (OP1+1),tA           ; want to look for floating
                                ; point number named 'A'
RST       rFindSym             ; Data pointer -> DE
                                ; System pointer -> HL
                                ; C if none
JR        C,..skip             ; does not exist, so skip
CALL     NzIfArchived          ; NZ if was in RAM already.
JR        Z,..skip             ; not archived, so no need to
                                ; unarchive
B_CALL    Arc_Unarc            ; unarchive variable.
.....
NzIfArchived:
LD        A,B                  ; B has page information, NZ
                                ; if archived.
OR        A
RET
```

ChkFindSym

Category: Memory

Description: Searches the symbol table structure for a variable.

This particular search routine must be used if the variable to search for is either a Program, AppVar, or Group. It will also work for variables of other types as long as the data type in OP1 input is correct.

This is used to determine if a variable is created and also to return pointers to both its symbol table entry and data storage area.

This will also indicate whether or not the variable is located in RAM or has been archived in Flash ROM.

Inputs:

Registers: (OP1) = one-byte, data type of variable to search for.
 This routine will fail if this data type is not correct.
 (OP1 + 1) to (OP1 + 8) = variable name

Flags: None

Others: None

Outputs:

Registers: CA flag = 1 if symbol was not found
 = 0 if symbol was found

Also if found:

ACC lower 5 bits = data type

ACC upper 3 bits = system flags about variable, do an 1Fh to get type only

B = 0 if variable is located in RAM else variable is archived

B = ROM page located on

If variable is archived then its data cannot be accessed directly, it must be unarchived first.

HL = pointer to the start of the variables symbol table entry

DE = pointer to the start of the variables data area if in RAM

Flags: None

Others: OP1 = variable name

Registers destroyed: All

Remarks: This will not find system variables that are preallocated in system RAM such as Xmin, Xmax etc. Use **RclSysTok** to retrieve their values.

Note: **ChkFindSym** will not find Applications.

(continued)

ChkFindSym *(continued)*

Example: Look for AppVar MYAPPVAR in the symbol table.
 If it exists and is archived then unarchive it and relook it up.
 If it does not exist ; create it with a size of 100 ; bytes.

```

Relook:
        LD          HL,varname
        B_CALL      Mov9ToOP1      ; OP1 = variable name
        B_CALL      ChkFindSym     ; look up
        JR          NC,varCreated  ; jump if it exists
;
        LD          HL,100         ; size to create at
        B_CALL      CreateAppVar   ; create it, HL = pointer to
                                   ; sym entry, DE = pointer to
                                   ; data
        PUSH        HL
        PUSH        DE             ; save during move
        B_CALL      OP4ToOP1      ; OP1 = name
        POP         DE             ; restore
        POP         HL
        JR          done
VarCreated:
        LD          A,B            ; check for archived
        OR          A              ; in RAM ?
        JR          Z,done         ; yes
        B_CALL      Arc_Unarc     ; unarchive if enough RAM
        JR          relook        ; look up pointers again in
                                   ; RAM now done:
        RET
;
VarName:
        DB          AppVarObj,'MYAPPVAR',0
  
```

CleanAll

Category: Memory

Description: Deletes all temporary variables from RAM.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Temporary variables are all deleted

**Registers
destroyed:** All

Remarks: This routine should only be used when there are no temporary variables that exist and are still being used. See the Temporary Variables section in Chapter 2 for further information. See the **ParseInp** and **MemChk** routines.

Example:

CloseProg

Category: Memory

Description: This routine is used after **EditProg** to return unused RAM back to free RAM. The size bytes of the variable are updated by this routine. An application should not update them.

Inputs:

Registers: Each of these are two-bytes:

(iMathPtr1) = pointer to the start of the variables data storage area

(iMathPtr2) = pointer to the byte following the variable data, this will be used to calculate the new size of the variable

(iMathPtr3) = pointer to the byte AFTER the last byte of free RAM inserted

(iMathPtr4) = size of RAM block moved to allow the RAM to be inserted
DO NOT CHANGE THIS VALUE.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: The variable's size is changed. Unused RAM returned to free RAM. Normal allocating and deallocating of RAM can resume.

Registers destroyed: All

Remarks:

Example:

CmpSyms

Category: Memory

Description: Compares Name @HL with Name @DE.

Inputs:

Registers: HL = end of first name in RAM
DE = end of second name in RAM
B = length of name

Flags: None

Others: None

Outputs:

Registers: C = number of letters that match
C = original B if all letters match

Flags: Carry set if Sym2 (HL) > Sym1 (DE)

Others: None

Registers destroyed: AF, BC, DE, HL

Remarks: The names must be the same size. The name lengths should have already been compared before calling this routine.

Example: ; See if the name last used for the Xlist variable in statistics is
; the name "ZEBRA"

```

LD      HL,StZebra
RST     rMov9ToOP1      ; Move 9 bytes to OP1:
                        ; "ZEBRA" + junk

LD      DE,OP1+4
LD      HL,StatX+4
LD      B,5              ; compare 5 bytes
B_CALL  CmpSyms          ; If C = 5 then OP1 = StatX
                        ; name

LD      A,C
CP      5
JR      Z,Match
JR      NoMatch

StZebra DB  "ZEBRA"
```

Create0Equ

Category: Memory

Description: Creates an equation variable of size 0 in RAM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of equation to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. See section on Creating Variables.

Example: Create an empty Y1 equation.

```

                                LD          HL,Y1name
                                RST          rMov9ToOP1  ; OP1 = name
                                B_CALL      Create0Equ   ; if returns then variable created

Y1name:                        DB          EquObj,tVarEqu,tY1,0,0

```

CreateAppVar

Category: Memory

Description: Creates an AppVar variable in RAM.

Inputs:

Registers: HL = size of AppVar to create in bytes

Flags: None

Others: OP1 = name of AppVar to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. Users can only delete and link AppVars. They are intended for Apps to use for state saving upon exiting. See section on Creating Variables.

Example: Create AppVar DOG, 50 bytes in size.

```

                                LD      HL,DOGname
                                RST      rMov9ToOP1      ; OP1 = name
                                ;
                                LD      HL,50
                                B_CALL   CreateAppVar    ; if returns then variable
                                                         ; created

DOGname:      DB      AppVarObj,'DOG',0

```

CreateCList

Category: Memory

Description: Creates a complex list variable in RAM.

Inputs:

Registers: HL = number of elements in the list

Flags: None

Others: OP1 = name of list to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization of the elements is done, assume random. See section on Creating Variables.

Example: Create complex list L1 with 50 elements.

```

                LD        HL,L1name
                RST        rMov9ToOP1    ; OP1 = name
;
                LD        HL,50
                B_CALL     CreateCList    ; if returns then variable
                                           ; created

L1name:        DB        CListObj,tVarLst,tL1,0,0

```

CreateCplx

Category: Memory

Description: Creates a complex variable in RAM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of complex to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. This should not be used to create temp storage space, A-Z * theta. No initialization is done, assume random. See section on Creating Variables.

Example: Create complex A.

```

                                LD      HL,Aname
                                RST      rMov9ToOP1      ; OP1 = name
                                ;
                                B_CALL   CreateCplx      ; if returns then variable
                                                         ; created

Aname:      DB      CplxObj, 'A', 0, 0
```

CreateEqu

Category: Memory

Description: Creates an equation variable in RAM.

Inputs:

Registers: HL = size of equation to create in bytes

Flags: None

Others: OP1 = name of equation to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. See section on Creating Variables.

Example: Create Y1 equation 50 bytes in size.

```

                                LD      HL,Y1name
                                RST      rMov9ToOP1  ; OP1 = name
                                ;
                                LD      HL,50
                                B_CALL   CreateEqu    ; if returns then variable created

Y1name:                        DB      EquObj,tVarEqu,tY1,0,0

```

CreatePair

Category: Memory

Description: Creates a pair of parametric graph equations.

There should never be a situation where only 1 of a pair of parametric equations is created without the other. This routine will check that there is enough memory to create both equations before creating any.

Inputs:

Registers: HL = size to create the equation specified in OP1, either xt or yt. The member of the pair not specified will be created empty.

Flags: None

Others: OP1 = pair member name to create with the specified size

Outputs:

Registers: HL = size of pair member specified

Flags: None

Others: OP1 = pair member name specified
OP4 = pair member name not specified

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM to create the pair.
If xt# is specified then yt# is created empty. If yt# is specified then xt# is created empty.
No checks are made for duplicate or valid names. No initialization is done, assume random. See section on Creating Variables.

Example: Create parametric pair of equations xt1 and yt1, yt1 at size 50.

```

                LD      HL, ytlname
                RST     rMov9ToOP1 ; OP1 = name
;
                LD      HL, 50
                B_CALL  CreatePair ; if returns then variables
                                ; created
                                ; OP1 = yt1, OP4 = xt1, HL = 50

ytlname:       DB      EquObj, tVarEqu, tyt1, 0, 0

```


CreatePict

Category: Memory

Description: Creates a picture variable in RAM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of picture to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. The size of a Pic var is 756 bytes, it does not allocate space for the last row of pixels, that row is never used by the system graph routines.

If you need to save a bitmap of the entire display to a variable then an AppVar should be used. The only drawback to using an AppVar is that the Pic could not be displayed by the user when the app is not executing.

No initialization is done, assume random. See section on Creating Variables.

Example: Create Pic Pic1.

```

                LD        HL,Pic1name
                RST       rMov9ToOP1    ; OP1 = name
;
                B_CALL    CreatePict    ; if returns then variable
                                         ; created

Pic1name:      DB        PictObj,tVarPict,tPic1,0,0

```

CreateProg

Category: Memory

Description: Creates a program variable in RAM.

Inputs:

Registers: HL = size of program to create in bytes

Flags: None

Others: OP1 = name of program to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. See section on Creating Variables.

Example: Create Program DOG, 50 bytes in size.

```

                                LD      HL,DOGname
                                RST      rMov9ToOP1          ; OP1 = name
                                ;
                                LD      HL,50
                                B_CALL   CreateProg          ; if returns then
                                                                ; variable created

DOGname:      DB      ProgObj,'DOG',0

```

CreateProtProg

Category: Memory

Description: Creates a protected program variable in RAM.

Inputs:

Registers: HL = size of program to create in bytes

Flags: None

Others: OP1 = name of program to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization is done, assume random. Users cannot delete or edit protected programs, they can be deleted from an application. See section on Creating Variables.

Example: Create protected Program DOG, 50 bytes in size.

```

                                LD      HL,DOGname
                                RST      rMov9ToOP1      ; OP1 = name
                                ;
                                LD      HL,50
                                B_CALL   CreateProtProg  ; if returns then variable
                                                         ; created
DOGname:      DB      ProtProgObj,'DOG',0

```

CreateReal

Category: Memory

Description: Creates a real variable in RAM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of real to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. This should not be used to create temp storage space, only A-Z * theta. No initialization is done, assume random. See section on Creating Variables.

Example: Create real A.

```

                                LD      HL,Aname
                                RST      rMov9ToOP1      ; OP1 = name
                                ;
                                B_CALL   CreateReal      ; if returns then variable
                                                         ; created

Aname:      DB      RealObj,'A',0,0
```

CreateRList

Category: Memory

Description: Creates a real list variable in RAM.

Inputs:

Registers: HL = number of elements in the list

Flags: None

Others: OP1 = name of list to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization of the elements is done, assume random. See section on Creating Variables.

Example: Create real list CAT with 50 elements.

```

                LD      HL,CATname
                RST      rMov9ToOP1    ; OP1 = name
;
                LD      HL,50
                B_CALL   CreateRList    ; if returns then variable
                                         ; created

CATname:       DB      RListObj,tVarLst,'CAT',0

```

CreateRMat

Category: Memory

Description: Creates a real matrix variable in RAM.

Inputs:

Registers: HL = dimension of matrix, (row,col), 99 is maximum row or column

Flags: None

Others: OP1 = name of matrix to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, dimension

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization of the elements is done, assume random. See section on Creating Variables.

Example: Create matrix [A] with 5 rows and 8 columns.

```

                LD        HL,MatAname
                RST        rMov9ToOP1    ; OP1 = name
;
                LD        HL,5*256+8    ; 5 x 8
                B_CALL    CreateRMat    ; if returns then variable
                                   ; created

MatAname:      DB        MatObj,tVarMat,tMatA,0,0

```

CreateStrng

Category: Memory

Description: Creates a string variable in RAM.

Inputs:

Registers: HL = number bytes in string

Flags: None

Others: OP1 = name of string to create

Outputs:

Registers: HL = pointer to variable's symbol table entry
DE = pointer to variable's data storage, size bytes

Flags: None

Others: OP4 = variable's name

Registers destroyed: OP1 and OP2

Remarks: Memory error if not enough free RAM. No checks are made for duplicate or valid names. No initialization of the string contents is done, assume random. See section on Creating Variables.

Example: Create string Str1 100 bytes in length.

```

                LD      HL,Str1name
                RST      rMov9ToOP1      ; OP1 = name
;
                LD      HL,100           ; size of string
                B_CALL   CreateStrng     ; if returns then variable
                                         ; created

Str1name:      DB      StrngObj,tVarStrng,tStr1,0,0

```

DataSize

Category: Memory

Description: Computes the size, in bytes, of the data portion of a variable in RAM.

Inputs:

Registers: ACC = data type
HL = pointer to first byte of data storage

Flags: None

Others: None

Outputs:

Registers: DE = size of data storage in bytes
HL = intact

Flags: None

Others: None

Registers destroyed: A, BC

Remarks: This routine cannot be used on archived variables or applications.

If the variable's data area has size information, like a list has two-bytes for number of elements, then those bytes are included in the computation.

Example:

```

; Find the size in bytes of the data area for list L1.
L1Name:
        DB          ListObj,tVarLst,tL1,0,0
;
        LD          HL,L1name
        RST         rMov9ToOP1    ; OP1 = L1
;
        B_CALL     FindSym        ; find in symbol table,
                                   ; DE = pointer to data
        AND         1Fh           ; ACC = data type information,
                                   ; real or complex list
        EX         DE,HL          ; HL = pointer to data storage
        B_CALL     DataSize       ; DE = size of data storage
                                   ; If L1 were a real list with 5
                                   ; elements then the size
                                   ; returned would be 47 bytes.

;      5 elements *9 for each = 45
;      2 size bytes           =  2
;                               ---
;                               47

```


DataSizeA

Category: Memory

Description: Computes the size, in bytes, of the data portion of a variable that has two size bytes as part of its data storage.
This application applies to equations, lists, matrices, programs, AppVars.

Inputs:

Registers: ACC = data type
BC = two byte size information: dimension, number of bytes, number of elements

Flags: None

Others: None

Outputs:

Registers: DE = size of data storage in bytes

Flags: None

Others: None

Registers destroyed: All

Remarks: If the variable's data area has size information, like a list has two bytes for number of elements, then those bytes are included in the computation.

Example:

```

; Find the size in bytes of a complex list with 5 elements:
      LF          A,CListObj      ; ACC = data type information,
                                   ; cplx list
      LD          BC,5            ; number elements
;
      B_CALL      DataSizeA      ; DE = size of data storage
;
;
;      5 elements *18 for each = 90
;      2 size bytes           =  2
;                               ----
;                               92

```

DeallocFPS

Category: Memory

Description: Removes space in nine-byte chunks from the Floating Point Stack.

Inputs:

Registers: HL = number of chunks to remove

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: FPS (Floating Point Stack top) decreased by $HL * 9$

**Registers
destroyed:** DE, HL

Remarks: No values are removed from the deallocated space.

Example:

DeallocFPS1

Category: Memory

Description: Removes space in bytes from the Floating Point Stack.

Inputs:

Registers: DE = number of bytes to remove

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: FPS (Floating Point Stack top) decreased by HL

**Registers
destroyed:** HL

Remarks: No values are removed from the deallocated space.

Example:

DelMem

Category: Memory

Description: Deletes RAM from an existing variable. This routine will only delete the RAM. If the variable deleting from has a size field, it is NOT UPDATED. Updating must be done by the application.

Inputs:

Registers: HL = address of first byte to delete
DE = number of bytes to delete

Flags: None

Others: None

Outputs:

Registers: DE = intact
BC = amount deleted
RAM deleted

Flags: None

Others: None

Registers destroyed: All

Remarks: See **InsertMem** routine.

(continued)

DelMem *(continued)*

Example: Delete 10 bytes at the beginning of an AppVar.

```

;
;          LD          HL,AppVarName
;          RST          rMov9ToOP1      ; OP1 = name of AppVar
;          B_CALL      ChkFindSym      ; look up in symTable
;          JR           NC,..Created    ; jump if it exists
;
;          B_JUMP      ErrUndefined    ; error if not there
;
;  DE = pointer to size bytes of AppVar
;
Created:
;          PUSH        DE              ; save pointer to start of
;                                     ; size bytes of data
;
;          INC         DE
;          INC         DE              ; move DE to 1st byte of
;                                     ; AppVar Data
;
;          LD          HL,10           ; number bytes to insert
;
;          EX          DE,HL           ; HL = pointer to start of
;                                     ; delete, DE number bytes
;          B_CALL      DelMem          ; delete the memory
;
;          POP         HL              ; HL = pointer to size bytes
;          PUSH        HL              ; save
;
;          B_CALL      ldHLind         ; HL = size of AppVar,
;                                     ; number bytes
;
;          XOR         A               ; clear CA
;
;          LD          BC,10
;          SBC         HL,BC           ; decr by amount deleted
;
;          EX          DE,HL
;          POP         HL              ; pointer to size bytes
;                                     ; location
;
;          LD          (HL),E
;          INC         HL
;          LD          (HL),D           ; write new size.
;
;
AppVarName: DB      AppVarObj, 'AVAR', 0

```

DelVar

Category: Memory

Description: Deletes a variable stored in RAM.

Inputs: All of the inputs for this routine are the outputs of **FindSym** and **ChkFindSym**. It is common to call one of these routines and then call **DelVar** immediately after.

Registers: HL = pointer to start of symbol table entry of variable
DE = pointer to start of data storage of variable
B = 0 if variable resides in RAM else it is the page in the archive it is stored

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: OP1 – OP6 are preserved.
Variable's symbol entry and data are deleted.
Graph is marked dirty if variable was used during graphing.
All global memory pointers are adjusted.
Error if the variable resides in the archive.

Example:

```

; Delete the variable 'A' if it exists
      LD      HL, AName
      RST     rMov9ToOP1    ; OP1 = variable a
;
      B_CALL  FindSym      ; look up
      JR      C,..deleted  ; jump if variable is not
                           ; created
;
      B_CALL  DelVar
..Deleted:
AName:
      DB      RealObj, 'A', 0, 0

```

DelVarArc

Category: Memory

Description: Deletes a variable from RAM or the archive.

Inputs:

Registers: HL = pointer to symbol table entry of variable to delete
DE = pointer to start of data for variable
B = archived status
0 = RAM otherwise the ROM page in Flash for the variable

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Variable's symbol entry and data deleted if in RAM, otherwise the symbol table entry is only deleted and the variable data is marked for deletion on the next garbage collection.

Graph is marked dirty if variable was used during graphing.

All global memory pointers are adjusted.

Registers destroyed: All

Remarks: See **DelVar** and **DelVarNoArc** routines.

Example:

DelVarNoArc

Category: Memory

Description: Deletes variable from RAM.
No archive checking performed.

Inputs:

Registers: (HL) = sign digit of symbol table entry
DE = data pointer to data

Flags: None

Others: None

Outputs:

Registers: None

Flags: Regraph flag set if varGraphRef flag of symbol was set.

Others: None

Registers destroyed: All

Remarks: See **DelVar** for more information.
This routine should only be called if you are sure that your variable will never be archived. Generally, it is better to use the **DelVarArc** or **DelVar** routines.
Variable's symbol entry and data are deleted.
Graph is marked dirty if variable was used during graphing.
All global memory pointers are adjusted.
Error if the variable resides in the archive.

Example:

```

; Delete the variable 'A' if it exists:
        LD          HL,Aname
        RST         rMov9ToOP1      ; OP1 = variable a
;
        B_CALL     FindSym          ; look up
        JR         C,...deleted     ; jump if variable is not
                                   ; created
;
        B_CALL     DelVarNoArc
Deleted:
Aname:
        DB         RealObj,'A',0,0

```


EditProg

Category: Memory

Description: This routine will insert all of free RAM into a Program, Equation, or AppVar. The intent is for the variable to be able to be edited without having to continuously allocate and deallocate memory. Once the edit is completed, a call to **CloseProg** is made to return what is not used back to free RAM.

Inputs:

Registers: DE = pointer to start of variables data storage area

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Each of following are two-bytes:

(iMathPtr1) = pointer to the start of the variables data storage area.
THIS MUST STAY INTACT WHILE THE EDIT IS IN SESSION.

(iMathPtr2) = pointer to the byte following the variable data. This is the next location the data area can grow into.

(iMathPtr3) = pointer to the byte AFTER the last byte of free RAM inserted. The data being input cannot be written into this RAM location.

(iMathPtr4) = size of RAM block moved to allow the RAM to be inserted.
DO NOT CHANGE THIS VALUE.

Registers destroyed: All

Remarks: The application can must change the pointer value in (iMathPtr2) as the variables data size grows or shrinks. This value is needed by the close routine.

No memory allocation/deallocation can be done in this state.

Contents of variables may be copied or changed, but not their sizes.

The Floating Point Stack may be copied to/from, but not grown or shrunk.

The hardware stack may change, calls, RET, push, and pop.

Example:

EnoughMem

Category: Memory

Description: Checks if an imputed amount of RAM is available. This routine will also attempt to free RAM that is taken by temporary variables that have been marked dirty but not yet deleted.

Inputs:

Registers: HL = amount of RAM to check for being available

Flags: None

Others: None

Outputs:

Registers: DE = amount of RAM to check for being available

Flags: CA = one (set) if there is insufficient RAM available

Others: None

Registers destroyed: All

RAM used: None

Remarks: No error is generated.
See **MemChk**.

Example:

Exch9

Category: Memory

Description: Exchanges (swaps) two nine-byte blocks of memory.

Inputs:

Registers: DE = address of start of one nine-byte block
HL = address of start of second nine-byte block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Nine bytes originally at DE are now at original HL
Nine bytes originally at HL are now at original DE

Registers destroyed: A, BC, DE, HL

Remarks: None

Example:

ExLp

Category: Memory

Description: Exchanges blocks of memory of up to 256 bytes.

Inputs:

Registers: B = number of bytes; 0 = 256
DE = address of start of one nine-byte block
HL = address of start of second nine-byte block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Block originally at DE is now at original HL
Block originally at HL is now at original DE

Registers destroyed: A, BC, DE, HL

Remarks: None

Example:

FindAlphaDn

Category: Memory

Description: This is used to search the symbol table, for all of the variables of a certain type, alphabetically in descending order.

Each call to this routine returns the variable name preceding the one input in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = variable name to find the previous before, usually output from the last call to this routine.

(OP1) must have the type of variable searching for set.

The name input in order to have the very last name for a certain type varies by the variable's type:

Real, Complex, Programs, AppVars, Group Vars:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
Object Type	0FEh	?	?	?	?	?	?	?

All other types:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
Object Type	variable token	0FEh	?	?	?	?	?	?

Outputs:

Registers: If a previous variable name is found then:
HL = pointer to the symbol table entry of the variable found

Flags: CA = 0 if a previous variable name was found
= 1 if no previous variable name exists

Others: If a previous variable name is found then:
OP1 and OP3 = the variable name found

Otherwise :
OP1 = variable name input

Registers destroyed: All

(continued)

FindAlphaDn *(continued)*

- RAM used:** OP2, OP3
upDownPtr — two byte pointer
- Remarks:** ProgObj, ProtProgObj, and TempProgObj are grouped together.
ListObj and CListObj are grouped together.
NewEquObj and EquObj are grouped together.
See **FindAlphaUp**, **SrchVLstUp**, **SrchVLstDn**.
- Example:** Find all of the programs that are currently created, search alphabetically in descending order.

```

FindPrograms:
    B_CALL    ZeroOP1
    LD        A,ProgObj
    LD        (OP1),A        ; looking for a list
    LD        A,0FEh        ; name = FEh, so the last
                            ; program alphabetically is
                            ; found
    LD        (OP1+1),A
FindLoop:
    B_CALL    FindAlphaDn    ; see if find another program
                            ; name
    RET       C              ; return if no more program
                            ; names not found yet
;
;  OP1 = next list name
;
    JR        FindLoop      ; find previous using one just
                            ; found as input

```

FindAlphaUp

Category: Memory

Description: This is used to search the symbol table, for all of the variables of a certain type, alphabetically in ascending order.

Each call to this routine returns the next variable name following the one input in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = variable name to find the next after, usually output from the last call to this routine.

(OP1) must have the type of variable searching for set.

The name input in order to have the very first name for a certain type varies by the variable's type:

Real, Complex, Programs, AppVars, Group Vars:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
Object Type	00	?	?	?	?	?	?	?

All other types:

OP1	+1	+2	+3	+4	+5	+6	+7	+8
Object Type	variable token	0FFh	?	?	?	?	?	?

Outputs:

Registers: If a next variable name is found then:
HL = pointer to the symbol table entry of the variable found

Flags: CA = 0 if a next variable name was found
= 1 if no next variable name exists

Others: If a next variable name is found then:
OP1 and OP3 = the variable name found

Otherwise:
OP1 = variable name input

Registers destroyed: All

(continued)

FindAlphaUp *(continued)*

RAM used: OP2, OP3
upDownPtr — two byte pointer

Remarks: ProgObj, ProtProgObj and TempProgObj are grouped together.
ListObj and CListObj are grouped together.
NewEquObj and EquObj are grouped together.
See **FindAlphaDn**, **SrchVLstUp**, **SrchVLstDn**.

Example: Find all of the lists that are currently created, search alphabetically in ascending order.

```
FindLists:
    B_CALL    ZeroOP1
    LD        A,ListObj
    LD        (OP1),A        ; looking for a list
    LD        A,tVarLst      ; list designator token
    LD        (OP1+1),A      ;
    LD        A,0FFh         ; set name to FFh, so that the
                             ; first list alphabetically is
                             ; found
    LD        (OP1+1),A

FindLoop:
    B_CALL    FindAlphaUp    ; see if find another list name
    RET       C              ; return if no more list names
                             ; not found yet
;
;   OP1 = next list name
;
    JR        FindLoop      ; find next using one just found
                             ; as input
```


FindApp

Category: Memory

Description: Searches for an application that may be stored in Flash ROM.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of application to search for

Outputs:

Registers: A = ROM page application starts on if found

Flags: CA = 0 if application exists
CA = 1 if application does not exist

Others: None

**Registers
destroyed:** All

RAM used: appSearchPage (two-bytes)

Remarks:

Example:

FindAppNumPages

Category: Memory

Description: Finds the number of 16K pages an application uses in archive memory

Inputs:

Registers: A = first page of application

Flags: None

Others: None

Outputs:

Registers: A = first page of application
C = number of 16K pages the application uses

Flags: None

Others: None

Registers destroyed: BC, DE

Remarks: If an application was not found on the given page, C will equal 0.

Example:

```

IN          A,(memPageAPort)    ; gets the current memory
                                   ; page for app. Make sure
                                   ; this is on the first page
                                   ; of a multi-page
                                   ; application.
B_CALL      FindAppNumPages      ; finds the total number of
                                   ; pages the application
                                   ; uses in archive memory.
LD          A,C                  ; A = number of pages

```

For multi-page apps, create a routine that will reside on the first page of the application that will return the memory page.
i.e., Get_First_Page:

```

IN          A,(memPageAPort)    ; get the memory page of
                                   ; the first application
                                   ; page.
RET

```

FindAppDn

Category: Memory

Description: Searches for the next application in Flash ROM whose name is alphabetically less than the name in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = the name to find an application less than
If searching for all of the application names in descending alphabetical order then this name is either the previous one found or the initial name used to start the search.

To initialize the search to find the last application name alphabetically, set (OP1 + 1) = 0FEh.

Outputs:

Registers: None

Flags: CA = 1 if no application with a lesser name exists. The previous found is the first alphabetically.

CA = 0 if an application less than OP1 was found.

Others: OP1 = application name found if one exists.

Registers destroyed: All

RAM used: OP2, OP3, appSearchPage (two-bytes)

Remarks: No information about what ROM page the application resides on is returned. To get this information a **FindApp** needs to be done.

Example: A loop that finds all of the application names in descending order.

```

        B_CALL    ZerroOP1      ; initialize OP1 for 1st search
        LD        A,0FEh
        LD        OP1+1),A      ; set OP1 = name > any valid
                                   ; name

    ..loop:
        B_CALL    FindAppDn     ; look for next lesser
                                   ; alphabetically
        JR        NC,..loop     ; jump if found one, go look for
                                   ; next one

    ;
        RET

```

FindAppUp

Category: Memory

Description: Searches for the next application in Flash ROM whose name is alphabetically greater than the name in OP1.

Inputs:

Registers: None

Flags: None

Others: OP1 = the name to find an application greater than

If searching for all of the application names in ascending alphabetical order then this name is either the previous one found or the initial name used to start the search.

To initialize the search set OP1 = all 0's with a system call to **ZerroOP1**.

Outputs:

Registers: None

Flags: CA = 1 if no application with a greater name exists. The previous found is the last alphabetically.

CA = 0 if an application greater than OP1 was found

Others: OP1 = application name found if one exists

Registers destroyed: All

RAM used: OP2, OP3, appSearchPage (two-bytes)

Remarks: No information about what ROM page the application resides on is returned. To get this information a **FindApp** needs to be done.

Example: A loop that finds all of the application names in ascending order.

```

                                B_CALL      ZerroOP1      ; initialize OP1 for 1st search
..loop:
                                B_CALL      FindAppUp      ; look for next higher
                                                ; alphabetically
                                JR           NC,..loop      ; jump if found one, go look for
                                                ; next one
                                ;
                                RET

```

FindSym

Category: Memory

Description: Searches the symbol table structure for a variable.

This search routine is used to find variables that are not programs, AppVar, or Groups. See **ChkFindSym**.

This is used to determine if a variable is created and also to return pointers to both its symbol table entry and data storage area.

This will also indicate whether or not the variable is located in RAM or has been archived in Flash ROM.

Inputs:

Registers: (OP1 + 1) to (OP1 + 6) = variable name
See documentation on variable naming conventions.

Flags: None

Others: None

Outputs:

Registers: CA flag = 1 if symbol was not found
= 0 if symbol was found

If symbol is found, additional outputs are:

ACC lower 5 bits = data type

ACC upper 3 bits = system flags about variable. Mask via “AND” with a value of 1Fh to obtain data type only.

B = 0 if variable is located in RAM else variable is archived

B = ROM page located on

If variable is archived then its data cannot be accessed directly, it must be unarchived first.

HL = pointer to the start of the variables symbol table entry

DE = pointer to the start of the variables data area if in RAM

Flags: None

Others: OP1 = variable name

Registers destroyed: All

Remarks: This will not find system variables that are preallocated in system RAM such as Xmin, Xmax etc. Use **RclSysTok** to retrieve their values.

This will not find applications.

(continued)

FindSym *(continued)*

Example: ; Look for List L1 in the symbol table.
 ; If it exists and is archived then unarchive it and relook it up.
 ; If it does not exist create it as a real list of 10 elements.
 Relook:

```

                LD      HL,L1name
                B_CALL  Mov9ToOP1      ; OP1 = variable name
                B_CALL  FindSym        ; look up
                JR      NC,varCreated  ; jump if it exists
;
                LD      HL,10          ; size to create at data
                B_CALL  CreatorList
                PUSH    HL
                PUSH    DE            ; save during move
                B_CALL  OP4ToOP1      ; OP1 = name
                POP     DE            ; restore
                POP     HL
                JR      done
;
VarCreated:
                LD      A,B            ; check for archived
                OR      A              ; in RAM ?
                JR      Z,done         ; yes
;
                B_CALL  Arc_Unarc     ; unarchive if enough RAM
                JR      Relook        ; look up pointers again in
                                     ; RAM now

DONE:
                RET
;
L1name:
                DB      ListObj,tVarLst,tL1,0

```

FixTempCnt

Category: Memory

Description: Resets pTempCnt back to a input value, and delete all temps with name counters greater than or equal to that value.

Inputs:

Registers: DE = value to pTempCnt to

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: (pTempCnt) = DE

All temps created with pTempCnt >= input DE are deleted. For example, if input DE = 5 then temps with counter value 5 or greater \$0500 will be deleted. \$0600...

Registers destroyed: All

RAM used: pTempCnt

Remarks: See the Temporary Variables section in Chapter 2. Also, see the **CleanAll** routine.

Example:

FlashToRam

Category: Memory

Description: Copies bytes from Flash to RAM.

Inputs:

Registers: A = page of source (Flash)
HL = offset of source (Flash)
DE = RAM location of destination
BC = number of bytes to copy

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks: Certain pages in Flash cannot be copied. This routine will wrap to the next page if the offset = 8000h. A will be incremented to the next page, and HL will be reset to 4000h, and the copying will go on.

Example:

InsertMem

Category: Memory

Description: Inserts RAM into an existing variable.

This routine will only insert the RAM — it stays uninitialized and if the variable inserting into has a size field, it is NOT UPDATED. Updating must be done by the application.

A check for enough free RAM must be done by the application. This routine assumes the RAM is available.

Inputs:

Registers: HL = number of bytes of RAM to insert, no check is made for enough free RAM.

DE = point of insertion address — this cannot be the first byte of the variable's data — if it is, its symbol table entry will not have the correct pointer to the data.

Flags: None

Others: None

Outputs:

Registers: DE = intact, the point of insertion address

Flags: None

Others: RAM inserted into variable.

Registers destroyed: All

Remarks: See **DelMem**.

(continued)

InsertMem *(continued)*

Example: Insert 10 bytes at the beginning of an Application Variable.

```

        LD      HL,10          ; number bytes to insert
        B_CALL  ErrNotEnoughMem ; error if 10 bytes are not
                                ; free
;
        LD      HL,AppVarName
        RST     rMov9ToOPl      ; OPl = name of AppVar
        B_CALL  ChkFindSym      ; look up in symTable
        JR      NC,..Created     ; jump if it exists
;
        B_JUMP  ErrUndefined     ; error if not there
;
; DE = pointer to size bytes of AppVar
;
..Created:
        PUSH    DE              ; save pointer to start of
                                ; size bytes of data
;
        INC     DE
        INC     DE              ; move DE past size bytes
;
        LD      HL,10          ; number bytes to insert
        B_CALL  InsertMem       ; insert the memory
;
        POP     HL              ; HL = pointer to size bytes
        PUSH    HL              ; save
;
        B_CALL  ldHLlnd         ; HL = size of AppVar,
                                ; number bytes
        LD      BC,10
        ADD     HL,BC           ; increase by 10, amount
                                ; inserted
;
        EX      DE,HL
        POP     HL              ; pointer to size bytes
                                ; location
;
        LD      (HL),E
        INC     HL
        LD      (HL),D          ; write new size.
;
;
AppVarName: DB      AppVarObj,'AVAR',0

```

LoadCIndPaged

Category: Memory

Description: Reads a byte of data from any ROM page. Main use is for applications to read data from variables that are archived, without having to unarchive them to RAM first.

Inputs:

Registers: B = ROM page to read byte from
HL = address of byte on the ROM page,
(4000h–7FFFh)

Flags: None

Others: None

Outputs:

Registers: C = byte of data from input ROM page and Offset

Flags: None

Others: None

Registers destroyed: C

Remarks: B, HL are not changed. See the **LoadDEIndPaged** routine. Also, see the Accessing Archived Variables Without Unarchiving section in Chapter 2.

Example: Read the byte of data from ROM page 0Ch, address 4006h.

```

LD      B,0ch      ; ROM page
LD      HL,4006h    ; offset
;
B_CALL  LoadCIndPaged ; C = byte
RET
```

LoadDEIndPaged

Category: Memory

Description: Read two consecutive bytes of data from any ROM page. The main use of this routine is for applications to read data from variables that are archived, without having to unarchive them to RAM first.

Inputs:

Registers: B = ROM page of first of two bytes to read
HL = address of byte on the ROM page,
(4000h–7FFFh)

Flags: None

Others: None

Outputs:

Registers: E = first byte read
D = second byte read

Flags: None

Others: None

Registers destroyed: DE, C

Remarks: B, HL are set to the address of the second byte read. If the second byte of data is not on the same ROM page as the first, the switch to the next ROM page is handled. See the **LoadCIndPaged** routine. Also, see the Accessing Archived Variables Without Unarchiving section in Chapter 2.

Example: Read two bytes of data from ROM page 0Ch, address 4006h.

```

LD      B,0ch          ; ROM page
LD      HL,4006h        ; offset
;
B_CALL  LoadDEIndPaged  ; D = byte @ (4007h),
                        ; E = byte @(4006h)
RET
```

MemChk

Category: Memory

Description: Returns the amount of RAM currently available.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: HL = amount of RAM available, in bytes

Flags: None

Others: None

Registers destroyed: BC, HL

Remarks: If a system editor is open, this will always return 0 bytes available. System edits use all of free RAM during the edit.

The amount returned may be inaccurate if there are any temporary variables that are marked as dirty but not yet deleted. There are two ways/options to solve this:

- The routine **CleanAll** can be used to remove all temporary variables. This is fine as long as an application is not using temporary variables. Temporary variables are returned by the parser if the result is not RealObj or CplxObj, make sure that none are still in use.
- Use the routine **EnoughMem** instead, it will delete only temps that are marked dirty.

Example: Delete all temporary variables and then check if there is at least 100 bytes available.

```

B_CALL    CleanAll      ; delete all temporary
                        ; variables
B_CALL    MemChk        ; HL = amount of mem free
;
LD         DE,100
OR         A             ; CA = 0
SBC       HL,DE         ; if CA = 1 then less than 100
                        ; bytes are available
JR        C,Not_100     ; jump if < 100

```

PagedGet

Category: Memory

Description: Used for reading data from the archive with the Caching technique. This routine will return the next byte and also refill the cache when it is emptied.

A call to the **SetupPagedPtr** routine must be done once before using this routine to retrieve data from the archive.

Inputs:

Registers: None

Flags: None

Others: These are initially set by the **SetupPagedPtr** routine and are updated each time a call is made to the **PagedGet** routine. Applications do not need to modify these RAM locations.

(pagedPN) = current Flash page.

(pagedGetPtr) = current Flash address.

Outputs:

Registers: ACC = byte read

Flags: None

Others: None

Registers destroyed: ACC

Remarks: Crossing ROM page boundaries is handled. See the **SetupPagedPtr**, **LoadCIndPaged**, and **LoadDEIndPaged** routines. Also, see the Accessing Archived Variables Without Unarchiving section in Chapter 2.

Example:

```

LD      B,PageToGet
LD      DE,AddressToGet
B_CALL  SetupPagedPtr      ; setup paged get
;

B_CALL  PagedGet            ; ACC = byte from archive
LD      E,A                ; E = byte
;

B_CALL  PagedGet            ;
;

LD      D,A                ; DE = 2 bytes read from
                           ; archive

```

RcIGDB2

Category: Memory

Description: Recalls graph database.

Inputs:

Registers: A = tVarGDB

Flags: None

Others: OP1 = data base name
(chkDelPtr1) contains data pointer

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:**

Remarks: Acts exactly as the user controlled RcIGDB command: Restores graph mode stored in the GDB and replaces all equation variables with those stored in the GDB and all range values with those stored in the GDB.

Example:

```

                                ; Restore GDB2 if it exists:
B_CALL    ZeroOP1              ; zero out OP1
LD         HL,GDB2Name         ; name -> OP1
LD         DE,OP1
LD         BC,03
LDIR
B_CALL    FindSym              ; find & point to symbol.
RET       C                    ; abort if does not exist.
B_CALL    RcIGDB2              ; restore graph data base.
....
GDB2Name:
DB         GDBObj,tVarGDB,tGDB2  (008h,061h,001h)
```

RcIN

Category: Memory

Description: Recalls the contents of variable N if it exists.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: System error if N does not exist.

OP1 = contents of N if RealObj

OP1/OP2 = contents of N if CplxObj

**Registers
destroyed:** All

RAM used: OP1 – OP2

Remarks:

Example:

RclVarSym

Category: Memory

Description: Recalls the contents of variable A – Z or THETA.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of variable to recall

Outputs:

Registers: None

Flags: None

Others: System error if variable does not exist.

If a variable other than A – Z or THETA, then nothing is done.

OP1 = contents of variable if RealObj

OP1/OP2 = contents of variable if CplxObj

**Registers
destroyed:** All

RAM used: OP1 – OP2

Remarks:

Example:

RcIX

Category: Memory

Description: Recalls the contents of variable X if it exists.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: System error if X does not exist.

OP1 = contents of X if RealObj

OP1/OP2 = contents of X if CplxObj

**Registers
destroyed:** All

RAM used: OP1 – OP2

Remarks:

Example:

RcIY

Category: Memory

Description: Recalls the contents of variable Y if it exists.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: System error if Y does not exist.

OP1 = contents of Y if RealObj

OP1/OP2 = contents of Y if CplxObj

**Registers
destroyed:** All

RAM used: OP1 – OP2

Remarks:

Example:

RedimMat

Category: Memory

Description: Redimensions an existing matrix.

Inputs:

Registers: HL = new dimension of matrix wanted

Flags: None

Others: OP1 = name of matrix

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All, iMathPtr1, insDelPtr

RAM used: OP1, OP3

Remarks: If not enough room, then a memory error will occur.

The space is allocated/deallocated. The pointers are adjusted accordingly. All the new elements are set to 0. The old values of the elements that are not removed are kept. A Matrix cannot be modified if it is archived.

Example:

```

B_CALL    ZeroOP1      ; zero out OP1
LD        HL,MatrixA
LD        DE,OP1
LD        BC,3
LDIR                      ; load matrix name into OP1
B_CALL    ChkFindSym   ; find matrix variable name
JR        C,..skip     ; if not found, skip over work
LD        A, B         ;
OR        A            ; see if archived
JR        NZ,..skip    ; skip if variable archived
LD        HL,0505h
                        ; redimensionalize matrix to 5x5
B_CALL    RedimMat
..skip:
RET
MatrixA:  DB          MatObj,tVarMat,MataA

```

SetupPagedPtr

Category: Memory

Description: Initializes the process of reading data from the archive using the caching method.

The **PagedGet** routine is used to read data from the archive after this initialization routine is called.

Inputs:

Registers: Start address of the first byte of data to be read
 B = ROM page of the first byte
 DE = address of first byte, on the ROM page
 (4000h–7FFFh)

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: These outputs are inputs to the **PagedGet** routine. An application should not change these values directly.

pagedCount = 0 on first call
 pagedPN = current Flash page
 pagedGetPtr = current Flash address

Registers destroyed: None

Remarks: See the **PagedGet** routine. Also, see the Accessing Archived Variables Without Unarchiving section in Chapter 2.

Example:

```

LD      B,PageToGet
LD      DE,AddressToGet
B_CALL  SetupPagedPtr    ; setup paged get
;
B_CALL  PagedGet          ; ACC = byte from archive
LD      E,A              ; E = byte
;
B_CALL  PagedGet          ;
;
LD      D,A              ; DE = 2 bytes read from
                        ; archive
```

SrchVLstDn, SrchVLstUp

Category: Memory

Description: Searches the I/O var list in the backward/forward direction, next lower alphabetically, and by type in the following order:

PROGRAM,ProtPtrg	05h,06h
DATABASE	08h
PICTURE	07h
LIST,Clist	01h,0Dh
MATRIX	02h
YVARS	03h
AppVars	15h
Group	17h
WINDOW	0Bh
ZSTO	0Ch
TABLE RANGE	0Dh
REAL	00h
Cplx	0Ch
String	04h
Apps	14h

Inputs:

Registers: OP1 = last name and type found in variable format

Flags: inGroup, (IY + groupFlags) should be reset
inDelete, (IY + ioDelFlag) should be reset

Others: (varClass) should be set to 9 to search through the entire list.

Outputs:

Registers: HL = pointer to symbol table entry if found

Flags: CA = 0 if found
CA = 1 if did not find anything

Others: OP1 = var format of next variable if found

Registers destroyed: All registers

Remarks: This calls **FindAlphaUp/FindAlphaDn** to find variables within each variable type.

Example:

StMatEI

Category: Memory

Description: Stores an element to a matrix. Convert matrix or element to complex if necessary.

Inputs:

Registers: BC = column number
DE = row number

Flags: None

Others: OP1 = existing matrix variable name
FPST = value to store (real or complex)

Outputs:

Registers: None

Flags: graphDraw set if graph reference flag was on.

Others: OP1 = value originally on FP stack
FPST was popped, value no longer on FPST
Value was stored to the matrix

Registers destroyed: All

Remarks:

Example:

StoAns

Category: Memory

Description: Stores OP1 to Ans variable.

Inputs:

Registers: None

Flags: None

Others: OP1[,OP2] = value if real [complex]
Otherwise OP1 = name of variable that contains the data to store into Ans

Outputs:

Registers: None

Flags: None

Others: Data stored if possible
OP1[,OP2] = original contents if real[complex]
else OP1 = Ans variable name

Registers destroyed: FPS, OP1, OP2, OP4

Remarks: If input was a parser temporary (\$P) variable, it is marked dirty (to be deleted by memory management).
A memory error occurs if there is not enough room to store the value.
Ans is the same system variable that is found by pressing [2nd] [Ans] on the calculator keyboard.
Use **RclAns** to recall the contents of Ans.

Example:

StoGDB2

Category: Memory

Description: Stores the current graph mode settings and equations into a system graph database variable.

Inputs:

Registers: None

Flags: None

Others: OP1 = graph database name to store to

Outputs:

Registers: None

Flags: None

Others: GDB created or modified

**Registers
destroyed:** All

RAM used: (ioData) buffer used to store name temporarily.

Remarks: This creates the graph database if it did not exist already. If it did exist, it is resized to fit the size of the variables to be stored.

Example:

StoN

Category: Memory

Description: Stores OP1 to sequence variable n.

Inputs:

Registers: None

Flags: None

Others: OP1 = a real number, positive integer

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

**Registers
destroyed:** All

RAM used: OP1, OP2, OP4

Remarks: This does not store to variable N.
This will store to the system variable n used in Sequence graphing.
To recall, see **RcIN**.

Example:

StoOther

Category: Memory

Description: General purpose routine that stores data to user created variables that are not of type ProgObj, GDBObj, GroupObj, AppObj or PictObj.

Also, this routine should not be used to store to system variables such as Xmin.

Inputs:

Registers: None

Flags: None

Others: OP1 = name and type of variable to store to.
(OP1) = data type, followed by the name.
FPST = data to store if not storing to CplxObj
FPS1/FPS2 = data to store if storing to CplxObj

If the variable storing to is RealObj or CplxObj, then the data storing CANNOT be another variable. The FPS must contain the literal data stored.

If the variable storing to is not RealObj or CplxObj, then the data storing MUST be another variable. This variable can either be user created or a temporary variable returned by the parser after executing an expression.

If the variable storing to is already created, then it must reside in RAM and not the archive.

Outputs:

Registers: None

Flags: Both the graph and the table can be marked dirty if the variable stored to was used in a graph equation.

Others: Error if the data is not the correct type to be stored to the variable — for example, store list data to a matrix.

Error if the variable storing to is archived.

Error if not enough memory.

If no errors:

If the variable storing to was not created on input, this routine will create it.

Data stored to the variable.

OP1/OP2 = data that was stored.

The data is removed from the FPS.

Registers destroyed: All

(continued)

StoOther *(continued)*

Remarks: See the **StoSysTok** routine. See Chapter 2 for Error Handlers and Floating Point Stack.

Example: Store list L1 to list L3.

```

                LD      HL,L1name
                B_CALL  Mov9ToOP1      ; OP1 = L1 name
                B_CALL  PushRealO1     ; FPST = L1 name
            ;
                LD      A,tL3          ; token for L3
                LD      (OP1+2),A      ; change OP1 to L3 name
            ;
                B_CALL  StoOther        ; store L1 -> L3
                RET
            ;
L1name:
                DB      ListObj,tVarLst,tL1,0

```

StoR

Category: Memory

Description: Stores OP1[,OP2] -> user variable R.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

**Registers
destroyed:** All

RAM used: OP1, OP2, OP4

Remarks: Note that there is not a RclR routine, but one can be made by:

```
B_CALL  RName      ; set OP1 to R name
B_CALL  RclVarSym   ; do recall
```

Example: ; This sets R to 1:

```
B_CALL  OP1Set1
B_CALL  StoR        ; INIT R = 1
RET
```

StoSysTok

Category: Memory

Description: Stores a value in OP1 to system variable specified by token number in the accumulator.

Inputs:

Registers: A = system variable token number
OP1 = real number to save

Flags: None

Others: None

Outputs:

Registers: OP1 = contents of system variable

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example:

```
                                ; Store -3 into Xmin
B_CALL      OP1Set3            ; register OP1 = floating point 3
B_CALL      InvOP1S           ; negate FP number in OP1,
                                ; OP1 = -3
LD          A,XMINT           ; ACC = Xmin variable token value
B_CALL      StoSysTok         ; store OP1 to Xmin,
```

StoT

Category: Memory

Description: Stores OP1[,OP2] to user variable T.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

Registers destroyed: All

RAM used: OP1, OP2, OP4

Remarks: Note that there is not a RclT routine, but one can be made by:

```
B_CALL    TName      ; set OP1 to T name
B_CALL    RclVarSym   ; do recall
```

Example: ;

```
; This sets T to 0. :B_CALL
; OP1Set0
B_CALL    StoT        ; INIT T = 0
RET
```

StoTheta

Category: Memory

Description: Stores OP1[,OP2] to user variable Theta.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

**Registers
destroyed:** All

RAM used: OP1, OP2, OP4

Remarks: Note that there is not a RclTheta routine, but one can be made by:

```
B_CALL    ThetaName    ; set OP1 to Theta name
B_CALL    RclVarSym    ; do recall
```

Example:

```
;
; This sets Theta to 2...
B_CALL    OP1Set2
B_CALL    StoTheta     ; INIT Theta = 2
RET
```


StoX

Category: Memory

Description: Stores OP1[,OP2] to user variable X.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

Registers destroyed: All

RAM used: OP1, OP2, OP4

Remarks: See **RclX** to recall contents of X.

Example: ;

```

                                ; This sets X to 2:
B_CALL      OP1Set2
B_CALL      StoX              ; INIT X = 2
RET
```

StoY

Category: Memory

Description: Stores OP1[,OP2] to user variable Y.

Inputs:

Registers: None

Flags: None

Others: OP1 = real value to store
or
OP1/OP2 = complex value to store

Outputs:

Registers: None

Flags: None

Others: Sets chkDelPtr3 = system table pointer
Sets chkDelPtr1 = data pointer

**Registers
destroyed:** All

RAM used: OP1, OP2, OP4

Remarks: See **RclY** to recall contents of Y.

Example: ;

```
                                ; This sets Y to 2:
B_CALL      OP1Set2
B_CALL      StoY              ; INIT Y = 2
RET
```

A

System Routines — Parser

BinOPExec	559
FiveExec.....	561
FourExec	563
ParseInp	565
RclSysTok	567
ThreeExec	568
UnOPExec.....	570

BinOPExec

Category: Parser

Description: Executes functions that have two arguments as inputs.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = second argument
FPST = first argument (Floating Point Stack Top), see example

Outputs:

Registers: None

Flags: None

Others: OP1 = result

**Registers
destroyed:** All

Remarks: Checks for valid argument types are done.
The values pushed onto the FPS are removed.
This entry point should only be used if direct access to a particular function is not available.

It can also be used in cases of mixed argument types. Like the example below where a real is added to a list.
Valid arguments can be obtained from the *TI-83 Plus Guidebook*.

(continued)

BinOPExec *(continued)*

Example:

```
.5 + L1
      LD      HL,Point5
      RST     rMov9ToOP1      ; OP1 = .5
      B_CALL  PushOP1        ; OP1 -> FPST, or OP1/OP2 is
                               ; complex number
;
      LD      HL,L1name
      RST     rMov9ToOP1      ; OP1 = L1 name
;
      LD      A,OPAdd         ; function is addition
      B_CALL  BinOPExec       ; OP1 = result of .5 + L1
L1name: DB      RListobj,tVarLst,tL1,0,0
```

BinOPExec equates and functions

<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>
OPBal	bal(OPSum	sum(OPProd	prod(
OPBinCdf	binomcdf(OPBinPdf	binompdf(OPIrr	irr(
OPFinNom	>Nom(OPFinEff	>Eff	OPFinDbd	dbd(
OPRandNrm	randNorm(OPstDev	stdDev(OPVariance	variance(
OPPrn	îPrn(OPIntr	îInt(OPRandBin	randBin(
OPNormalPdf	normalpdf(OPINormal	invNorm(OPNormal	normalcdf(
OPPoiPdf	poissonpdf(OPPoiCdf	poissoncdf(OPGeoCdf	geometcdf(
OPGeOPdf	geometpdf(OPChiPdf	xýpdf(OPTpdf	tpdf(
OPAdd	+	OPSub	-	OPMult	*
OPDiv	ö	OPPower	^	OPXroot	xûy
OPEq	=	OPRound2	round(OPConst	Fill(
OPAUG	augment(OPMax	max(OPMin	min(
OPLCm	lcm(OPGcd	gcd(tEvalF	u(beg,end
tMedian	median(tMean	mean(OPRandInt	randInt(
OPAnd	and	OPOr	or	OPXor	xor
OPNcr	nCr	OPNpr	nPr	OPLt	<
OPLe	ó	OPGt	>	OPGe	ö
OPRand1	randM(OPInstr	inString(OPPxtst	Pxl-Test(
OPRtOPr	R>Pr(OPRtOPo	R>Pé(OPPtorx	P>Rx(
OPPtoRy	P>Ry(

Note: For tEvalF there are really three inputs but execution still goes through the entry point for two arguments. The Equation name needs to be pushed onto the FPS first, then the second argument and the third in OP1. This is only valid in Sequential graph mode.

The second argument is the start value.

The third argument is the end value.

A list of results is returned.

FiveExec

Category: Parser

Description: Executes functions that have five arguments as input.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = fifth argument
FPST = fourth argument (pushed onto FPS fourth)
FPS1 = third argument (pushed onto FPS third)
FPS2 = second argument (pushed onto FPS second)
FPS3 = first argument (pushed onto FPS first)

Outputs:

Registers: None

Flags: None

Others: OP1 = result

**Registers
destroyed:** All

Remarks: Checks for valid argument types are done.
The values pushed onto the FPS are removed.

This entry point should only be used if direct access to a particular function is not available.
Valid arguments can be gotten from the *TI-83 Plus Guidebook*.

(continued)

FiveExec *(continued)*

Example:

```

                                ; fin(Y1, X, 2, 4, .5);
LD      HL,Y1name
RST     rMov9ToOP1      ; OP1 = Y1 name
B_CALL  PushOP1         ; save to FPST;
B_CALL  XName           ; OP1 = X var name
B_CALL  PushOP1         ; push onto FPST, Y1 -> FPS1;
B_CALL  OP1Set2         ; OP1 = 2
B_CALL  PushOP1         ; push onto FPST, Y1 -> FPS2,
                                ; X -> FPS1;
B_CALL  OP1Set4         ; OP1 = 4
B_CALL  PushOP1         ; ->FPST, Y1->FPS3, X->FPS2,
                                ; 2->FPS1, 4->FPST;

LD      HL,point5
RST     rMov9ToOP1      ; OP1 = .5;
LD      A,OPFmin 1      ; function is fMin(
B_CALL  FiveExec        ; OP1 = result
Y1Name: DB      EquObj,tVarEqu,tY1,0,0
Point5: DB      0,80h,50h,0,0,0,0,0,0

```

FiveExec equates and functions

<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>
OPSeq	seq(OPQuad	fnInt(OPFmin	fmin(
OPFmax	fMax(

FourExec

Category: Parser

Description: Executes functions that have four arguments as input.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = fourth argument
FPST = third argument (pushed onto FPS third)
FPS1 = second argument (pushed onto FPS second)
FPS2 = first argument (pushed onto FPS first)

Outputs:

Registers: None

Flags: None

Others: OP1 = result

**Registers
destroyed:** All

Remarks: Checks for valid argument types are done.
The values pushed onto the FPS are removed.
This entry point should only be used if direct access to a particular function is not available.
Valid arguments can be obtained from the *TI-83 Plus Guidebook*.

(continued)

FourExec *(continued)*

Example:

```

                                ; nDeriv(Y1, X, 2, .5);
LD      HL,Y1Name
RST     rMov9ToOP1      ; OP1 = Y1 name
B_CALL  PushOP1         ; save to FPST;
B_CALL  XName           ; OP1 = X var name
B_CALL  PushOP1         ; push onto FPST, Y1 -> FPS1;
B_CALL  OP1Set2         ; OP1 = 2
B_CALL  PushOP1         ; push onto FPST, Y1 -> FPS2,
                                ; X -> FPS1;

LD      HL,point5
RST     rMov9ToOP1      ; OP1 = .5;
LD      A,OPDeriv81     ; function is nDeriv
B_CALL  FourExec        ; OP1 = result
Y1Name: DB      EquObj,tVarEqu,tY1,0,0
Point5: DB      0,80h,50h,0,0,0,0,0,0

```

FourExec equates and functions

<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>
OPNpv	npv(OPNormal	normalcdf(OPMltRadd	*row+(
OPSeq	seq(OPQuad	fnInt(OPDeriv81	nDeriv(
OPSolve	solve(OPFmin	fMin(OPFmax	fMax(
OPDf	Fcdf(

ParseInp

Category: Parser

Description: Executes an equation or program stored in a variable.

Inputs:

Registers: None

Flags: None

Others: OP1 = name of equation or program to execute

Outputs:

Registers: None

Flags: None

Others: If executed an equation, then OP1 and Ans contain the result.

If executed a program, then no result is returned.

Errors will be generated during parsing — to avoid them from being displayed, install an error handler before parsing.

Registers destroyed: All

Remarks: See the Parsing Function, Temporary Variables section in Chapter 2 for further information.

(continued)

ParseInp *(continued)*

Example: Parse the graph equation y1 and store the answer in Y. Install an error handler around the parsing and the storing to catch any errors.

RET CA = 0 if OK, else RET CA = 1.

```

                LD          HL,y1Name
                RST          rMov9ToOP1    ; OP1 = y1 name
;
; if an error while parsing go to this label
                AppOnErr     ErrorHan      ; error handler installed,
                                           ; (macro)
;
                B_CALL       ParseInp     ; execute the equation
;
; returns if no error
;
                B_CALL       CkOP1Real    ; check if RealObj
                JR           Z,storit     ; jump if it is real
;
                AppOffErr                    ; remove the error handler
;
; come here if any error was detected
; error handler is removed when the error occurred
;
ErrorHan:
                B_CALL       CleanAll     ; clean any temp vars created by
                                           ; parser
                SCF                               ; CA = 1 signals failure
                RET
;
storit:
                B_CALL       StoY         ; store to Y, RET if no error,
                                           ; else ErrorHan
;
                AppOffErr                    ; remove error handler
;
                B_CALL       CleanAll     ; clean any temp vars created by
                                           ; parser
;
                CP           A           ; CA = 0 for no error
                RET
;
y1Name:
                DB           EquObj,tVarEqu,tY1,0,0

```

RclSysTok

Category: Parser

Description: Recalls a value in system variable specified by token number in the accumulator to OP1.

Inputs:

Registers: A = system variable token number

Flags: None

Others: None

Outputs:

Registers: OP1 = contents of system variable

Flags: None

Others: None

**Registers
destroyed:**

Remarks:

Example:

```
LD      A,XMINt
B_CALL  RclSysTok      ; OP1 = contents of Xmin
```

ThreeExec

Category: Parser

Description: Executes functions that have three arguments as input.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = third argument
FPST = second argument (pushed onto FPS second)
FPS1 = first argument (pushed onto FPS first)

Outputs:

Registers: None

Flags: None

Others: OP1 = result

**Registers
destroyed:** All

Remarks: Checks for valid argument types are done.
The values pushed onto the FPS are removed.

This entry point should only be used if direct access to a particular function is not available.

Valid arguments can be obtained from the *TI-83 Plus Guidebook*.

(continued)

ThreeExec *(continued)*

Example:

```

                                ; row + ([A],1,2)
LD      HL,MatAName
RST     rMov9ToOP1      ; OP1 = [A] name
B_CALL  PushOP1         ; save to FPST;
B_CALL  OP1Set1         ; OP1 = 1
B_CALL  PushOP1         ; push onto FPST, mat name
                                ; moves to FPS1;
B_CALL  OP1Set2         ; OP1 = 2;
LD      A,OPRAdd        ; function is row +
B_CALL  ThreeExec       ; OP1 = result, a temp Matrix
                                ; variable
MatAName:  DB      MatObj,tVarMat,tMatA,0,0

```

ThreeExec equates and functions

<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>
OPPrn	îPrn(OPIntr	îInt(OPBinpdf	binompdf(
OPBincdf	binomcdf(OPIrr	irr(OPNpv	npv(
OPSum	sum(OPProd	prod(OPNormalPdf	normalpdf(
OPRandNrm	randNorm(OPRandBin	randBin(OPRandInt	randInt(
OPINormal	invNorm(OPInstr	inString(OPNormal	normalcdf(
OPDt	tcdf(OPFpdf	Fpdf(Opchi	χcdf(
OPSubstr	sub(OPDeriv81	nDeriv(tEvalF	u#(
OPRadd	row+(OPRswap	rowSwap(OPRmlt	row*(
OPMltRadd	*row+(OPSolve	solve(

Note: For tEvalF there are really four inputs but execution still goes through the entry point for three arguments. The Equation name needs to be pushed onto the FPS first, then the second argument and then third, and then the fourth in OP1. This is only valid in Sequential graph mode.

The second argument is the start value.

The third argument is the end value.

The fourth argument is the step size.

A list of results is returned.

UnOPExec

Category: Parser

Description: Executes functions that have one argument as the input.

Inputs:

Registers: ACC = function to execute (see table below)

Flags: None

Others: OP1 = argument

Outputs:

Registers: None

Flags: None

Others: OP1 = result

**Registers
destroyed:** All

Remarks: This entry point should only be used if direct access to a particular function is not available.

It is also useful to use this entry point when arguments are not simply real numbers. See example below.

Valid arguments can be obtained from the *TI-83 Plus Guidebook*.

(continued)

UnOPExec *(continued)*

Example:

```

                                ; sin(L1)
LD          HL,L1name
RST         rMov9ToOP1        ; OP1 = L1 name;
LD          A,OPSin           ; function is addition
B_CALL     UnOPExec          ; OP1 = result of sin(L1) a
                                ; temp list variable
L1name:     DB                RListObj,tVarLst,tL1,0,0

```

UnOPExec equates and functions

<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>	<u>Equate</u>	<u>Function</u>
OPLog	log(OPTenX	10^X(OPLn	ln(
OPExToX	e^X(OPNot	not(OPSin	sin(
OPAsin	sin-1(OPCos	cos(Pacos	cos-1(
OPTan	tan(OPAtan	tan-1(OPSinh	sinh(
OPAsinh	sinh-1(OPCosh	cosh(OPAcosh	cosh-1(
OPTanh	tanh(OPAtanh	tanh-1(OPInverse	recipicol
OPDet	det(OPSgroot	û	OPSquare	^2
Opnegate	(-)	OPIpart	iPart(OPFpart	fPart(
OPIntgr	int(tEvalF	y#(value	OPConj	conj(
OPFact	!	OPAbs	abs(OPIdent	identity(
OPTranspose	mat transpose	OPSum	sum(OPProd	prod(
OPMin	min(OPMax	max(OPTofrac	>Frac
OPReal	real(OPImag	imag(OPAngle	angle(
OPExpr	expr(OPRound2	round(OPLength	length(
OPCube	^3	OPCbrt	3û	OPDim	dim(
OPRad	^r	OPDeg	ø	tMean	mean(
tMedian	median(OPRef	ref(OPRref	rref(
OPCumSum	cumSum(OPNormalPdf	normalPdf(OPInormal	invNorm(
OPDeltalst	-List(OPBal	bal(OPStdev	stdDev(
OPVariance	variance(OPRand	rand		

Note: For tEvalF there are really two inputs but execution still goes through the entry point for one argument. The Equation name needs to be pushed onto the FPS first, and the second argument in OP1.

This is valid in all graph modes.

The second argument is the value to evaluate at.

A

System Routines — Screen

ForceFullScreen	573
-----------------------	-----

ForceFullScreen

Category: Screen

Description: Switches the TI-83 Plus to Full Screen mode if currently In Horizontal or Vertical split mode.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** All

Remarks: Graph is dirtied if mode switched.

Example:

A

System Routines — Statistics

DelRes.....	575
OneVar.....	576
Rcl_StatVar	577

DelRes

Category: Statistics

Description: Invalidates the statistic results.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Statistic result variables marked as undefined.
RegEq variable is deleted.

**Registers
destroyed:** All

Remarks: Note that this routine does not set the graphDraw flag even if the stat result variable is used in a graph equation. This is a known problem.

Example: `B_CALL DelRes ; invalidate stat results`

OneVar

Category: Statistics

Description: Executes one-variable statistics.

Inputs:

Registers: ACC = number of arguments input

Flags: No_Del_Stat, (IY + more_flags) = 1 if:
Stat results that are not associated with one-variable stats are not to be deleted when this routine executes.
Also no Min's, Max's, or Quartiles will be computed.
Otherwise: previous statistic results are cleared.

Others: If ACC = 1 then OP1 = data list name.
If ACC = 2 then OP1 = frequency list name.
FPST = data list name.
Dimensions must match if two arguments.

Outputs:

Registers: None

Flags: None

Others: If no errors then one-variable stat output variables are updated.

Registers destroyed: All

Remarks: If the input lists have a formula associated with them this routine will not execute it and update the list values. This must be done by the calling routine.

See **Find_Parse_Formula**.

Example: Run one-variable stats on data list L1 and freq. list L2.

```

LD      HL,L1name
RST     rMov9ToOP1      ; OP1 = L1
RST     rPushReal01     ; data ->FPST
;
LD      HL,L2name
RST     rMov9ToOP1      ; OP1 = L2
;
B_CALL  OneVar          ; execute 1-variable stats
;
RET
L1name:  DB      ListObj,tVarLst,tL1,0,0
L2name:  DB      ListObj,tVarLst,tL2,0,0

```

Rcl_StatVar

Category: Statistics

Description: Recalls a statistic result variable to OP1.

Inputs:

Registers: ACC = stat variable to recall token value. These are listed in the TI83plus.inc file.

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = stat variable value, floating-point number

Registers destroyed: All but the ACC.

Remarks: The statistic variables are validated by running a regression or one/two variable statistic commands.

This routine does not check that the statistic variables are valid. Recalling one when not valid may result in random values.

Example: Recall statistic result variable X mean, assume statistic have been computed.

```
LD          A,tXMean      ; token value for XMean
B_CALL      Rcl_StatVar   ; recall contents to OP1
```

A

System Routines — Utility

AnsName.....	580
ConvDim00.....	581
CpHLDE	582
DisableApd	583
EnableApd.....	584
EOP1NotReal	585
Equ_or_NewEqu	586
GetBaseVer.....	587
GetTokLen.....	588
JForceCmdNoChar.....	589
JForceGraphKey	590
JForceGraphNoKey.....	591
MemClear.....	592
MemSet.....	593
Mov7B, Mov8B, Mov9B, Mov10B, Mov18B	594
Mov9OP1OP2	595
Mov9OP2Cp.....	596
Mov9ToOP1	597
Mov9ToOP2	598
MovFrOP1	599
OP1ExOP2, OP1ExOP3, OP1ExOP4, OP1ExOP5, OP1ExOP6, OP2ExOP4, OP2ExOP5, OP2ExOP6, OP5ExOP6	600
OP1ToOP2, OP1ToOP3, OP1ToOP4, OP1ToOP5, OP1ToOP6, OP2ToOP1, OP2ToOP3, OP2ToOP4, OP2ToOP5, OP2ToOP6, OP3ToOP1, OP3ToOP2, OP3ToOP4, OP3ToOP5, OP4ToOP1, OP4ToOP2, OP4ToOP3, OP4ToOP5, OP4ToOP6, OP5ToOP1, OP5ToOP2, OP5ToOP3, OP5ToOP4, OP5ToOP6, OP6ToOP1, OP6ToOP2, OP6ToOP5.....	601
PosNo0Int.....	602
RclAns	603
ReloadAppEntryVecs.....	604
SetXXOP1	605

(continued)

Contents *(continued)*

SetXXOP2	606
SetXXXXOP2	607
StoRand	608
StrCopy	609
StrLength.....	610

AnsName

Category: Utility

Description: Loads OP1 with the variable name Ans.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 contains the variable name Ans.

**Registers
destroyed:** All

Remarks:

Example: B_CALL AnsName ; load OP1 with Ans variable

ConvDim00

Category: Utility

Description: Converts floating-point number in OP1 to a two-byte value and compares that value with an input two-byte value.

Inputs:

Registers: HL = two-byte test value

Flags: None

Others: OP1 = floating-point value, must be a positive integer < 10,000

Outputs:

Registers: If no error on the input:
A = LSB hex value of OP1
DE = entire hex value of OP1

Flags: None

Others: None

**Registers
destroyed:** All

Remarks:

Example: Test OP1 = positive integer < or = 400:

```
LD      HL,400d      ; test value
B_CALL  ConvDim00
```

CpHLDE

Category: Utility

Description: Non destructives compare of registers HL and DE.

Inputs:

Registers: HL = two-byte value
DE = two-byte value

Flags: None

Others: None

Outputs:

Registers: HL, DE intact

Flags: CA = 1 if DE > HL
Z = 1 if HL = DE
CA = 0 if HL ≥ DE

Others: None

**Registers
destroyed:** None

Remarks:

Example: B_CALL CpHLDE

DisableApd

Category: Utility

Description: Turns off Auto Power Down feature.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: apdAble, (IY + apdFlags) is reset

**Registers
destroyed:** None

Remarks: Applications should re-enable APD before exiting. See **EnableApd**.

Example:

EnableApd

Category: Utility

Description: Turns on Auto Power Down.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

**Registers
destroyed:** None

Remarks: The TI-83 Plus will now power down if not used for approximately four minutes.

Example:

EOP1NotReal

Category: Utility

Description: Tests object in OP1 to be a real data type. If it is not, then jump to the system error DATA TYPE.

Inputs:

Registers: None

Flags: None

Others: (OP1) = objects data type byte

Outputs:

Registers: None

Flags: None

Others: Error if not OP1 — it does not have the data type RealObj.

Registers destroyed: A

Remarks:

Example:

Equ_or_NewEqu

Category: Utility

Description: Sees if A = EquObj or NewEquObj type.

Inputs:

Registers: A = type, can have flags set

Flags: None

Others: None

Outputs:

Registers: A = type with flags reset

Flags: Z set if A = EquObj or NewEquObj type

Others: None

**Registers
destroyed:** None

Remarks:

Example:

```
; see if ACC is EquObj or NewEquObj
Equ_or_NewEqu::
    AND    1Fh
    CP     EquObj
    RET    Z
    CP     NewEquObj
    RET
```

GetBaseVer

Category: Utility

Description: Returns current operating system version number.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: A = major version number
B = minor version number

Flags: None

Others: None

Registers destroyed: A, B

Remarks:

Example: For Operating system 1.00: A = 1, B = 0.

GetTokLen

Category: Utility

Description: Copy a token's string to OP3 and return the number of characters in the string.

Inputs:

Registers: HL = pointer to either a one or two byte token

Flags: None

Others: None

Outputs:

Registers: A = number of characters in the token's string
BC = also contains the number of characters in the token's string
HL = address of OP3, location the string was copied to

Flags: None

Others: String copied to RAM, starting at OP3

Registers destroyed: All

RAM used: OP3 – OP3 + (length of string)

Remarks:

Example: Find the number of characters in the 'Sin(' token string.

```

LD      A,tSin      ; Sin( token
LD      (OP1),A
LD      HL,OP1      ; pointer to token
;
B_CALL  GetTokLen   ; ACC = 4, the length of 'Sin('

```

JForceCmdNoChar

Category: Utility

Description: Exits the Application and returns to the home screen.

This should not be used to exit an application if the TI-83 Plus system monitor is closing the application due to link activity or turning off.

This routine will be the used in most applications to Close the application and return control to the TI-83 Plus system.

Before an application jumps to this entry point it must make certain the systems monitor vectors are set to the Application loader context.

See Entering and Exiting an Application Properly.

Inputs:

Registers: None

Flags: None

Others: Monitor vectors should be set to the Application loader.

Outputs:

Registers: None

Flags: None

Others: The home screen is given control.

Registers

destroyed: All

Remarks: Only use a B_JUMP with this entry point.

This can be used by an application anytime — the return stack does not need to be at any certain level. This routine will set the stack level back to a safe level.

ASM PROGRAMS SHOULD NOT USE THIS ROUTINE TO EXIT BACK TO THE SYSTEM.

Example: Set the monitor vectors to the Application loader and exit the application and return control to the home screen.

```
Exit_App:
    B_CALL    ReloadAppEntryVecs    ; load the monitor vectors
                                           ; to App loader
;
    B_JUMP    JForceCmdNoChar      ; exit the app and
                                           ; initiate home screen
```

JForceGraphKey

Category: Utility

Description: Exits the Application and returns to the graph screen with a key to be executed in the graph screen.

This should not be used to exit an application if the TI-83 Plus system monitor is closing the application due to link activity or turning off.

This routine will be the used in most applications to Close the application and return control to the TI-83 Plus system.

Before an application jumps to this entry point it must make certain the systems monitor vectors are set to the Application loader context.

See Entering and Exiting an Application Properly.

Inputs:

Registers: ACC = key to execute in the graph screen

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers All

destroyed:

Remarks: Only use a B_JUMP with this entry point.

This can be use by an application anytime — the return stack does not need to be at any certain level. This routine will set the stack level back to a safe level.

ASM PROGRAMS SHOULD NOT USE THIS ROUTINE TO EXIT BACK TO THE SYSTEM.

Example: Set the monitor vectors to the Application loader and exit the application and enter trace mode.

```
Exit_App:
        B_CALL      ReloadAppEntryVecs    ; load the monitor vectors
                                           ; to App loader
;
        LD          A,kTrace
        B_JUMP      JForceGraphKey        ; exit the app enter trace
                                           ; mode
```

JForceGraphNoKey

Category: Utility

Description: Exits the Application and returns to the graph screen.

This should not be used to exit an application if the TI-83 Plus system monitor is closing the application due to link activity or turning off.

This routine will be the used in most applications to close the application and return control to the TI-83 Plus system.

Before an application jumps to this entry point it must make certain the systems monitor vectors are set to the Application loader context.

See Entering and Exiting an Application Properly.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers destroyed: All

Remarks: Only use a B_JUMP with this entry point.

This can be use by an application anytime — the return stack does not need to be at any certain level. This routine will set the stack level back to a safe level.

ASM PROGRAMS SHOULD NOT USE THIS ROUTINE TO EXIT BACK TO THE SYSTEM.

Example: Set the monitor vectors to the Application loader and exit the application and give control to the graph context.

```
Exit_App:
        B_CALL      ReloadAppEntryVecs    ; load the monitor vectors
                                           ; to App loader
;
        LD          A,kTrace
        B_JUMP      JForceGraphNoKey     ; exit the app
```

MemClear

Category: Utility

Description: Clears a memory block (to 00h's).

Input:

Registers: BC = number of bytes in block
HL = address of first byte in memory block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Memory block cleared

Registers destroyed: A, BC, DE, HL

Remarks: BC must be > 1

Example: TBD

MemSet

Category: Utility

Description: Sets a memory block to a given value.

Inputs:

Registers: A = value to set all bytes in memory block
BC = number of bytes in block
HL = address of first byte in memory block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Memory block set

Registers destroyed: BC, DE, HL

Remarks: BC must be > 1

Example: TBD

Mov7B, Mov8B, Mov9B, Mov10B, Mov18B

Category: Utility

Description: Copies a short memory block where X = MovXB, where X is the number of bytes.

Inputs:

Registers: HL = start of source block
DE = start of destination block

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Block starting at original HL copied to area starting at original DE.

Registers destroyed: BC, DE, HL

Remarks:

Example:

Mov9OP1OP2

Category: Utility

Description: Copies a block of 18 bytes of RAM/ROM to OP1/OP2, with the first nine-bytes to OP1 and the second nine-bytes to OP2.

This is most commonly used to copy a complex element of either a list or matrix to OP1/OP2, skipping the 10th and 11th bytes of OP1.

Inputs:

Registers: HL = pointer to start of 18 bytes to copy

Flags: None

Others: None

Outputs:

Registers: DE = DE + 18

Flags: None

Others: First nine-bytes OP1 and first nine-bytes of OP2 contain the 18 bytes copied.

Registers destroyed: All but ACC

Remarks:

Example: Copy the first element of complex list L1 to OP1/OP2:

```
LD          HL,L1name
RST        rMov9ToOP1    ; OP1 = L1 name
B_CALL     FindSym       ; look up, DE = pointer to data
EX         DE,HL         ; HL = pointer to data
INC        HL
INC        HL             ; HL = pointer to 1st element
;
B_CALL     Mov9OP1OP2     ; OP1 = real part, OP2 = image
; part, of element 1
RET
```


Mov9OP2Cp

Category: Utility

Description: Copies a floating-point number from RAM/ROM to OP2 and compares it to a floating-point number in OP1.

Inputs:

Registers: HL = pointer to floating point to copy to OP2

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: Z = 1 if OP1 = OP2
Z = 0, CA = 1: OP1 < OP2
Z = 0, CA = 0: OP1 ≥ OP2

Others: OP1 = intact
OP2 = floating-point number copied

Registers destroyed: All

Remarks: Both OP1 and the float copied to OP2 are preserved.

Example: Copy the first element of real list L1 to OP2 and compare it to a floating-point number in OP1.

```

LD          HL,L1name
RST        rMov9ToOP1      ; OP1 = L1 name
B_CALL     FindSym         ; look up, DE = pointer to data
EX         DE,HL           ; HL = pointer to data
INC        HL
INC        HL              ; HL = pointer to 1st element
;
B_CALL     Mov9OP2Cp        ; copy element to OP2 and
                           ; compare to OP1
RET
```

Mov9ToOP1

Category: Utility

Description: Copies nine-bytes of RAM/ROM to OP1.

Inputs:

Registers: HL = pointer to the nine-bytes to copy

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 contains the nine-bytes

Registers destroyed: All but ACC

Remarks:

Example: B_CALL Mov9ToOP1

Mov9ToOP2

Category: Utility

Description: Copies nine-bytes of RAM/ROM to OP2.

Inputs:

Registers: HL = pointer to the nine-bytes to copy

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP2 contains the nine-bytes

Registers destroyed: All but ACC

Remarks:

Example: B_CALL Mov9ToOP2

MovFrOP1

Category: Utility

Description: Copies OP1 (nine bytes) to another RAM location.

Inputs:

Registers: DE = pointer to destination of move

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: HL = OP1 + 9
DE = DE + 9
OP1 copied to (DE)

Registers destroyed: All but ACC

Remarks:

Example:

OP1ExOP2, OP1ExOP3, OP1ExOP4, OP1ExOP5, OP1ExOP6, OP2ExOP4, OP2ExOP5, OP2ExOP6, OP5ExOP6

Category: Utility

Description: Exchanges 11-byte contents of OP(x) with OP(y).

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP(X) = former contents of OP(Y)
OP(Y) = former contents of OP(X)

Registers destroyed: A, BC, DE, HL

Remarks: Combinations Available:

(y)	OP1	OP2	OP3	OP4	OP5	OP6
(x)						
OP1		X	X	X	X	X
OP2				X	X	X
OP3						
OP4						
OP5						X
OP6						

Example: ; Exchange contents of OP2 and OP4
B_CALL OP2ExOP4

OP1ToOP2, OP1ToOP3, OP1ToOP4, OP1ToOP5,
 OP1ToOP6, OP2ToOP1, OP2ToOP3, OP2ToOP4,
 OP2ToOP5, OP2ToOP6, OP3ToOP1, OP3ToOP2,
 OP3ToOP4, OP3ToOP5, OP4ToOP1, OP4ToOP2,
 OP4ToOP3, OP4ToOP5, OP4ToOP6, OP5ToOP1,
 OP5ToOP2, OP5ToOP3, OP5ToOP4, OP5ToOP6,
 OP6ToOP1, OP6ToOP2, OP6ToOP5

Category: Utility

Description: Copies 11 bytes from OP(x) to OP(y).

Inputs:

Registers: None

Flags: None

Others: OP(x)

Outputs:

Registers: None

Flags: None

Others: OP(y) = former contents of OP(x)

Registers destroyed: BC, DE, HL

Remarks: Combinations Available:

Dest(y) Source(x)	OP1	OP2	OP3	OP4	OP5	OP6
OP1		X	X	X	X	X
OP2	X		X	X	X	X
OP3	X	X		X	X	
OP4	X	X	X		X	X
OP5	X	X	X	X		X
OP6	X	X			X	

Example: B_CALL OP1ToOP3

PosNo0Int

Category: Utility

Description: Checks if OP1 is a positive non-zero integer floating point.

Inputs:

Registers: None

Flags: None

Others: OP1 = floating-point number

Outputs:

Registers: None

Flags: Z = 1 if OP1 = positive non 0 integer
Z = 0 if non integer or negative or 0

Others: None

**Registers
destroyed:** ACC

Remarks:

Example:

RclAns

Category: Utility

Description: Recalls answer to OP1[,OP2] or at least set up pointers to it.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1[,OP2] if real [or complex]

Registers destroyed: AF, BC, DE, HL,

Remarks: Entire code:

```
CALL    AnsName    ; see these routines for more
                    ; info
JP      RclVarSym   ; see these routines for more
                    ; info
```

AnsName puts the name of Ans into
 OP1 = 00h,tAns,00h,00h,.....00h
 = 00h,072h,00h,00h,.....00h

RclVarSym will recall the contents of the variable to OP1 if it is real, to OP1 and OP2 if the variable is complex and otherwise leaves the name as is in OP1 and returns HL as the symbol table pointer and DE as the data pointer as in **ChkFindSym**.

Example:

```
B_CALL  RclAns      ; This example presumes that
                    ; you already know that Ans is
                    ; a Real number.
LD      A,9          ; display up to 8 digits
B_CALL  DispOP1A     ;
```


ReloadAppEntryVecs

Category: Utility

Description: Sets the system monitor vector table to the Application loader context.

This routine is used by advanced applications that override the system monitor vector table. This routine should be called by the application just before exiting.

This routine should only be used by applications, not ASM programs.

Inputs:

Registers: None

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: Monitor system vectors are now set to the application loader.

Registers destroyed: All

Remarks:

Example: Assume we have an application that overrode the monitor vectors and our application is exiting because the user pressed the [Quit] key.

```

ChkForQuit:
                CP            kQuit            ; quit key?
                JR            NZ,notQuit        ; jump if no
;
                B_CALL        ReloadAppEntryVecs ; restore monitor to
                                                ; application loader
                B_JUMP        JForceCmdNoChar   ; switch to the home
                                                ; screen
;

```

SetXXOP1

Category: Utility

Description: Sets OP1 equal to a floating-point integer between 0 and 99.

Inputs:

Registers: ACC = integer value to set OP1 equal to

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP1 = floating-point integer between 0 – 99

**Registers
destroyed:** All

RAM used: OP1

Remarks: No error checking is done for invalid input.

Example: Set OP1 = 75.

```
LD      A,75
B_CALL  SetXXOP1      ; OP1 = floating point 75
```

SetXXOP2

Category: Utility

Description: Sets OP2 equal to a floating-point integer between 0 and 99.

Inputs:

Registers: ACC = integer value to set OP2 equal to

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP2 = floating-point integer between 0 – 99

**Registers
destroyed:** All

RAM used: OP2

Remarks: No error checking is done for invalid input.

Example: Set OP2 = 75.

```
LD      A,75
B_CALL  SetXXOP2      ; OP2 = floating point 75
```

SetXXXXOP2

Category: Utility

Description: Sets OP2 equal to a floating-point integer between 0 and 65535.

Inputs:

Registers: HL = integer value to set OP2 equal to

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: OP2 = floating-point integer between 0 – 65535

**Registers
destroyed:** All

RAM used: OP2

Remarks:

Example: Set OP2 = 7523.

```
LD      HL, 7523
B_CALL  SetXXXXOP2    ; OP2 = floating point 7523
```

StoRand

Category: Utility

Description: Initializes random number seeds on OP1 value.

Inputs:

Registers: None

Flags: None

Others: OP1 = real number $0e0 \dots < 1E9$

Outputs:

Registers: None

Flags: None

Others: OP1 = same value as unmodified input.

**Registers
destroyed:** All

RAM used: OP1, OP2, OP6

Remarks: Storing a 0 to the seed will reinitialize the random number generator to its original state from the factory.

The input value in OP1 must be a real number, but it does not have to fall within the specified range. If it does not, it will be modified (exponent reduced, sign changed, and truncated) to fit in the range.

Example:

StrCopy

Category: Utility

Description: Copy a null-terminated string in memory.

Inputs:

Registers: HL = starting address of source string

DE = starting address of destination

Flags: None

Others: None

Outputs:

Registers: None

Flags: None

Others: None

Registers A, DE, HL

Destroyed:

Remarks: This is like a C language StrCpy() function.

When complete:

- HL is left pointing to the null terminator of the source string.
- DE is left pointing to the null terminator of the destination string.

Example:

StrLength

Category: Utility

Description: Returns the length of a zero (0) terminated string residing in RAM.

Inputs:

Registers: HL = pointer to start of zero terminated string, in RAM

Flags: None

Others: None

Outputs:

Registers: BC = length of string, not including terminating 0

Flags: None

Others: None

**Registers
destroyed:** BC

Remarks:

Example:

A

System Routines — Miscellaneous

ConvOP1 612

ConvOP1

Category: Miscellaneous

Description: Converts a floating-point number in OP1 to a two-byte hexadecimal number in DE.

Inputs:

Registers: OP1 = floating-point number

Flags: None

Others: None

Outputs:

Registers: A = LSB hex value
DE = entire hex value
If OP1 exponent > 3 error

Flags: None

Others: None

**Registers
destroyed:**


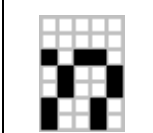

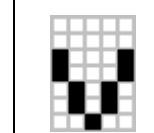
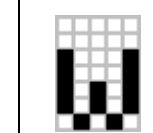
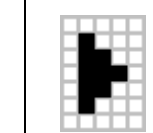
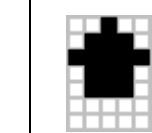
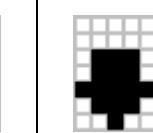
Remarks:

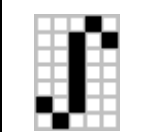
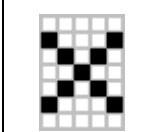
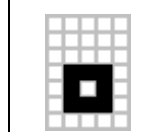
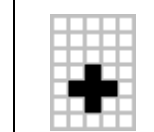
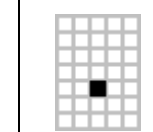
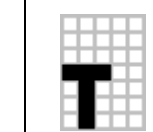
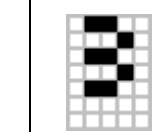

Example:

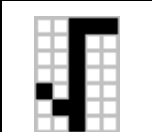
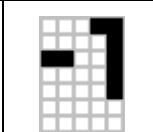
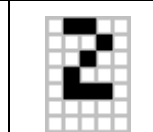
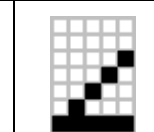
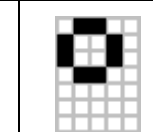
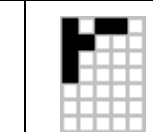
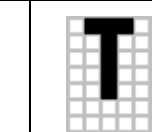
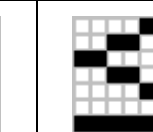
Appendix B

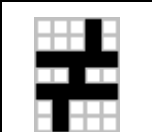
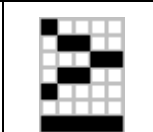
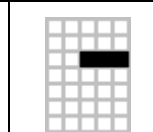
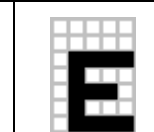
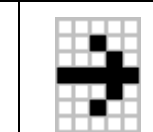

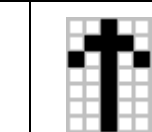
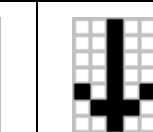
TI-83 Plus “Large” Character Fonts

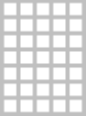
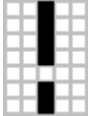
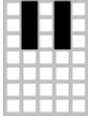
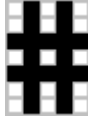


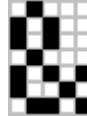
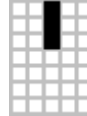
The font map below shows each character code, the symbolic name, and the character map.

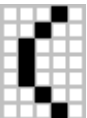
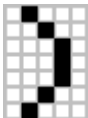
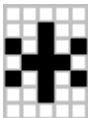

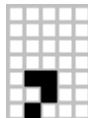

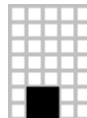
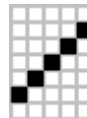
00h NOT USED	01h LrecurN	02h LrecurU	03h LrecurV	04h LrecurW	05h Lconvert	06h LsqUp	07h LsqDown
							


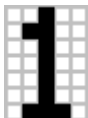
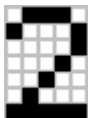





08h Lintegral	09h Lcross	0Ah LboxIcon	0Bh LcrossIcon	0Ch LdotIcon	0Dh LsubT	0Eh LcubeR	0Fh LhexF
							

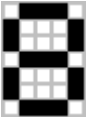
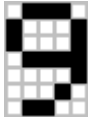
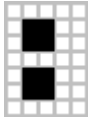
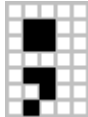
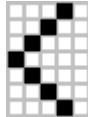

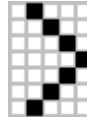
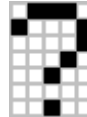
10h Lroot	11h Linverse	12h Lsquare	13h Langle	14h Ldegree	15h Lradian	16h Ltranspose	17h LLE
							




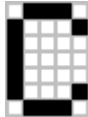




18h LNE	19h LGE	1Ah Lneg	1Bh Lexponent	1Ch Lstore	1Dh Lten	1Eh LupArrow	1Fh LdownArrow
							




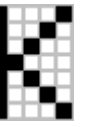



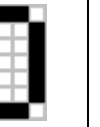
20h Lspace	21h Lexclam	22h Lquote	23h Lpound	24h Lfourth	25h Lpercent	26h Lampersand	27h Lapostrophe
							






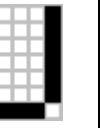


28h LlParen	29h LrParen	2Ah Lasterisk	2Bh LplusSign	2Ch Lcomma	2Dh Ldash	2Eh Lperiod	2Fh Lslash
							

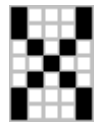
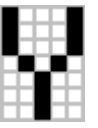




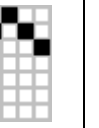

30h L0	31h L1	32h L2	33h L3	34h L4	35h L5	36h L6	37h L7
							

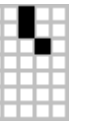

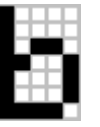



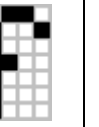

38h L8	39h L9	3Ah Lcolon	3Bh Lsemicolon	3Ch LLT	3Dh LEQ	3Eh LGT	3Fh Lquestion
							

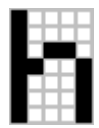
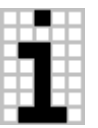

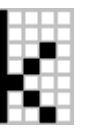

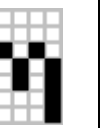

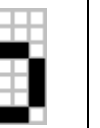
40h LatSign	41h LcapA	42h LcapB	43h LcapC	44h LcapD	45h LcapE	46h LcapF	47h LcapG
							

48h LcapH	49h LcapI	4Ah LcapJ	4Bh LcapK	4Ch LcapL	4Dh LcapM	4Eh LcapN	4Fh LcapO
							

50h LcapP	51h LcapQ	52h LcapR	53h LcapS	54h LcapT	55h LcapU	56h LcapV	57h LcapW
							

58h LcapX	59h LcapY	5Ah LcapZ	5Bh Ltheta	5Ch Lbackslash	5Dh LrBrack	5Eh Lcaret	5Fh Lunderscore
							

60h Lbackquote	61h La	62h Lb	63h Lc	64h Ld	65h Le	66h Lf	67h Lg
							

68h Lh	69h Li	6Ah Lj	6Bh Lk	6Ch Ll	6Dh Lm	6Eh Ln	6Fh Lo
							

70h Lp	71h Lq	72h Lr	73h Ls	74h Lt	75h Lu	76h Lv	77h Lw

78h Lx	79h Ly	7Ah Lz	7Bh LlBrace	7Ch Lbar	7Dh LrBrace	7Eh Ltilde	7Fh LinVEQ

80h Lsub0	81h Lsub1	82h Lsub2	83h Lsub3	84h Lsub4	85h Lsub5	86h Lsub6	87h Lsub7

88h Lsub8	89h Lsub9	8Ah LcapAAcute	8Bh LcapAGrave	8Ch LcapACaret	8Dh LcapADier	8Eh LaAcute	8Fh LaGrave

90h LaCaret	91h LaDier	92h LcapEAcute	93h LcapEGrave	94h LcapECaret	95h LcapEDier	96h LeAcute	97h LeGrave

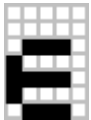
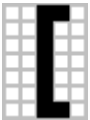
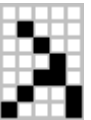



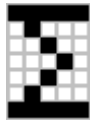

98h LeCaret	99h LeDier	9Ah LcapIAcute	9Bh LcapIGrave	9Ch LcapICaret	9Dh LcapIDier	9Eh LiAcute	9Fh LiGrave

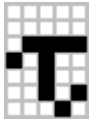
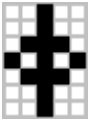



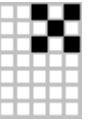
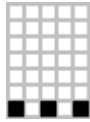
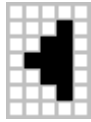
A0h LiCaret	A1h LiDier	A2h LcapOAcute	A3h LcapOGrave	A4h LcapOCaret	A5h LcapODier	A6h LoAcute	A7h LoGrave

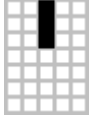
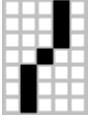
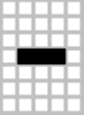
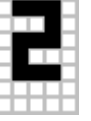
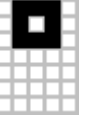
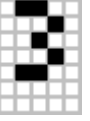

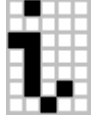
A8h LoCaret	A9h LoDier	AAh LcapUAcute	ABh LcapUGrave	ACH LcapUCaret	ADh LcapUDier	Aeh LuAcute	Afh LuGrave


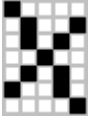


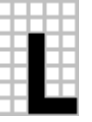

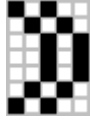
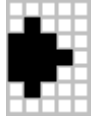
B0h LuCaret	B1h LuDier	B2h LcapCCed	B3h LcCed	B4h LcapNTilde	B5h LnTilde	B6h Laccent	B7h Lgrave








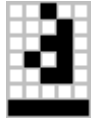
B8h Ldieresis	B9h LquesDown	BAh LexclamDown	BBh Lalpha	BCh Lbeta	BDh Lgamma	BEh LcapDelta	BFh Ldelta

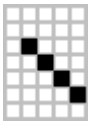
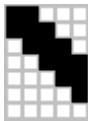




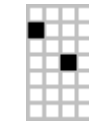

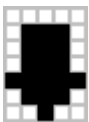
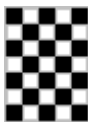
C0h Lepsilon	C1h LlBrack	C2h Llambda	C3h Lmu	C4h Lpi	C5h Lrho	C6h LcapSigma	C7h Lsigma
							

C8h Ltau	C9h Lphi	CAh LcapOmega	CBh LxMean	CCh LyMean	CDh LsupX	CEh Lellipsis	CFh Lleft
							

D0h Lblock	D1h Lper	D2h Lhyphen	D3h Larea	D4h Ltemp	D5h Lcube	D6h Lenter	D7h LimagI
							

D8h Lphat	D9h Lchi	DAh LstatF	DBh Llne	DCh LlistL	DDh LfinanN	DEh L2_r_paren	DFh LblockArrow
							

E0h LcurO	E1h LcurO2	E2h LcurOcapA	E3h LcurOa	E4h LcurI	E5h LcurI2	E6h LcurIcapA	E7h LcurIa
							

E8h LGline	E9h LGthick	EAh LGabove	EBh LGbelow	ECh LGpath	EDh LGanimate	EEh LGdot	EFh LUpBlk
							
F0h LDnBlk	F1h LcurFull						
							

Appendix C

TI-83 Plus “Small” Character Fonts



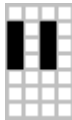
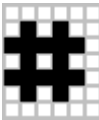

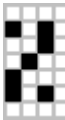
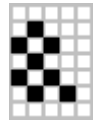

The font map below shows each character code, the symbolic name, and the character map. Most characters are five pixels high, but a few are longer. The character widths are variable, e.g. a space has a width of one pixel whereas an asterisk has width of five pixels. Character maps usually include one blank pixel column on the right side to ensure spacing when printing strings.



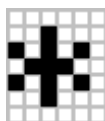


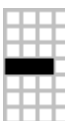


00h NOT USED	01h SrecurN	02h SrecurU	03h SrecurV	04h SrecurW	05h Sconvert	06h SFourSpaces	07h SsqDown




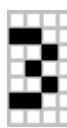




08h Sintegral	09h Scross	0Ah SboxIcon	0Bh ScrossIcon	0Ch SdotIcon	0Dh SsubT	0Eh ScubeR	0Fh ShexF




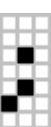




10h Sroot	11h Sinverse	12h Ssquare	13h Sangle	14h Sdegree	15h Sradian	16h Stranspose	17h SLE

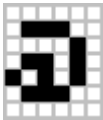
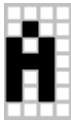

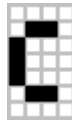




18h SNE	19h SGE	1Ah Sneg	1Bh Sexponent	1Ch Sstore	1Dh Sten	1Eh SupArrow	1Fh SdownArrow









20h Sspace	21h Sexclam	22h Squote	23h Spound	24h Sdollar	25h Spercent	26h Sampersand	27h Sapostrophe
							




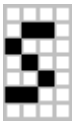
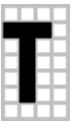

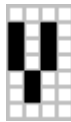

28h SlParen	29h SrParen	2Ah Sasterisk	2Bh SplusSign	2Ch Scomma	2Dh Sdash	2Eh Speriod	2Fh Sslash
							

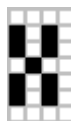
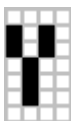

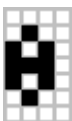
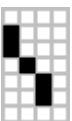

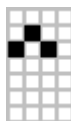

30h s0	31h s1	32h s2	33h s3	34h s4	35h s5	36h s6	37h s7
							

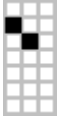


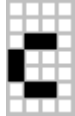
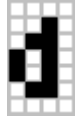
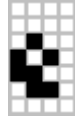

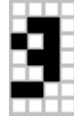
38h s8	39h s9	3Ah Scolon	3Bh Ssemicolon	3Ch SLT	3Dh SEQ	3Eh SGT	3Fh Squestion
							



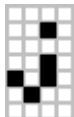


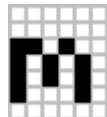

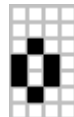
40h SatSign	41h ScapA	42h ScapB	43h ScapC	44h ScapD	45h ScapE	46h ScapF	47h ScapG
							

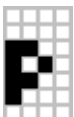
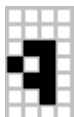

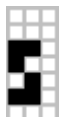


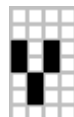
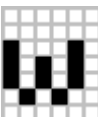
48h ScapH	49h ScapI	4Ah ScapJ	4Bh ScapK	4Ch ScapL	4Dh ScapM	4Eh ScapN	4Fh ScapO
							

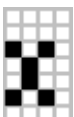
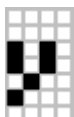
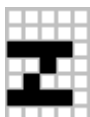
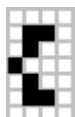

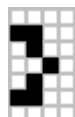
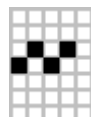
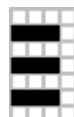
50h ScapP	51h ScapQ	52h ScapR	53h ScapS	54h ScapT	55h ScapU	56h ScapV	57h ScapW
							




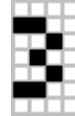




58h ScapX	59h ScapY	5Ah ScapZ	5Bh Stheta	5Ch Sbackslach	5Dh SrBrack	5Eh Scaret	5Fh Sunderscore
							




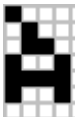


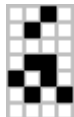

60h Sbackquote	61h SmallA	62h SmallB	63h SmallC	64h SmallD	65h SmallE	66h SmallF	67h SmallG
							

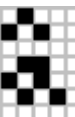
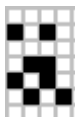






68h SmallH	69h SmallI	6Ah SmallJ	6Bh SmallK	6Ch SmallL	6Dh SmallM	6Eh SmallN	6Fh SmallO
							

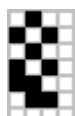
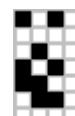




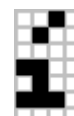
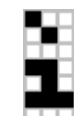
70h SmallP	71h SmallQ	72h SmallR	73h SmallS	74h SmallT	75h SmallU	76h SmallV	77h SmallW
							

78h SmallX	79h SmallY	7Ah SmallZ	7Bh SlBrace	7Ch Sbar	7Dh SrBrace	7Eh Stilde	7Fh SinvEQ
							

80h Ssub0	81h Ssub1	82h Ssub2	83h Ssub3	84h Ssub4	85h Ssub5	86h Ssub6	87h Ssub7
							

88h Ssub8	89h Ssub9	8Ah ScapAAcute	8Bh ScapAGrave	8Ch ScapACaret	8Dh ScapADier	8Eh SaAcute	8Fh SaGrave
							

90h SaCaret	91h SaDier	92h ScapEAcute	93h ScapEGrave	94h ScapECaret	95h ScapEDier	96h SeAcute	97h SeGrave
							

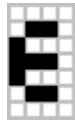

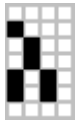





98h SeCaret	99h SeDier	9Ah ScapIAcute	9Bh ScapIGrave	9Ch ScapICaret	9Dh ScapIDier	9Eh SiAcute	9Fh SiGrave
							


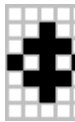

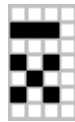
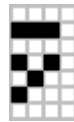
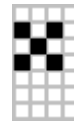

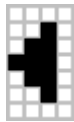
A0h SiCaret	A1h SiDier	A2h ScapOAcute	A3h ScapOGrave	A4h ScapOCaret	A5h ScapODier	A6h SoAcute	A7h SoGrave

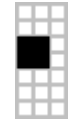
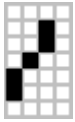
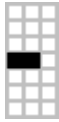
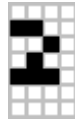
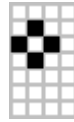
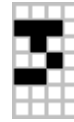
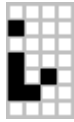
A8h SoCaret	A9h SoDier	AAh ScapUAcute	ABh ScapUGrave	ACh ScapUCaret	ADh ScapUDier	AEnh SuAcute	AFh SuGrave

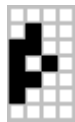
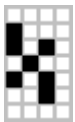

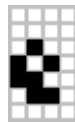



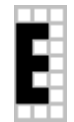
B0h SuCaret	B1h SuDier	B2h ScapCCed	B3h ScCed	B4h ScapNTilde	B5h SnTilde	B6h Saccent	B7h Sgrave


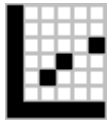
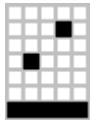
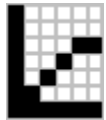
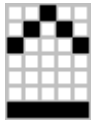

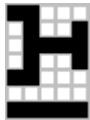
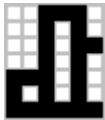
B8h Sdieresis	B9h SquesDown	BAh SexclamDown	BBh Salpha	BCh Sbeta	BDh Sgamma	BEh ScapDelta	BFh Sdelta


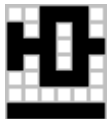
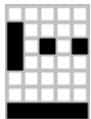
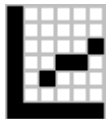
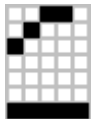
C0h Sepsilon	C1h SlBrack	C2h Slambda	C3h Smu	C4h Spi	C5h Srho	C6h ScapSigma	C7h Ssigma
							

C8h Stau	C9h Sphi	CAh ScapOmega	CBh SxMean	CCh SyMean	CDh SsupX	CEh Sellipsis	CFh Sleft
							

D0h Sblock	D1h Sper	D2h Shyphen	D3h Sarea	D4h Stemp	D5h Scube	D6h Senter	D7h SimagI
							

D8h Sphat	D9h Schi	DAh SstatF	DBh Slne	DCh SlistL	DDh SfinanN	DEh S2_r_paren	DFh SnarrowCapE
							

E0h SListLock	E1h Sscatter1	E2h Sscatter2	E3h Sxylne1	E4h Sxylne2	E5h Sboxplot1	E6h Sboxplot2	E7h Shist1
							

E8h Shist2	E9h SmodBox1	EAh SmodBox2	EBh Snormal1	ECh Snormal2			
							

R

Reference List — System Routines

A

AbsO1O2Cp.....	368. See Math
AbsO1PAbsO2.....	369. See Math
ACos	370. See Math
ACosh	371. See Math
ACosRad.....	372. See Math
AdrLEle	351. See List
AdrMEle	352. See List
AdrMRow	483. See Matrix
AlIEq	259. See Graphing and Drawing
AllocFPS	241. See Floating Point Stack
AllocFPS1	242. See Floating Point Stack
Angle.....	373. See Math
AnsName	580. See Utility
ApdSetup	344. See Keyboard
AppGetCalc.....	337. See IO
AppGetCbl	338. See IO
Arc_Unarc.....	488. See Memory
ASin	374. See Math
ASinH.....	375. See Math
ASinRad.....	376. See Math
ATan	377. See Math
ATan2	378. See Math
ATan2Rad.....	379. See Math
ATanH.....	380. See Math
ATanRad.....	381. See Math

B

BinOPExec.....	559. See Parser
Bit_VertSplit	159. See Display
BufClr	260. See Graphing and Drawing
BufCpy	261. See Graphing and Drawing

C

CAbs	382. See Math
CAdd	383. See Math
CanAlphIns	345. See Keyboard
CDiv	384. See Math
CDivByReal	385. See Math
CEtoX	386. See Math
CFrac	387. See Math
CheckSplitFlag	160. See Display
ChkFindSym	489. See Memory
CIntgr	388. See Math
CircCmd	262. See Graphing and Drawing
CkInt	389. See Math
CkOdd	390. See Math
CkOP1C0	391. See Math
CkOP1Cplx	392. See Math
CkOP1FP0	393. See Math
CkOP1Pos	394. See Math
CkOP1Real	395. See Math
CkOP2FP0	396. See Math
CkOP2Pos	397. See Math
CkOP2Real	398. See Math
CkPosInt	399. See Math
CkValidNum	400. See Math
CleanAll	491. See Memory
ClearRect	264. See Graphing and Drawing
ClearRow	161. See Display
CLine	265. See Graphing and Drawing
CLineS	267. See Graphing and Drawing
CLN	401. See Math
CLog	402. See Math
CloseEditBufNoR	203. See Edit
CloseProg	492. See Memory
ClrGraphRef	269. See Graphing and Drawing
ClrLCD	162. See Display
ClrLCDFull	163. See Display
ClrLp	403. See Math
ClrOP1S	404. See Math
ClrOP2S	164. See Display

ClrScrn	165. See Display
ClrScrnFull	166. See Display
ClrTxtShd	167. See Display
CMltByReal	405. See Math
CmpSyms	493. See Memory
CMult	406. See Math
Conj	407. See Math
ConvDim	353. See List
ConvDim00	581. See Utility
ConvLcToLr	354. See List
ConvLrToLc	355. See List
ConvOP1	612. See Miscellaneous
COP1Set0	408. See Math
Cos	409. See Math
CosH	410. See Math
CpHLDE	582. See Utility
CPoint	270. See Graphing and Drawing
CPointS	272. See Graphing and Drawing
CpOP1OP2	411. See Math
CpOP4OP3	412. See Math
CpyO1ToFPS1	244. See Floating Point Stack
CpyO1ToFPS2	244. See Floating Point Stack
CpyO1ToFPS3	244. See Floating Point Stack
CpyO1ToFPS4	244. See Floating Point Stack
CpyO1ToFPS5	244. See Floating Point Stack
CpyO1ToFPS6	244. See Floating Point Stack
CpyO1ToFPS7	244. See Floating Point Stack
CpyO1ToFPST	244. See Floating Point Stack
CpyO2ToFPS1	244. See Floating Point Stack
CpyO2ToFPS2	244. See Floating Point Stack
CpyO2ToFPS3	244. See Floating Point Stack
CpyO2ToFPS4	244. See Floating Point Stack
CpyO2ToFPST	244. See Floating Point Stack
CpyO3ToFPS1	244. See Floating Point Stack
CpyO3ToFPS2	244. See Floating Point Stack
CpyO3ToFPST	244. See Floating Point Stack
CpyO5ToFPS1	244. See Floating Point Stack
CpyO5ToFPS3	244. See Floating Point Stack
CpyO6ToFPS2	244. See Floating Point Stack

CpyO6ToFPST	244. See Floating Point Stack
CpyStack	243. See Floating Point Stack
CpyTo1FPS1	245. See Floating Point Stack
CpyTo1FPS10	245. See Floating Point Stack
CpyTo1FPS11	245. See Floating Point Stack
CpyTo1FPS2	245. See Floating Point Stack
CpyTo1FPS3	245. See Floating Point Stack
CpyTo1FPS4	245. See Floating Point Stack
CpyTo1FPS5	245. See Floating Point Stack
CpyTo1FPS6	245. See Floating Point Stack
CpyTo1FPS7	245. See Floating Point Stack
CpyTo1FPS8	245. See Floating Point Stack
CpyTo1FPS9	245. See Floating Point Stack
CpyTo1FPST	245. See Floating Point Stack
CpyTo2FPS1	245. See Floating Point Stack
CpyTo2FPS2	245. See Floating Point Stack
CpyTo2FPS3	245. See Floating Point Stack
CpyTo2FPS4	245. See Floating Point Stack
CpyTo2FPS5	245. See Floating Point Stack
CpyTo2FPS6	245. See Floating Point Stack
CpyTo2FPS7	245. See Floating Point Stack
CpyTo2FPS8	245. See Floating Point Stack
CpyTo2FPST	245. See Floating Point Stack
CpyTo3FPS1	245. See Floating Point Stack
CpyTo3FPS2	245. See Floating Point Stack
CpyTo3FPST	245. See Floating Point Stack
CpyTo4FPST	245. See Floating Point Stack
CpyTo5FPST	245. See Floating Point Stack
CpyTo6FPS2	245. See Floating Point Stack
CpyTo6FPS3	245. See Floating Point Stack
CpyTo6FPST	245. See Floating Point Stack
CpyToFPS1	247. See Floating Point Stack
CpyToFPS2	248. See Floating Point Stack
CpyToFPS3	249. See Floating Point Stack
CpyToFPST	246. See Floating Point Stack
CpyToStack	250. See Floating Point Stack
Create0Equ	494. See Memory
CreateAppVar	495. See Memory
CreateCList	496. See Memory

CreateCplx	497. See Memory
CreateEqu	498. See Memory
CreatePair	499. See Memory
CreatePict	500. See Memory
CreateProg	501. See Memory
CreateProtProg	502. See Memory
CreateReal	503. See Memory
CreateRList	504. See Memory
CreateRMat	505. See Memory
CreateStrng	506. See Memory
CRecip	413. See Math
CSqRoot	414. See Math
CSquare	415. See Math
CSub	416. See Math
CTenX	417. See Math
CTrunc	418. See Math
Cube	419. See Math
CursorOff	204. See Edit
CursorOn	205. See Edit
CXrootY	420. See Math
CYtoX	421. See Math

D

DarkLine	274. See Graphing and Drawing
DarkPnt	276. See Graphing and Drawing
DataSize	507. See Memory
DataSizeA	508. See Memory
DeallocFPS	509. See Memory
DeallocFPS1	510. See Memory
DecO1Exp	422. See Math
DelListEl	356. See List
DelMem	511. See Memory
DelRes	575. See Statistics
DelVar	513. See Memory
DelVarArc	514. See Memory
DelVarNoArc	515. See Memory
DisableApd	583. See Utility
Disp	278. See Graphing and Drawing
DispDone	168. See Display
DispEOL	206. See Edit

DispHL	169. See Display
DisplayImage	170. See Display
DispOP1A	172. See Display
DivHLBy10	334. See Interrupt
DivHLByA.....	335. See Interrupt
DrawCirc2	279. See Graphing and Drawing
DrawCmd.....	281. See Graphing and Drawing
DrawRectBorder	282. See Graphing and Drawing
DrawRectBorderClear	283. See Graphing and Drawing
DToR	423. See Math

E

EditProg	516. See Memory
EnableApd	584. See Utility
EnoughMem.....	517. See Memory
EOP1NotReal	585. See Utility
Equ_or_NewEqu.....	586. See Utility
EraseEOL	173. See Display
EraseRectBorder	284. See Graphing and Drawing
ErrArgument.....	209. See Error
ErrBadGuess	210. See Error
ErrBreak.....	211. See Error
ErrD_OP1_0	212. See Error
ErrD_OP1_LE_0	213. See Error
ErrD_OP1Not_R	214. See Error
ErrD_OP1NotPos.....	215. See Error
ErrD_OP1NotPosInt.....	216. See Error
ErrDataType.....	217. See Error
ErrDimension	218. See Error
ErrDimMismatch	219. See Error
ErrDivBy0.....	220. See Error
ErrDomain.....	221. See Error
ErrIncrement	222. See Error
ErrInvalid.....	223. See Error
ErrIterations	224. See Error
ErrLinkXmit	225. See Error
ErrMemory	226. See Error
ErrNon_Real	227. See Error
ErrNonReal	228. See Error
ErrNotEnoughMem	229. See Error

ErrOverflow	230. See Error
ErrSignChange	231. See Error
ErrSingularMat	232. See Error
ErrStat	233. See Error
ErrStatPlot	234. See Error
ErrSyntax	235. See Error
ErrTolTooSmall	236. See Error
ErrUndefined	237. See Error
EToX	424. See Math
Exch9	518. See Memory
ExLp	519. See Memory
ExpToHex	425. See Math

F

Factorial	426. See Math
FillRect	285. See Graphing and Drawing
FillRectPattern	287. See Graphing and Drawing
Find_Parse_Formula	357. See List
FindAlphaDn	520. See Memory
FindAlphaUp	522. See Memory
FindApp	524. See Memory
FindAppDn	526. See Memory
FindAppNumPages	525. See Memory
FindAppUp	527. See Memory
FindSym	528. See Memory
FiveExec	561. See Parser
FixTempCnt	530. See Memory
FlashToRam	531. See Memory
ForceFullScreen	573. See Screen
FormBase	174. See Display
FormDCplx	176. See Display
FormEReal	178. See Display
FormReal	179. See Display
FourExec	563. See Parser
FPAdd	427. See Math
FPDiv	428. See Math
FPMult	429. See Math
FPRecip	430. See Math
FPSquare	431. See Math
FPSub	432. See Math

Frac..... 433. See Math

G

GetBaseVer587. See Utility

GetCSC 346. See Keyboard

GetKey 349. See Keyboard

GetLToOP1358. See List

GetMToOP1484. See Matrix

GetTokLen588. See Utility

GrBufClr289. See Graphing and Drawing

GrBufCpy290. See Graphing and Drawing

GrphCirc291. See Graphing and Drawing

H

HLTimes9..... 434. See Math

HorizCmd292. See Graphing and Drawing

HTimesL 435. See Math

I

IBounds.....293. See Graphing and Drawing

IBoundsFull294. See Graphing and Drawing

ILine295. See Graphing and Drawing

IncLstSize359. See List

InsertList361. See List

InsertMem 532. See Memory

Int..... 436. See Math

Intgr 437. See Math

InvCmd297. See Graphing and Drawing

InvertRect298. See Graphing and Drawing

InvOP1S 438. See Math

InvOP1SC 439. See Math

InvOP2S 440. See Math

InvSub..... 441. See Math

IOffset299. See Graphing and Drawing

IPoint.....300. See Graphing and Drawing

J

JError 238. See Error

JErrorNo 239. See Error

JForceCmdNoChar589. See Utility

JForceGraphKey590. See Utility

JForceGraphNoKey591. See Utility

K

KeyToString 207. See Edit

L

LineCmd 302. See Graphing and Drawing

LnX 442. See Math

Load_SFont 181. See Display

LoadClndPaged 534. See Memory

LoadDElndPaged 535. See Memory

LoadPattern 180. See Display

LogX 443. See Math

M

Max 444. See Math

MemChk 536. See Memory

MemClear 592. See Utility

MemSet 593. See Utility

Min 445. See Math

Minus1 446. See Math

Mov10B 594. See Utility

Mov18B 594. See Utility

Mov7B 594. See Utility

Mov8B 594. See Utility

Mov9B 594. See Utility

Mov9OP1OP2 595. See Utility

Mov9OP2Cp 596. See Utility

Mov9ToOP1 597. See Utility

Mov9ToOP2 598. See Utility

MovFrOP1 599. See Utility

O

OneVar 576. See Statistics

OP1ExOP2 600. See Utility

OP1ExOP3 600. See Utility

OP1ExOP4 600. See Utility

OP1ExOP5 600. See Utility

OP1ExOP6 600. See Utility

OP1ExpToDec 447. See Math

OP1Set0 448. See Math

OP1Set1	448. See Math
OP1Set2	448. See Math
OP1Set3	448. See Math
OP1Set4	448. See Math
OP1ToOP2	601. See Utility
OP1ToOP3	601. See Utility
OP1ToOP4	601. See Utility
OP1ToOP5	601. See Utility
OP1ToOP6	601. See Utility
OP2ExOP4	600. See Utility
OP2ExOP5	600. See Utility
OP2ExOP6	600. See Utility
OP2Set0	448. See Math
OP2Set1	448. See Math
OP2Set2	448. See Math
OP2Set3	448. See Math
OP2Set4	448. See Math
OP2Set5	448. See Math
OP2Set60	448. See Math
OP2Set8	449. See Math
OP2SetA	450. See Math
OP2ToOP1	601. See Utility
OP2ToOP3	601. See Utility
OP2ToOP4	601. See Utility
OP2ToOP5	601. See Utility
OP2ToOP6	601. See Utility
OP3Set0	448. See Math
OP3Set1	448. See Math
OP3Set2	448. See Math
OP3ToOP1	601. See Utility
OP3ToOP2	601. See Utility
OP3ToOP4	601. See Utility
OP3ToOP5	601. See Utility
OP4Set0	448. See Math
OP4Set1	448. See Math
OP4ToOP1	601. See Utility
OP4ToOP2	601. See Utility
OP4ToOP3	601. See Utility
OP4ToOP5	601. See Utility

OP4ToOP6	601. See Utility
OP5ExOP6	600. See Utility
OP5Set0	448. See Math
OP5ToOP1	601. See Utility
OP5ToOP2	601. See Utility
OP5ToOP3	601. See Utility
OP5ToOP4	601. See Utility
OP5ToOP6	601. See Utility
OP6ToOP1	601. See Utility
OP6ToOP2	601. See Utility
OP6ToOP5	601. See Utility
OutputExpr.....	182. See Display

P

PagedGet.....	537. See Memory
ParseInp	565. See Parser
PDspGrph	304. See Graphing and Drawing
PixelTest	305. See Graphing and Drawing
Plus1	451. See Math
PointCmd	306. See Graphing and Drawing
PointOn	308. See Graphing and Drawing
PopOP1	251. See Floating Point Stack
PopOP3	251. See Floating Point Stack
PopOP5	251. See Floating Point Stack
PopReal	252. See Floating Point Stack
PopRealO1	253. See Floating Point Stack
PopRealO2	253. See Floating Point Stack
PopRealO3	253. See Floating Point Stack
PopRealO4	253. See Floating Point Stack
PopRealO5	253. See Floating Point Stack
PopRealO6	253. See Floating Point Stack
PosNo0Int	602. See Utility
PtoR.....	452. See Math
PushOP1	254. See Floating Point Stack
PushOP3	254. See Floating Point Stack
PushOP5	254. See Floating Point Stack
PushReal	255. See Floating Point Stack
PushRealO1.....	256. See Floating Point Stack
PushRealO2.....	256. See Floating Point Stack
PushRealO3.....	256. See Floating Point Stack

PushRealO4.....	256. See Floating Point Stack
PushRealO5.....	256. See Floating Point Stack
PushRealO6.....	256. See Floating Point Stack
PutC.....	183. See Display
PutMap	184. See Display
PutPS.....	185. See Display
PutS	187. See Display
PutTokString	189. See Display
PutToL	363. See List
PutToMat	485. See Matrix

R

RandInit	453. See Math
Random	454. See Math
Rcl_StatVar.....	577. See Statistics
RclAns	603. See Utility
RclGDB2.....	538. See Memory
RclIN	539. See Memory
RclSysTok.....	567. See Parser
RclVarSym.....	540. See Memory
RclX	541. See Memory
RclY	542. See Memory
Rec1stByte.....	339. See IO
Rec1stByteNC	340. See IO
RecAByteIO	341. See IO
RedimMat	543. See Memory
Regraph	309. See Graphing and Drawing
ReloadAppEntryVecs	604. See Utility
RestoreDisp	190. See Display
RName.....	455. See Math
RndGuard	456. See Math
RnFx	457. See Math
Round	458. See Math
RToD	459. See Math
RToP.....	460. See Math
RunIndicOff.....	191. See Display
RunIndicOn.....	192. See Display

S

SaveDisp.....	193. See Display
SendABYTE	342. See IO
SetAllPlots.....	310. See Graphing and Drawing
SetFuncM	311. See Graphing and Drawing
SetNorm_Vals.....	194. See Display
SetParM.....	312. See Graphing and Drawing
SetPolM	313. See Graphing and Drawing
SetSeqM	314. See Graphing and Drawing
SetTblGraphDraw	315. See Graphing and Drawing
SetupPagedPtr.....	544. See Memory
SetXXOP1.....	605. See Utility
SetXXOP2.....	606. See Utility
SetXXXXOP2.....	607. See Utility
SFont_Len	195. See Display
Sin.....	461. See Math
SinCosRad.....	462. See Math
SinH.....	463. See Math
SinHCosH	464. See Math
SqRoot.....	465. See Math
SrchVLstDn, SrchVLstUp	545. See Memory
SStringLength	196. See Display
StMatEl	546. See Memory
StoAns	547. See Memory
StoGDB2.....	548. See Memory
StoN.....	549. See Memory
StoOther	550. See Memory
StoR.....	552. See Memory
StoRand.....	608. See Utility
StoSysTok.....	553. See Memory
StoT	554. See Memory
StoTheta	555. See Memory
StoX	556. See Memory
StoY	557. See Memory
StrCopy	609. See Utility
StrLength	610. See Utility

T

Tan.....	466. See Math
TanH.....	467. See Math
TanLnF.....	316. See Graphing and Drawing
TenX.....	468. See Math
ThetaName.....	469. See Math
ThreeExec.....	568. See Parser
Times2.....	470. See Math
TimesPt5.....	471. See Math
TName.....	472. See Math
ToFrac.....	473. See Math
Trunc.....	474. See Math

U

UCLineS.....	317. See Graphing and Drawing
UnLineCmd.....	318. See Graphing and Drawing
UnOPExec.....	570. See Parser

V

VertCmd.....	319. See Graphing and Drawing
VPutMap.....	197. See Display
VPutS.....	198. See Display
VPutSN.....	200. See Display
VtoWHLDE.....	320. See Graphing and Drawing

X

Xftol.....	321. See Graphing and Drawing
Xitof.....	322. See Graphing and Drawing
XName.....	475. See Math
XRootY.....	476. See Math

Y

Yftol.....	323. See Graphing and Drawing
YName.....	477. See Math
YToX.....	478. See Math

Z

Zero16D.....	479. See Math
ZeroOP.....	480. See Math
ZeroOP1.....	481. See Math
ZeroOP2.....	481. See Math

ZeroOP3	481. See Math
ZmDecml	324. See Graphing and Drawing
ZmFit.....	325. See Graphing and Drawing
ZmInt.....	326. See Graphing and Drawing
ZmPrev	327. See Graphing and Drawing
ZmSquare	328. See Graphing and Drawing
ZmStats	329. See Graphing and Drawing
ZmTrig	330. See Graphing and Drawing
ZmUsr	331. See Graphing and Drawing
ZooDefault	332. See Graphing and Drawing