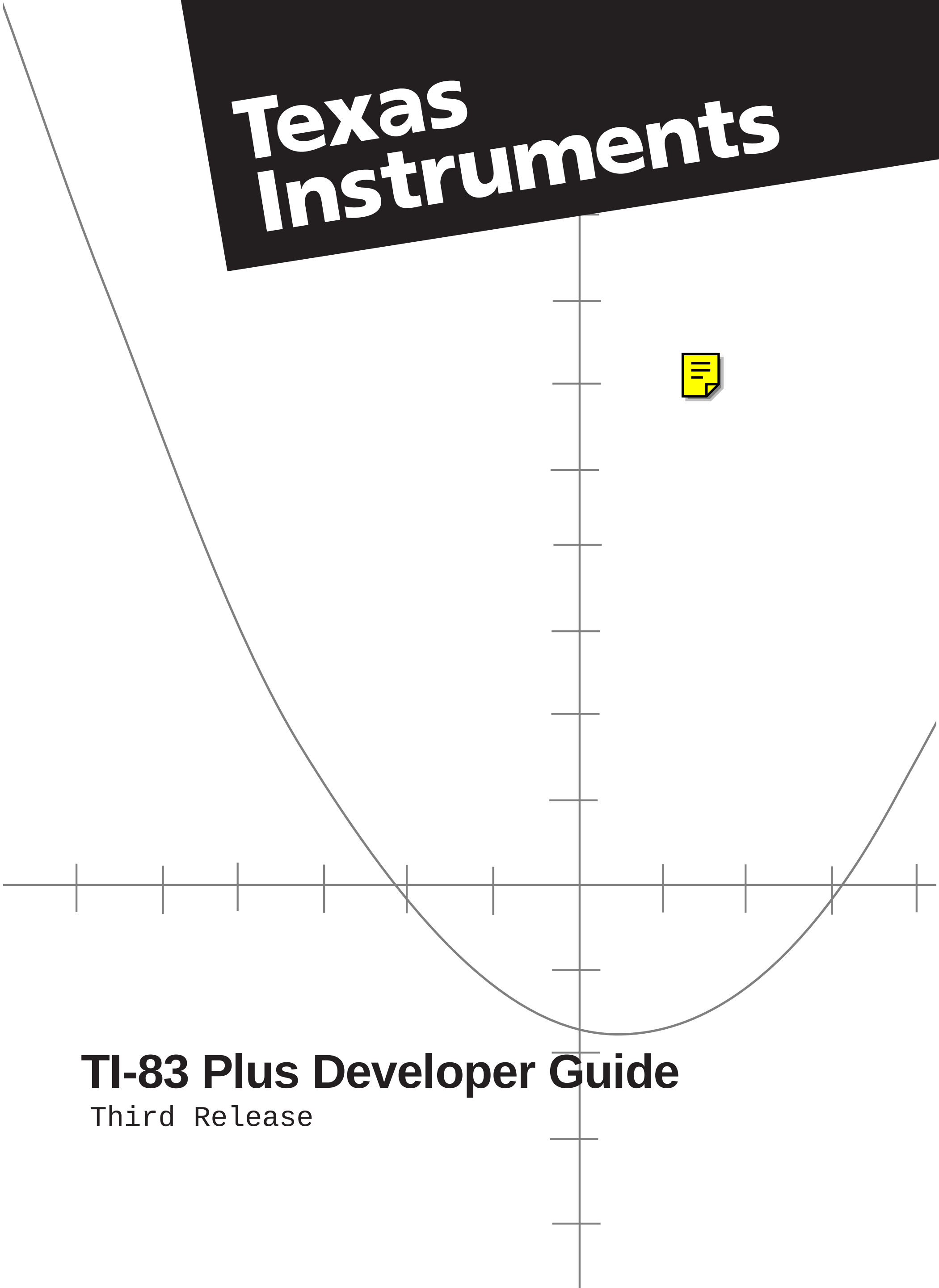
Texas Instruments



# TI-83 Plus Developer Guide

Third Release

## Important information

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

# Table of Contents

## Table of Contents (continued)

# Table of Contents (continued)

## Table of Contents (continued)

## Table of Contents (continued)

# Figures

# Tables

# 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.

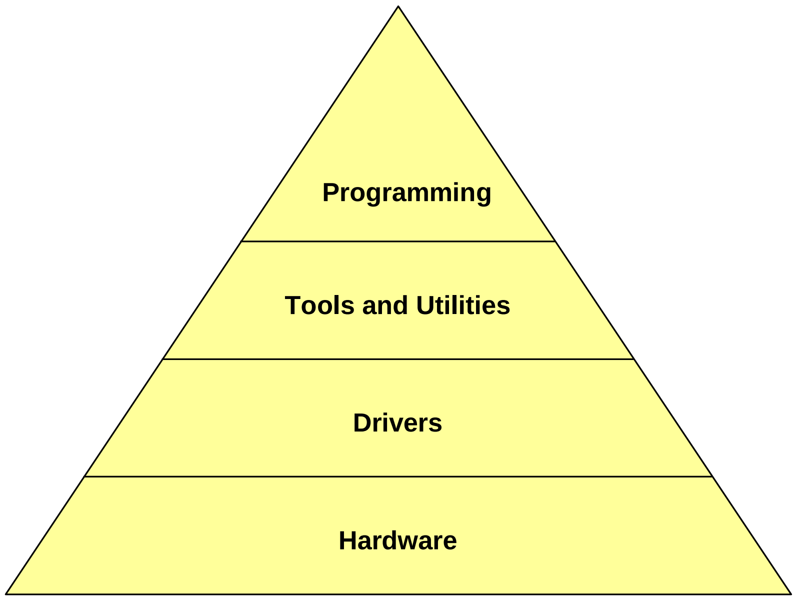


**Fig. 2.1: TI-83 Plus Architecture**

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.

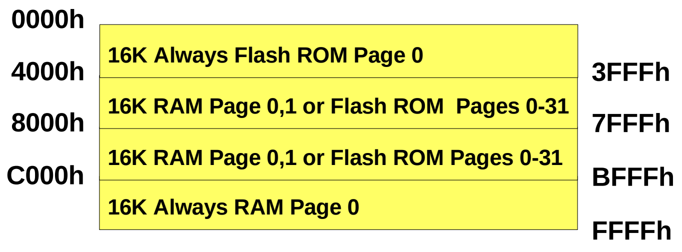

0000h
16K Always Flash ROM Page 0
3FFFh
4000h
16K RAM Page 0,1 or Flash ROM Pages 0-31
7FFFh
8000h
16K RAM Page 0,1 or Flash ROM Pages 0-31
BFFFh
C000h
16K Always RAM Page 0
FFFFh

**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.

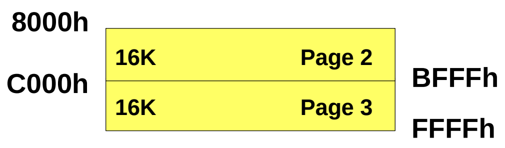


Fig. 2.2: TI-83 Plus RAM

## 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.

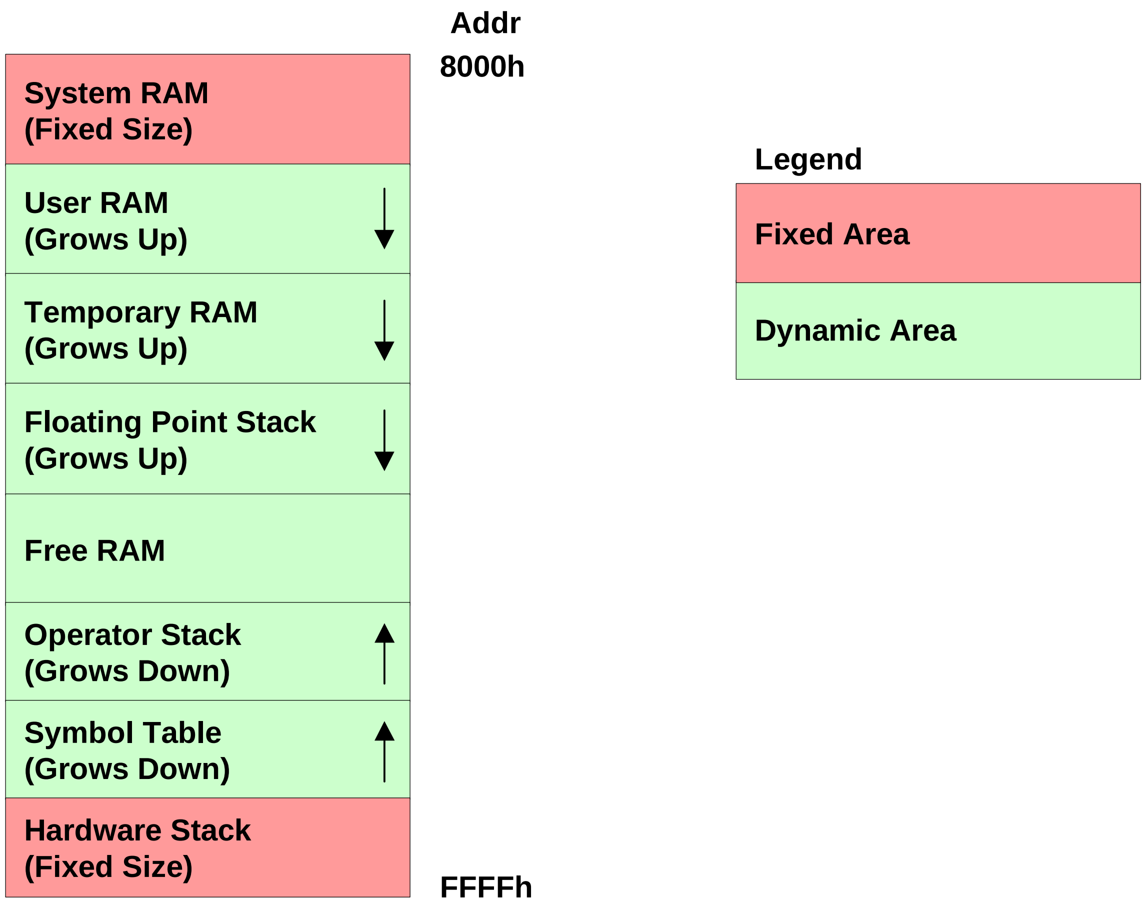


Fig. 2.4: TI-83 Plus RAM Structure

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

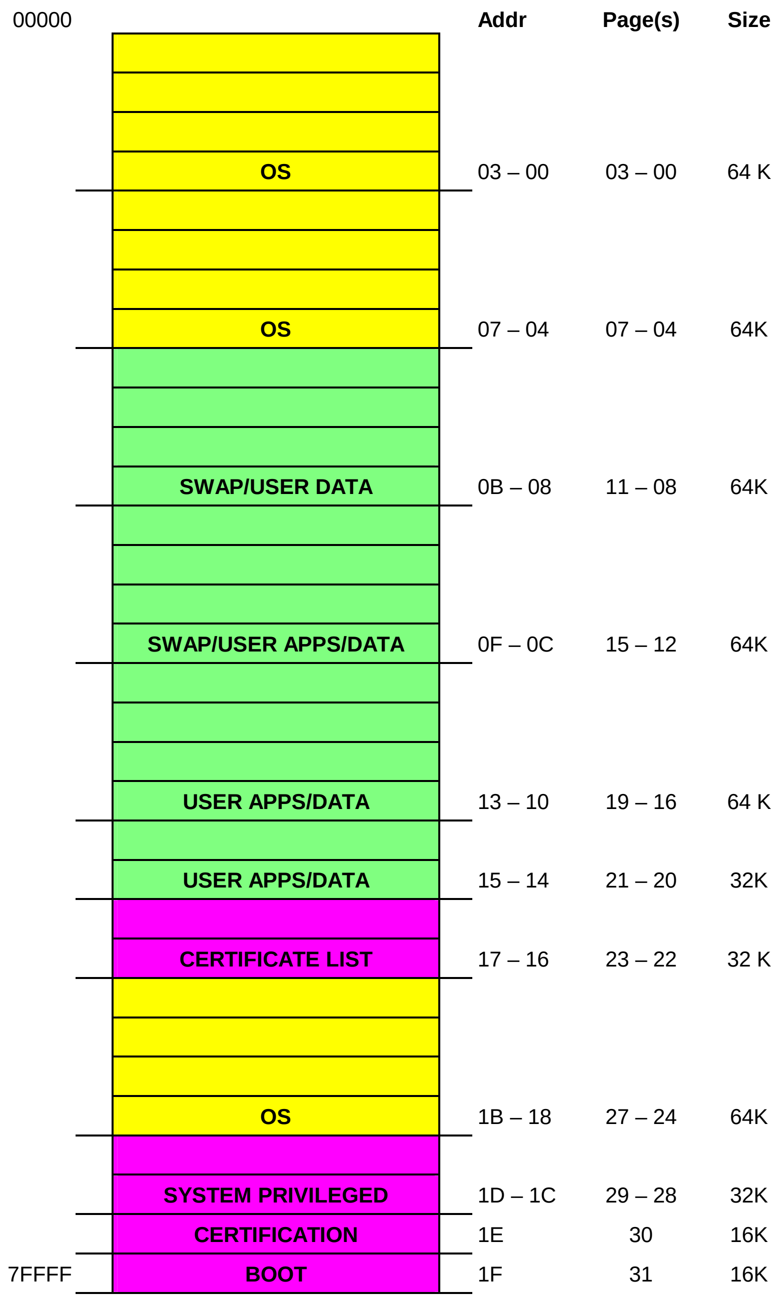
This is the area to which the Z80 Stack Pointers (SP) register points. This stack area is 400 bytes. The Hardware Stack starts at address FFFFh and it grows from high to low memory.

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.

## Flash ROM Structure

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.



| | Addr | Page(s) | Size |
|---|---|---|---|
| OS | 03 – 00 | 03 – 00 | 64 K |
| OS | 07 – 04 | 07 – 04 | 64K |
| SWAP/USER DATA | 0B – 08 | 11 – 08 | 64K |
| SWAP/USER APPS/DATA | 0F – 0C | 15 – 12 | 64K |
| USER APPS/DATA | 13 – 10 | 19 – 16 | 64 K |
| USER APPS/DATA | 15 – 14 | 21 – 20 | 32K |
| CERTIFICATE LIST | 17 – 16 | 23 – 22 | 32 K |
| OS | 1B – 18 | 27 – 24 | 64K |
| SYSTEM PRIVILEGED | 1D – 1C | 29 – 28 | 32K |
| CERTIFICATION | 1E | 30 | 16K |
| BOOT | 1F | 31 | 16K |

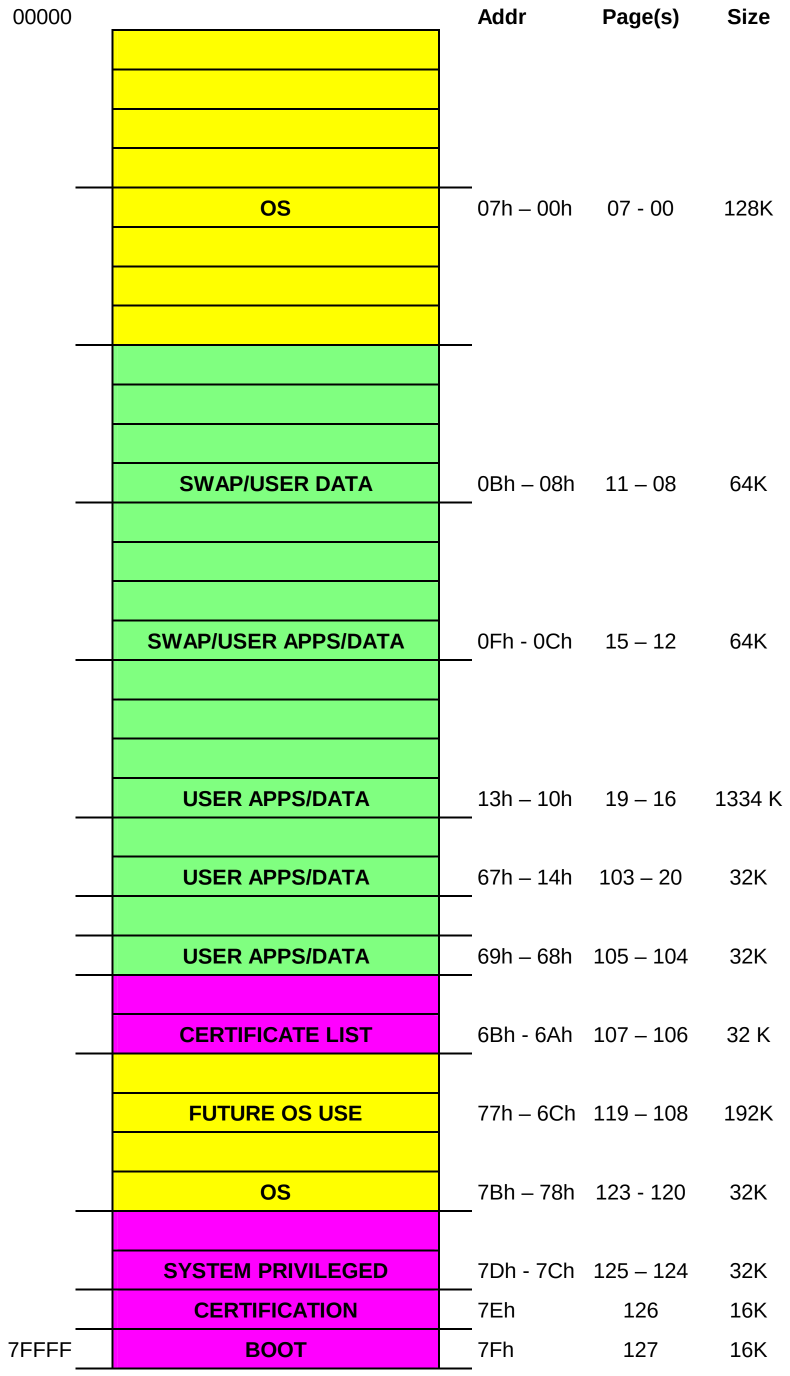
**Fig. 2.5: TI-83 Plus  Flash ROM Structure**

**Legend**

| | |
|---|---|
| 🟩 | **SWAP and/or User APPS Area** |
| 🟨 | **Update System (OS) Area** |
| 🟪 | **Fixed Area — changeable only by TI** |

The TI-83 Plus Silver Edition Flash ROM is composed of 2048K (2M) bytes divided into 128 pages, each of which is 16K bytes in size.  The structure is generally the same as the TI-83 Plus except for the inclusion of 96 additional 16K pages (24 additional 64K sectors).

The TI-83 Plus Flash structure chart (Fig. 2.5) is correct up to page 14h; at that point, the TI-83 Plus Silver Edition includes more data pages.  The TI-83 Plus Silver Edition also has OS residing at the high 8 pages of Flash, 78h…7Fh.  The TI-83 Plus high memory is 18h… 1Fh.



| | Addr | Page(s) | Size |
|---|---|---|---|
| OS | 07h – 00h | 07 - 00 | 128K |
| SWAP/USER DATA | 0Bh – 08h | 11 – 08 | 64K |
| SWAP/USER APPS/DATA | 0Fh - 0Ch | 15 – 12 | 64K |
| USER APPS/DATA | 13h – 10h | 19 – 16 | 1334 K |
| USER APPS/DATA | 67h – 14h | 103 – 20 | 32K |
| USER APPS/DATA | 69h – 68h | 105 – 104 | 32K |
| CERTIFICATE LIST | 6Bh - 6Ah | 107 – 106 | 32 K |
| FUTURE OS USE | 77h – 6Ch | 119 – 108 | 192K |
| OS | 7Bh – 78h | 123 - 120 | 32K |
| SYSTEM PRIVILEGED | 7Dh - 7Ch | 125 – 124 | 32K |
| CERTIFICATION | 7Eh | 126 | 16K |
| BOOT | 7Fh | 127 | 16K |

**Legend**

| | |
|---|---|
| 🟩 | SWAP and/or User APPS Area |
| 🟨 | Update System (OS) Area |
| 🟪 | Fixed Area — changeable only by TI |

**Fig. 2.5b: TI-83 Plus Silver Edition 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 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 and 2.5b.

## 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 the TI-83 Plus System Routine Documentation (separate document). 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 restart 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 the System Routine Documentation.

## 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<br>1 = degree angle mode | |
| **plotLoc** | plotFlags | 0 = write to display and buffer<br>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<br>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<br>1 = graph needs to be updated | |
| **grfDot** | grfDBFlags | 0 = graph connected draw mode<br>1 = graph dot draw mode | |
| **grfSimul** | grfDBFlags | 0 = sequential graph draw mode<br>1 = simultaneous graph draw mode | |
| **grfGrid** | grfDBFlags | 0 = graph mode grid off<br>1 = graph mode grid on | |
| **grfPolar** | grfDBFlags | 0 = graph — rectangular coordinates<br>1 = graph — polar coordinates | |
| **grfNoCoord** | grfDBFlags | 0 = graph coordinates off<br>1 = graph coordinates on | |
| **grfNoAxis** | grfDBFlags | 0 = graph draw axis<br>1 = graph no axis | |
| **grfLabel** | grfDBFlags | 0 = graph labels off<br>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 *break*. |
| **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<br>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<br>0 = com did not fail | |
| **apdRunning** | apdFlags | 1 = APD™ is running<br>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<br>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 the System Routine Documentation. |
| **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 the System Routine Documentation. |
| **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 the System Routine Documentation 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 | SetXXXXOP2 | | |

**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 | XRESt |
| 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.

| **9 Bytes** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **0** | **+1** | **+2** | **+3** | **+4** | **+5** | **+6** | **+7** | **+8** |
| T | EXP | DD | DD | DD | DD | DD | DD | DD |

First byte of mantissa

**Table 2.9: Floating-Point Number Format**

T = object type where:

| **Bit** | **Description** |
|---|---|
| 0 – 4 | 0 if a real variable's data,<br>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  0C 80 20 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  0C 80 50 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    element (1,1)
       00 80 20 00 00 00 00 00 00    element (1,2)
       00 80 30 00 00 00 00 00 00    element (1,3)
       00 80 40 00 00 00 00 00 00    element (2,1)
       00 80 50 00 00 00 00 00 00    element (2,2)
       00 80 60 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 = 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, TI Connect™ 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:

| **Bit** | **Flag** |
|---|---|
| 0 – 4 | Object Type |
| 5 | Future use |
| 6 | Future use |
| 7 | Future use |

* See also: Symbol Table Structure

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

| **Object Type Value** | **Object Type** | **Object Type Equate** |
|---|---|---|
| 00h | Real | RealObj |
| 01h | List | ListObj |
| 02h | Matrix | MatObj |
| 03h | Equation | EquObj |
| 04h | String | StrngObj |
| 05h | Program | ProgObj |
| 06h | Protected Program | ProtProgObj |
| 07h | Picture | PictObj |
| 08h | Graph Database | GDBObj |
| 0Bh | New EQU Obj | NewEquObj |
| 0Ch | Complex Obj | CplxObj |
| 0Dh | Complex List Obj | CListObj |
| 14h | Application Obj | AppObj |
| 15h | AppVar Obj | AppVarObj |
| 17h | 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 θ

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 θ**

| **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) |
| twn | 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 | twn 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. List names are 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 | 00 | ? |

**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.
- The graphing equations *always* exist, and therefore must be deleted before recreating them. Always create the equation immediately after deleting it to avoid system crashes.
- 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 contain 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 the System Routine Documentation 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 the System Routine Documentation.

**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 complex, OP1 = real part; OP2 = imaginary part.

**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 the System Routine Documentation.

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

```
             B_CALL      ZeroOP1       ; OP1 = 00000000000
             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 without immediately recreating them. 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 by any of the Symbol Table search routines to reflect whether or not a variable is archived. |
| **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 by any of the Symbol Table search routines to reflect whether or not a variable is archived. |
| **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                     ; remove error handler
;
             OR          A                 ; CA = 0 if successful
             RET
MatAName:
             DB          MatObj, tVarMat, tMatA, 0
;
; control comes here if memory error during create
;
Failed:
             SCF                           ; CA = 1 if not successful
             RET
```

## Archiving and Unarchiving

Applications can use the Flash archive area in the same way as users do during normal system operation. Variables can be archived - moved from RAM to the archive area. They can also be unarchived - removed from the archive area and placed into RAM. 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 condition. Be sure to check where variables are located, RAM or Archive, before using them as inputs to system routines.

- **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 the System Routine Documentation 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        Z              ; 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                 ; remove error handler
errorHand:
              RET
```

## *Related Routines*

| | |
|---|---|
| **ChkFindSym** | Searches the Symbol Table for a variable. |
| **MemChk** | Returns the amount of free RAM available. |

See the System Routine Documentation 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         rMov9ToOP1           ; OP1 = AppVar name
             B_CALL      ChkFindSym           ; 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      AppVarObj, ‘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 the System Routine Documentation 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 the System Routine Documentation 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 the System Routine Documentation 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 the System Routine Documentation 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).

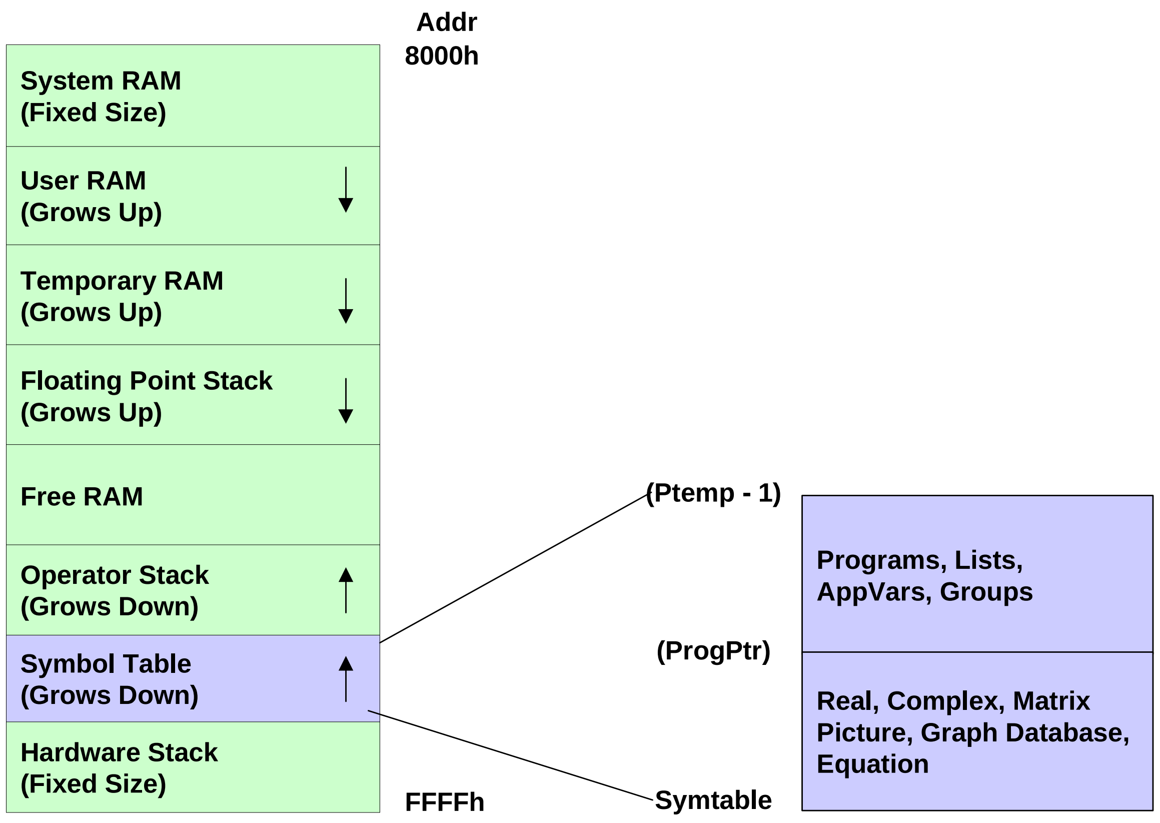

Addr
8000h
System RAM (Fixed Size)
User RAM (Grows Up)
Temporary RAM (Grows Up)
Floating Point Stack (Grows Up)
Free RAM
Operator Stack (Grows Down)
Symbol Table (Grows Down)
Hardware Stack (Fixed Size)
FFFFh
(Ptemp - 1)
(ProgPtr)
Symtable
Programs, Lists, AppVars, Groups
Real, Complex, Matrix Picture, Graph Database, Equation

**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:

| Bit | Flag |
|---|---|
| 0 – 4 | Object Type |
| 5 | Graph equation selected |
| 6 | Variable used during graphing |
| 7 | Link transfer flag |

| Object Type Value | Object Type | Object Type Equate |
|---|---|---|
| 00h | Real | RealObj |
| 01h | List | ListObj |
| 02h | Matrix | MatObj |
| 03h | Equation | EquObj |
| 04h | String | StrngObj |
| 05h | Program | ProgObj |
| 06h | Protected Program | ProtProgObj |
| 07h | Picture | PictObj |
| 08h | Graph Database | GDBObj |
| 0Bh | New EQU Obj | NewEquObj |
| 0Ch | Complex Obj | CplxObj |
| 0Dh | Complex List Obj | CListObj |
| 14h | Application Obj | AppObj |
| 15h | AppVar Obj | AppVarObj |
| 17h | 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 that of the product 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 # | ?<br>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,movetonext     ; yes, get NL to find next entry
;
             CP         ProgObj          ; current entry a program
             JR         Z,movetonext     ; yes, get NL to find next entry
;
             CP         ProtProgObj      ; current entry a program
             JR         Z,movetonext     ; yes, get NL to find next entry
;
             CP         TempProjObj      ; current entry a program
             JR         Z,movetonext     ; yes, get NL to find next entry
;
             CP         groupprogobj     ; current entry a group var
             JR         Z,movetonext     ; 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,movetonext1   ; no
Movetonext:
             LD         E,(HL)           ; DE = length of name
             INC        E                ; DE = length of name + 1
;
; move HL to next symbol table entry sign digit
;
Movetonext1:
             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 multiples of nine bytes ONLY. This entry size is the size of a floating-point number.

This does not mean that only floating-point numbers may be pushed onto the stack. The content of the nine bytes, in most cases, can be random data. 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.

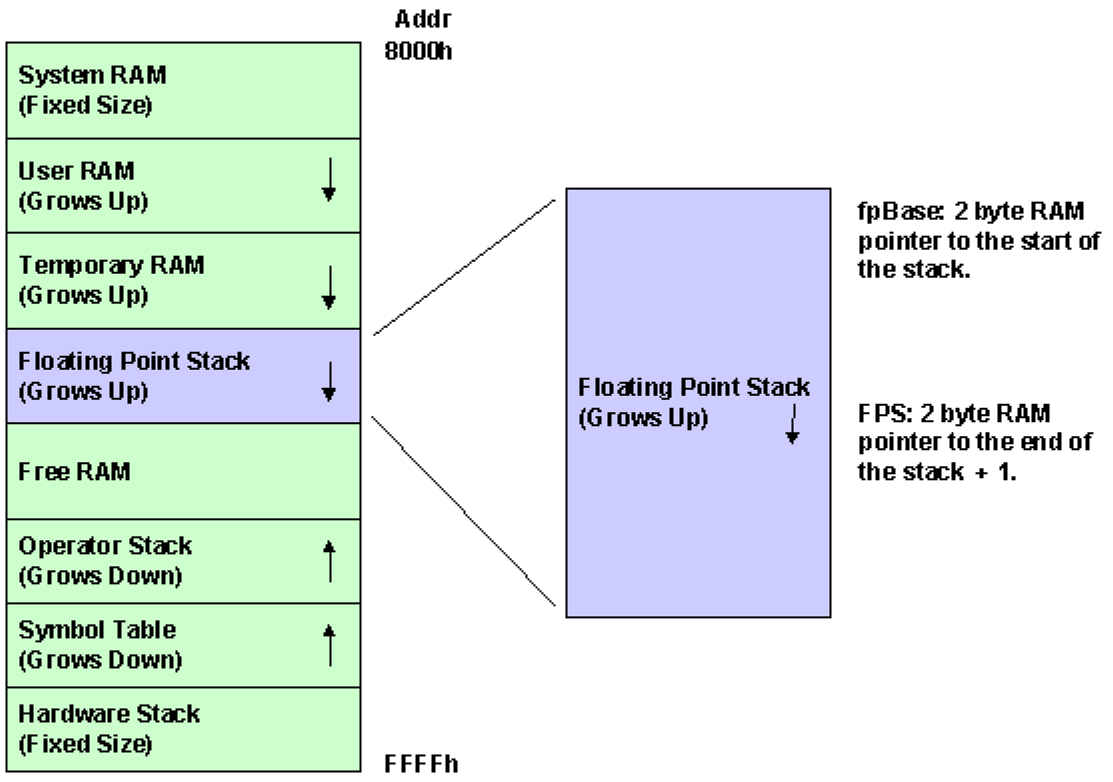

Addr
8000h
System RAM (Fixed Size)
User RAM (Grows Up)
Temporary RAM (Grows Up)
Floating Point Stack (Grows Up)
Free RAM
Operator Stack (Grows Down)
Symbol Table (Grows Down)
Hardware Stack (Fixed Size)
FFFFh
Floating Point Stack (Grows Up)
fpBase: 2 byte RAM pointer to the start of the stack.
FPS: 2 byte RAM pointer to the end of the stack + 1.

**Fig. 2.7: TI-83 Plus System RAM**

## Naming Convention

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

**FPS** = **F**loating **P**oint **S**tack

**FPST** = **F**loating **P**oint **S**tack **T**op. This is the last nine bytes of the FPS.

**FPS1** = **F**loating **P**oint **S**tack 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.

<u>Address</u>

```
(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, it will 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.

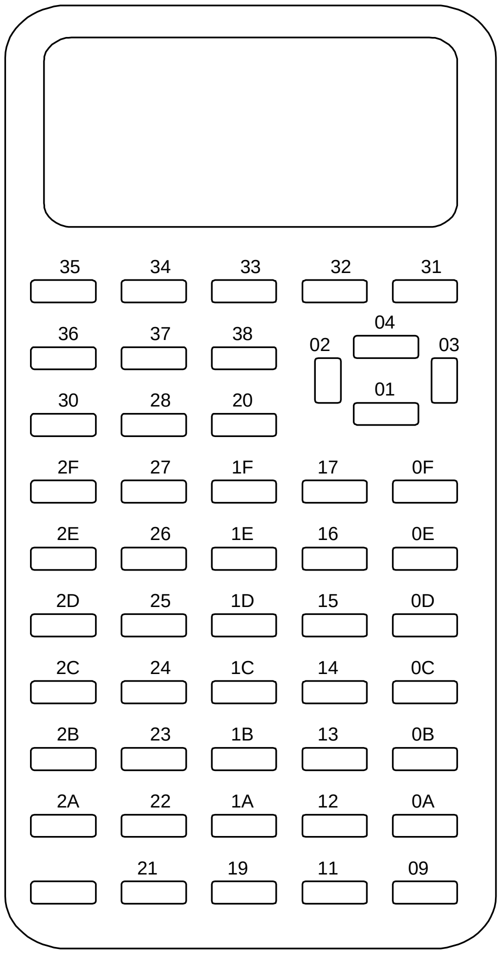


**Fig. 2.8: Calculator Scan Code**

**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 = $\boxed{\texttt{ON}}$ 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 GRAPH LINK™ or TI Connect™ 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 $\boxed{\text{ENTER}}$ or $\boxed{\text{ON}}$ 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, points, lines, etc.
- 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.

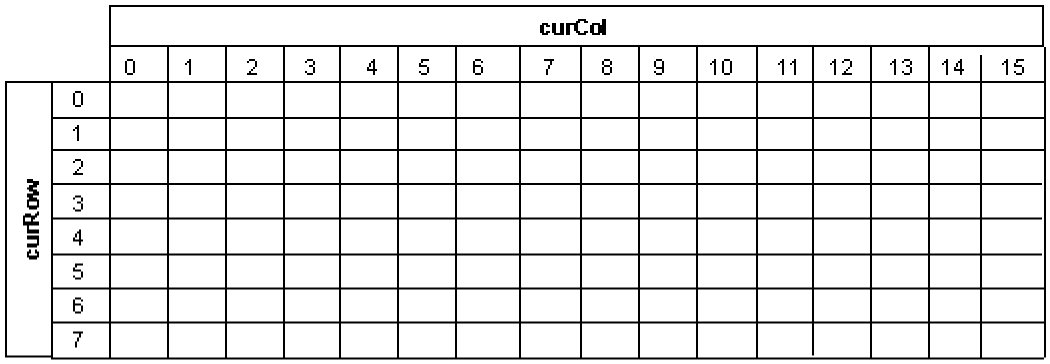


**Fig. 2.9: Home Screen Display Mapping**

- **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. As characters are sent to the display, character font codes will be written to corresponding locations in this buffer. 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 **PutS** routine can be used without first copying strings to RAM by coding a local version of the routine in the application. See the System Routine Documentation for the source code to this routine.

See the Display Utility Routines section.
See the Formatting Numeric Values for Display section.
See the System Routine Documentation 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.

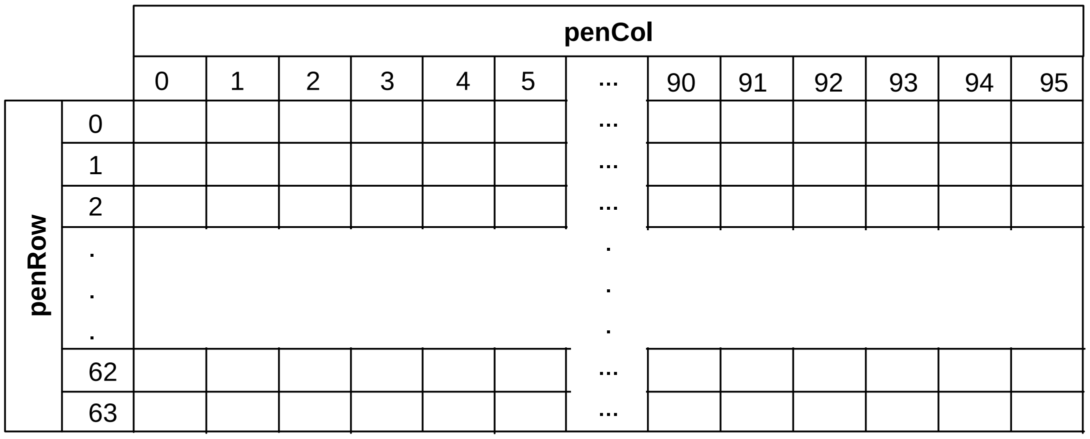


**Fig. 2.10: Pen Display Mapping**

– **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 coding a local version of the routines in the application. See the System Routine Documentation 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 the System Routine Documentation 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^θ**i 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 the System Routine Documentation 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.

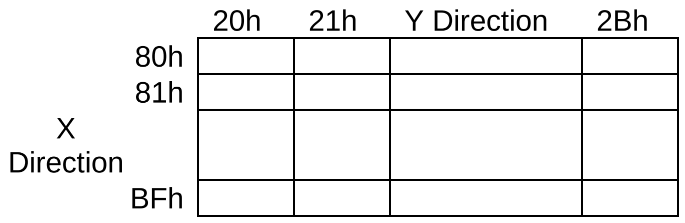


**Fig. 2.11: Command Values**

  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.

– The LCD has a delay requirement of approximately 10us between operations. The following routine should provide adequate delay on the TI-83 Plus (not Silver Edition).

```
lcd_busy:
             PUSH        AF
             INC         HL
             DEC         HL
             POP         AF
             RET
```

– If the application is run on the Silver Edition at fast speed, the above routine will not provide a long enough delay.  There are three options for solving this problem.

- Triple or quadruple the delay time of the in-line code.  This will solve the problem, but it may reoccur if another faster version is produced.
- Do B_CALL LCD_BUSY.  This is guaranteed to work, but may slow down a display intensive application.
- Use a CALL LCD_BUSY_QUICK, where LCD_BUSY_QUICK is equated to 000Bh.  This is a new entry point that does not require the system overhead of a B_CALL.  This call also works on earlier TI-83 Plus versions, but runs slightly faster than the required 10us and modifies the z/nz status flag.  To use this on all versions, wrap it in another routine that saves and restores the flag register.

```
lcd_busy_2:
             PUSH        AF
             CALL        LCD_BUSY_QUICK   ; = 000Bh
             POP         AF
             RET
```

This will ensure that the routine runs on both the TI-83 Plus and Silver Edition with minimal additional time delays.

– 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_2
OUT         (lcdinstport),A
```

Bottom Row

```
LD          A,0BFh              ; last row
CALL        lcd_busy_2
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_2
OUT         (lcdinstport),A
```

  Last byte of row

```
LD          A,2Bh               ; last byte of row
CALL        lcd_busy_2
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_2
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_2
             OUT         (lcdinstport),A     ; set X to row 5
;
             LD          A,07h
             CALL        lcd_busy_2
             OUT         (lcdinstport),A     ; set Y auto increment mode
;
             CALL        lcd_busy_2
             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_2
             IN          A,(lcddataport)     ; dummy read since we changed
                                             ; X, Y position
Loop:
             CALL        lcd_busy_2
             IN          A,(lcddataport)     ; read byte, auto increment Y
;
             LD          (HL),A
             INC         HL
             DJNZ        Loop
;
             LD          A,05h
             CALL        Lcd_busy_2
             OUT         (lcdinstport),A     ; set X auto increment mode
```

- Writing to the display driver RAM

```
             CALL        lcd_busy_2
             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_2
             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_2
             OUT         (lcdinstport),A    ; set new x
             LD          A,20h
             CALL        lcd_busy_2
             OUT         (lcdinstport),A    ; set to first column
             LD          B,12               ; 12 columns
loop2:
             LD          A,(HL)             ; get source
             INC         HL
             CALL        lcd_busy_2
             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_2
             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_2           ; delay for LCD driver
             OUT         (lcdinstport),A      ; set contrast
             RET
```

> **Note**: Adjusting the contrast in this manner 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 (Xmin, Xmax, Ymin, Ymax) 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. 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 the System Routine Documentation.

| | |
|---|---|
| **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, ***plotSScreen***. The portion of the buffer cleared is determined by the split mode setting. |
| **GrBufCpy** | Displays the graph backup buffer, ***plotSScreen***. 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 stat plots. |
| **SetTblGraphDraw** | Sets the graph to dirty, which causes a complete regraph the next time the graph is brought to the display. |

## Stat Plots

Stat plots provide a way to display data stored in list variables. The SetAllPlots routine will select or deselect all stat plots. Each stat plot has a portion of System RAM allocated to store its settings. To select/deselect or change settings for an individual stat plot, you must modify this RAM.

There are three bytes that determine if a plot is on or off:

| | |
|---|---|
| **P1FrqOnOff** | Plot 1, 0 = off; 1 = on |
| **P2FrqOnOff** | Plot 2, 0 = off; 1 = on |
| **P3FrqOnOff** | Plot 3, 0 = off; 1 = on |

The high 4 bits of these bytes determine which axis the data will be plotted on if the plot type is Normal Probability Plot. 0 = X axis, 1 = Y axis.

There are three bytes that determine the type of plot to be drawn:

| | |
|---|---|
| **P1Type** | Plot 1 type |
| **P2Type** | Plot 2 type |
| **P3Type** | Plot 3 type |

0 = Scatter Plot

1 = XY Line

2 = Modified Box Plot

3 = Histogram

4 = Box Plot

5 = Normal Probability Plot

Like the on/off bytes, the type bytes have a second purpose.  The high four bits of the type bytes determine the mark or icon used in the stat plot.

0 = Box icon

1 = Cross icon

2 = Dot icon

Each stat plot has three five-byte locations to store the names of lists used in the plot. The list names do not include tVarLst, and must be zero-terminated if less than five bytes.

| | |
|---|---|
| **SavX1List** | Plot 1 X list |
| **SavY1List** | Plot 1 Y list |
| **SavF1List** | Plot 1 Frequency List |
| **SavX2List** | Plot 2 X list |
| **SavY2List** | Plot 2 Y list |
| **SavF2List** | Plot 2 Frequency List |
| **SavX3List** | Plot 3 X list |
| **SavY3List** | Plot 3 Y list |
| **SavF3List** | Plot 3 Frequency List |

Split screen settings will affect how plots are drawn.  System errors will be generated if the plots are not set up correctly.

## 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.

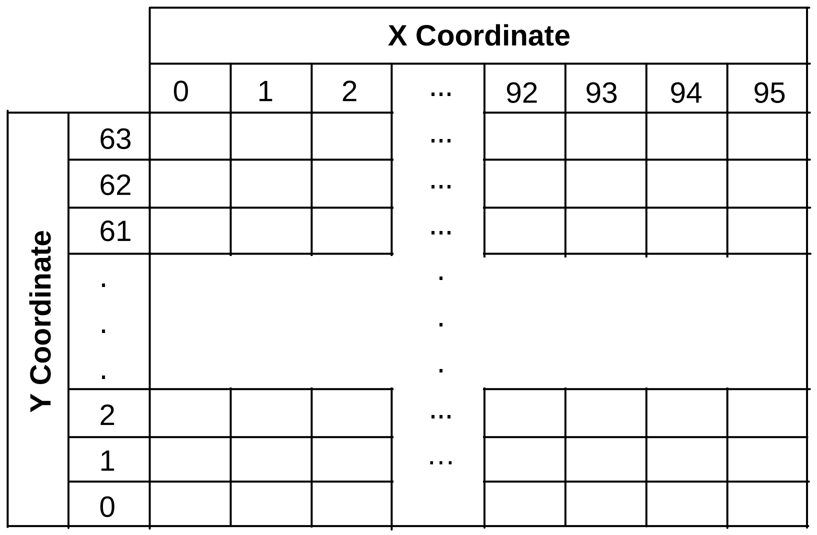


**Fig. 2.12: Pixel Coordinates**

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. The System Routine Documentation 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***.<br>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***. The System Routine Documentation 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 ***plotSScreen*****,** 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** | 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.<br>Also returns the bit number in that byte for that pixel.<br>Additionally, computes the row and column commands to set the LCD driver to the display byte for that pixel. |

## 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.

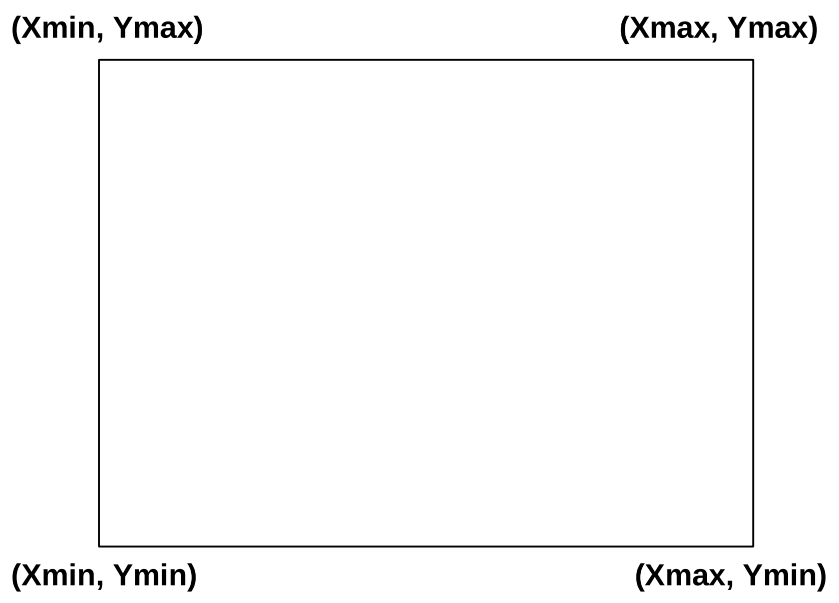


**Fig. 2.13: Graph WINDOW Setting**

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 the System Routine Documentation 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 selected 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 ***plotSScreen*** 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** | $\Delta$X 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 $\Delta$X 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 the System Routine Documentation for more details.

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

```
             CALL        l1name                      ; L1
             RES         onInterrupt,(IY+onFlags)    ; clear break
             B_CALL      SendVarCmd                  ; send L1 to start up
                                                     ; CBL
             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        l1name                      ; 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        l1name
             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
;
L1name:
             LD          HL,L1name
             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 $\boxed{\text{ON}}$ 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 $\boxed{\text{ON}}$ 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 the System Routine Documentation 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
dblp1:
              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, dblp1                  ; jump if debounce not done
;
              AppOnErr    Linkfail                   ; set error handler
              SET         indicOnly,(IY+indicFlags)  ; no key scan
              B_CALL      RecAByteIO
EndexIO:
              RES         indicOnly,(IY+indicFlags)  ; read the byte
              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
;
```

**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.

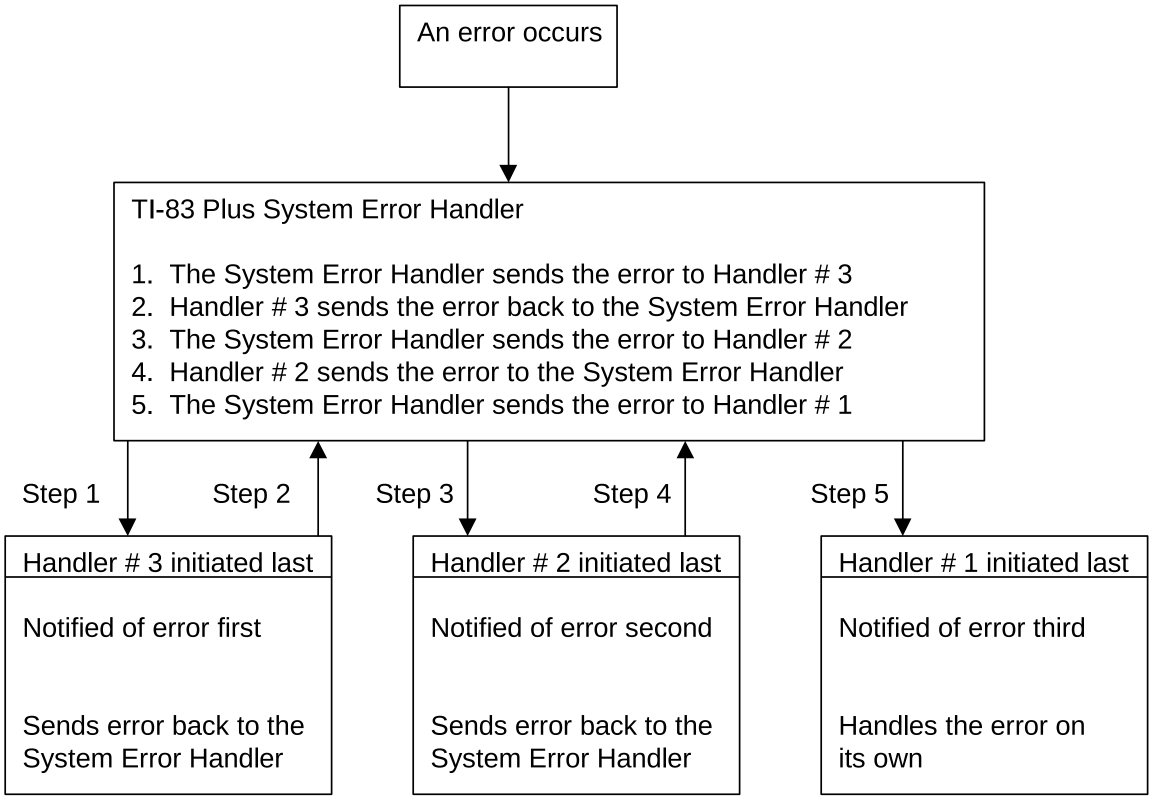

An error occurs
TI-83 Plus System Error Handler
1. The System Error Handler sends the error to Handler # 3
2. Handler # 3 sends the error back to the System Error Handler
3. The System Error Handler sends the error to Handler # 2
4. Handler # 2 sends the error to the System Error Handler
5. The System Error Handler sends the error to Handler # 1
Step 1
Step 2
Step 3
Step 4
Step 5
Handler # 3 initiated last
Notified of error first
Sends error back to the System Error Handler
Handler # 2 initiated last
Notified of error second
Sends error back to the System Error Handler
Handler # 1 initiated last
Notified of error third
Handles the error on its own

**Fig. 2.14: Error Flow**

See the System Routine Documentation 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 the System Routine Documentation, 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 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 (rectangular coordinates) and re^θI (polar coordinates), 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 the System Routine Documentation, 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 | SquareRoot (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)^ 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 = abs(OP1/OP2) |
| Conj | Conj(OP1/OP2) |
| Angle | OP1 = Angle(OP1/OP2) |
| CIntgr | Intgr(OP1/OP2) |
| CTrunc | integer part(OP1/OP2) |
| CFrac | fractional part(OP1/OP2) |
| RToP | (OP1/OP2) rectangular to polar |
| PToR | (OP1/OP2) polar to rectangular |
| ATan2 | OP1 — ATan2(OP1/OP2) where OP1 = imaginary part, OP2 = 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

| | |
|---|---|
| Sin(L1) | Sine of list L1 |
| 4 * [A] | 4 times matrix [A] |
| (1+2i) + L3 | 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 the System Routine Documentation 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
;
y1Name:
             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:

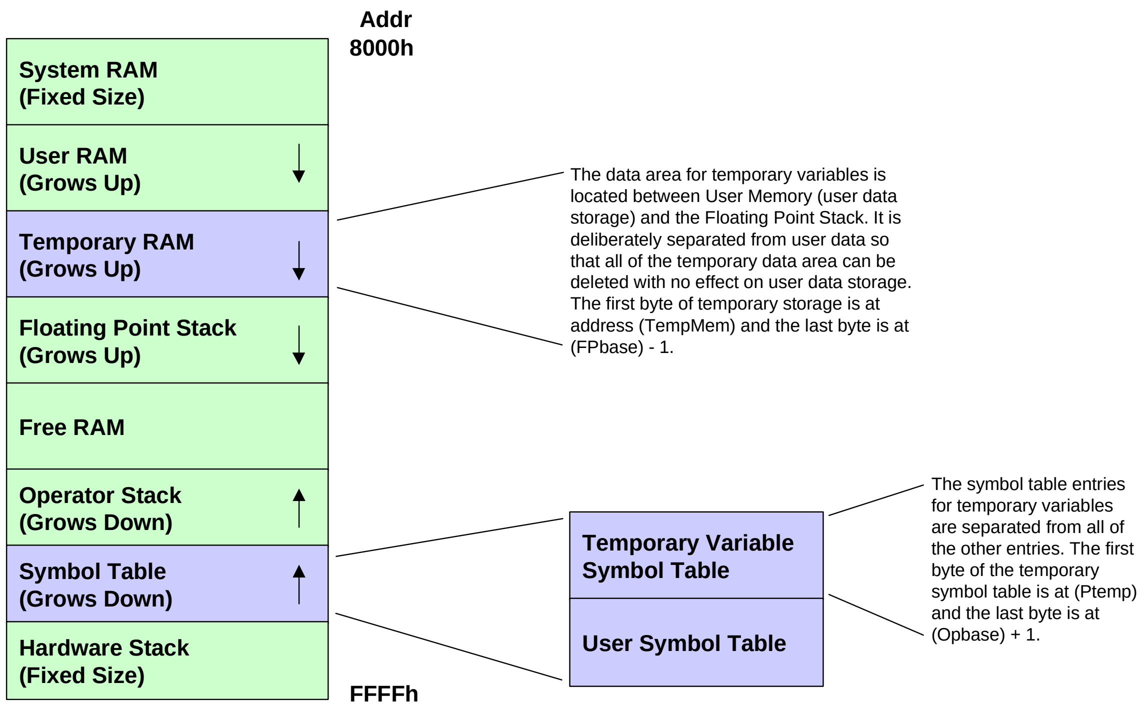
| **OP1** | **+1** | **+2** | **+3** | **+4** | **+5** | **+6** | **+7** | **+8** |
|---|---|---|---|---|---|---|---|---|
| ListObj 01h | $ 24h | 04h | 00h | ? | ? | ? | ? | ? |

**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.


Addr
8000h
System RAM (Fixed Size)
User RAM (Grows Up)
Temporary RAM (Grows Up)
Floating Point Stack (Grows Up)
Free RAM
Operator Stack (Grows Down)
Symbol Table (Grows Down)
Hardware Stack (Fixed Size)
FFFFh
Temporary Variable Symbol Table
User Symbol Table

The data area for temporary variables is located between User Memory (user data storage) and the Floating Point Stack. It is deliberately separated from user data so that all of the temporary data area can be deleted with no effect on user data storage. The first byte of temporary storage is at address (TempMem) and the last byte is at (FPbase) - 1.

The symbol table entries for temporary variables are separated from all of the other entries. The first byte of the temporary symbol table is at (Ptemp) and the last byte is at (Opbase) + 1.

**Fig. 2.15: TI-83 Plus System RAM**

## Using Temporary Variables

Temporary variables can be used the 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 System Routine 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 if 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 home 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 **StoAns** 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.

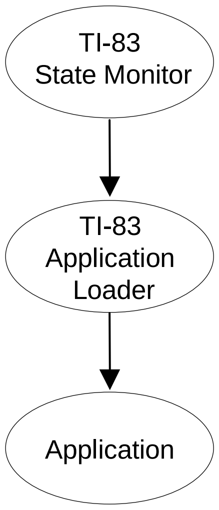


**Fig. 2.16: Control Flow**

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.

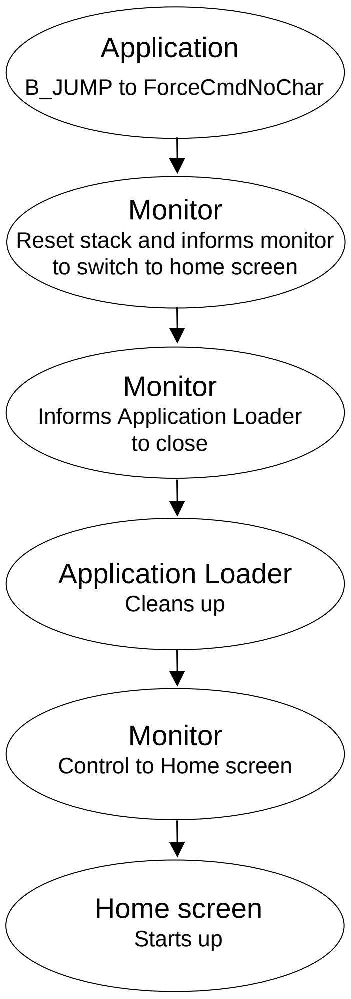


**Fig. 2.17: Event Sequence**

## 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.

**Note:** The Put Away cannot be stopped by the application. Once notified by the monitor, the application must terminate.

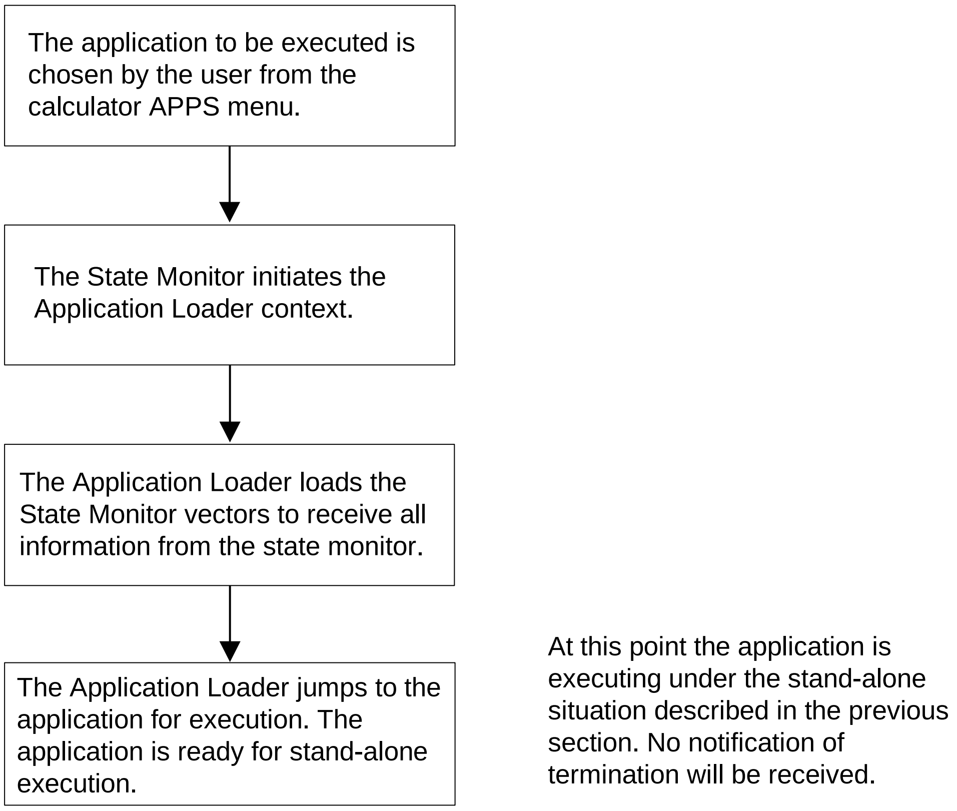
### How is the application notified?

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.



The application to be executed is chosen by the user from the calculator APPS menu.

The State Monitor initiates the Application Loader context.

The Application Loader loads the State Monitor vectors to receive all information from the state monitor.

The Application Loader jumps to the application for execution. The application is ready for stand-alone execution.

At this point the application is executing under the stand-alone situation described in the previous section. No notification of termination will be received.

**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.
;
             RES        plotLoc, (IY+plotFlags)    ; draw to display & buffer
             RES        textWrite, (IY+sGrFlags)   ; small font written to
                                                   ; display
; 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, TurningOff             ; 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.
;
             LD         A, iall                    ; all interrupts on
             OUT        (intrptEnPort), A
             B_CALL     LCD_DRIVERON               ; turn on LCD
             SET        onRunning, (IY+onFlags)    ; on interrupt running
             EI                                    ; enable interrupts

             B_JUMP     JForceCmdNoChar            ; force to home screen
;
TurningOff:
             B_JUMP     Putaway                    ; force App loader to do its
                                                   ; put away
```

# 3 Application Development Process

The following chart provides an overview of the steps necessary to create a TI-83 Plus application. A simple application is used to walk you through the detailed steps. Use the chart as a general guide. This process assumes that you are running Windows 95 operating system and that you have access to a text editor such as Notepad.

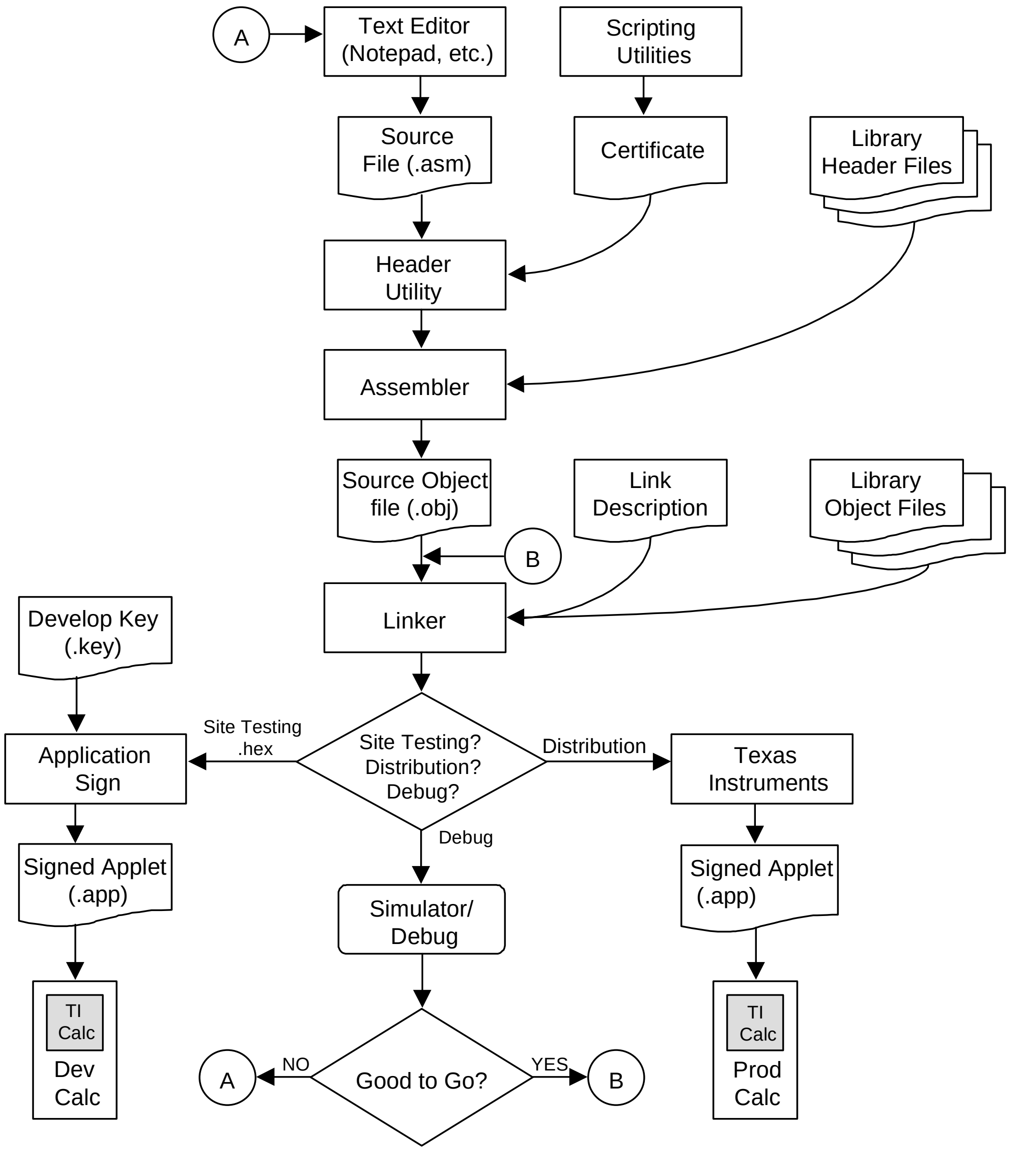


**Fig. 3.1: Application Development Flow**

## 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 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 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 digital signature. All Flash apps to be distributed must be digitally signed before they will be accepted by the operating system. Applications can be signed as freeware or authenticated applications. Freeware applications can run on any TI-83 Plus or Silver Edition calculator. The 0104.key file and Wappsign utility are provided with the SDK and can be used to sign applications as freeware. Authenticated applications require a certificate on the calculator and must be signed by TI.

## 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 address 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.

Texas Instruments has provide the AppHeader utility to aid in the creation of multiple page applications. You can download the AppHeader utility and User's Guide from http://education.ti.com/developers.

## 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

#### 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**.**

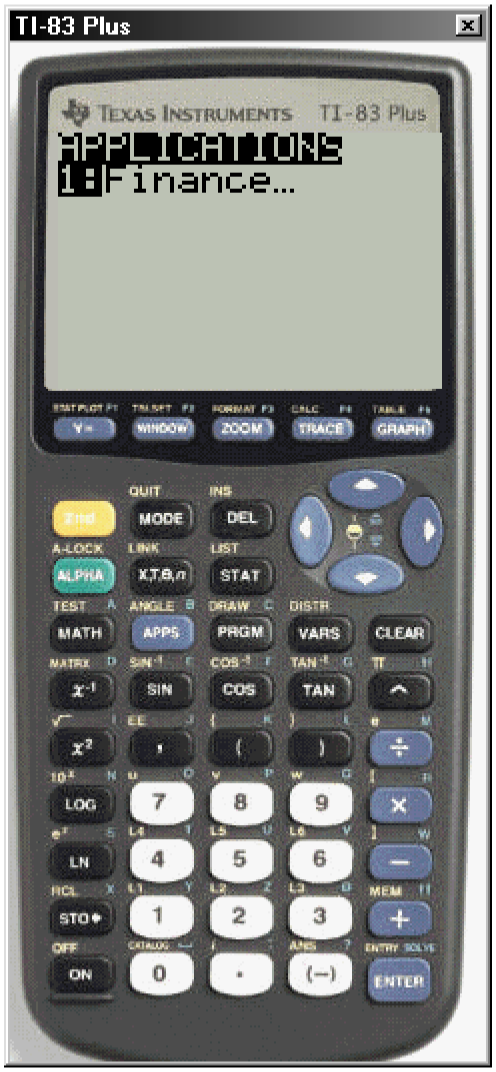
## Building the Application

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)
   ```

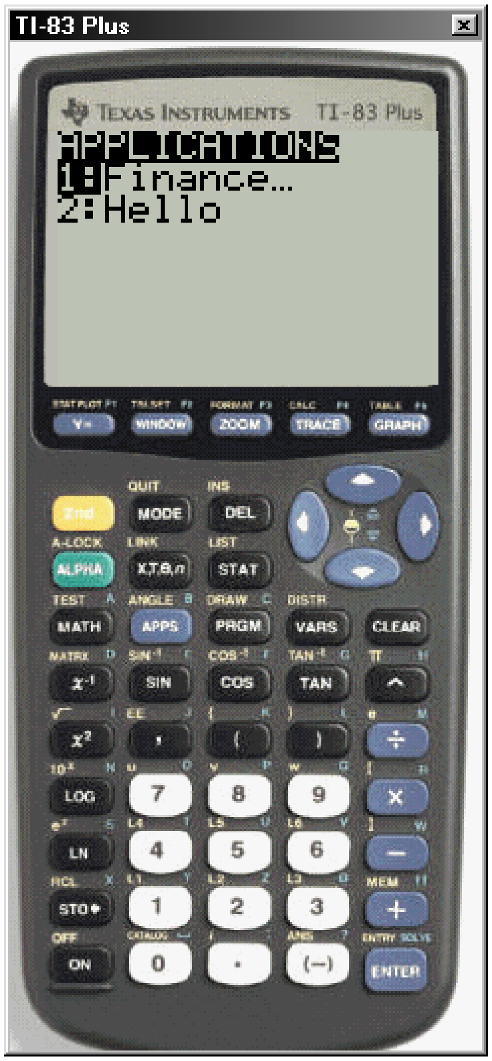
## Loading the Application into the Simulator

1. Start the TI Flash Debugger.
2. Select File, and then New, then TI-83 Plus.
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**.



```
APPLICATIONS
1:Finance…
2:Hello
```

Next:

1. Click the **2** 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.

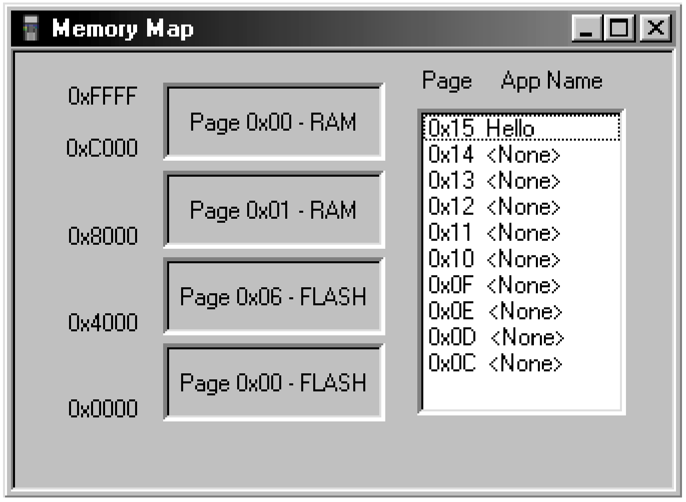
## Debugging the Application

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.

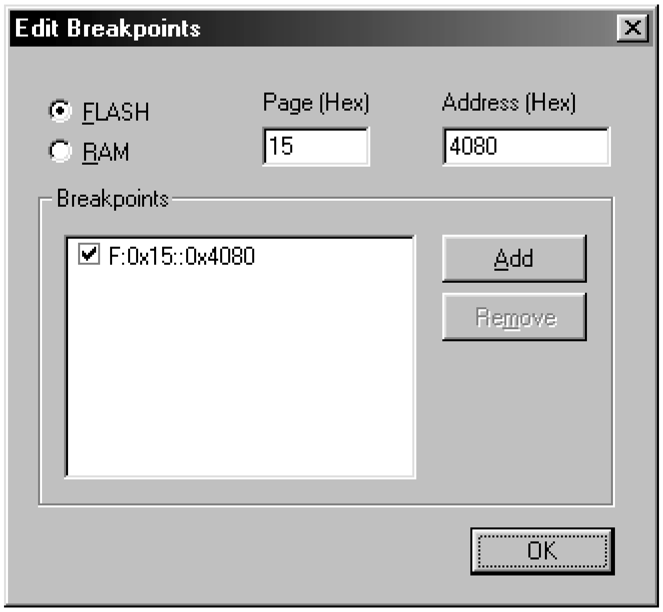
This view shows us that the Hello application is on page 0x15 of Flash.

1. Select Debug, and then Breakpoints.
2. Update the Edit Breakpoints dialog box so that it looks like the following:



| | |
|---|---|
| 0xFFFF | |
| | Page 0x00 - RAM |
| 0xC000 | |
| | Page 0x01 - RAM |
| 0x8000 | |
| | Page 0x06 - FLASH |
| 0x4000 | |
| | Page 0x00 - FLASH |
| 0x0000 | |

| Page | App Name |
|---|---|
| 0x15 | Hello |
| 0x14 | <None> |
| 0x13 | <None> |
| 0x12 | <None> |
| 0x11 | <None> |
| 0x10 | <None> |
| 0x0F | <None> |
| 0x0E | <None> |
| 0x0D | <None> |
| 0x0C | <None> |



[✓ FLASH] RAM

Page (Hex): 15

Address (Hex): 4080

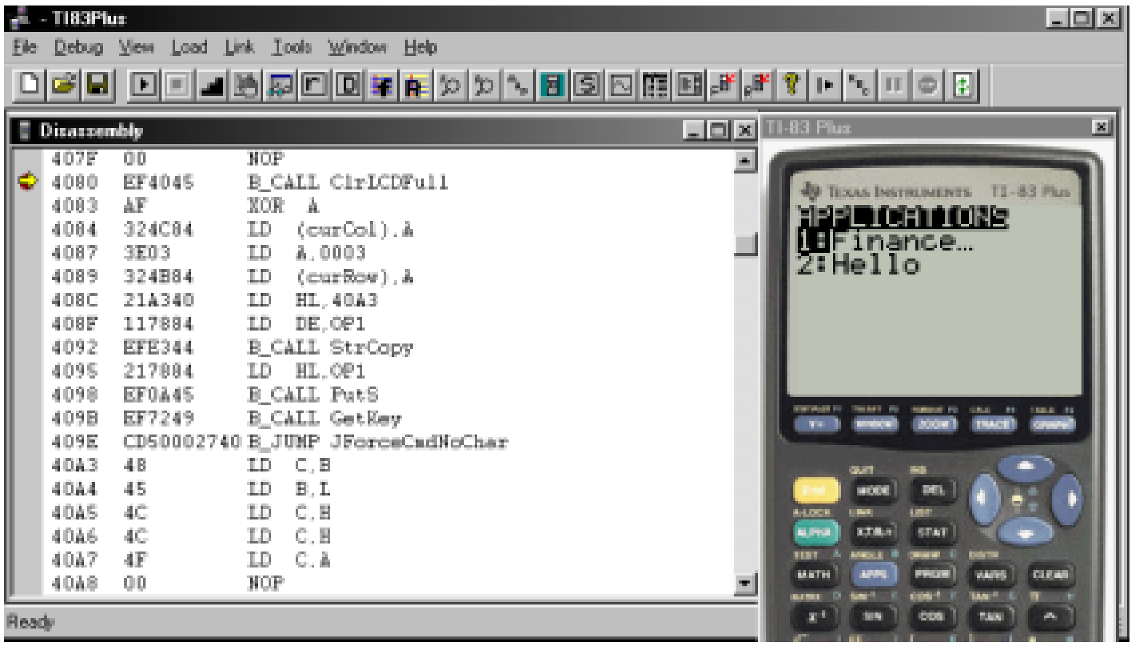
Breakpoints

☑ F:0x15::0x4080

**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:

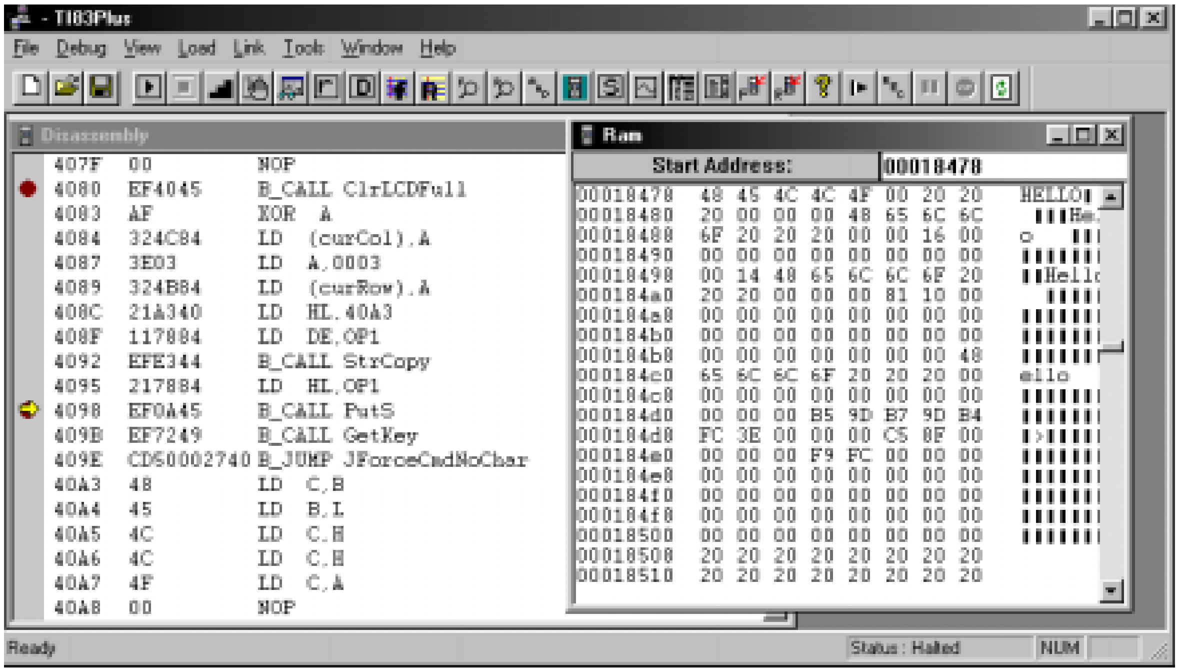
1. Click OK to exit the Edit Breakpoints dialog.
2. Select Debug, and then Go.
3. Click on the APPS key of the calculator. Note that the Status of the Debugger is Running.
4. Click on the **2** key of the calculator. The status of the Debugger will change to Halted when the breakpoint is reached.



```
407F  00         NOP
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.40A3
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
40A3  48         LD   C.B
40A4  45         LD   B.L
40A5  4C         LD   C.H
40A6  4C         LD   C.H
40A7  4F         LD   C.A
40A8  00         NOP
```

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 Address field enter OP1.



```
407F   00          NOP
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,40A3
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   CD60002740 B_JUMP JForceCmdNoChar
40A3   48          LD   C,B
40A4   45          LD   B,L
40A5   4C          LD   C,H
40A6   4C          LD   C,H
40A7   4F          LD   C,A
40A8   00          NOP
```

```
Start Address: 00018478

00018478   48 45 4C 4C 4F 00 20 20    HELLO▮
00018480   20 00 00 00 48 65 6C 6C     ▮▮▮Hel
00018488   6F 20 20 20 00 00 16 00    o   ▮▮▮
00018490   00 00 00 00 00 00 00 00    ▮▮▮▮▮▮▮
00018498   00 14 48 65 6C 6C 6F 20    ▮▮Hello
000184a0   20 20 00 00 00 81 10 00      ▮▮▮▮▮
000184a8   00 00 00 00 00 00 00 00    ▮▮▮▮▮▮▮
000184b0   00 00 00 00 00 00 00 00    ▮▮▮▮▮▮▮
000184b8   00 00 00 00 00 00 00 48    ▮▮▮▮▮▮▮
000184c0   65 6C 6C 6F 20 20 20 00    ello
000184c8   00 00 00 00 00 00 00 00    ▮▮▮▮▮▮▮
000184d0   00 00 00 B5 9D B7 9D B4    ▮▮▮▮▮▮▮
000184d8   FC 3E 00 00 00 C5 0F 00    ▮>▮▮▮▮▮
000184e0   00 00 00 F9 FC 00 00 00    ▮▮▮▮▮▮▮
000184e8   00 00 00 00 00 00 00 00    ▮▮▮▮▮▮▮
000184f0   00 00 00 00 00 00 00 00    ▮▮▮▮▮▮▮
000184f8   00 00 00 00 00 00 00 00    ▮▮▮▮▮▮▮
00018500   00 00 00 00 00 00 00 00    ▮▮▮▮▮▮▮
00018508   20 20 20 20 20 20 20 20
00018510   20 20 20 20 20 20 20 20
```

Finally:

1. Change byte 18479 from 45 E to 4F O, 18480 from 4C L to 57 W, 18481 from 4C L to 44 D and 18482 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 **2** 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.

1. Select Debug, and then Stop.
2. Select View, and then Flash.
3. In the Start Address field enter 1540A3. The application is on page 0x15. If we look at the hello.lst file, we will see that the Hello string begins at offset 40A3.
4. Change the byte at address 1540A3 to 0x53, 1540A4 to 0x54, 1540A5 to 0x41, 1540A6 to 0x52 and 1540A7 to 0x53.
5. Select Debug, and then Go.
6. Click the APPS key on the calculator.
7. Click the **2** key on the calculator.
8. The calculator will display STARS (as in the Dallas Stars, the 1999 Stanley Cup Champions) each time the application runs.
9. Select File, and then Close to close the debug session. A dialog box will appear asking if you want to save changes.
10. Click the Yes button.
11. The Save As dialog box will appear. Save debug session to C:\Mydemo\mydemo.83d.
12. Select File, and then Exit to exit the Debugger.

## Signing the Application

Texas Instruments has provided the Wappsign (Windows appsign) utility to allow you to easily sign your applications. Please refer to the Wappsign User's Guide for more information.

## Downloading the Application

You can use the TI GRAPH LINK™ program or TI Connect™ to download the app to the calculators.

# 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 3.62 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. Download the installer 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

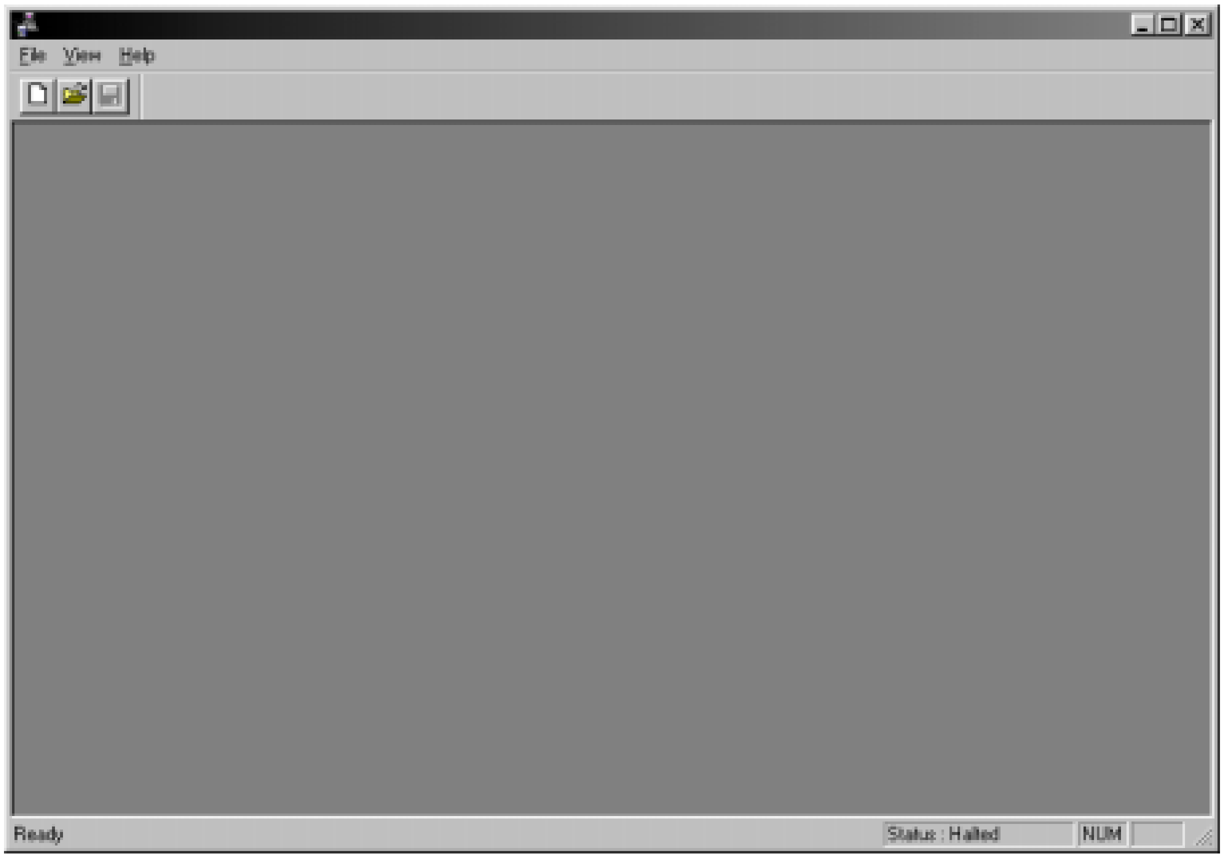
The TI-83 Plus simulator provides the capability to simulate the TI-83 Plus calculator to allow debugging of applications. The following is a detailed description of the various menu options, screens, and operations.

## Installation

To install the TI Flash Debugger, run the installation file that has been furnished with the SDK package.

## Getting Started

Click on Start, then Programs, then TI-83 Plus Flash Debugger. 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).

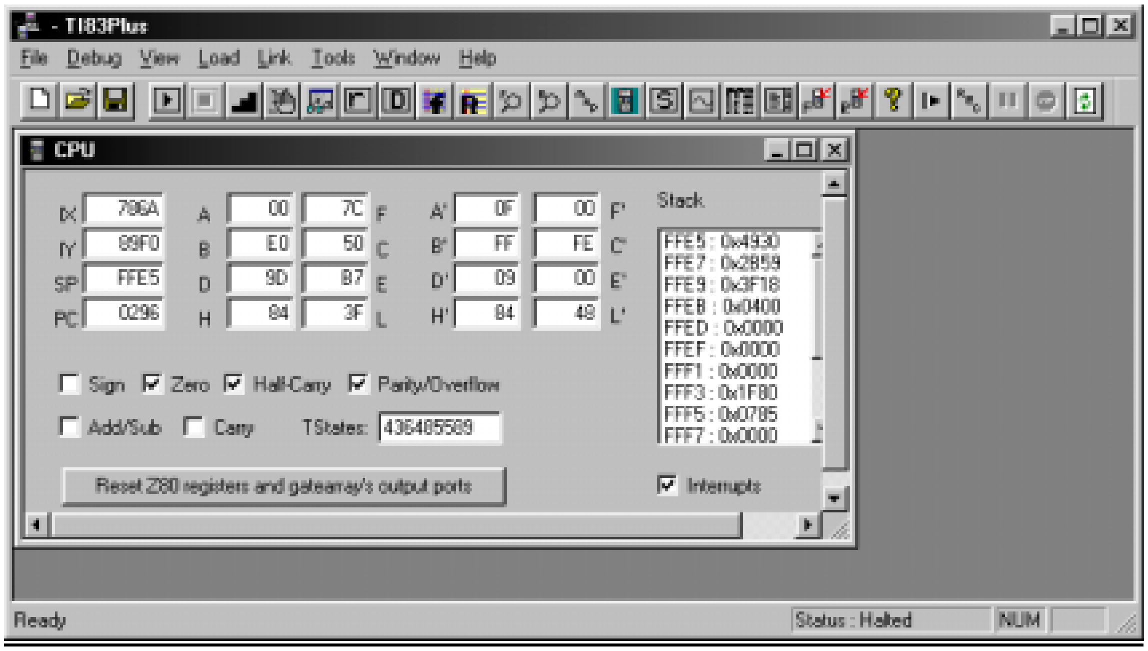
The status bar at the bottom of the window indicates the status of the debugger and simulator. The left side of the status bar indicates the status of the debugger (i.e., Ready). The first box on the right side of the status bar indicates the status of the simulator. In this case, the status of the simulator 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, you must select the calculator model (TI-73, TI-83 Plus, or TI-83 Plus Silver Edition) you wish to simulate. Once you have selected a calculator model, the following CPU view is presented with additional selections on the menu bar and tool bar as noted below.


- TI83Plus
File Debug View Load Link Tools Window Help
CPU
Reset Z80 registers and gatearray's output ports

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| IX | 786A | A | 00 | 7C | F | A' | 0F | 00 | F' |
| IY | 89F0 | B | E0 | 50 | C | B' | FF | FE | C' |
| SP | FFE5 | D | 9D | 87 | E | D' | 09 | 00 | E' |
| PC | 0296 | H | 84 | 3F | L | H' | 84 | 48 | L' |

Stack
```
FFE5 : 0x4930
FFE7 : 0x2859
FFE9 : 0x3F18
FFEB : 0x0400
FFED : 0x0000
FFEF : 0x0000
FFF1 : 0x0000
FFF3 : 0x1F80
FFF5 : 0x0785
FFF7 : 0x0000
```

☐ Sign ☑ Zero ☑ Half-Carry ☑ Parity/Overflow

☐ Add/Sub ☐ Carry TStates: 436485589

☑ Interrupts

Ready Status : Halted NUM

**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** | F11 | Allows single instruction stepping |
| **Step Over** | F10 | Steps over CALL and B_CALL instructions. |
| **Breakpoints...** | Alt+F9 | |
| | Edit Breakpoints Dialog Box | |
| **Address Watch Points…** | | Alt+F8 |
| | Address Watch Points Dialog Box | |
| **Trace Options...** | Alt+F7 | |
| | Trace Option Dialog Box | |
| **Enable IO Trace** | | |
| | IO Trace Option Dialog Box | |

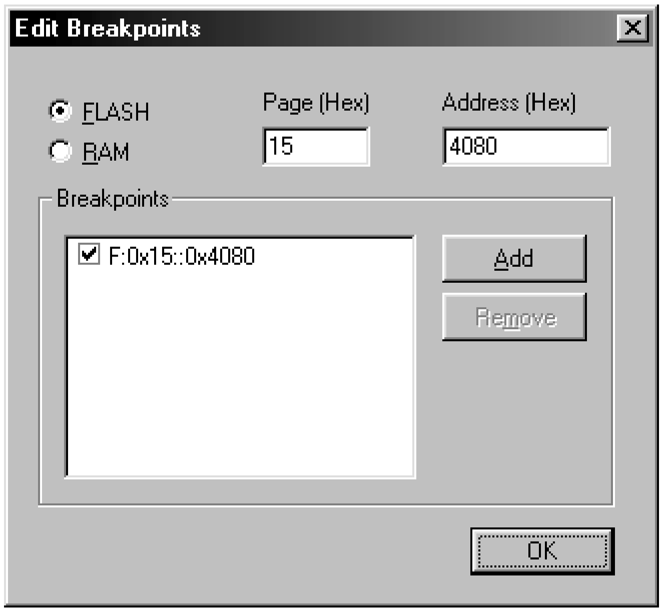
**View**

| | | |
|---|---|---|
| **CPU** | Alt+0 | |
| **Disassembly** | Alt+1 | |
| **Flash** | Alt+2 | |
| **RAM** | Alt+3 | |
| **Flash Monitor** | Alt+4 | |
| **RAM Monitor** | Alt+5 | |
| **Memory Map** | Alt+6 | |
| **Calculator** | Alt+7 | |
| **Symbol Table** | Alt+8 | |
| **GateArray IO Ports** | | Alt+C |

|  |  |  |
|---|---|---|
|  | **Display** | Alt+9 |
|  | **Trace Log** | Alt+A |
|  | **IO Buffer** | Alt+B |
|  | **OP Table** | Alt+C |
|  | **Toolbar** (selected) |  |
|  | **Status bar** (selected) |  |
|  | **Calc On Top** |  |
|  | **Clear Flash Monitor** |  |
|  | **Clear RAM Monitor** |  |
| **Window** |  |  |
|  | **Cascade** |  |
|  | **Tile** |  |
|  | **1 CPU** |  |
| **Load** |  |  |
|  | **Application...** | Ctrl+F |
|  | Load Application (Hex) File Dialog Box |  |
|  | **RAM File...** | Ctrl+R |
|  | Load RAM File Dialog Box |  |
| **Link** |  |  |
|  | **Setting** | Ctrl+L |
|  | Link Settings Dialog Box |  |
| **Tools** |  |  |
|  | **Key Press Recording Setup…** |  |
|  | **Start Key Press Recording** |  |
|  | **End Key Press Recording** (grayed out) |  |
|  | **Key Press Playing Setup…** |  |
|  | **Start Key Press Playing** |  |
|  | **End Key Press Playing** (grayed out) |  |
|  | **Mouse Cursor Tracking Enable** |  |
|  | **Save Current Calculator Screen** |  |
|  | **Display a Calculator Screen** |  |
|  | **Compare Two Calculator Screen** |  |
| **Help** |  |  |
|  | **About TI Flash Debugger** |  |

## 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.



Edit Breakpoints

[✓ FLASH] RAM

Page (Hex): 15

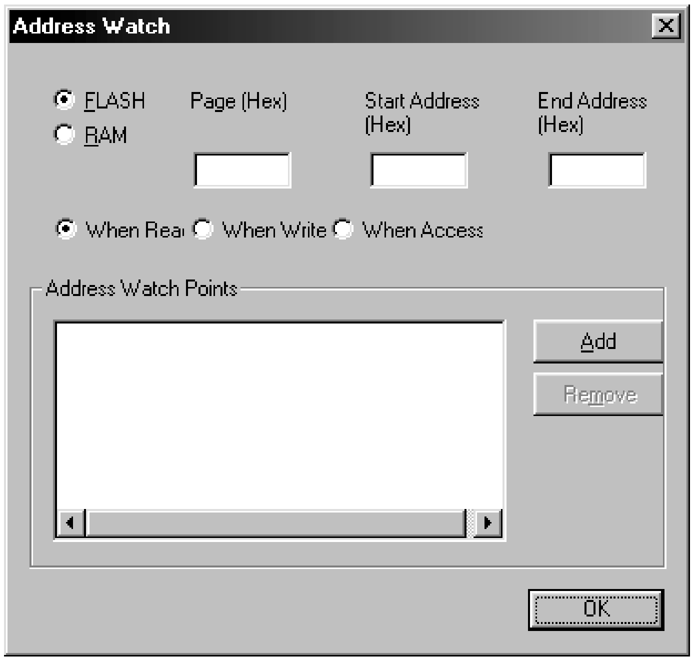
Address (Hex): 4080

Breakpoints

☑ F:0x15::0x4080

## Address Watch Points

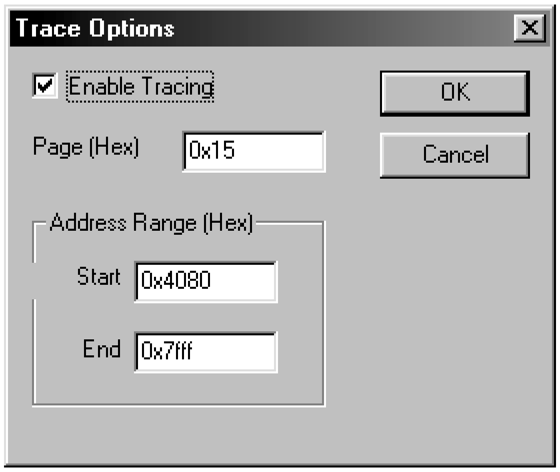
Address watch points will notify you if an address in RAM or Flash has been read from, written to, or accessed.



Address Watch: [✓ FLASH] RAM; [✓ When Rea] When Write When Access

## Trace Options

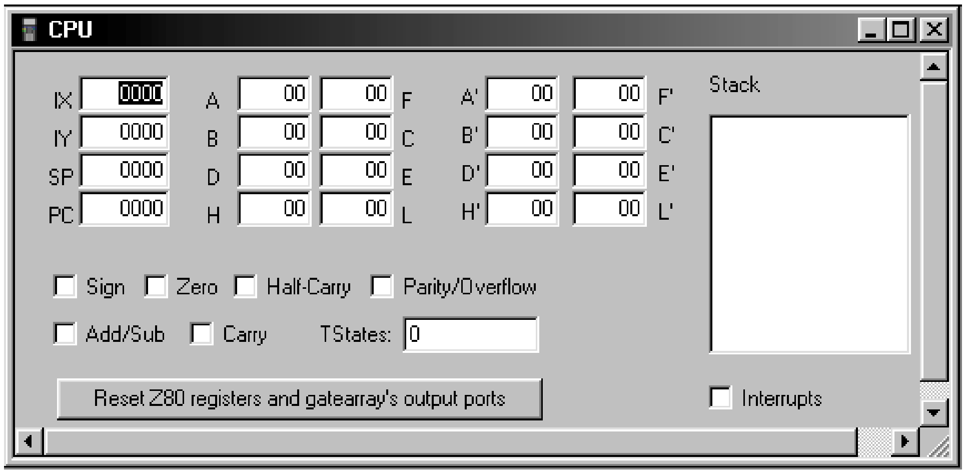
This dialog box presents options to be considered in performing a trace such as page, and address ranges.



Trace Options: [✓ Enable Tracing]; Page (Hex) 0x15; Address Range (Hex): Start 0x4080, End 0x7fff

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 |

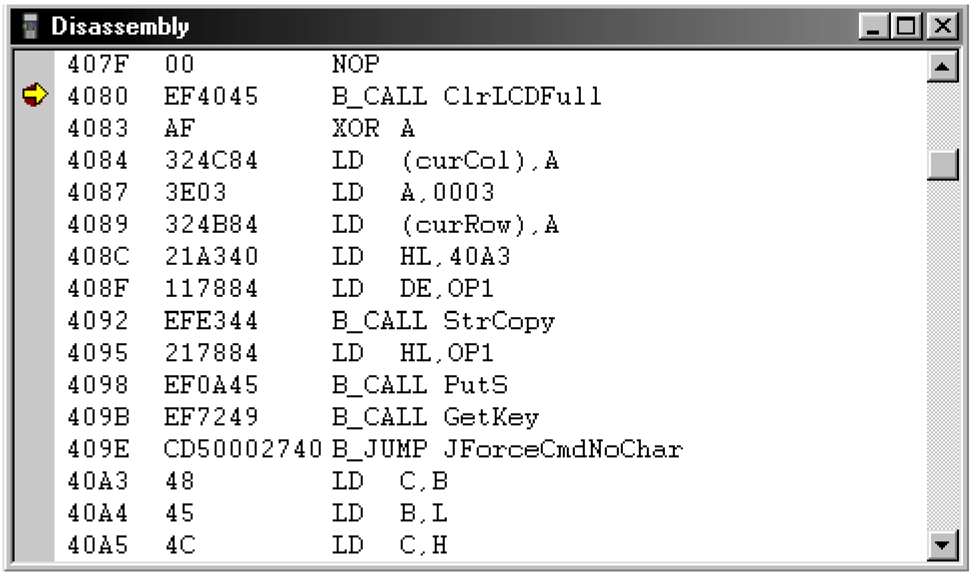
| | |
|---|---|
| Half Carry | Half Carry |
| Carry | Carry |
| Add/Sub | Flag set if a subtraction operation occurred, otherwise is reset for any other operation. |
| Tstate | Time State — counts the number of time periods. |

Reset 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

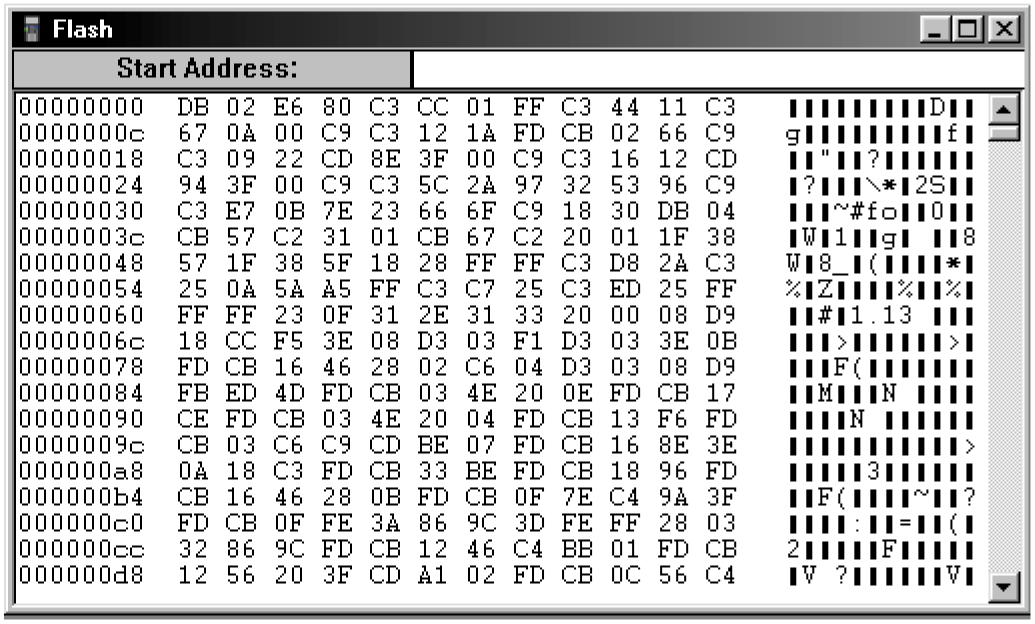
Contains the address, byte code, and instructions of the disassembled code. Breakpoints can be set and cleared from this screen by use of the right mouse click. This window is automatically invoked when the Debugger STOP key is pressed.



```
407F  00          NOP
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,40A3
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
40A3  48          LD   C,B
40A4  45          LD   B,L
40A5  4C          LD   C,H
```

## Flash View Window

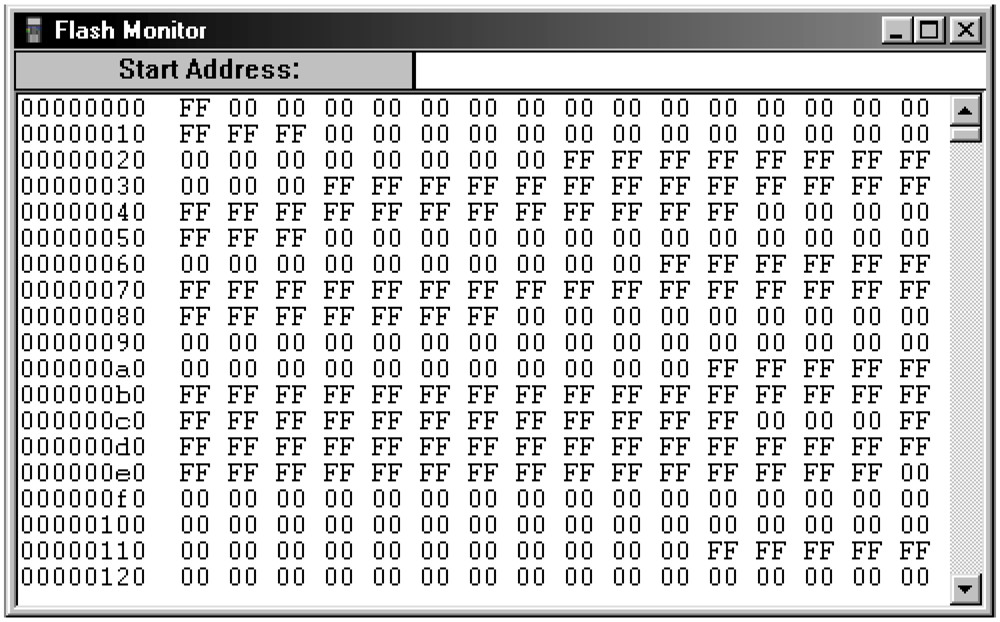
Displays the entire contents of Flash memory. This is the Edit/View screen. The ASCII representation of data is in a column on the right. The Start Address edit box is used to view addresses by entering the desired page/address and pressing enter. Right clicking in the window allows you to toggle between physical addressing and logical addressing modes.



```
00000000  DB 02 E6 80 C3 CC 01 FF C3 44 11 C3   ▌▌▌▌▌▌▌▌▌D▌▌
0000000c  67 0A 00 C9 C3 12 1A FD CB 02 66 C9   g▌▌▌▌▌▌▌▌▌f▌
00000018  C3 09 22 CD 8E 3F 00 C9 C3 16 12 CD   ▌▌"▌▌?▌▌▌▌▌▌
00000024  94 3F 00 C9 C3 5C 2A 97 32 53 96 C9   ▌?▌▌▌\*▌2S▌▌
00000030  C3 E7 0B 7E 23 66 6F C9 18 30 DB 04   ▌▌▌~#fo▌▌0▌▌
0000003c  CB 57 C2 31 01 CB 67 C2 20 01 1F 38   ▌W▌1▌▌g▌ ▌▌8
00000048  57 1F 38 5F 18 28 FF FF C3 D8 2A C3   W▌8_▌(▌▌▌▌*▌
00000054  25 0A 5A A5 FF C3 C7 25 C3 ED 25 FF   %▌Z▌▌▌%▌▌%▌
00000060  FF FF 23 0F 31 2E 31 33 20 00 08 D9   ▌▌#▌1.13 ▌▌▌
0000006c  18 CC F5 3E 08 D3 03 F1 D3 03 3E 0B   ▌▌▌>▌▌▌▌▌▌>▌
00000078  FD CB 16 46 28 02 C6 04 D3 03 08 D9   ▌▌▌F(▌▌▌▌▌▌▌
00000084  FB ED 4D FD CB 03 4E 20 0E FD CB 17   ▌▌M▌▌▌N ▌▌▌▌
00000090  CE FD CB 03 4E 20 04 FD CB 13 F6 FD   ▌▌▌▌N ▌▌▌▌▌▌
0000009c  CB 03 C6 C9 CD BE 07 FD CB 16 8E 3E   ▌▌▌▌▌▌▌▌▌▌▌>
000000a8  0A 18 C3 FD CB 33 BE FD CB 18 96 FD   ▌▌▌▌▌3▌▌▌▌▌▌
000000b4  CB 16 46 28 0B FD CB 0F 7E C4 9A 3F   ▌▌F(▌▌▌▌~▌▌?
000000c0  FD CB 0F FE 3A 86 9C 3D FE FF 28 03   ▌▌▌▌:▌▌=▌▌(▌
000000cc  32 86 9C FD CB 12 46 C4 BB 01 FD CB   2▌▌▌▌▌F▌▌▌▌▌
000000d8  12 56 20 3F CD A1 02 FD CB 0C 56 C4   ▌V ?▌▌▌▌▌▌V▌
```

## Flash Monitor Window

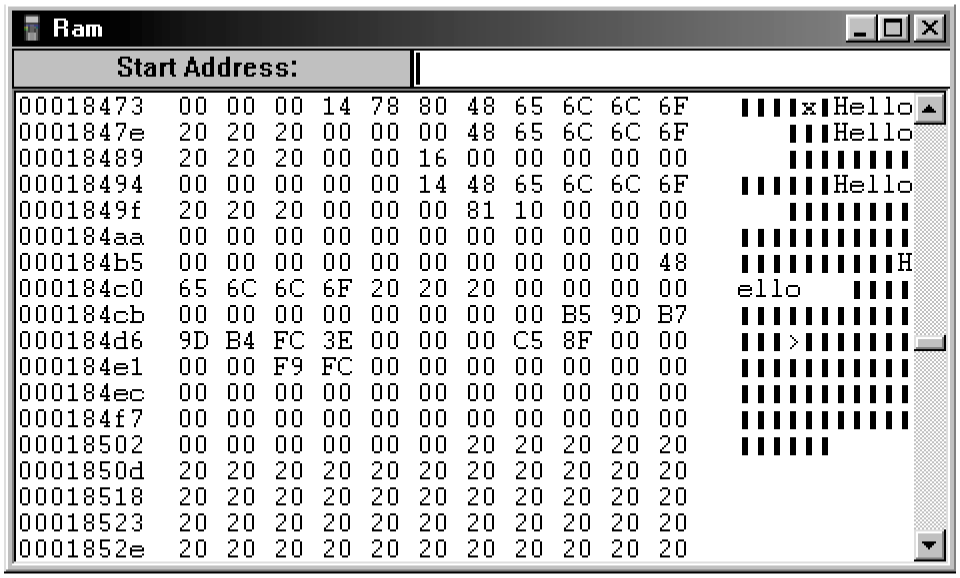
The Flash monitor notifies you if a location in Flash ROM has been read from, written to, or both. The Start Address edit box is used to view addresses by entering the desired page/address and pressing enter. Right clicking in the window allows you to toggle between physical addressing and logical addressing modes, and to clear the monitor. If a location has not been accessed, it will contain 00. When the location has been read from, it will contain 11. If the location has been written to, it will contain 99. If the location has been both read from, and written to since the monitor was cleared, then it will contain FF. Selecting View, then Clear Flash Monitor resets all locations to 00.



```
00000000   FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000010   FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020   00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF
00000030   00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF
00000040   FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 00
00000050   FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060   00 00 00 00 00 00 00 00 00 00 FF FF FF FF FF FF
00000070   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000080   FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00 00
00000090   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000a0   00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF FF
000000b0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
000000c0   FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 FF
000000d0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
000000e0   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 00
000000f0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000100   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000110   00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF FF
00000120   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

## RAM View Window

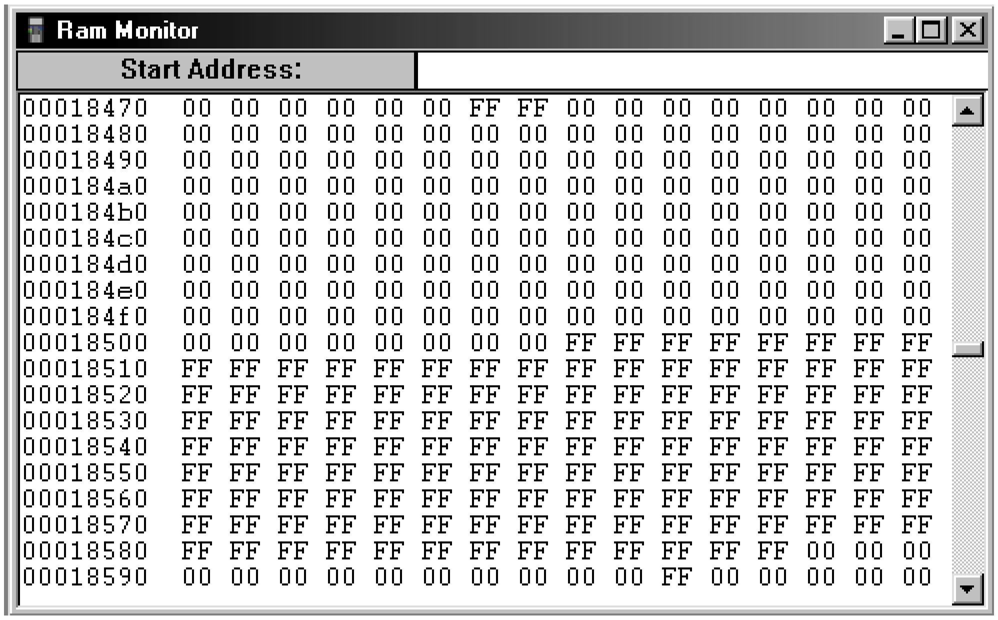
Displays the entire contents of RAM. This is the Edit/View screen. The ASCII representation of data is in a column on the right. The Start Address edit box is used to view addresses by entering the desired page/address and pressing enter. Right clicking in the window allows you to toggle between physical addressing and logical addressing modes.



```
00018473   00 00 00 14 78 80 48 65 6C 6C 6F   ▌▌▌▌x▌Hello
0001847e   20 20 20 00 00 00 48 65 6C 6C 6F      ▌▌▌Hello
00018489   20 20 20 00 00 16 00 00 00 00 00      ▌▌▌▌▌▌▌▌
00018494   00 00 00 00 00 14 48 65 6C 6C 6F   ▌▌▌▌▌▌Hello
0001849f   20 20 20 00 00 00 81 10 00 00 00      ▌▌▌▌▌▌▌▌
000184aa   00 00 00 00 00 00 00 00 00 00 00   ▌▌▌▌▌▌▌▌▌▌▌
000184b5   00 00 00 00 00 00 00 00 00 00 48   ▌▌▌▌▌▌▌▌▌▌H
000184c0   65 6C 6C 6F 20 20 20 00 00 00 00   ello   ▌▌▌▌
000184cb   00 00 00 00 00 00 00 00 B5 9D B7   ▌▌▌▌▌▌▌▌▌▌▌
000184d6   9D B4 FC 3E 00 00 00 C5 8F 00 00   ▌▌▌>▌▌▌▌▌▌▌
000184e1   00 00 F9 FC 00 00 00 00 00 00 00   ▌▌▌▌▌▌▌▌▌▌▌
000184ec   00 00 00 00 00 00 00 00 00 00 00   ▌▌▌▌▌▌▌▌▌▌▌
000184f7   00 00 00 00 00 00 00 00 00 00 00   ▌▌▌▌▌▌▌▌▌▌▌
00018502   00 00 00 00 00 00 20 20 20 20 20   ▌▌▌▌▌▌
0001850d   20 20 20 20 20 20 20 20 20 20 20
00018518   20 20 20 20 20 20 20 20 20 20 20
00018523   20 20 20 20 20 20 20 20 20 20 20
0001852e   20 20 20 20 20 20 20 20 20 20 20
```

## RAM Monitor Window

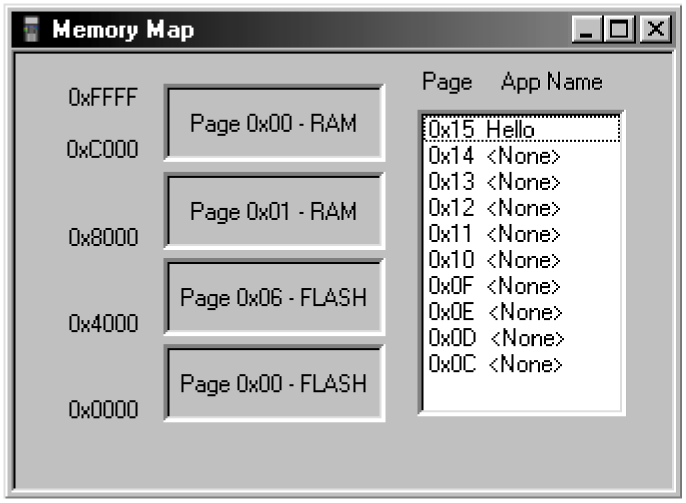
The RAM monitor notifies you if a location in RAM has been read from, written to, or both. The Start Address edit box is used to view addresses by entering the desired page/address and pressing enter. Right clicking in the window allows you to toggle between physical addressing and logical addressing modes, and to clear the monitor. If a location has not been accessed, it will contain 00. When the location has been read from, it will contain 11. If the location has been written to, it will contain 99. If the location has been both read from, and written to since the monitor was cleared, then it will contain FF. Selecting View, then Clear RAM Monitor resets all locations to 00.



```
00018470  00 00 00 00 00 00 FF FF 00 00 00 00 00 00 00 00
00018480  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00018490  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000184a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000184b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000184c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000184d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000184e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000184f0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00018500  00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF
00018510  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00018520  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00018530  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00018540  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00018550  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00018560  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00018570  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00018580  FF FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00
00018590  00 00 00 00 00 00 00 00 00 00 FF 00 00 00 00 00
```

## Memory Map Window

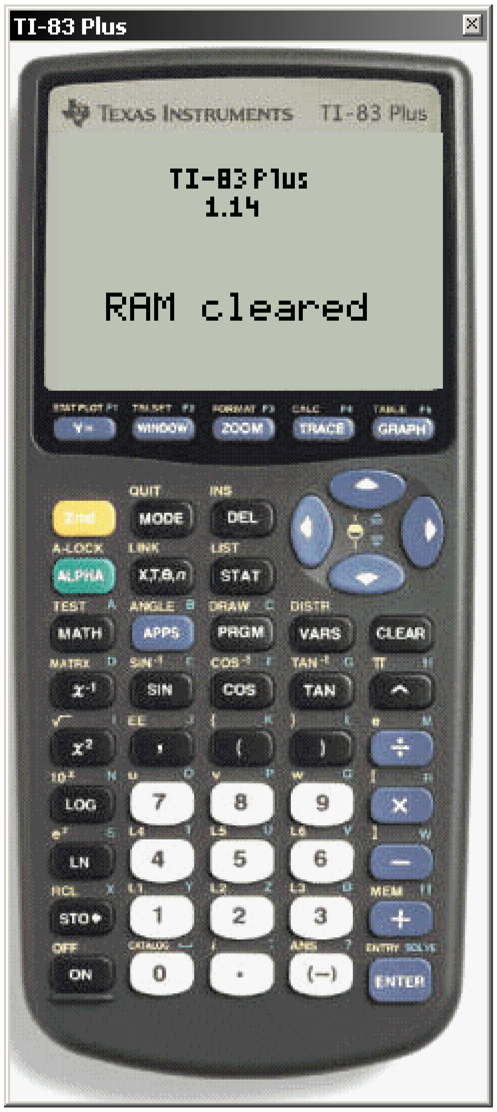
Shows which pages of Flash and RAM are currently mapped in the Z80 address space.



| Page | App Name |
|---|---|
| 0x15 | Hello |
| 0x14 | <None> |
| 0x13 | <None> |
| 0x12 | <None> |
| 0x11 | <None> |
| 0x10 | <None> |
| 0x0F | <None> |
| 0x0E | <None> |
| 0x0D | <None> |
| 0x0C | <None> |

## Calculator Simulator Window

The following screen shot contains an active simulated TI-83 Plus calculator. 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.



The input to the TI-83 Plus calculator window 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 pckeys.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.

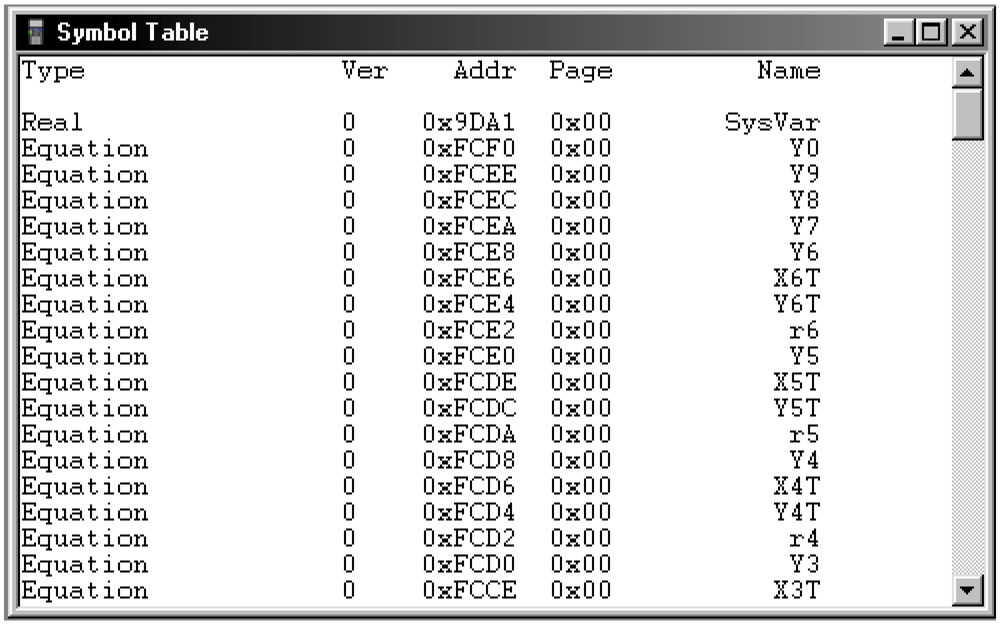
  The pckeys.cfg file contains the PC keyboard keys with their hex values.

  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.

## Symbol Table

Displays information about all variables in the symbol table. The symbol table window shows the variable type, version, data area start address, page the variable is located on (0x00 if the variable resides in RAM), and the name of the variable. Double-clicking on an entry will bring you to that entry's data storage area.

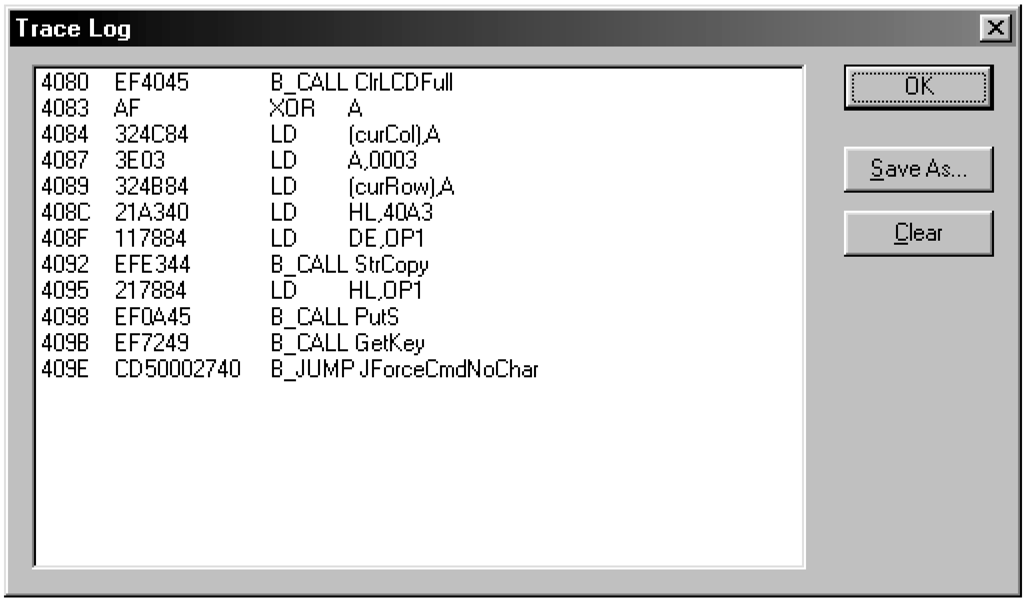


```
Type                Ver    Addr    Page        Name

Real                0     0x9DA1   0x00      SysVar
Equation            0     0xFCF0   0x00          Y0
Equation            0     0xFCEE   0x00          Y9
Equation            0     0xFCEC   0x00          Y8
Equation            0     0xFCEA   0x00          Y7
Equation            0     0xFCE8   0x00          Y6
Equation            0     0xFCE6   0x00         X6T
Equation            0     0xFCE4   0x00         Y6T
Equation            0     0xFCE2   0x00          r6
Equation            0     0xFCE0   0x00          Y5
Equation            0     0xFCDE   0x00         X5T
Equation            0     0xFCDC   0x00         Y5T
Equation            0     0xFCDA   0x00          r5
Equation            0     0xFCD8   0x00          Y4
Equation            0     0xFCD6   0x00         X4T
Equation            0     0xFCD4   0x00         Y4T
Equation            0     0xFCD2   0x00          r4
Equation            0     0xFCD0   0x00          Y3
Equation            0     0xFCCE   0x00         X3T
```

## Trace Log Window

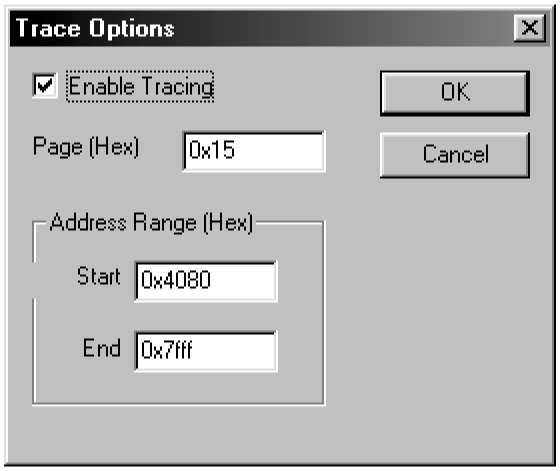
Displays the output of a trace — the execution of instructions within a developer definable address space.


Trace Log

```
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,40A3
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
```

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. |


Trace Options

Enable Tracing: ✓

Page (Hex): 0x15

Address Range (Hex)

Start: 0x4080

End: 0x7fff

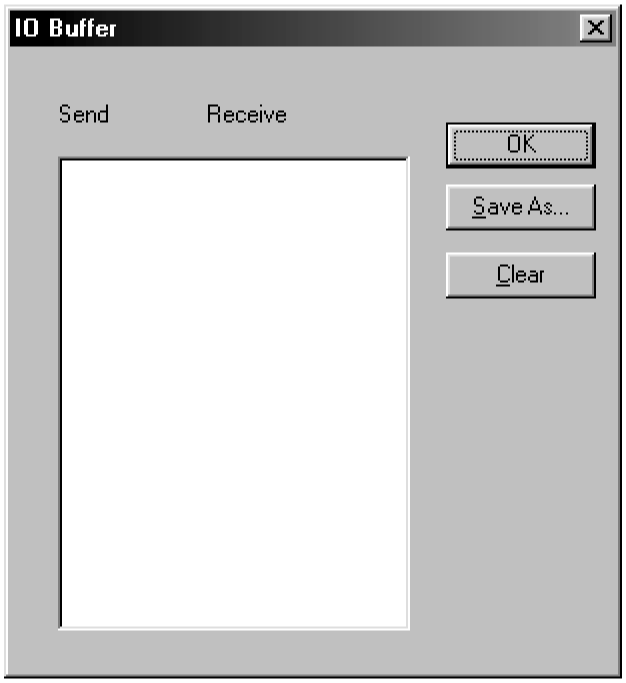
Here is how it works:

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 buffer is cleared.

## IO Buffer Window

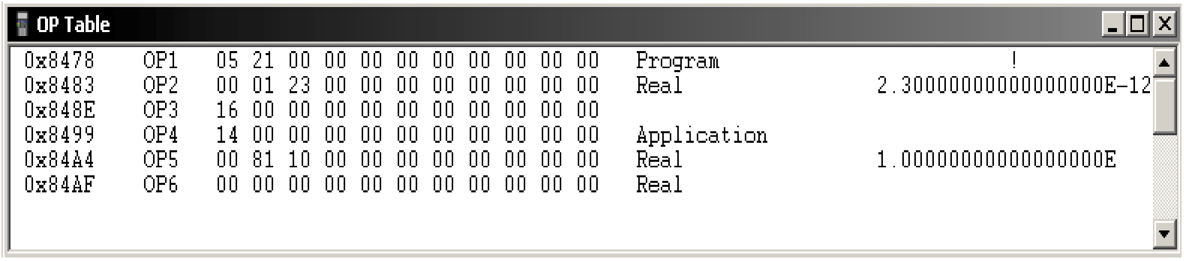
Displays all data sent or received through the input/output port.



From the IO Buffer dialog box, the developer can save [Save As..] the contents of the trace buffer. Using the [Clear] button, the buffer is cleared.

## OP Table Window

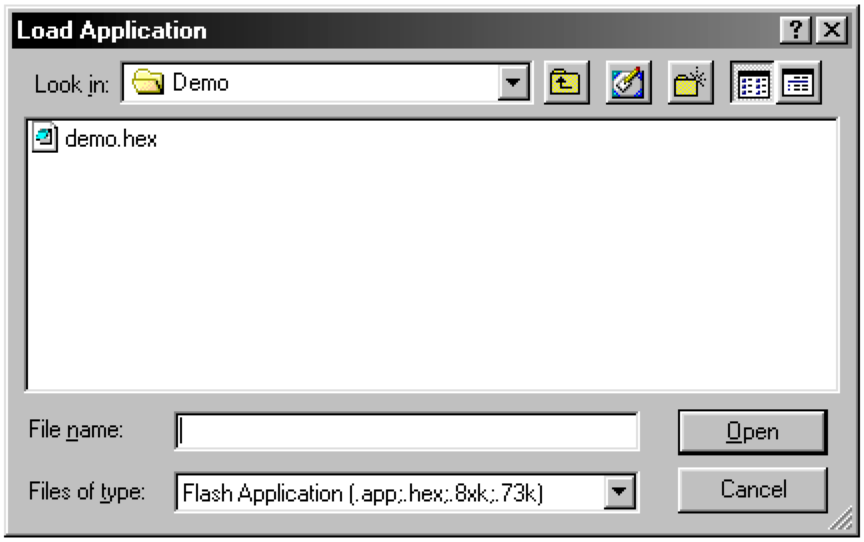
Displays the contents of the OP1 – OP6 RAM registers. If a register contains a floating point number or variable name, the data type is shown and the register's contents are decoded and displayed.



```
0x8478   OP1   05 21 00 00 00 00 00 00 00 00 00   Program                    !
0x8483   OP2   00 01 23 00 00 00 00 00 00 00 00   Real              2.30000000000000000E-12
0x848E   OP3   16 00 00 00 00 00 00 00 00 00 00
0x8499   OP4   14 00 00 00 00 00 00 00 00 00 00   Application
0x84A4   OP5   00 81 10 00 00 00 00 00 00 00 00   Real              1.00000000000000000E
0x84AF   OP6   00 00 00 00 00 00 00 00 00 00 00   Real
```

## Loading Applications and RAM Files

Selecting the Load/Application... menu item allows you to load an Application.



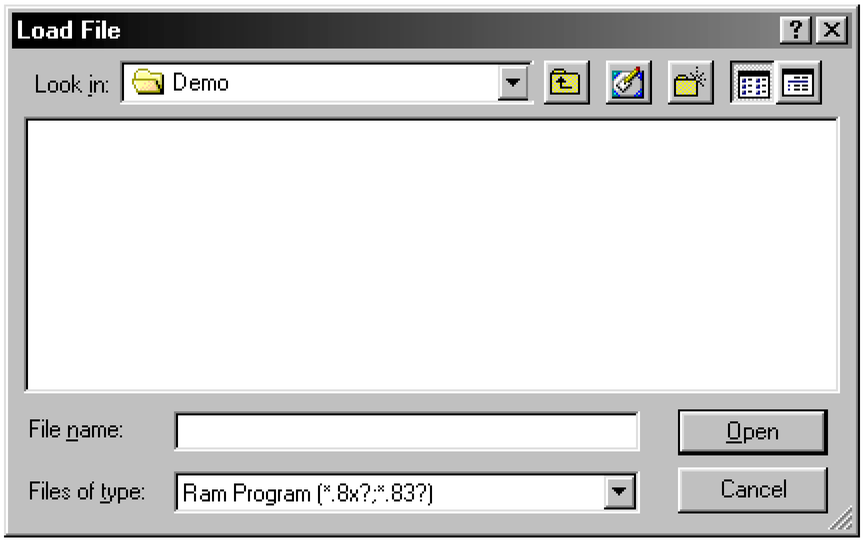
Demo

demo.hex

File name:

Files of type: Flash Application (.app;.hex;.8xk;.73k)

Selecting the Load/RAM File… menu item allows you to load a RAM file.



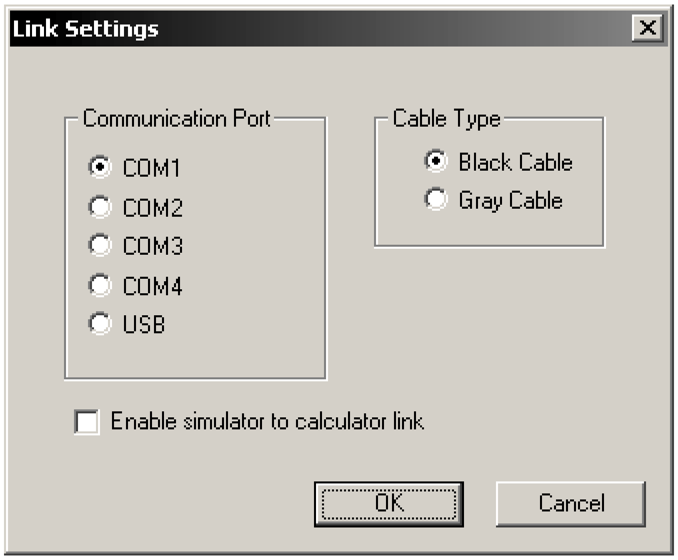
Demo

File name:

Files of type: Ram Program (*.8x?;*.83?)

## Link Settings

The Link Settings dialog box allows you to configure communications settings. Selecting Enable simulator to calculator link will allow you to send and receive data to an external device (calculator, CBL, CBL2, CBR, etc.) through the TI-GRAPH LINK cable.


Link Settings

Communication Port
- [✓ COM1]
- COM2
- COM3
- COM4
- USB

Cable Type
- [✓ Black Cable]
- Gray Cable

☐ Enable simulator to calculator link

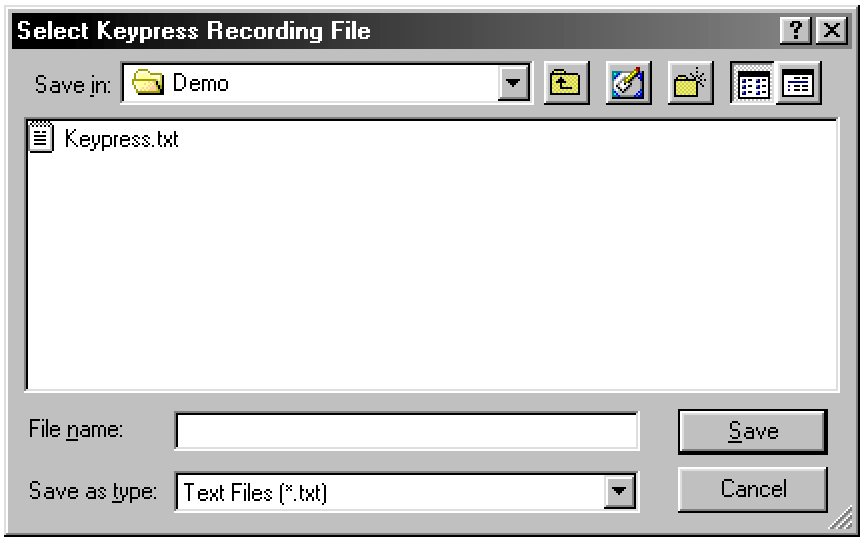
Troubleshooting link errors:

1. Make sure that the cable is firmly connected to both the serial port and the external device.
2. Make sure that the serial port is enabled, and that the COM port is not in use by another device.
3. Close any conflicting software programs (TI-GRAPH LINK™, TI Connect™ software, some personal organizer software, etc.).

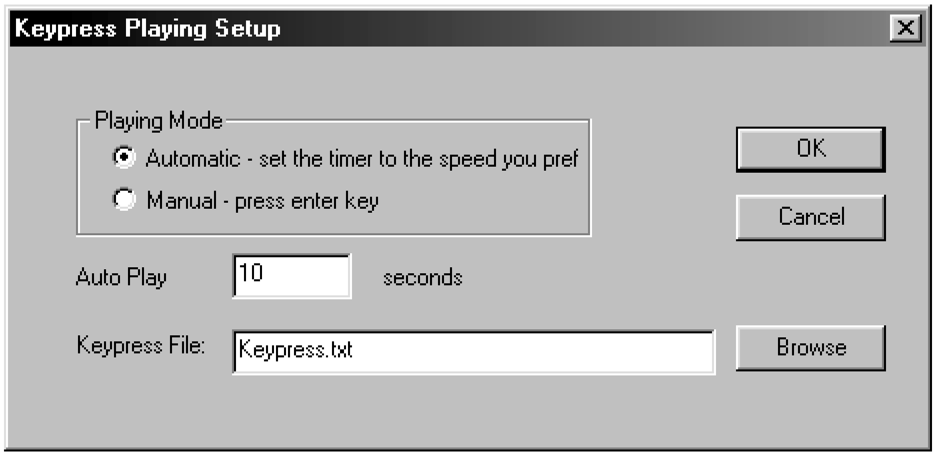
For more information, refer to the TI-GRAPH LINK™ or TI Connect™ documentation.

## Key Press Recording and Playback

This option allows you to record a series of key presses and play them back at a specified rate. Select Tools, then Start Key Press Recording to start recording. All key presses will be saved into a file named Keypress.txt. Select Tools, then End Key Press Recording to stop recording. Selecting Tools, then Key Press Recording Setup... allows you to save the key presses into a different file.


Select Keypress Recording File
Save in: Demo
Keypress.txt
File name:
Save as type: Text Files (*.txt)
Save
Cancel

Selecting Tools, then Key Press Playing Setup… will bring up the Key Press Playing Setup dialog box. You can select between Automatic and Manual playback, choose a different keypress file, and select the time between key presses (Automatic mode).



[✓ Automatic - set the timer to the speed you pref]
Manual - press enter key

Auto Play: 10 seconds
Keypress File: Keypress.txt

Select Tools, then Start Key Press Playing to start playing the key presses. When the end of the key press file is reached, a message will prompt you to either play the key presses again or stop.

Selecting the Mouse Cursor Tracking Enable option will put the mouse cursor on the keys as they are being played back.

## Save/Display/Compare Calculator Screens

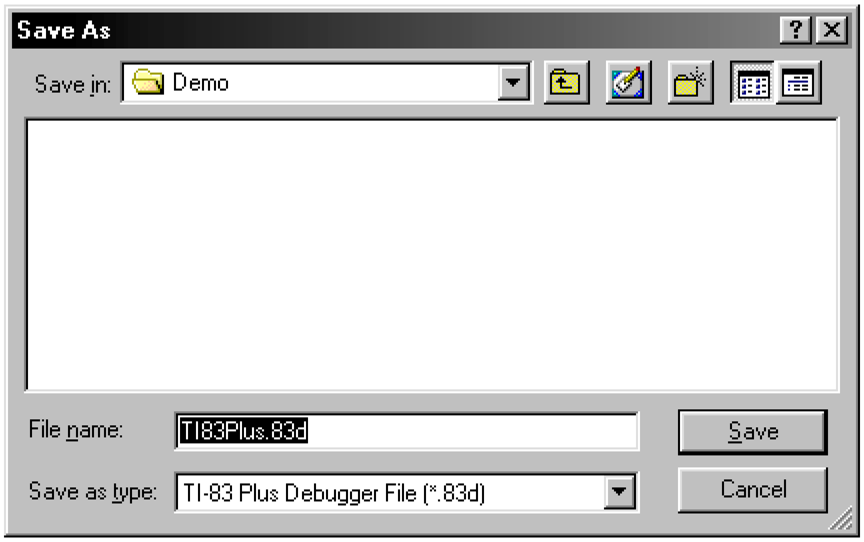
Select Tools, then Save Current Calculator Screen to save the current calculator screen into a file (*.dat). Select Tools, then Display a Calculator Screen to display a saved calculator screen. Select Tools, then Compare Two Calculator Screen to compare two saved calculator screens.

**Note:** The Tools menu is also available by right-clicking on the calculator window.

## 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://education.ti.com/developer/deselect.html

# 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⎡** | **A**utomatic **P**ower **D**own⎡. |
| **API** | **A**pplication **P**rogrammer's **I**nterface—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** | **As**sembly **A**pplication **P**rogram—a RAM-resident application. |
| **ASCII** | **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange—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** | A system of counting using 0's and 1's. The first seven digits and the decimal equivalents are: |

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 10 | 2 |
| 11 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

See also Hexadecimal.

| | |
|---|---|
| **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. |

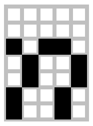
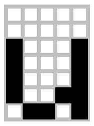
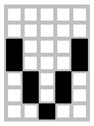
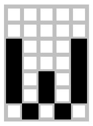
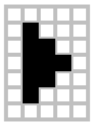
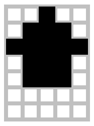
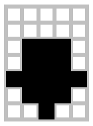
| | |
|---|---|
| **Garbage collection** | A procedure that automatically determines what memory a program is no longer using and recycles it for other use. This is also known as **automatic storage (or memory) reclamation**. |
| **TI.GRAPH LINK⎡** | An optional accessory that links a calculator to a 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** | **I**ntegrated **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. |

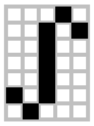
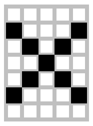
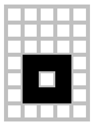
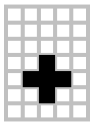
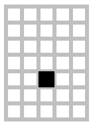
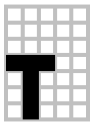
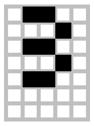
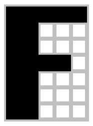
| | |
|---|---|
| **Marked Dirty** | The graph is marked as needing to be updated. The next system routine that will affect the graph contents will cause the system to regraph all of the equations selected thereby making the graph clean. |
| **Memory** | Memory is where data is stored. On the TI–83 Plus, 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. |

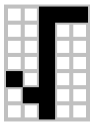
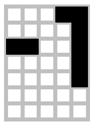
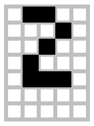
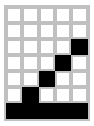
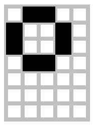
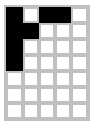
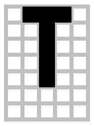
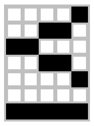
| | |
|---|---|
| | Payment of the registration fee to the author will bring the user a license to continue using the software. Most authors will include other materials in return for the registration fee—like printed manuals, technical support, bonus or additional software, or upgrades. Shareware is 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. |

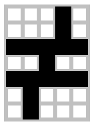
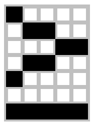
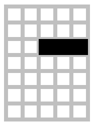
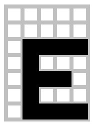
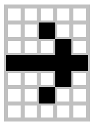
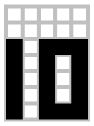
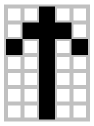
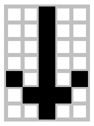
# Appendix A TI-83 Plus “Large” Character Fonts

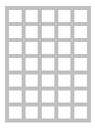
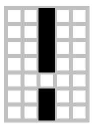
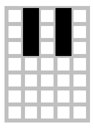
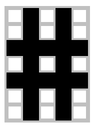
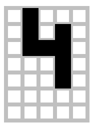
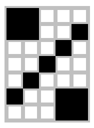
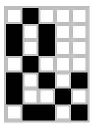
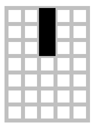
The font map below shows each character code, the symbolic name, and the character map.

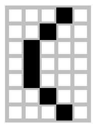
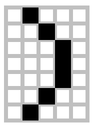
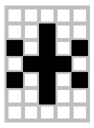
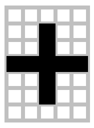
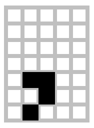
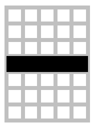
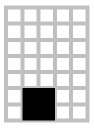
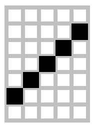
| 00h | 01h | 02h | 03h | 04h | 05h | 06h | 07h |
|---|---|---|---|---|---|---|---|
| NOT USED | LrecurN | LrecurU | LrecurV | LrecurW | Lconvert | LsqUp | LsqDown |
|  |  |  |  |  |  |  |  |

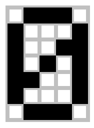
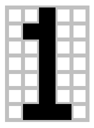
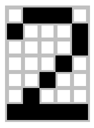
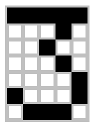
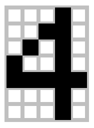
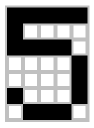
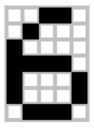
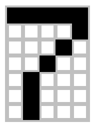
| 08h | 09h | 0Ah | 0Bh | 0Ch | 0Dh | 0Eh | 0Fh |
|---|---|---|---|---|---|---|---|
| Lintegral | Lcross | LboxIcon | LcrossIcon | LdotIcon | LsubT | LcubeR | LhexF |
|  |  |  |  |  |  |  |  |

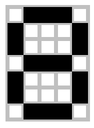
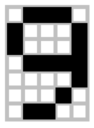
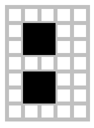
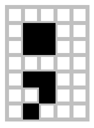
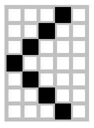
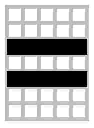
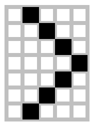
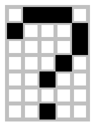
| 10h | 11h | 12h | 13h | 14h | 15h | 16h | 17h |
|---|---|---|---|---|---|---|---|
| Lroot | Linverse | Lsquare | Langle | Ldegree | Lradian | Ltranspose | LLE |
|  |  |  |  |  |  |  |  |

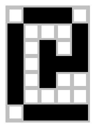
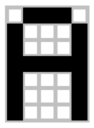
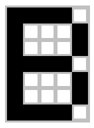
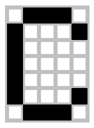
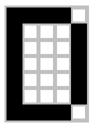
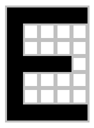
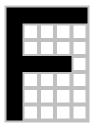
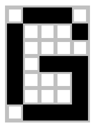
| 18h | 19h | 1Ah | 1Bh | 1Ch | 1Dh | 1Eh | 1Fh |
|---|---|---|---|---|---|---|---|
| LNE | LGE | Lneg | Lexponent | Lstore | Lten | LupArrow | LdownArrow |
|  |  |  |  |  |  |  |  |

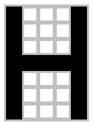
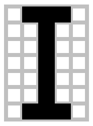
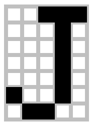
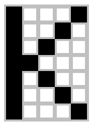
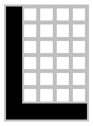
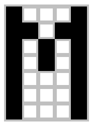
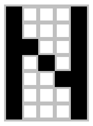
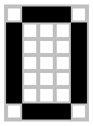
| 20h | 21h | 22h | 23h | 24h | 25h | 26h | 27h |
|---|---|---|---|---|---|---|---|
| Lspace | Lexclam | Lquote | Lpound | Lfourth | Lpercent | Lampersand | Lapostrophe |
|  |  |  |  |  |  |  |  |

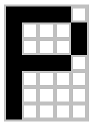
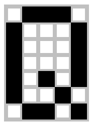
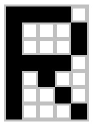
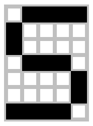
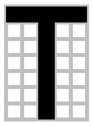
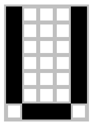
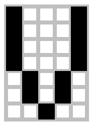
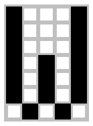
| 28h | 29h | 2Ah | 2Bh | 2Ch | 2Dh | 2Eh | 2Fh |
|---|---|---|---|---|---|---|---|
| LlParen | LrParen | Lasterisk | LplusSign | Lcomma | Ldash | Lperiod | Lslash |
|  |  |  |  |  |  |  |  |

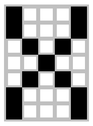
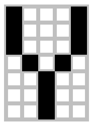
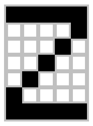
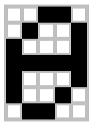
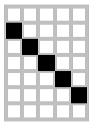
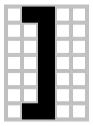
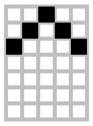
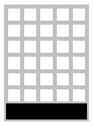
| 30h | 31h | 32h | 33H | 34H | 35H | 36H | 37H |
|---|---|---|---|---|---|---|---|
| L0 | L1 | L2 | L3 | L4 | L5 | L6 | L7 |
|  |  |  |  |  |  |  |  |

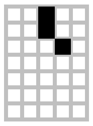
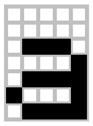
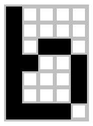
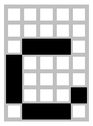
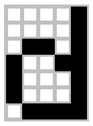
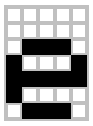
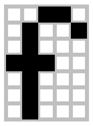
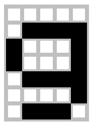
| 38H | 39H | 3Ah | 3Bh | 3Ch | 3Dh | 3Eh | 3Fh |
|---|---|---|---|---|---|---|---|
| L8 | L9 | Lcolon | Lsemicolon | LLT | LEQ | LGT | Lquestion |
|  |  |  |  |  |  |  |  |

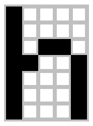
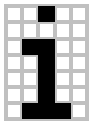
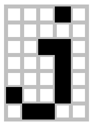
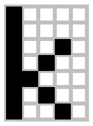
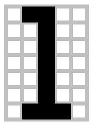
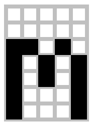
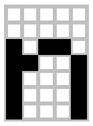
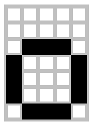
| 40h | 41h | 42h | 43h | 44h | 45h | 46h | 47h |
|---|---|---|---|---|---|---|---|
| LatSign | LcapA | LcapB | LcapC | LcapD | LcapE | LcapF | LcapG |
|  |  |  |  |  |  |  |  |

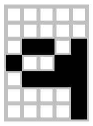
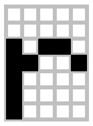
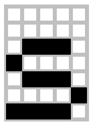
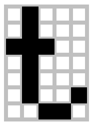
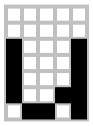
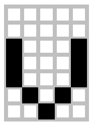
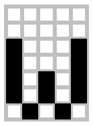
| 48h<br>LcapH | 49h<br>LcapI | 4Ah<br>LcapJ | 4Bh<br>LcapK | 4Ch<br>LcapL | 4Dh<br>LcapM | 4Eh<br>LcapN | 4Fh<br>LcapO |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

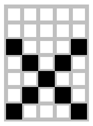
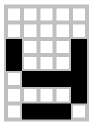
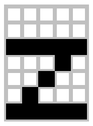
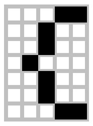
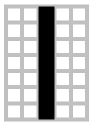
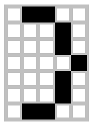
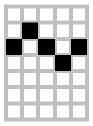
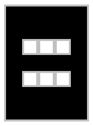
| 50h<br>LcapP | 51h<br>LcapQ | 52h<br>LcapR | 53h<br>LcapS | 54h<br>LcapT | 55h<br>LcapU | 56h<br>LcapV | 57h<br>LcapW |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

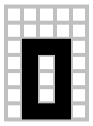
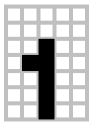
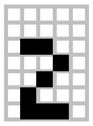
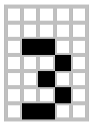
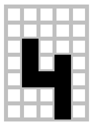
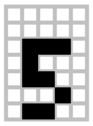
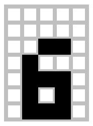
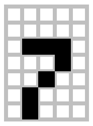
| 58h<br>LcapX | 59h<br>LcapY | 5Ah<br>LcapZ | 5Bh<br>Ltheta | 5Ch<br>Lbackslash | 5Dh<br>LrBrack | 5Eh<br>Lcaret | 5Fh<br>Lunderscore |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

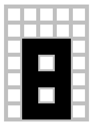
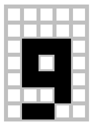
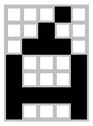
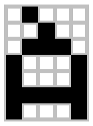
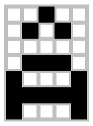
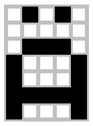
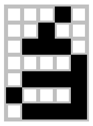
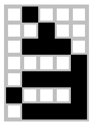
| 60h<br>Lbackquote | 61h<br>La | 62h<br>Lb | 63h<br>Lc | 64h<br>Ld | 65h<br>Le | 66h<br>Lf | 67h<br>Lg |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

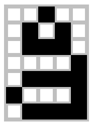
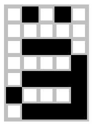
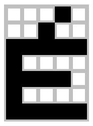
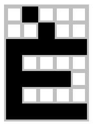
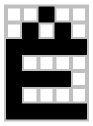
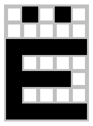
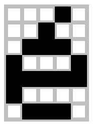
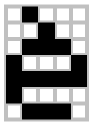
| 68h<br>Lh | 69h<br>Li | 6Ah<br>Lj | 6Bh<br>Lk | 6Ch<br>Ll | 6Dh<br>Lm | 6Eh<br>Ln | 6Fh<br>Lo |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

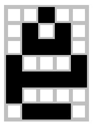
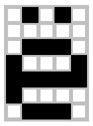
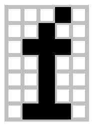
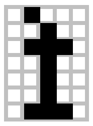
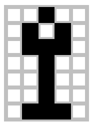
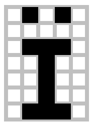
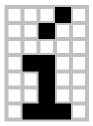
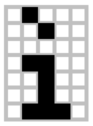
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 70h | 71h | 72h | 73h | 74h | 75h | 76h | 77h |
| Lp | Lq | Lr | Ls | Lt | Lu | Lv | Lw |
| |  |  |  |  |  |  |  |

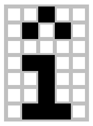
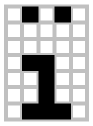
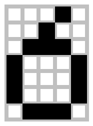
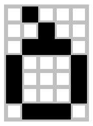
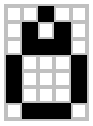
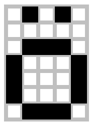
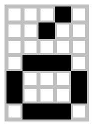
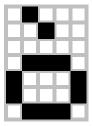
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 78h | 79h | 7Ah | 7Bh | 7Ch | 7Dh | 7Eh | 7Fh |
| Lx | Ly | Lz | LlBrace | Lbar | LrBrace | Ltilde | LinvEQ |
|  |  |  |  |  |  |  |  |

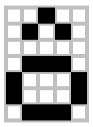
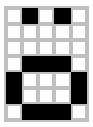
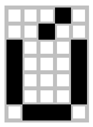
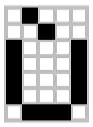
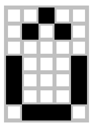
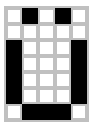
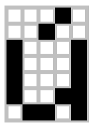
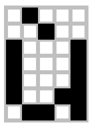
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 80h | 81h | 82h | 83h | 84h | 85h | 86h | 87h |
| Lsub0 | Lsub1 | Lsub2 | Lsub3 | Lsub4 | Lsub5 | Lsub6 | Lsub7 |
|  |  |  |  |  |  |  |  |

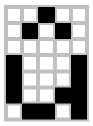
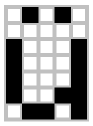
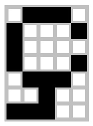
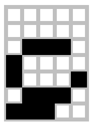
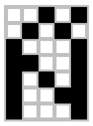
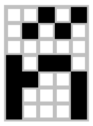
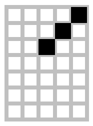
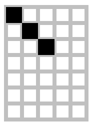
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 88h | 89h | 8Ah | 8Bh | 8Ch | 8Dh | 8Eh | 8Fh |
| Lsub8 | Lsub9 | LcapAAcute | LcapAGrave | LcapACaret | LcapADier | LaAcute | LaGrave |
|  |  |  |  |  |  |  |  |

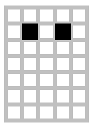
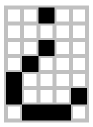
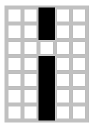
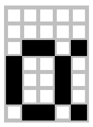
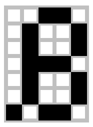
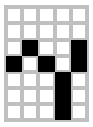
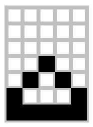
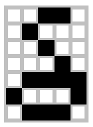
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 90h | 91h | 92h | 93h | 94h | 95h | 96h | 97h |
| LaCaret | LaDier | LcapEAcute | LcapEGrave | LcapECaret | LcapEDier | LeAcute | LeGrave |
|  |  |  |  |  |  |  |  |

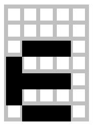
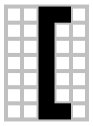
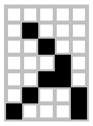
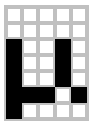
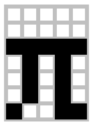
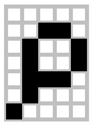
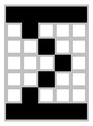
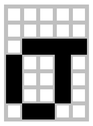
| 98h | 99h | 9Ah | 9Bh | 9Ch | 9Dh | 9Eh | 9Fh |
|---|---|---|---|---|---|---|---|
| LeCaret | LeDier | LcapIAcute | LcapIGrave | LcapICaret | LcapIDier | LiAcute | LiGrave |
|  |  |  |  |  |  |  |  |

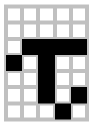
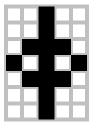
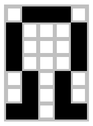
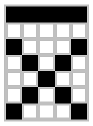
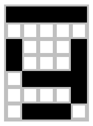
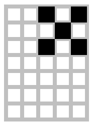
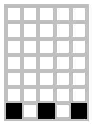
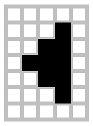
| A0h | A1h | A2h | A3h | A4h | A5h | A6h | A7h |
|---|---|---|---|---|---|---|---|
| LiCaret | LiDier | LcapOAcute | LcapOGrave | LcapOCaret | LcapODier | LoAcute | LoGrave |
|  |  |  |  |  |  |  |  |

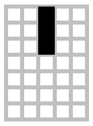
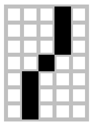
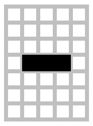
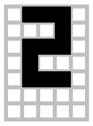
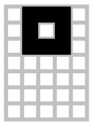
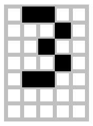
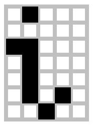
| A8h | A9h | AAh | ABh | ACh | ADh | AEh | AFh |
|---|---|---|---|---|---|---|---|
| LoCaret | LoDier | LcapUAcute | LcapUGrave | LcapUCaret | LcapUDier | LuAcute | LuGrave |
|  |  |  |  |  |  |  |  |

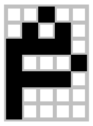
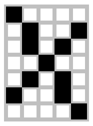
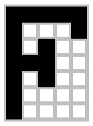
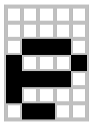
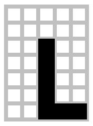
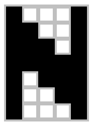
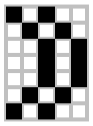
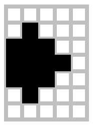
| B0h | B1h | B2h | B3h | B4h | B5h | B6h | B7h |
|---|---|---|---|---|---|---|---|
| LuCaret | LuDier | LcapCCed | LcCed | LcapNTilde | LnTilde | Laccent | Lgrave |
|  |  |  |  |  |  |  |  |

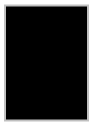
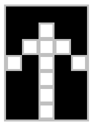
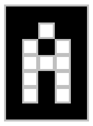
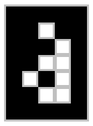
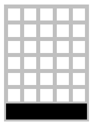
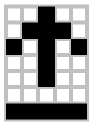
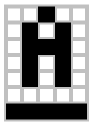
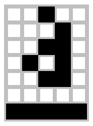
| B8h | B9h | BAh | BBh | BCh | BDh | BEh | BFh |
|---|---|---|---|---|---|---|---|
| Ldieresis | LquesDown | LexclamDown | Lalpha | Lbeta | Lgamma | LcapDelta | Ldelta |
|  |  |  |  |  |  |  |  |

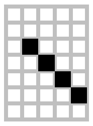
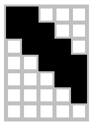
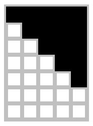
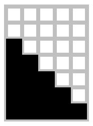
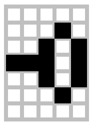
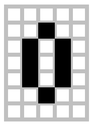
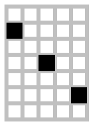
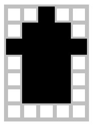
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| C0h | C1h | C2h | C3h | C4h | C5h | C6h | C7h |
| Lepsilon | LlBrack | Llambda | Lmu | Lpi | Lrho | LcapSigma | Lsigma |
|  |  |  |  |  |  |  |  |

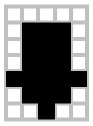
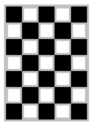
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| C8h | C9h | CAh | CBh | CCh | CDh | CEh | CFh |
| Ltau | Lphi | LcapOmega | LxMean | LyMean | LsupX | Lellipsis | Lleft |
|  |  |  |  |  |  |  |  |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| D0h | D1h | D2h | D3h | D4h | D5h | D6h | D7h |
| Lblock | Lper | Lhyphen | Larea | Ltemp | Lcube | Lenter | LimagI |
|  |  |  |  |  |  | |  |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| D8h | D9h | DAh | DBh | DCh | DDh | DEh | DFh |
| Lphat | Lchi | LstatF | Llne | LlistL | LfinanN | L2_r_paren | LblockArrow |
|  |  |  |  |  |  |  |  |

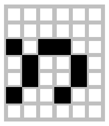
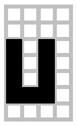
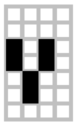
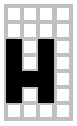
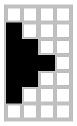
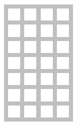
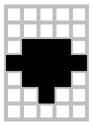
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| E0h | E1h | E2h | E3h | E4h | E5h | E6h | E7h |
| LcurO | LcurO2 | LcurOcapA | LcurOa | LcurI | LcurI2 | LcurIcapA | LcurIa |
|  |  |  |  |  |  |  |  |

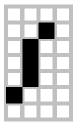
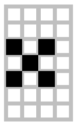
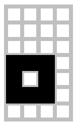
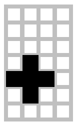
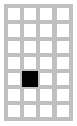
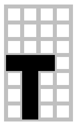
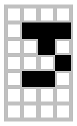
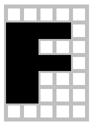
| E8h<br>LGline | E9h<br>LGthick | EAh<br>LGabove | EBh<br>LGbelow | ECh<br>LGpath | EDh<br>LGanimate | EEh<br>LGdot | EFh<br>LUpBlk |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

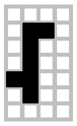
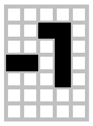
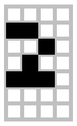
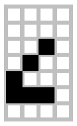
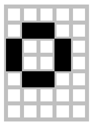
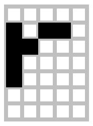
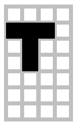
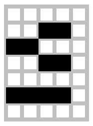
| F0h<br>LDnBlk | F1h<br>LcurFull |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

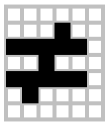
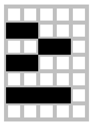
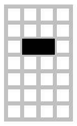
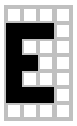
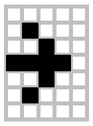
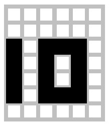
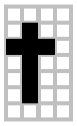
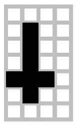
# Appendix B TI–83 Plus “Small” Character Fonts

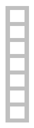
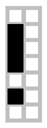
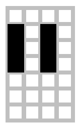
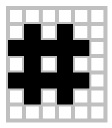
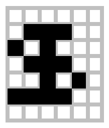
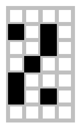
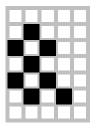
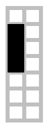
The font map below shows each character code, the symbolic name, and the character map. Most characters are five pixels high, but a few are longer. The character widths are variable, e.g. a space has a width of one pixel whereas an asterisk has width of five pixels. Character maps usually include one blank pixel column on the right side to ensure spacing when printing strings.

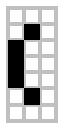
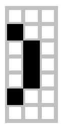
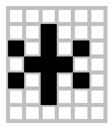
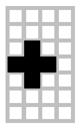
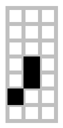
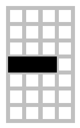
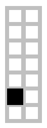
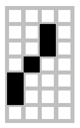
| 00h | 01h | 02h | 03h | 04h | 05h | 06h | 07h |
|---|---|---|---|---|---|---|---|
| NOT USED | SrecurN | SrecurU | SrecurV | SrecurW | Sconvert | SFourSpaces | SsqDown |
|  |  |  |  |  |  |  |  |

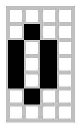
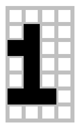
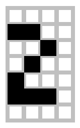
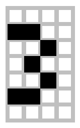
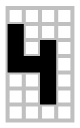
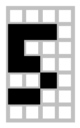
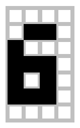
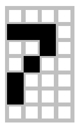
| 08h | 09h | 0Ah | 0Bh | 0Ch | 0Dh | 0Eh | 0Fh |
|---|---|---|---|---|---|---|---|
| Sintegral | Scross | SboxIcon | ScrossIcon | SdotIcon | SsubT | ScubeR | ShexF |
|  |  |  |  |  |  |  |  |

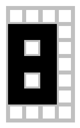
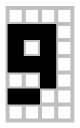
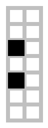
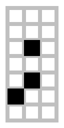
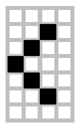
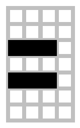
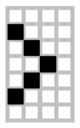
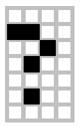
| 10h | 11h | 12h | 13h | 14h | 15h | 16h | 17h |
|---|---|---|---|---|---|---|---|
| Sroot | Sinverse | Ssquare | Sangle | Sdegree | Sradian | Stranspose | SLE |
|  |  |  |  |  |  |  |  |

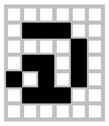
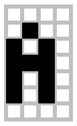
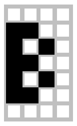
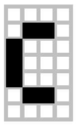
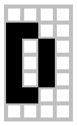
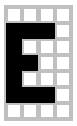
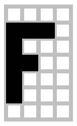
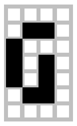
| 18h | 19h | 1Ah | 1Bh | 1Ch | 1Dh | 1Eh | 1Fh |
|---|---|---|---|---|---|---|---|
| SNE | SGE | Sneg | Sexponent | Sstore | Sten | SupArrow | SdownArrow |
|  |  |  |  |  |  |  |  |

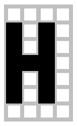
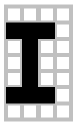
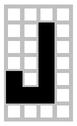
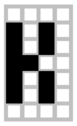
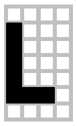
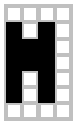
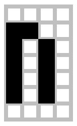
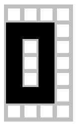
| 20h | 21h | 22h | 23h | 24h | 25h | 26h | 27h |
|---|---|---|---|---|---|---|---|
| Sspace | Sexclam | Squote | Spound | Sdollar | Spercent | Sampersand | Sapostrophe |
|  |  |  |  |  |  |  |  |

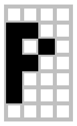
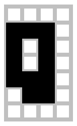
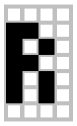
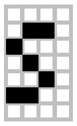
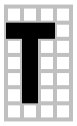
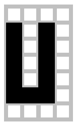
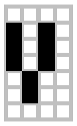
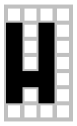
| 28h | 29h | 2Ah | 2Bh | 2Ch | 2Dh | 2Eh | 2Fh |
|---|---|---|---|---|---|---|---|
| SlParen | SrParen | Sasterisk | SplusSign | Scomma | Sdash | Speriod | Sslash |
|  |  |  |  |  |  |  |  |

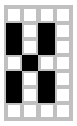
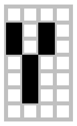
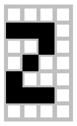
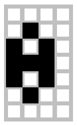
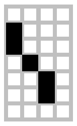
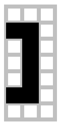
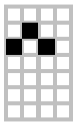
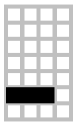
| 30h | 31h | 32h | 33h | 34h | 35h | 36h | 37h |
|---|---|---|---|---|---|---|---|
| S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|  |  |  |  |  |  |  |  |

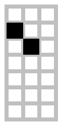
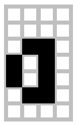
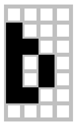
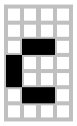
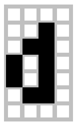
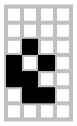
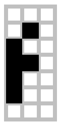
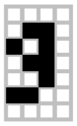
| 38h | 39h | 3Ah | 3Bh | 3Ch | 3Dh | 3Eh | 3Fh |
|---|---|---|---|---|---|---|---|
| S8 | S9 | Scolon | Ssemicolon | SLT | SEQ | SGT | Squestion |
|  |  |  |  |  |  |  |  |

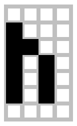
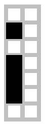
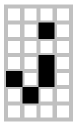
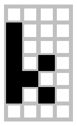
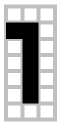
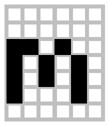
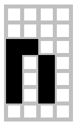
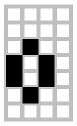
| 40h<br>SatSign | 41h<br>ScapA | 42h<br>ScapB | 43h<br>ScapC | 44h<br>ScapD | 45h<br>ScapE | 46h<br>ScapF | 47h<br>ScapG |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

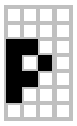
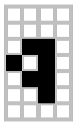
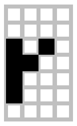
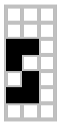
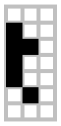
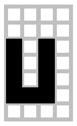
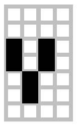
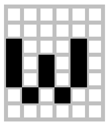
| 48h<br>ScapH | 49h<br>ScapI | 4Ah<br>ScapJ | 4Bh<br>ScapK | 4Ch<br>ScapL | 4Dh<br>ScapM | 4Eh<br>ScapN | 4Fh<br>ScapO |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

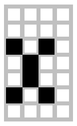
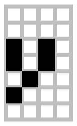
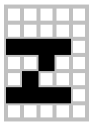
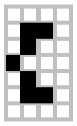
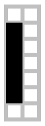
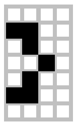
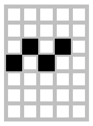
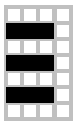
| 50h<br>ScapP | 51h<br>ScapQ | 52h<br>ScapR | 53h<br>ScapS | 54h<br>ScapT | 55h<br>ScapU | 56h<br>ScapV | 57h<br>ScapW |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

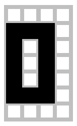
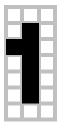
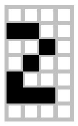
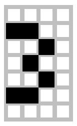
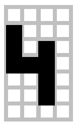
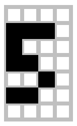
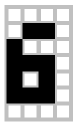
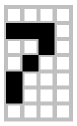
| 58h<br>ScapX | 59h<br>ScapY | 5Ah<br>ScapZ | 5Bh<br>Stheta | 5Ch<br>Sbackslash | 5Dh<br>SrBrack | 5Eh<br>Scaret | 5Fh<br>Sunderscore |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

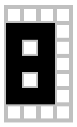
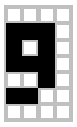
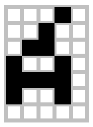
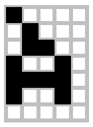
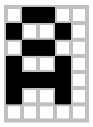
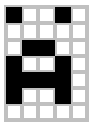
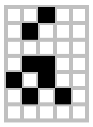
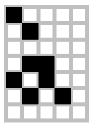
| 60h Sbackquote | 61h SmallA | 62h SmallB | 63h SmallC | 64h SmallD | 65h SmallE | 66h SmallF | 67h SmallG |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

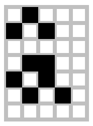
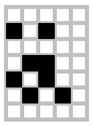
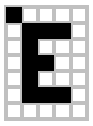
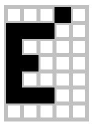
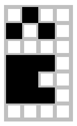
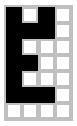
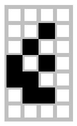
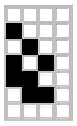
| 68h SmallH | 69h SmallI | 6Ah SmallJ | 6Bh SmallK | 6Ch SmallL | 6Dh SmallM | 6Eh SmallN | 6Fh SmallO |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

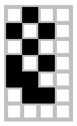
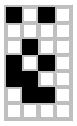
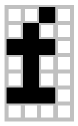
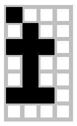
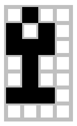
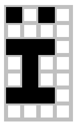
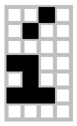
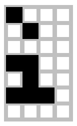
| 70h SmallP | 71h SmallQ | 72h SmallR | 73h SmallS | 74h SmallT | 75h SmallU | 76h SmallV | 77h SmallW |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

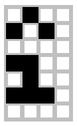
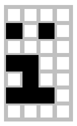
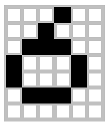
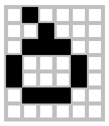
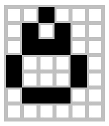
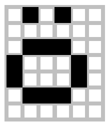
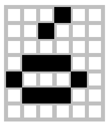
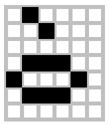
| 78h SmallX | 79h SmallY | 7Ah SmallZ | 7Bh SlBrace | 7Ch Sbar | 7Dh SrBrace | 7Eh Stilde | 7Fh SinvEQ |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

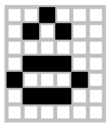
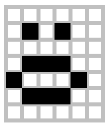
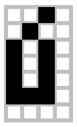
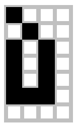
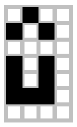
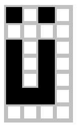
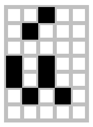
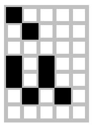
| 80h<br>Ssub0 | 81h<br>Ssub1 | 82h<br>Ssub2 | 83h<br>Ssub3 | 84h<br>Ssub4 | 85h<br>Ssub5 | 86h<br>Ssub6 | 87h<br>Ssub7 |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

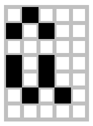
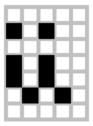
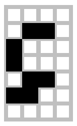
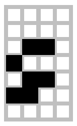
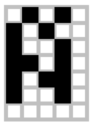
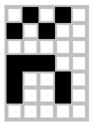
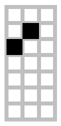
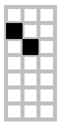
| 88h<br>Ssub8 | 89h<br>Ssub9 | 8Ah<br>ScapAAcute | 8Bh<br>ScapAGrave | 8Ch<br>ScapACaret | 8Dh<br>ScapADier | 8Eh<br>SaAcute | 8Fh<br>SaGrave |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

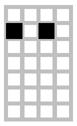
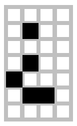
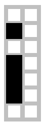
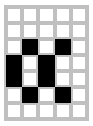
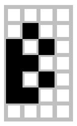
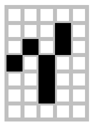
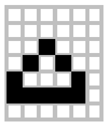
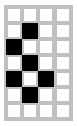
| 90h<br>SaCaret | 91h<br>SaDier | 92h<br>ScapEGrave | 93h<br>ScapEAcute | 94h<br>ScapECaret | 95h<br>ScapEDier | 96h<br>SeAcute | 97h<br>SeGrave |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

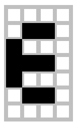
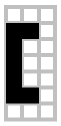
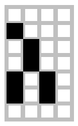
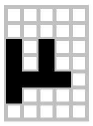
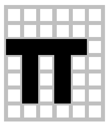
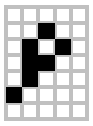
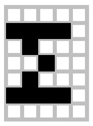
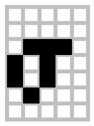
| 98h<br>SeCaret | 99h<br>SeDier | 9Ah<br>ScapIAcute | 9Bh<br>ScapIGrave | 9Ch<br>ScapICaret | 9Dh<br>ScapIDier | 9Eh<br>SiAcute | 9Fh<br>SiGrave |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

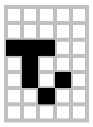
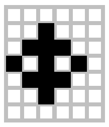
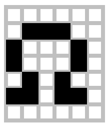
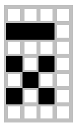
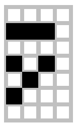
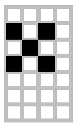
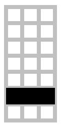
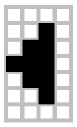
| A0h | A1h | A2h | A3h | A4h | A5h | A6h | A7h |
|---|---|---|---|---|---|---|---|
| SiCaret | SiDier | ScapOAcute | ScapOGrave | ScapOCaret | ScapODier | SoAcute | SoGrave |
|  |  |  |  |  |  |  |  |

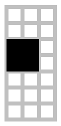
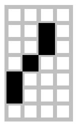
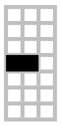
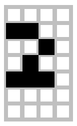
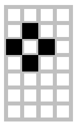
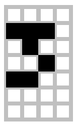
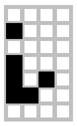
| A8h | A9h | AAh | ABh | ACh | ADh | AEh | AFh |
|---|---|---|---|---|---|---|---|
| SoCaret | SoDier | ScapUAcute | ScapUGrave | ScapUCaret | ScapUDier | SuAcute | SuGrave |
|  |  |  |  |  |  |  |  |

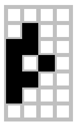
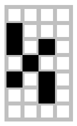
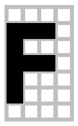
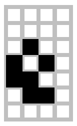
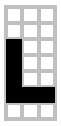
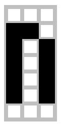
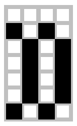
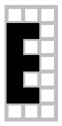
| B0h | B1h | B2h | B3h | B4h | B5h | B6h | B7h |
|---|---|---|---|---|---|---|---|
| SuCaret | SuDier | ScapCCed | ScCed | ScapNTilde | SnTilde | Saccent | Sgrave |
|  |  |  |  |  |  |  |  |

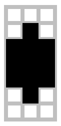
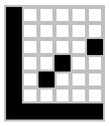
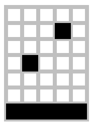
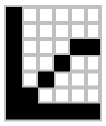
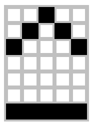
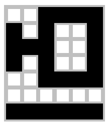
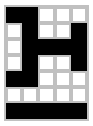
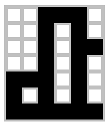
| B8h | B9h | BAh | BBh | BCh | BDh | BEh | BFh |
|---|---|---|---|---|---|---|---|
| Sdieresis | SquesDown | SexclamDown | Salpha | Sbeta | Sgamma | ScapDelta | Sdelta |
|  |  |  |  |  |  |  |  |

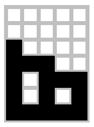
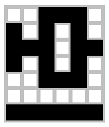
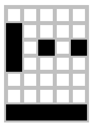
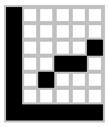
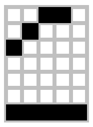
| C0h | C1h | C2h | C3h | C4h | C5h | C6h | C7h |
|---|---|---|---|---|---|---|---|
| Sepsilon | SlBrack | Slambda | Smu | Spi | Srho | ScapSigma | Ssigma |
|  |  |  |  |  |  |  |  |

| C8h | C9h | CAh | CBh | CCh | CDh | CEh | CFh |
|---|---|---|---|---|---|---|---|
| Stau | Sphi | ScapOmega | SxMean | SyMean | SsupX | Sellipsis | Sleft |
|  |  |  |  |  |  |  |  |

| D0h | D1h | D2h | D3h | D4h | D5h | D6h | D7h |
|---|---|---|---|---|---|---|---|
| Sblock | Sper | Shyphen | Sarea | Stemp | Scube | Senter | SimagI |
|  |  |  |  |  |  |  |  |

| D8h | D9h | DAh | DBh | DCh | DDh | DEh | DFh |
|---|---|---|---|---|---|---|---|
| Sphat | Schi | SstatF | Slne | SlistL | SfinanN | S2_r_paren | SnarrowCapE |
|  |  |  |  |  |  |  |  |

| E0h SListLock | E1h Sscatter1 | E2h Sscatter2 | E3h Sxyline1 | E4h Sxyline2 | E5h Sboxplot1 | E6h Sboxplot2 | E7h Shist1 |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |

| E8h Shist2 | E9h SmodBox1 | EAh SmodBox2 | EBh Snormal1 | ECh Snormal2 | | | |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  | | | |

Texas Instruments U.S.A.
7800 Banner Dr.
Dallas, TX  75251

Texas Instruments Holland B.V.
Rutherfordweg 102
3542CG Utrecht-The Netherlands

Printed by: