TI-84 PSE Final Project Report

Robert Boerwinkle & Julia Pitz
Computer Science Department
Missouri University of Science and Technology
Rolla, MO 65401

Introduction

The goal of this project is to create an operating system for the Texas Instruments 84 Plus Silver Edition (TI-84 PSE). The TI line of graphing calculators uses the Zilog Z80 (released 1976), and the PSE is no exception. The Z80 is a powerful processor, and was used in early PC's - like the 1982 ZX spectrum.[1] It has 8-bit words, 16-bit addresses, and a featureful interrupt system.[2]

Unfortunately, the TI-84 PSE does not have many of the peripherals needed for a truly general purpose system. Many modern operating system features are enabled by advances in hardware. Modern segmentation, for example, is reliant on a hardware Memory Management Unit (MMU). The TI-84 PSE only does simple paging and has no segment protections.

This document describes the layout, rationale, abilities, and limitations of the operating system. Many of these are dependent on the hardware of the TI-84 PSE. Some details (memory layout, port functions, etc.) are similar to other calculators, but are only guaranteed for this model. Many values are in hexadecimal, indicated by a "$". As the Z80 is little-endian, this system adopts the same convention.

Memory Layout

There are 2 types of memory: Flash[3] and RAM. The Z80 can only access 16-bit addresses, so the TI-84 PSE has simple paging to enable a larger memory space. Both the Flash and RAM are split up into equal pages; these pages

[1] Wikipedia contributors, "Zilog Z80," *Wikipedia, The Free Encyclopedia*, December 3, 2024, https://en.wikipedia.org/w/index.php?title=Zilog_Z80&oldid=1260959179
[2] "Z80 CPU User Manual," *Zilog*, August 2016, https://www.zilog.com/docs/z80/um0080.pdf
[3] Sometimes referred to as "Flash ROM" or just "ROM".

are assigned to “banks” by writing to specific ports. The 2 high bits address the bank, and the 14 low bits address the offset within the bank/page. There are 4 special Flash pages: $7C, $7D, $7E, and $7F. Pages $7C and $7D are privileged pages. Any code which writes to privileged ports must be executed from these pages. Pages $7E and $7F come preloaded with the certificate and boot code. These are used to load the operating system and check its signature.[4] These 2 pages should normally never be accessed, and are not modified in this project.[5]

Memory Spaces		
Memory Type	Flash	RAM
Executable?	Yes	Sometimes*, Never First Page
Volatile?	No	Yes**
Writable?	Sometimes*	Yes
Page Size	2^{14} = 16KiB	2^{14} = 16KiB
Address Range	$00 – $7F	$80 – $87
Number of Pages	2^{7} = 128	2^{3} = 8
Total Space	2^{21} = 2MiB	2^{17} = 128KiB
Special Pages	$7C & $7D: Privileged Pages $7E: Certificate $7F: Boot Code	$80: No Code Execution
* These are controlled by some special ports. ** The RAM is typically kept intact during power-off by a backup battery.		

[4] WikiTI contributors, “83Plus:Certificate/Headers,” *WikiTI*, April 2015, https://wikiti.brandonw.net/index.php?title=83Plus:OS:Certificate/Headers
[5] WikiTI contributors, “83Plus:Memory Mapping,” *WikiTI*, May 2019, https://wikiti.brandonw.net/index.php?title=83Plus:Memory_Mapping

Memory Banks During Memory Mode 0				
Bank Number	0	1	2	3
Address Range	$0000 – $3FFF	$4000 – $7FFF	$8000 – $BFFF	$C000 – $FFFF
Points To	Page $00	Any Page	Any Page	Any RAM Page
Controlled By	N/A	Port 6	Port 7	Port 5

Software

This project uses a suite of software chosen for Debian GNU/Linux. Some software was built from source. Additionally, “os2tools” was only available in the form of a Visual Basic application, so it was rewritten in Python.

Software Used			
Name	Debian Package	Usage	Links
TiLP	tilp2	Connecting to calculator	http://lpg.ticalc.org/prj_tilp/index.html
TilEm	tilem2	Emulation	http://lpg.ticalc.org/prj_tilem/
rom8x	N.A., built from source	Building roms	https://www.ticalc.org/archives/files/fileinfo/373/37341.html
os2tools	N.A., rewritten from source	Building os files	https://www.ticalc.org/archives/files/fileinfo/420/42044.html
spasm	N.A., built from source	Z80 Assembler	https://github.com/alberthdev/spasm-ng

Emulator and ROM

A ROM is not critical, but very helpful. TilEm emulates the hardware, but the boot code and certificate cannot be distributed without Texas Instument's permission. The ROM contains a copy of the entire flash memory of the calculator. To obtain the ROM, rom8x was used. This software comes with an extensive readme on its operation. In short, it has 2 assembly programs (for the TI-84 PSE). They each need to be sent to the calculator (with TiLP), and then run (with the "ASM" command). They generate files which can then be copied back to the PC (again, with TiLP). Then rom8x combines them with an OS file from TI to produce a ".rom" file. This ROM can then be run with TilEm. At the end of the process, TilEm should boot into TIOS, as originally distributed. After the calculator is running the ROM, it can receive an ".8xu" Operating System file.

Assembling Code

Compile Z80 assembly files into binary files with spasm. Combine the binary files with "Build8XU.py"

The project was built on top of preexisting hardware interface subroutines. These were developed by Drew DeVault for KnightOS and packaged into a minimal OS named AsmOS.[6] Each memory page is compiled separately. This is because (except for relative jumps) code has to know what bank it is loaded into to perform a jump to the right place. The majority of the kernel is on the first Flash page (always loaded into bank 0), so it can be compiled into one binary file. The privileged page and user programs can be loaded into any bank, so they must be compiled with the appropriate ".org $xxxx" instruction at the beginning. This project uses the spasm Z80 assembler. Then they are packed with os2tools.

[6] https://www.cemetech.net/downloads/files/629/x629

Kernel Layout

Memory Page Assignments	
Page	Contents
$00	Kernel code space
$01 - $7B	User code space
$7C	Privileged kernel code space
$7D	Unused
$7E - $7F	Certificate and boot code (unmodified)
$80	Kernel memory space
$81 - $87	User memory space

The PID is the Process Identifier. A PID of $00 indicates no process. PID $01-$07 are the user processes. This also indicates which page of RAM they are allotted ($80 + PID). The “ready queue” is a small queue which indicates which processes are ready to run (by PID). The first entry can be used to indicate the currently running process.

Each process also has a Process Control Block (PCB). This contains miscellaneous information about the process. The PID can be used to index into this table. Notably, this does not contain most of the CPU state, which is stored in user memory space on top of the stack. The stack pointer (SP) for every memory page is stored at the last address (Memory bank offset + $3FFE). Whenever a new memory page is loaded, it is best to adopt its SP. This is somewhat contrary to the classical approach of “ld SP, $0000”, so it is set up for user processes when they are initialized. User programs simply need to use the stack as they normally would (taking care not to reset the SP to $0000 or over-pop).

Top	**Stack Layout for CPU State**									Btm.
HL’	DE’	BC’	AF’	IY	IX	HL	DE	BC	AF	PC

PCB Layout		
Offset	Size	Contents
$00	$01	Code Memory Page
$01	$02	Input Address
$03	$02	Pane Buffer Address
	$05	Total size

The LCD Screen

The LCD screen is addressed by X- and Y-coordinates. Each coordinate corresponds to 8 pixels in a horizontal row. The screen is 96 by 64 pixels, so the X coordinates go from 0 to 11 and the Y coordinates go from 0 to 63. This screen space is divided into 3 sections. The bottom 5 rows of pixels are controlled by the kernel. The rest is split into 2 panes, a left and right. Each process has a graphical buffer which can be displayed in 1 of the panes at any time by the user (called the “pane buffer”). Note that there is no global graphical buffer.

User processes can assign a “pane buffer address” which indicates the start of their pane buffer anywhere in user address space. The buffer itself is a 6-column by 59-row array in column-major order (354 bytes in total). The column-major was chosen because the screen is split vertically. The LCD driver can be put into Y-increment mode, such that every subsequent byte written goes to the next row. During a pane update, the pointer only needs to be repositioned 6 times.

The Keyboard

The keyboard is heavily customized around the TIOS and its function as a calculator. This operating system uses a custom character set designed around a more general system, so a key conversion is required.

There is an input box which can hold a line of text at the far left of the kernel screen space. This is used by the control keys, but can also be sent to a user process. The user process must have first requested input. This input request blocks the process until it receives a line.

Each key has a scancode. Most keys type a character into the input box. The ALPHA key switches between an “alpha” and “numeric” character set. This is modeled after the TIOS input method. Some keys instead call a kernel subroutine when pressed, like selecting a pane, controlling processes, or the ALPHA key itself.

Kernel Control Keys		
Key	Function	Description
Y=	LEFT PANE	Selects the left pane.
GRAPH	RIGHT PANE	Selects the right pane.
2ND	SPAWN PROC	Reads 2 hexadecimal chars from input, spawns a process at that code memory.
MODE	SELECT PROC	Reads 1 hexadecimal char from input, assigns that PID to display on the selected pane.
DEL	KILL PROC	Reads 1 hexadecimal char from input, kills that PID.
ALPHA	ALPHA/NUM	Toggles between alpha and numeric input modes.
LEFT	LEFT	Moves the input cursor a char to the left.
RIGHT	RIGHT	Moves the input cursor a char to the right.
CLEAR	CLEAR	Clears the input buffer.
ENTER	SUBMIT	Sends the input buffer to the process assigned to the selected pane.
ON	ON/OFF	Toggles the calculator on and off (note that the 2ND button is not required, as in TIOS).

Keyboard Layout				Hardware Appearance			Scancode		
				Numeric Function			Alpha Function		
Y=	$21	WIND	$22	ZOOM	$23	TRACE	$24	GRAPH	$25
LEFT PANE								RIGHT PANE	
2ND	$26	MODE	$27	DEL	$28	UP	$29	RIGHT	$2A
SPAWN PROC		SELECT PROC		KILL PROC				RIGHT	
ALPHA	$2B	XT0N	$2C	STAT	$2D	LEFT	$2E	DOWN	$2F
ALPHA/NUM						LEFT			
MATH	$01	APPS	$02	PRGM	$03	VARS	$04	CLEAR	$30
P-CUL	A	P-CUR	B	P-HOR	C	PATET	SPACE	CLEAR	
X^-1	$05	SIN	$06	COS	$07	TAN	$08	^	$09
P-CDL	D	P-CDR	E	<	F	>	G	^	H
X^2	$0A	,	$0B	(	$0C	)	$0D	/	$0E
P-TU	I	P-VER	J	(	K	)	L	/	M
LOG	$0F	7	$10	8	$11	9	$12	*	$13
P-TR	N	7	O	8	P	9	Q	*	R
LN	$14	4	$15	5	$16	6	$17	-	$18
P-TD	S	4	T	5	U	6	V	-	W
STO	$19	1	$1A	2	$1B	3	$1C	+	$1D
P-TL	X	1	Y	2	Z	3	#	+	‘
ON	N/A	0	$1E	.	$1F	(-)	$20	ENTER	$31
ON/OFF		0	_	.	:	=	!	SUBMIT	

Kernel Memory Space Layout		
Offset	Size	Contents
$0000	$08	Ready Queue of PID’s
$0008	$05 * $08 = $28	PCB Table
$0030	$02	Stack Pointer loading address
$0030	$01	PID in Left Pane
$0031	$01	PID in Right Pane
$0032	$01	Pane Flags
$0033	$18	Kernel Pane
$004B	$162	Placeholder Pane
$3FFE	N/A	Bottom of Kernel Stack*
$3FFE	$02	Stack Pointer Storage*
* These values are the same, as the SP decrements before writing values		

Pane Flag Bits				
7	6	5	4	3 - 0
N/A	Key pressed last frame? 0: no; 1: yes	Keyboard Mode 0: alpha 1: numeric	Pane selected 0: Left 1: Right	Cursor position in input box. $0-$B inclusive.

Filesystem and Process Loading

All files are stored in a separate Flash page. Processes are loaded by page number. Notably, this cannot be the kernel page, privileged pages, or the boot / certificate pages. The process is assigned a PID and appended to the ready queue. The user memory space is loaded into bank 4 and the user code space is loaded into bank 2.

Scheduler

At a regular interval, the scheduler interrupt is triggered. The interrupt pushes the old program counter (PC) to the stack, and hands code execution to the scheduler, which executes the following:

- Push the rest of the old CPU state to the stack.
- Write the old SP to the end of the user memory page.
- Load kernel memory space into bank 3.
- Cycle the ready queue, identifying the new PID.
- Load the new code page into bank 1.
- Load the new memory page into bank 3.
- Read the new SP from the end of the user memory page.
- Pop the rest of the new CPU state from the stack.

At the end of the scheduler, the “ret” instruction is executed, popping the new PC from the stack, handing code execution to the newly activated process.

System Calls

System calls are accomplished through accumulating the arguments in a contiguous block with register IX pointing to the first byte (this can be in any of the currently loaded pages), and executing the “rst $08”[7] instruction. Every system call has a code and some parameters. All values are little-endian.

The “DrawText” system call does not interface with kernel-specific data or functions, but is included in place of a shared library.

[7] This functions like “call $08”, but takes slightly less code space.

System Call Layout			
Name	Offset	Size	$Code = Description / Parameter Name
Shutdown	$00	$01	$00 = Shutdown until the ON button is pressed
Spawn	$00	$01	$02 = Spawn process
	$01	$01	Flash page
	$02	$02	Address of resulting PID
Kill	$00	$01	$04 = Kill process
	$01	$01	PID
GetPID	$00	$01	$06 = Gets the calling process PID
	$01	$02	Return address
SetPaneBuffer	$00	$01	$08 = Sets the address of the pane buffer
	$01	$02	Pane address
DrawText	$00	$01	$0A = Draw text to the process’s pane buffer
	$01	$02	Address of the text
WaitInput	$00	$01	$0C = Waits until text is submitted
	$01	$02	Return address

Conclusion

The system has many limitations, but has a solid foundation for a general-purpose operating system. The next steps for improving the operating system would include system calls to support writing new files. This is the biggest hurdle to making this project a full fledged operating system.