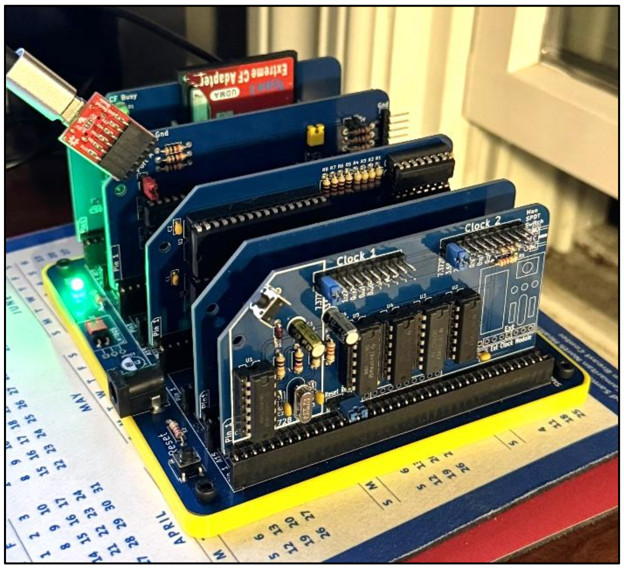
Assembly Language Programming for the RC2014 Zed.

by Bruce E. Hall



It’s been 40+ years since I’ve touched a Z80 microcomputer. I don’t know what came over me when I visited Tindie.com and clicked the ‘Purchase Now’ button for a RC2014 Zed. But two weeks later, my kit was assembled, and I was ready to relive some Z80 goodness.

There is much to enjoy right away: a built-in monitor/debugger. Microsoft BASIC. A full CP/M operating system. Serial communication. XMODEM file transfer. Out of the box, the RC2014 Zed is ready to run a wide array of software.

The path forward is less clear if you are new to retrocomputing, like me, and want to do assembly language programming.

You’ve come to the right place if you use Microsoft Windows and want to compile assembly language programs on your PC for the RC2014 Zed. I will explain what you need and how to start writing your first assembly language program. I assume that you have some knowledge of programming and understand your way around bits, bytes, and hexadecimal numbers.

What is the RC2014 Zed?

It is a Z80 microcomputer built on the RC2014 bus. It contains RomWBW firmware on a 512K ROM/512K RAM module, a Z80-SIO module for serial communication, and a CF module for mass storage.

Get yours at z80kits.com

The first half of this writeup is “Getting to know the RC2014 Zed”. It is meant for newbies who are meeting retro hardware for the first time. Never rush into programming without knowing your programming environment. If you are already a seasoned user, feel free to browse or skip to part 2.

Part 1: Getting to know the RC2014 Zed.

What you need for Part 1:

- Assembled and working RC2014 Zed microcomputer, running RomWBW
- Serial-to-USB adapter with HW flow control (I use Sparkfun Dev-15096).
- Matching USB cable
- PC running Microsoft Windows
- Tera Term software for Windows
- A copy of the RomWBW distribution, downloaded from GitHub.

To burn your own ROMs, I also recommend the following:

- XGecu T48 programmer
- Extra SST39SF040 FLASH chip (512Kx8), DIP-32 package

A few more suggested items, but not required:

- Extra CF card
- SD-to-CF adapter
- CF card reader
- Protective base for RC2014

Connecting to the RC2014 Zed

By now, you've probably figured out how to connect your PC to the RC2014. But if you've just finished assembling your kit and are wondering what to do next, here you go:

First, set up a terminal program on your computer. I am a long time PuTTY user. But I have to say, Tera Term is hard to beat.   Give it a try!

Tera Term setup

1. Get a copy of the .exe installer. Install and run the software. TeraTerm (github.com)
2. Go to serial port setup. Set speed to 115200 and flow control to RTS/CTS.   No transmit delays.
3. Go to terminal setup: Change font to Terminal/Regular/12 point
4. Go to windows setup: Change text color to Green (R0/G255/B0)
5. Save setup to the default configuration file.

The RC2014 Serial Port uses hardware flow control.   When sending data from the RC2014 to the PC, speed is not an issue: the PC is faster and never misses a character.   It *is* an issue, however, when sending data from the PC to the RC2014. The RC2014 uses the RTS line to pause the PC until it is ready to accept the next character.

My homemade USB-to-Serial adapter does not have hardware flow control. So, I purchased one that does (Sparkfun Dev-15096). It uses the standard FTDI pinout, including the CTS line on pin 2, which the RC2014 needs. Bottom-line: your adapter *must* have HW flow control. If it doesn't, you won't be able transfer files to the RC2014.

Next, physically connect your PC to the RC2014. Use a USB cable to connect the PC to the USB-to-serial converter. Then plug the converter into Port 0 of the Dual Serial (SIO) board. My converter plugs directly into the port; some converters require jump wires. The top-most pin of Port 0 is the ground pin. Below that, in order, are RTS, VCC, Rx, Tx and N/C. These pins attach to the converter's GND, CTS, Tx, Rx, and DTR pins, respectively.

RC2014: Getting to know you.

There is a lot to discover before starting down the Assembly Language rabbit hole.

There is a Github repository for the RC2014. The repository is confusing for a beginner 'Zed owner, since much of the content does not apply to RomWBW firmware or the 512K RAM/ROM module.

The RC2014 Zed boots to a ROM bootloader. At the bootloader prompt, type 'H' to see bootloader commands or 'L' to see bootloader applications. For a good review of these applications, please read "ROM applications.pdf" in the docs folder of the RomWBW distribution. Below are a few additional notes.

M: Monitor is homebase for assembly language programming, especially if you are doing it outside of CP/M. From here you can load, view, and run your code. The most useful commands are L (load), R(run), T (transfer), and D (dump). Type X to exit and return to the bootloader. The strange '8E>' prompt tells you that the first 32K of memory is mapped to RAM bank 8E. Feel free to put code anywhere from $0200 to $EE00. The monitor itself lives at $EE00-FE00. The bank switching command is unavailable to you because the 'Zed operates in Interrupt Mode 1.

Z or C: Z-System DOS vs. CP/M. ZSDOS was sold as an "enhanced replacement" for CP/M 2.2. Here are a few of the additional features that ZSDOS provides:

1. ZSDOS programs contain built-in help, accessed with a double slash following the program name. For example, type "ZXD //" to see info about the extended directory program.
2. Error messages are in plain English.
3. Support for larger files/drives
4. File time and date stamping

5. File attributes
6. DOS search path

For me, CP/M 2.2 works well enough. If I were writing programs that use BDOS, perhaps I'd better appreciate ZSDOS's additional capabilities.

F: Forth. CamelForth has its own website. Check out a Forth tutorial here. From the bootloader, press 'F' to enter Forth and 'bye' (lower case) to leave it. You cannot load or save Forth programs from this ROM version.

B or T: Basic comes in two flavors, a full-featured Nascom version of Microsoft Basic and the minimal Tasty Basic. Tasty Basic is not that tasty; stick with the Nascom version. There is a manual for Nascom Basic in the doc/contrib folder of RomWBW. Here is a quickie demo program:

```
10 FOR I = 1 TO 10
20 PRINT TAB(I) "Hello"
30 NEXT
```

Press 'B' to start Basic. At the Memory top? prompt, press <enter>. Type in this demo or copy the text above and paste it into Basic via the Tera Term Alt-V command. Then type 'run' to run it. (Note: Tasty Basic does not support the TAB function.) When you are done, 'bye' will return you to the monitor.

Like Forth, the ROM-based versions of BASIC do not allow you to load or save programs. Run either of them from CP/M (or ZSDOS), however, and you will be able to LOAD "filename" and SAVE "filename". See below for more BASIC commands.

E: Easter Egg. You won't find an 'E' command on the list of Boot Loader apps, but… give it a try!

Booting an Operating System

You may boot CP/M 2.2 or ZSDOS from ROM, as described above. Or you can boot them from disk. The disk contains several additional OS choices. On my RC2014, the disk is a CF card, which is mapped to disk unit #2. To boot an OS from disk, type <device>.<slice> where the OS resides. Here are the device.slice combinations on my CF card:

• 2.0 (disk unit 2, slice 0)	CP/M v2.2
• 2.1 (disk unit 2, slice 1)	ZSDOS
• 2.2 (disk unit 2, slice 2)	Z-Com v2.0
• 2.3 (disk unit 3, slice 3)	CP/M v3.0
• 2.4 (disk unit 2, slice 4)	ZPM3

So, what's a slice? More on that below.

Using Microsoft Basic under CP/M

Here are a few BASIC commands to get you started.

To:	Type:
Start BASIC	MBASIC <enter>
Exit BASIC and return to CP/M	SYSTEM <enter>
Run a Basic program, non-interactively	MBASIC "<filename>"
Renumber a program, starting at 100	RENUM 100
Load a program from disk	LOAD "<filename>"
Save the current program to disk	SAVE "<filename>" When BASIC stores a file, it embeds non-printable characters.
Delete a file (from within BASIC):	KILL "<filename"
Erase current program & start anew	NEW
List the current program	LIST
Edit a program line	EDIT <linenumber> While in edit, press <space> to advance forward through line, press <esc> to leave, press <i> to insert a character, press <d> to delete a character, press <return> to quit editing.
Terminate a running program	<ctrl>C
Set the terminal width	WIDTH <value>

You can copy program listings into interactive MBASIC in the following manner:

1. Select the program text & copy to Windows clipboard in the usual manner (<ctrl>C).
2. Paste the text into Tera Term using <alt>V.
3. Once the listing has been pasted, it can be saved to disk from within Basic, as above.

Fun with ANSI Control Codes

I've avoided BASIC for decades. But now it's time to play and add a little color to our monochromatic terminal. The terminal's foreground and background colors can be controlled by sending ANSI escape codes. In general, these codes consist of the escape character + left bracket "[" + the remaining code. Some examples:

Code	Description
<esc>[3#m	Set foreground color to # (0..7). For example, “<esc>[33m” is yellow text.
<esc>[9#m	Set bright foreground color to # (0..7)
<esc>[4#m	Set background color to # (0..7). Example: “<esc>[41m” is red background
<esc>[10#m	Set bright background color to # (0..7)
<esc>[38;5;#m	Set foreground color to # (0..255)
<esc>[48;5;#m	Set background color to # (0..255)
<esc>[38;2;r;g;bm	Set foreground color to RGB
<esc>[48;2;r;g;bm	Set background color to RGB
<esc>[2J	Clear the screen with background color
<esc>[H	Put cursor at 0,0
<esc>[r;cH	Put cursor at row, column

The basic colors are 0=black, 1=red, 2=green, 3=yellow, 4=blue, 5=purple, 6=cyan, and 7=white. From MBASIC, the escape character is sent by “PRINT CHR$(27);” The semicolon is required to stop a carriage return from being sent. Start BASIC, then enter a few PRINT statements using these codes. For example, PRINT CHR$(27);”[44m” will change the background color to blue. What does PRINT CHR$(27);”[2J” do?

Useful CP/M commands

CP/M looks and feels like MSDOS, but there’s enough difference to trip you up.

• DIR A:	Display list of files on drive A: Use wildcards for file matching.
• ERA <file>	Erase the file. Can use wildcards.
• PIP <newf>=<oldf>[optns]	Copy command. To copy to another drive: PIP H:=A:FILE.TXT
• REN <newfile>=<oldfile>	Rename the file.
• USER <n>, n=0-15	Switch to user area of disk drive, with its own directory.
• TYPE <filename>	Echo contents of file to console. Use <ctrl>S to pause.
• HELP <command>	Get help with CP/M commands.

Note: PIP has lots of options. Try: “HELP PIP OPTIONS”. When using PIP, do not put a space between the filename and “[option]”.

For a lot more information, RTFM in the RomWBW distribution: doc/CPM Manual.pdf. A manual for ZSDOS is there, too: doc/ZSDOS Manual.pdf.

CP/M Text Editors

I don't fancy text editing on my RC2014. Years ago, I used "vi" and loved it. But that is ancient history. I find it much easier to create and edit files on my PC, transferring them to the RC2014 as needed.

The standard CP/M text editor is a line editor called ED. It is terrible (my opinion only, mind you).

If you must edit files on the RC2014, try ZDE, which is a visual text editor. The escape command is <esc> (or ^K). Then press S (save), X (save and exit), Q (quit without saving), L (load a new file). Help with commands is <esc>h.

CP/M Disk Organization

If you are like me, you entered 'C' at the bootloader, watched CP/M 2.2 boot, then entered DIR to see a directory listing. And you noticed drives A: through J:. Did you look at what was on those other drives?

It's confusing. There are files in each drive. Why are files distributed throughout those drives? And why do the listings look similar? Many files exist in more than one directory. For example, why are there copies of MBASIC.COM on the B, C, D, E, and F drives?

All the answers are in the documentation; it just takes a while to find them.

Today, mass storage devices are much larger than the disk drives of the 1980's. The operating systems of yore could not handle file systems larger than 8 MB. Therefore, the compact flash on my system is divided into "slices", with each slice exactly 8 MB in size. Any of these slices can then be assigned to a drive letter, A: through P: (CP/M-like systems allow for up to 15 lettered drives.)

Initially, RomWBW maps the first eight (8) slices from your mass storage device(s) to drive letters. In addition, it maps two (2) non-disk drives, which are ROM and RAM. As you might guess, writing to a RAM drive is not writing to disk, but to RAM memory. A file saved to the RAM drive ceases to exist when power is lost. You can read from the ROM drive but cannot write to it.

Just as an interesting aside, type "STAT" at the CP/M prompt to see how much free space is available on each drive/slice currently in use. On my system:

```
C>stat
A: R/W, Space: 246k
C: R/W, Space: 6464k
E: R/W, Space: 5324k
F: R/W, Space: 6116k
G: R/W, Space: 5652k
H: R/W, Space: 7076k
```

Translation: about 6MB remains available on each 8MB drive. There is plenty of room.   The RAM drive is smaller.   Unused drives I: and J: are not shown.

Here is a table of the formatted slices on my combo-image CF card.   Yours may be different.

CF Slice#	Type in:	Boots to Operating System	Drive Letter (as seen in CP/M 2.2)
0 (IDE0:0)	‘2.0’ or ‘2’	CP/M 2.2	C:
1 (IDE0:1)	‘2.1’	ZSDOS 1.1	D:
2 (IDE0:2)	‘2.2’	NZCOM ZCPR 3.4	E:
3 (IDE0:3)	‘2.3’	CP/M 3	F:
4 (IDE0:4)	‘2.4’	ZPM3	G:
5 (IDE0:5)	‘2.5’	None. Wordstar & ZDE applications.	H:
6 (IDE0:6)	n/a	None. Empty drive for user storage.	I:
7 (IDE0:7)	n/a	None. Empty drive for user storage.	J:

The first five disk-based drives, C: through G:, have operating systems installed on them.

Said another way, the CF card contains 5 different operating systems: CP/M 2.2, ZSDOS, ZCPR 3.4, CP/M 3, and ZPM3. You can boot to each of them by entering 2, 2.1, 2.2, 2.3, 2.4, and 2.5 at the bootloader prompt, respectively, corresponding to the drive.slice where the OS resides.

Question: from the bootloader, how do you boot CP/M 3?   Answer: type 2.3<return>.

Once booted, you have access to all drives, including the non-disk (RAM and ROM) drives.   This means that you can view files used by the other operating systems. Which explains why there are so many copies of MBASIC (one copy in each OS filesystem).

Once booted into your OS of choice, type ASSIGN to see how the drive letters are mapped to the slice numbers. The assignments in CP/M 2.2 are NOT the same assignments in CP/M 3.   You can also use ASSIGN to change the drive letter assignments.

Question: If only 8 slices get assigned to drive letters, and each slice is 8MB in size, is the maximum file space limited to 64 MB?   Answer: No.   Eight slices are assigned to drive letters at boot, but you can enlist more slices at any time. It’s a two-step process.   First, use ASSIGN to establish a new drive, linking it to a memory storage slice.   Then, format the slice with a proper directory.   For example:

```
A> ASSIGN K:=IDE0:8
A>  CLRDIR K:
```

Now IDE0:8 is properly formatted, and drive K: will be available to your system.   For example, we can copy files to it:

```
A>  PIP K:=MAPPER.COM
```

copies the file mapper.com to drive K.   Your file is safe & sound in slice #8 of the Compact Flash drive IDE0:, but to see it after rebooting you'll need to re-assign this slice to K (or to some other drive letter).

Use CLRDIR with care! When you CLRDIR a drive, any files on that drive will be deleted.

CP/M subdirectories

Sorry, there aren't any.   Each drive is limited to a single directory.   You cannot create subdirectories in CP/M, such as "A:\GAMES\ZORK.COM".   For organizational purposes, however, each drive can be divided into 16 different user areas.   From the CP/M 2.2 manual,

> The USER command allows maintenance of separate files in the same directory. In the syntax line, n is an integer value in the range 0 to 15. On cold start, the operator is automatically logged into user area number 0, which is compatible with standard CP/M 1 directories. You can issue the USER command at any time to move to another logical area within the same directory. Drives that are logged-in while addressing one user number are automatically active when the operator moves to another. A user number is simply a prefix that accesses particular directory entries on the active disks. The active user number is maintained until changed by a subsequent USER command, or until a cold start when user 0 is again assumed.

So, type 'User 1' and you are presented with a new, empty directory! (Technically, you are seeing the same directory, but viewing only files associated with User 1.) You now have access only to files in the User 1 area. Switch back to 'User 0' and the directory reappears.   Hint: If at some point your directory seems wrong or suspiciously empty, try switching to User 0.

Typing DIR at the prompt shows you only User 0 files.   Don't assume that you are seeing all the files on that drive.   On my CF drive, there are interesting files in User areas 2, 3, and 4.   See the README.TXT file in User 0 for brief descriptions.

Firmware Backup

Making a copy of your firmware is useful and instructive. Follow the steps below.

1. Purchase the T48 programmer. It is available from the manufacturer XGecu and also available from many vendors on Amazon.
2. Download the T48 software. The manufacturer's website is difficult to navigate; It is easier to get it from GitHub here: XGecu_Software.
3. Use the free 7-zip utility to extract the software, which is in RAR format.
4. Download the latest release of the RomWBW firmware from GibHub.
5. In the Binary directory, locate the ROM image file: **RCZ80_std.rom**. This is a binary file.

6. Put a new SST39SF040 chip into the programmer. Note that the programming socket is larger than the chip; the chip must be positioned as far away from the lever as possible.
7. From the menu, choose Select IC -> Search and Select. Type 39SF040 into the search box and select the device named SST39SF040. Choose the plain variety, not the PLCC32 or TSOP32 variants.
8. Click "Load" and select the RCZ80_std.rom file. Use the following options: File Format = Binary, To Region = Default, Load Mode = Normal, Clear Buffer = to default.
9. Click "Prog." To see programming setup screen, then click on "Program". It takes about 25 seconds to program the entire 512 KB chip.
10. Power down the RC2014, then remove the old ROM.
11. Install the newly programmed chip into the RC2014 and restore power.

Compact Flash Backup

You don't need to backup your original CF card right away. Why not? RomWBW comes with all the tools required to make a new CF card. When you are ready, here is the procedure.

First, consider the following purchases:

- A new CF card.
- Alternatively, a SD-to-CF adapter, so that you can use SD cards.
- A CF card reader.

Then create a new CF card that contains the RomWBW software:

1. Locate the Win32 Disk Imager in the RomWBW/tools directory
2. Locate the "hd1k_combo.img" disk image file in RomWBW/Binary directory
3. Install a new CF card (or SD card with CF adapter) into your PC's CF reader.
4. Run Win32 disk imager, specifying the disk image file and the CF drive.
5. Turn off the RC2014 and remove the original CF card. Keep it safe.
6. Install the newly formatted CF card and reapply power.

Note that the CF cards are configured for CP/M. You cannot view or modify their contents under Windows. It *is* possible to create a CF card with DOS and CP/M partitions, but hey, I'm just a beginner.

Part 2: Assembly Language Programming

```
03 00 00 00  00 00 00 00  ..................
00 00 00 00  07 00 00 00  ..`.............H..
A0 03 40 00  00 00 00 00  ...............@....
00 00 00 00  FE FF FF 6F  ..................
F0 FF FF 6F  00 00 00 00  ......@........o...
00 00 00 00  00 00 00 00  t.@...............
00 00 00 00  00 00 00 00  ..................
00 00 00 00  00 00 00 00  ..................
36 04 40 00  00 00 00 00  ....P.`...........
00 00 00 00  47 43 43 3A  F.@.....V.@........
75 35 29 20  34 2E 36 2E   (Ubuntu/Linaro 4.6
00 2E 69 6E  74 65 72 70  3...symtab..strtab.
6C 64 2D 69  64 00 2E 67  ..note.ABI-tag..not
2E 76 65 72  73 69 6F 6E  nu.hash..dynsym..dy
65 6C 61 2E  70 6C 74 00  ..gnu.version_r..re
68 5F 66 72  61 6D 65 5F  .init..text..fini..
2E 6A 63 72  00 2E 64 79  hdr..eh_frame..ctor
73 73 00 2E  63 6F 6D 6D  namic..got..got.plt
00 00 00 00  00 00 00 00  ent...............
00 00 00 00  1B 00 00 00  ..................
```

What you need for Part 2. Everything from Part 1, plus:

- Visual Studio Code for Windows.
  (If you don't like VSC, try Notepad++)
- A copy of RomWBW from GitHub.
- A good reference book, such as:
  "Programming the Z80", by Rodnay Zaks

To burn your own ROMs, I also recommend the following:

- XGecu T48 programmer
- Extra SST39SF040 FLASH chip (512Kx8), DIP-32 package
- HxD Hex Editor

Visual Studio Code

I chose VSC because it looks good, is widely used, and is extensible. Too complicated? A much simpler code editor is Notepad++. I keep a copy of both.

1. Download the latest version from https://code.visualstudio.com/Download.
2. Upon opening an .asm file, VSC offered to install syntax highlighting. I chose "Z80 Assembly/Imanoloa"

Assemblers

There are many assembler choices. I don't know which are the best, but I did not want to suffer from "paralysis by analysis". I picked two:

1. TASM (Telemark Assembler) – because it is bundled with RomWBW.
2. z88dk – because it is in active development and includes a C compiler.

Of the two, TASM is the simpler one to get running. I am going to start with that.

If you want to develop <u>on</u> the RC2014, there is an 8080-assembler bundled with CP/M. I decided against using it, since I prefer coding and compiling on my PC.

TASM installation

1. Copy the assembler file from the RomWBW tools folder to a convenient spot: "c:\tasm".
2. Edit the environment variables to include this path.
3. Add the variable TASMTABS and set its value to c:\tasm
4. Open a command prompt and type tasm. Make sure it works. Check out the compiler options.
5. *RTFM*. It is "tasmman.htm"

TASM notes for those who don't read the manual:

- Invoke by "tasm -80 <options> <filename>".
- The object file format is Intel Hex by default (-g0).
- To produce a binary object file, use option -g3 (same as -b), which also forces option -c (contiguous block output). This option is useful for code destined for ROM: all locations will have a known value. Often used with -f, below.
- Fill option (-fFF) fills unused memory with $FF. Similarly, -f00 fills with zeroes.
- The Intel Hex file is 24 bytes per line by default. Change it to 16 bytes with -o10 (or 32 bytes with -o20).
- Numbers are decimal by default.
- Enter hex numbers like this: $E3F4, 0E3F4H (number cannot begin with alpha digit).
- Binary numbers: 01101B, %01101
- The location counter symbol is "$".
- String constants may be used with DB, BYTE, TEXT directives & can include \r, \n,\t, \\, \"
- Examples of Byte and Word directives. (.BYTE, .byte, and .db are equivalent)

```
.db $0F
.dw $FFE0
.db 'a'
.db "Hello", 13, 10, "World",0
.byte "Hello World\r\n"
```

- Labels are case sensitive, up to 32 characters in length.
- Symbols can be defined at compile time with -d option. For example, -dINTMODE1
- IFDEF/ELSE example:

```
#IFDEF INTMODE1
<code here>
#ELSE
<code here>
#ENDIF
```

- Code usually starts with the .ORG directive and ends with the .END directive:

```
.ORG $0100      ; start program at $0100
<code here>
.END            ; follows the last line of code/data
```

- You can add additional .ORG statements in the program listing. For example, ".ORG $+8" will reserve 8 bytes of space without assigning any values.

Your first assembler program: onechar.asm

Let's create a super-simple, "Hello, World" style assembler program which outputs a single character on the console screen. Here is the code for onechar.asm. You may type or copy/paste it into Visual Studio Code, or download it from my bhall66 GitHub page:

```
.ORG $0100

    LD  b, $01        ; HBIOS function $01 = CHARACTER OUTPUT (CIOOUT)
    LD  c, $80        ; Device Number $80 = current console
    LD  e, '~'        ; character to be sent to console
    RST 08            ; call the HBIOS routine
    RET

.END
```

The first line locates the program's ORiGin (starting point) at $0100. This is a very common place to start user applications, since the first 256 bytes of memory from $0000 to $00FF are usually reserved for interrupt vectors and restart instructions.

The next four lines are used to call the character-out routine in HBIOS. This routine is provided by RomWBW as part of its hardware API. The character to be displayed is loaded into register E. Calling the routine is accomplished by the RST 08 instruction. This instruction jumps to location $0008 and continues until a return instruction is encountered. HBIOS has previously set up location $0008 as a jump instruction to an appropriate handler routine.

All the HBIOS routines are documented in the RomWBW System Guide, Chapter 8.

The RET instruction at the end returns control back to the debug monitor. Finally, the .END directive informs the compiler that there is no more code or data to compile.

Compile the code in a windows command shell with "tasm -80 onechar.asm". The compiler will create an object file in the Intel Hex format. You can read the contents of this file with Notepad or similar application. It will contain two lines that look like this:

```
:08010000006010E801E7ECFC92E
:00000001FF
```

The first line indicates that 8 bytes of code should be placed at location $0100: 06 01 0E 80 1E 7E CF C9. This is the entire program. The last byte on the line ($2E) is the checksum. The second line indicates "end of file".

Compiling also creates a list file, onechar.lst. It is also readable by Notepad or similar application:

```
0021   0100              .ORG $0100
0022   0100
0023   0100 06 01            LD  b, $01        ; HBIOS function $01 = CHARACTER OUTPUT (CIOOUT)
0024   0102 0E 80            LD  c, $80        ; Device Number $80 = current console
```

```
0025   0104 1E 7E           LD  e, '~'        ; character to be sent to console
0026   0106 CF              RST 08            ; call the HBIOS routine
0027   0107 C9              RET
```

Each line in the listing represents a line in the source file. Numbers in the first column are line numbers. The second column represents the program counter, or memory location where the compiled code or data will be loaded. The machine code to the right of the program counter may be one, two, or three bytes in length. Finally, the source code and/or comments from the source file are shown for reference.

To transfer the onechar program to the RC2014, do the following:

1. Apply power and connect to the serial port using Tera Term.
2. At the boot prompt, enter 'M' for the debug monitor.
3. The monitor will return with the prompt "8E>" (8E is the user memory page) .
4. Press 'L' to load a program in Intel Hex format.
5. From Tera Term, go to 'File' -> 'Send File' and select "onechar.obj"
6. Press the Open button to begin the transfer.
7. The debug monitor will respond 'Loaded' when the transfer is completed.
8. Now run the program by typing 'R 100' in the debug monitor.

If successful, you will see a tilde character '~' displayed after the R 100. An additional line of numbers will be displayed below it. Your screen should look something like this:

```
8E>R 100~
00 0100 EE7E 0181
```

The second line shows the contents of the A, BC, DE, and HL registers after running your program. In this case, A=$00, BC=$0100, DE=$EE7E, and HL=$0181.

What do you think about Z80 assembly language? Are you already familiar with the concepts, or do you want to learn more? The classic book for learning more is "Programming the Z80" by Rodnay Zaks. It is a useful reference for the remaining programs in this tutorial. You can find copies of it on the Internet in all the usual places.

Bare-Metal, HBIOS, BDOS, or BASIC?

In the first program, we sent a character to the console using an HBIOS routine. What is HBIOS? It is code that provides a standard programming interface for the system hardware. Programs that call HBIOS for basic input and output (character I/O) will run on any system that uses RomWBW. HBIOS provides a first layer of hardware abstraction: our program will work on any RomWBW system, even on systems with different hardware.

Bare-metal programming, on the other hand, eschews all compatibility concerns and writes directly to the hardware. In doing so, we narrow our focus to specific hardware and lose compatibility with other systems. RomWBW users with 68B50 UARTs will not be able to run a program meant for the Z80-SIO chip.   Moreover, bare-metal programs require detailed knowledge of how to make the hardware work – nitty-gritty details normally handled by HBIOS. A principal advantage of bare-metal programs is speed. They are small and fast.

If we want to write programs for all CP/M users, we must avoid HBIOS calls since those are specific to RomWBW. Instead, we call on a component of CP/M called BDOS, or basic disk operating system. The BDOS further abstracts the hardware and adds additional capabilities such as file storage and retrieval. BDOS includes a set of standard function calls, just like HBIOS. Here is the onechar program rewritten for BDOS:

```
.ORG $0100          ; CP/M programs start at $0100

    LD  c, $02      ; BDOS function $02 = Console output
    LD  e, '~'      ; character to be sent to console
    CALL $0005      ; call the BDOS routine
    RET

.END
```

If we want to write programs that can be used by multiple operating systems, we could add another layer of abstraction and use higher-level languages, such as BASIC.   Here is the onechar program, written in BASIC:

```
10  PRINT “~”;
```

Advantages of BASIC programming are simplicity, ease of use, and compatibility across numerous operating systems. But layers of abstraction come at a price.   BASIC programs are often much slower than assembly.

As programmers we have choices.   Do we want blazingly fast code, specific to our hardware? Or do we want slower code that has broader applicability? In the examples that follow, I will explore low-level code that stays close to the hardware, running in a debug monitor/HBIOS environment.

One final note: the programs that follow were written by me. I’ve taken ideas from various sources, where applicable, and sprinkled them into the mix. There was little attempt to make my programs efficient, elegant, or optimized.   My goals were to get them to work on the RC2014, to relearn Z80 programming, and to demonstrate a few principles along the way. Please refer to the “Programming the Z80” book for more detail on the Z80 instructions.

Your second assembler program: printchar.asm

Sending a character to the screen is useful. But remembering how to set up registers for HBIOS is cumbersome. Let's repackage the print-character routine so that we load the character in Register A and call a named routine to print it. That's easier to remember. Let's also save the registers, so that the calling code doesn't have to worry which registers are saved and which aren't. Here is the code for printchar.asm:

```
.ORG $0100

    LD   A, '~'          ; display a tilde
    CALL printCh         ; do it
    RET                  ; and return

printCh:
    PUSH BC              ; save current registers
    PUSH DE
    PUSH HL
    LD   B, $01          ; HBIOS function $01 = CHARACTER OUTPUT (CIOOUT)
    LD   C, $80          ; Device Number $80 = current console
    LD   E, A            ; load E with character to be displayed
    RST  08              ; call the HBIOS routine
    POP  HL              ; restore registers after HBIOS call
    POP  DE
    POP  BC
    RET

.END
```

Compile it using "tasm -80 printchar.asm" as in the previous example. You can check the listing and output files if you like. Then upload it to the debug monitor using Tera Term, as above. If you do everything correctly, you should see a tilde on screen, the same as before. So, why bother with a longer program to do the same thing? Because it encapsulates the HBIOS routine and relieves us from remembering all the details. The printCh routine is easier to use.

Getting User Input: getchar.asm

Now let's consider how to get user input. The HBIOS call for getting a character is very similar. Compare the following two routines:

```
printCh:              getCh:
    PUSH BC               PUSH BC             ; save registers
    PUSH DE               PUSH DE
    PUSH HL               PUSH HL
    LD   B, $01           LD   B, $00         ; HBIOS fn 1=OUTPUT, 0=INPUT
    LD   C, $80           LD   C, $80         ; device $80 = current console
    LD   E, A                                 ; put char into E
    RST  08               RST  08             ; call HBIOS
                          LD   A, E           ; get char into A
    POP  HL               POP  HL             ; restore registers
    POP  DE               POP  DE
    POP  BC               POP  BC
    RET                   RET
```

The getCh code is nearly the same as printCh, except that it calls HBIOS function $00 (character input) instead of function $01 (character output).

With these two routines, can you write a program that gets character input and then prints it? Consider the following:

```
start:
   CALL  getCh             ; wait for a character
   CALL  printCh           ; print it
   RET                     ; and quit.
```

It works but it's boring, terminating after one keypress. With a few more instructions we can continuously print keyboard input:

```
start:
   CALL  getCh             ; get a character
   CP    13                ; if <enter> key
   RET   Z                 ; then quit
   CALL  printCh           ; otherwise, print it
   JR    start             ; and get next character
```

We need a way for the program to stop, so the line 'CP 13' compares each input character with the enter key (ASCII 13). As soon as <enter> is received, the 'RET Z' Return on Zero instruction causes the program to end. See getchar.asm for full program listing.

Question: What happens if we insert the instruction 'INC A' before the call to printCh? Try it!

Printing strings: printstr.asm

Now let's put printCh to use and write a program that displays strings. Here is the code for printStr.asm. For brevity I have omitted the '.ORG' directive at the beginning and '.END' at the end, but you should remember they are still required.

```
    LD   HL, str        ; point to the string
    CALL printStr       ; display the string
    RET                 ; and return

printStr:
    LD   A, (HL)        ; get character in the string
    OR   A              ; is it zero?
    RET  Z              ; if it is, we are done.
    CALL printCh        ; otherwise, display it
    INC  HL             ; move to next character
    JR   printStr       ; and loop until done

printCh:
    PUSH BC             ; save current registers
    PUSH DE
    PUSH HL
    LD   B, $01         ; HBIOS function $01 = CHARACTER OUTPUT (CIOOUT)
```

```
        LD   C, $80          ; Device Number $80 = current console
        LD   E, A            ; load E with character to be displayed
        RST  08              ; call the HBIOS routine
        POP  HL              ; restore registers after HBIOS call
        POP  DE
        POP  BC
        RET

    str:
        .db  13,10,"Hello, RC2014!",0
```

Let's review the code. The first three lines should be self-explanatory: point to a string, call a routine to print it, and then return. "LD HL, str" causes the HL register to contain the address of the string we wish to print. The printStr routine works by starting at the beginning of the string. One by one, each character is loaded into Register A and printed by our printCh routine. We advance to the next character by incrementing the HL register pair. And we stop whenever we encounter zero, the null character. The "OR A" instruction (Do a logical OR operation between the A and itself) is rather strange but serves to set the Z80's zero flag. Z80 load operations do not set the zero flag, so we use OR A.

The last line contains the string itself. ".db" stands for "define byte", meaning that byte data follows. It can be a single byte or a comma-separated list of bytes such as a decimal number (0-255), a character in single quotation marks ('g'), or string of characters in double-quotation marks ("hello"). Note that the last data item, 0, marks the end of our string. Without this zero, your code will print characters beyond the confines of your program until it encounters a zero - somewhere. Don't forget the zero.

Compile the code using "tasm -80 printstr.asm" and load it into the RC2014 as before. If all goes well, the string will print on the console.

Reading strings from the console: getstr1.asm

So far, we can write characters, read characters, and write strings. Reading strings turns out to be trickier than the preceding three programs, so let's tackle the code one step at a time.

First, create an area of memory – a buffer – to store the string. The buffer has a finite size, placing a limit on the largest string that can be entered. Consider the following lines of code:

```
maxLen   .equ  80        ; string length limit of 80 characters

buffer:
   .FILL maxLen,0       ; reserve <maxLen> bytes for input string
```

The .FILL directive reserves maxLen bytes of memory. We use the HL register pair to point to this memory buffer. Our program to read a string and then echo it back to the console takes the following form:

```
start:
   LD    HL, buffer      ; point to input string buffer
   CALL  getStr          ; and get string from user

   LD    HL, buffer      ; point to input string buffer
   CALL  printStr        ; and echo string to console
   RET
```

The only missing component is getStr. As a first approximation, I tried the following code, modeled on the getchar program above:

```
getStr:
   CALL  getChar         ; get a character into A
   CP    13              ; if it's <enter> key
   JR    Z, gs1          ; then quit
   LD    (HL), A         ; save character to buffer
   INC   HL              ; increment buffer pointer
   CALL  printCh         ; echo input to console
   JR    getStr          ; and get next character
gs1:
   XOR   A               ; terminate the string
   LD   (HL), A          ; with null character
   RET
```

This looks very similar to getchar.asm, doesn't it? Get a character, exit when <enter> pressed, otherwise print the character and jump back to the beginning. The two additions are highlighted in yellow. First, 'LD (HL), A' stores the character in the memory buffer. Once the character is stored in memory, 'INC HL' advances the HL pointer to the next available buffer location.

When the enter key is pressed, the string should be terminated with $00. 'LD A, 0' would be a perfectly fine way to zero out the accumulator, but there is a more efficient way: XOR A. An exclusive OR takes only 4 processor cycles and one byte of memory. Compare that with LD A, 0 which takes 7 cycles and 2 bytes of memory. The downside is that XOR A is less readable. Either instruction is fine.

The full program is contained in getstr1.asm. Compile and run it. Does it work? Are we done?

Reading Strings, Part 2: getstr2.asm

The preceding program reads a string from the console and echoes it – all in one line. Let's add carriage returns to separate the input and output lines. The crlf routine does nothing more than send <carriage return> and <line feed> characters to the console.

```
start:
   CALL  crlf            ; start on new line
   LD    HL, buffer      ; point to input string buffer
   CALL  getStr          ; and get string from user

   CALL  crlf            ; start on new line
   LD    HL, buffer      ; point to input string buffer
```

```
      CALL  printStr        ; and echo string to console
      RET
```

A bigger issue is error checking. What happens if the user types 300 characters? The program happily adds all those characters to a buffer which is only 80 characters wide. So where does the 300th character go? Answer: 220 bytes after the 80th character, beyond the confines of the program, into memory that may be in use by other software.

We need a method to limit user input to 80 characters. The actual limit is 79 characters since we must save one byte for the terminating zero. One solution is to count the characters as they arrive. After each one, check that we've not exceeded the count:

```
   getStr:
      LD    B, 0            ; B contains character count
   gs0:
      CALL  getCh           ; get a character into A
      CP    13              ; if it's <enter> key
      JR    Z, gs1          ; then quit
      LD    (HL), A         ; save character to buffer
      INC   HL              ; increment buffer pointer
      CALL  printCh         ; echo input to console
      INC   B               ; increment character counter
      LD    A, B
      CP    maxLen-1        ; reached max size yet?
      JR    C, gs0          ; no, so keep going
   gs1:
      XOR   A               ; terminate the string
      LD   (HL), A          ; with null character
      RET                   ; and quit
```

The added parts are highlighted. Register B is used for counting characters. Note that we increment the counter for each character and compare the count to maxLen-1. The compare instruction internally subtracts 79 from the count (without changing the count or accumulator) and sets the flags accordingly. If the count is less than 79, a carry is generated, and we loop back via a 'jump if carry' instruction to get the next character. On the 79th character, however, the terminating null is added, and the program quits with a full buffer of 80 characters. The full code is available as getstr2.asm. Idea: could we count the other way, 79 down to 0, and use the DJNZ instruction?

Reading strings, part 3: getstr3.asm

Unfortunately, we are not quite done. Try entering a string with a few typos at the end, then backspace to correct them. Does the program behave as expected? Were the incorrect characters erased from the screen? Were they removed from the output string? An end-user will not be happy with a program that cannot handle simple backspace edits.

We must do two things. First, the character must be erased from the screen. Second, we must update our buffer and character count to reflect the removed character. Here it is in code:

```
CALL   printCh           ; move cursor back one
LD     A, ' '            ; and print a space
CALL   printCh           ; to erase last character
LD     A, 8              ; now print <backspace> again
CALL   printCh           ; to move cursor back

DEC    HL                ; decrement buffer pointer
DEC    B                 ; decrement character counter
```

The backspace character is $08. The first 5 lines do the erasing in 3 steps: a) print a backspace to move the cursor over the offending character, b) print a space which ‘erases’ the character, and finally, c) print another backspace to move the cursor back once again.

The last two lines adjust the count and buffer pointer by decrementing each.

There is one final wrinkle: nothing should happen if the user presses <backspace> at the beginning of the string, right? Check out the final code in getstr3.asm.

Simple math: dogyears.asm

After making a collection of useful routines, it is handy to bundle them together for future use. I put my routines in a file called bhUtils.asm. Now, check out the silly and irreverent program “dogyears.asm”, which estimates a dog’s biological age as calendar-age x 7.

There is no built-in Z80 instruction for multiply. We can write our own multiply routine (or copy one off the ‘net), but for simple multiplication it’s easier to use addition and subtraction. The following code snippet takes an age in the HL register and multiplies it by 7:

```
LD     B, H              ; multiply age x 7 as follows:
LD     C, L              ; copy HL to BC
ADD    HL, HL            ; HL x 2
ADD    HL, HL            ; HL x 4
ADD    HL, HL            ; HL x 8
SBC    HL, BC            ; HL x 7 (subtract one, get it?)
CALL   printNum16
```

The first two lines copy HL to BC. Next, use the ADD HL, HL instruction to add HL to itself, thereby multiplying it by 2. Three of these ADD instructions have the effect of multiplying HL by eight (2 x 2 x 2 = 8). Finally, the SBC instruction subtracts one multiple, making the result HL x 7. You should probably clear the carry before SBC, but we will assume no carry here.

The next three sections discuss bare-metal programming, or writing directly to the peripheral hardware.

Bare-Metal programming, part 1: writing to the SIO chip.

What does bare-metal mean? In general, it means interfacing directly with machine hardware. In our case, it means writing characters directly to the SIO chip, rather than sending those characters through the HBIOS. Your code no longer depends on HBIOS to run. There is nothing between you and the Z80, SIO, etc. Your code is smaller, and it runs faster. The disadvantage is that you've lost the hardware abstraction layer. You must learn how to program the serial interface chip (SIO). And you've lost the ability to run the same code on systems that use a different serial interface chip.

Let's learn about the SIO chip. It is a 40-pin DIP chip that provides two independent serial ports, called channel A and channel B. In the simplest setup, both ports are configured to run at 15200 baud. Sending data to/from the chip involves writing/reading from IO ports associated with the chip. The device is configured by sending setup information via the command port. Serial data is transmitted using the data port. The standard RC2014 location of these ports is as follows:

SIO Port Details	Channel A	Channel B
Command/Status Port	Port $80	Port $82
Data Port	Port $81	Port $83

There are 8 writable registers associated with each command port. These registers are responsible for establishing the desired transmission characteristics, such as baud rate, parity, start bits, stop bits, interrupt behavior, etc. There are 3 readable registers that return status information, such as whether the transmit/receive buffers are full, or if there is a parity error in the data. All this information flows through the command port. The data port, on the other hand, is used for receiving keyboard input and sending console output over the USB cable.

Sending a command to a channel requires two operations. First, send a byte to the command port which points to the register and second, send the data to be written to that register. For example:

```
OUT ($80), $04      ; point to WR4
OUT ($80), $EA      ; set WR4 value to EA
```

Will set the value of SIO write register #4 (WR4) to $EA.

Consider modifying the printStr program to write directly to the SIO chip. What about the following?

```
printCh:
    OUT  ($81), A        ; write char to SIO data port $81
    RET
```

That's a great starting point: it *will* send data to the SIO chip, to be serialized and sent over Channel A. But it does not check to see if the SIO hardware is ready to process the character. Specifically, we must ensure the transmitter buffer is empty and ready to send another character. Otherwise, we overrun the transmitter buffer and data gets dropped from the output.

It's time to pull out the datasheet. There is a Z80 peripheral User Manual that describes the SIO, PIO, etc. Discussion of SIO chip starts on page 207. Asynchronous operation begins page 231. Programming on page 272. After nosing around, you'll find that pages 292-293 describe how bit 2 of RR0 indicates the status of the transmitter buffer. If this bit is 1, the buffer is full; if its 0, the buffer is empty and ready to receive the next character. Here is the modified code:

```
printCh:
    PUSH AF             ; temp save character
pcloop:
    IN   A, ($80)       ; get SIO status (RR0)
    AND  $04            ; check Tx-buffer-empty bit
    JR   Z, pcloop      ; wait until buffer empty
    POP  AF             ; retrieve char to send
    OUT  ($81), A       ; write char to display
    RET
```

Note that IN A,($81) gets the contents of Read Register 0, which includes the Tx buffer status on bit 2. The AND $04 operation isolates this Tx buffer bit, so that A=0 when the buffer bit is 0. The following JR Z instruction creates a loop that continues checking until the buffer bit turns to 1. A pair of PUSH/POP statements surrounding the loop are used to temporarily save the character while the SIO bit is checked.

Check out sio.asm for the full program. And, just for fun, look at sioColor.asm which expands the test string and changes its color.

Bare-Metal programming, part 2: initializing the SIO chip

The previous code assumes that the SIO chip has already been initialized by HBIOS. This initialization includes parameters such as serial data mode, baud rate, number of data bits, stop bits, and parity. But if we are starting from scratch, we must learn how to do that ourselves.

As mentioned above, there are 8 configuration registers for each channel in the SIO chip. According to the peripheral manual, a reset channel command should be issued first, followed by setting the parameters in WR4. The order after that is not spelled out, but WR2, WR3, and WR5 will need to be programmed. WR6 and WR7 are not used for asynchronous communication. Wading through the manual reveals the following nitty-gritty details:

1. First do a channel reset by writing $18 to WR0.
2. Do WR4 next, which sets up UART baud,sync,stop bits, parity. See page 284 for details.
    - 115200N1 = $C4 is encoded as follows:

Baud1	Baud0	Sync1	Sync0	Stop1	Stop0	Parity1	Parity0
1	1	0	0	0	1	0	0

- Parity: x0=none, 10=odd, 11=even. Set to 00 (no parity)
- Stop bits: 01=1 bit, 10=1.5 bits, 11=2 stop bits. Set to 1.
- Sync mode: 00=8 bit sync, 01=16 bit sync, 10=SDLC, 11=external. Set to 00.
- Baud rate: 00=clock freq, 01=clock/16, 10=clock/32, 11=clock/64. Set to 11 (7.3MHz/64)

3. In WR1, set interrupt and wait/ready modes. See page 277 for details.

- No interrupts = $00; Int on Received Chars = $18. Encoded as follows:
- Wait/Rdy bits determine function of the /Wait and /Ready pins. 0xx = disabled
- Rx Interrupt Modes: 00=disabled, 01=Int on 1st char, 1x=Int on all chars.
- Status Vector: determines which vector returned for Interrupt acknowledge.
- Tx Int Enable: 1=interrupt when Tx buffer becomes empty.
- Ext Int Enable: 1=interrupt if DCD/CTS,SYNC inputs change, etc.

Wait/Rdy 2	Wait/Rdy 1	Wait/Rdy 0	Rx Int Mode 1	Rx Int Mode 0	Status Vector	TX Int Enable	Ext Int Enable
0	0	0	0	0	0	0	0

4. In WR2, set up interrupt mode 2 vector for Ch B only. See page 281 for details.
5. Set up WR3, receiver control. Set to $E1, as follows. See page 282 for details.
    - b0 = receiver enable. Set to 1.
    - b1 = inhibit loading sync chars. Set to 0.
    - b2 = applies to SDLC mode. Set to 0.
    - b3 = Rx CRC enable. Applies to SDLC & sync mode. Set to 0.
    - b4 = Enter Hunt phase. Set to 0.
    - b5 = if 1, /DCD & /CTS become Rx & Tx enables, respectively.
    - b6/b7: 00 = 5 bits/char, 01=7 bits, 10 = 6 bits, 11=8 bits/char. Set to 11.
6. Set up WR5, transmitter control. Set to $EA, as below. See page 287 for details.
    - b0 – Tx CRC enable, set to 0
    - b1 = RTS, set to 1. When 1, /RTS is low. When 0, RTS goes high on Tx buffer empty.
    - b2 = CRC, set to 0.
    - b3 = Tx enable, set to 1.
    - b4 = Send break, set to 0.
    - b5/b6 = Tx bits/char, set to 11. 00=5 bits, 01=7bits, 10=6bits, 11=8bits
    - b6 = Tx bits/char 1
    - b7 = DTR, set to 1. When set, /DTR is active low.

The SIO registers are each programmed by sending two bytes to the command port: a pointer to the register (think of it as the register address), followed by the data for that register. We can do this with a series of OUT instructions to port $80, something like this:

```
OUT ($80), $00       ; point to WR0
OUT ($80), $18       ; issue a channel reset command

OUT ($80), $04       ; point to WR4
OUT ($80), $C4       ; set baud=115200, no parity, 1 stop bit

OUT ($80), $01       ; point to WR1
```

```
        OUT ($80), $18       ; set interrupt to all received chars
        …
```

Here is another method. The Z80 has a nifty instruction for sending multiple bytes to an IO port: **OTIR**. To use this instruction, load register C with the port number, register B with the number of bytes to send, and HL with a pointer to the start of the data. OTIR does the rest. The following code will initialize channel A of the SIO chip with "reasonable" values for the RC2014. It's neat and tidy.

```
sioInit:
    LD   C, $80           ; SIO command port, channel A
    LD   HL, initRegs     ; point to table of register values
    LD   B, 10            ; count of bytes to write
    OTIR                  ; write all values to SIO cmd port
    RET

initRegs:
    .db $00, $18          ; wr0: reset the channel
    .db $04, $C4          ; wr4: baud=115200, no parity, 1 stop bit
    .db $01, $18          ; wr1: no interrupts=$00, on received chars=$18
    .db $03, $E1          ; wr3: Rx 8 bits, Rx enabled, cts/dcd auto
    .db $05, $EA          ; wr5: Tx 8 bits, Tx enabled, dtr&rts ($E8 no RTS)
```

Should you initialize the SIO with multiple OUT instructions, or should you use OTIR? Either is fine; the choice is yours.

Put all the pieces in part 2 together, and you get program listing sio1.asm. Try compiling and running it as before. Try commenting out the initialization "CALL sioInit" and you should not notice any difference. This is because our initialization code sets up the SIO channel exactly as our RC2014 expects.

Now change the value in register WR4, so that the receive buffer does not generate an interrupt. Can you figure out how to do that? (Change the line "`.db $01,$18`" to "`.db $01,$00`"). The output will be the same, but… the keyboard does not respond anymore. This is because HBIOS expects the SIO chip to fire an interrupt every time a character arrives. No interrupt = no response. Press the <Reset> button to get back to normal.

Bare-Metal programming, part 3: Burn your code to ROM

Since this program doesn't need HBIOS, can we burn our code on to a ROM/PROM/Flash chip? ¡Sí, se puede! But there is a small problem with our 512K ROM/512K RAM board: when first powered up, it looks like 64K of ROM to the Z80. Until programmed otherwise, all that RAM is unavailable. And, without read/writable memory, the Z80 can't have a stack. And THAT means that our code cannot contain any PUSH/POP instructions, nor can it CALL any subroutines. We are limited to one "routine" from start to finish. It's quite limiting when you think about it. We need RAM to do useful programming.

We will try to unlock the RAM later, but for now – let's rewrite our code so that it doesn't use a stack. In other words, remove all CALLs, RETurns, PUSHes, and POPs. Here it is:

```
sioCmd  .equ    $80     ; port for SIO Channel A cmd/status
sioDat  .equ    $81     ; port for SIO Channel A data

.ORG $0000

    LD   C, sioCmd      ; SIO command port, channel A
    LD   HL, initRegs   ; point to table of register values
    LD   B, initLen     ; count of bytes to write
    OTIR                ; write all values to SIO cmd port

    LD   HL, lambStr    ; point to the test string
getCh:
    LD   A, (HL)        ; get character in the string
    OR   A              ; is it zero?
    JR   Z, done        ; if it is, we are done.
    LD   B, A           ; save char in reg B
pcloop:
    IN   A, (sioCmd)    ; get SIO status (RR0)
    AND  $04            ; check Tx-buffer-empty bit
    JR   Z, pcloop      ; wait until buffer empty
    LD   A, B           ; retrieve char
    OUT  (sioDat), A    ; write char to display
    INC  HL             ; move to next character
    JR   getCh          ; and loop until done

done:
    HALT
```

For brevity I do not include the initialization table or the test string; see sio2.asm for those details. Note that the program starts at address $0000 and ends with the HALT instruction – it starts at the beginning of the address space, and it does not transfer control to HBIOS.

Compile the code as usual. It is OK to compile this in Intel Hex format, just like the previous examples. My chip programmer is the T48 programmer from XGecu. The software is available online in RAR format; use the free 7-zip utility to extract it. You can download it from the manufacturer's website, but it is easier to get it from Github here: XGecu_Software.

The IC used as ROM on the memory board is the 512K flash chip "SST39SF040". Purchase a couple of those chips at Digikey, Mouser, or your favorite part supplier. Make sure to get the 32-pin DIP variety.

Chip Programming on the T48:

1. Insert the flash chip into the programmer ZIP socket. Notice that the chip does not fill the socket. The notched/pin1 side of the chip should be as close as possible to the Pwr light/USB port.
2. Start the Xgpro software. From the menu, choose Select IC -> Search and Select. Type 39SF040 into the search box and select the device named SST39SF040. Choose the plain variety, not the PLCC32 or TSOP32 choices.
3. From the menu, choose File -> Load File. Browse for and select "sio2.obj". In the File Format box select "Intel Hex", which is not the default choice. Click OK.
4. Verify that the main screen shows program code from address 0000 through 00B0. Most of that should be the test string in ASCII format.
5. Click on the 'PROG' button to program your device.

Replacing your ROM chip:

1. Turn off the power to the RC2014.
2. Remove the memory board
3. *Carefully* lift the original SST Flash chip (bottom left) from the board, using an IC removal tool. A flat screwdriver will also work.
4. *Carefully* install the newly programmed chip. Sometimes is helpful to straighten the pins before attempting insertion.
5. Reinstall the memory board and restore power to the RC2014.
6. Press the <Reset> button. The Z80 should execute your code from the ROM.

Each time you press the <Reset> button, the test string should be displayed on the console. On my system, I will occasionally need to restart Tera Term and/or the RC2014 after replacing the ROM chip.

The Memory Management Unit (MMU)

Schematic here: [512kROMRAM.pdf (b-cdn.net)]

The memory card contains 512K of RAM and 512K of ROM. But the Z80 can only access 64K of memory at a time. It is the job of the Memory Management Unit (MMU) to map 1MB of memory into the 64K address space. The following discussion applies only to the MMU on the 512K ROM/ 512K RAM board. This MMU design is known as Zeta2, named after the computer which popularized its use.

Memory Division	Memory Address
0	0000-3FFF
1	4000-7FFF
2	8000-BFFF
3	C000-FFFF

The MMU is hardware. It divides the Z80 memory space into four 16K divisions. The memory chips are each divided into 32 pages, each page containing 16K. By writing to certain IO ports, we can select which of the 32 RAM pages or 32 ROM pages gets mapped into each memory division. For example, ROM page 0 can map to memory division 0, and RAM pages 0-2 can map to memory divisions 1-3.

How is this done? Two 74HCT670 chips are used to form a four-byte bank register.

Each byte in the bank register contains 7 useful bits in form: 0ccp.pppp, where the 'c' bits are used to select the chip (ROM vs RAM) and the five 'p' bits are used to select one of the 32 memory pages in the chip. For example, setting a bank register to 0x01 will enable Rom Page 1. Setting it to 0x21 will enable Ram page 1.

Bank Register Value	16K physical memory selected
0x00	ROM – page 0
0x01	ROM – page 1
0x02	ROM – page 2
0x1F	ROM – page 31
0x20	RAM – page 0
0x21	RAM – page 1
0x3F	RAM – page 31

The four bank registers are written to by writing to IO port $70 - $73. A 74HCT138 is used to select the $70 base address via address lines A5-A7. The A0 and A1 address lines then select which one of the four bank registers is being written to. For example, writing to port $70 will select the Bank Register 0, which corresponds to the memory division 0. By writing 0x00 to port 70, we map ROM page 0 to memory address $0000-3FFF. By writing 0x21 to port $71, we map RAM page 1 to memory addresses $4000-7FFF.

Finally, we must select ROM page 0 on reset, and then later enable this banking system. Initialization is done with a 74HCT74 flip-flop which disables (tri-states) the bank register outputs. A resistor network to ground on these outputs ensures that all are at logic zero and that the ROM chip is enabled. On reset, the entire 64K address space is ROM. Once the four bank registers are written as above, Page Enable is set by writing port $74 (base address of 70, plus address line A2). The upper two address lines will then select one of the four bank registers and indirectly address one of the 32 RAM/32 ROM pages. It is a very clever scheme, requiring only 4 chips.

Configuring the MMU is a simple matter of writing to the four MMU registers and enabling them. Here is code for configuring 32K ROM followed by 32K of RAM:

```
LD   A, $00
OUT  (mmuReg0), A   ; Memory $0000-$3FFF = 00 (ROM page 0)
LD   A, $01
OUT  (mmuReg1), A   ; Memory $4000-$7FFF = 01 (ROM page 1)
LD   A, $20
OUT  (mmuReg2), A   ; Memory $8000-$BFFF = $20 (RAM page 0)
LD   A, $21
OUT  (mmuReg3), A   ; Memory $C000-$FFFF = $21 (RAM page 1)
LD   A, $01
OUT  (mmuEnbl), A   ; Enable MMU
```

The MMU Registers are at ports $78-7B, and the enable register is at port $7C.

After configuring the MMU, let's see if it worked.   We can write a test byte to address $8000 (which should be RAM) and then read it back.   If it's the same value, then we know the Z80 was successful writing to that location – a telltale sign that $8000 is now mapped to RAM and not ROM. The test is only 5 lines of code:

```
LD   HL, $8000      ; at memory location $8000
LD   A, $DC         ; use test byte $DC
LD   (HL), A        ; try saving the byte
CP   (HL)           ; then read it back & compare
JR   Z, itWorked    ; if same, RAM was written correctly
```

The file mmu.asm attempts to configure the MMU, then tests memory $8000 for RAM. Finally, it prints a simple message depending on whether it found RAM or not.   Compile it, burn it to ROM, and run it in the RC2014. If all goes well, it will configure your memory and confirm that RAM is now available to the Z80.   Hooray!

A simple memory mapper

If we can check one memory location for its type (ROM/RAM), we can check as many as we like. What about a mapper program that shows us the memory type throughout the 64K address space?   Something like this, where each 2K segment of memory is labelled 'r' for read-only (ROM) memory and each writable (RAM) segment is labelled 'W':

```
+-----------------------------------------+
| 0    8k   16k  24k  32k  40k  48k  56k  |
+-----------------------------------------+      (32K ROM / 32K RAM)
| rrrr rrrr rrrr rrrr WWWW WWWW WWWW WWWW |
+-----------------------------------------+
```

Fancy printing like that is hard to do without subroutines, so let's try for simpler output: a single line of 32 characters, one for each 2K memory segment.   See "mapper.asm" for the full program listing.

First, initialize the SIO for serial output:

```
initSIO:
    LD   C, sioCmd      ; SIO command port, channel A
    LD   HL, initRegs   ; point to table of register values
    LD   B, 10          ; count of bytes to write
    OTIR                ; write all values to SIO cmd port
```

Next, initialize the MMU so that we can see both ROM and RAM:

```
configMMU:
    LD   A, ROM0
    OUT  (mmuReg0), A   ; Memory $0000-$3FFF = ROM page 0
    LD   A, ROM1
    OUT  (mmuReg1), A   ; Memory $4000-$7FFF = ROM page 1
    LD   A, RAM0
```

```
        OUT  (mmuReg2), A    ; Memory $8000-$BFFF = RAM page 0
        LD   A, RAM1
        OUT  (mmuReg3), A    ; Memory $C000-$FFFF = RAM page 1
        LD   A, 1
        OUT  (mmuEnbl), A    ; Enable MMU
```

Next, set up our ramTest routine do look at 32 pages of 2K each, starting at $0000

```
    ramTest:
        LD   DE, $0800       ; page size of 2K
        LD   B, 32           ; evaluate 32 pages of memory (64K)
        LD   HL, $0000       ; starting at $0000
```

For each page, test the first byte and print r or W depending on test result:

```
    rtLoop:
        LD   C, (HL)         ; temp save memory contents
        LD   A, $DC          ; use test byte $DC
        LD   (HL), A         ; try saving the byte
        CP   (HL)            ; then read it back & compare
        LD   (HL), C         ; restore memory contents
        JR   Z, foundRAM      ; if same, RAM was written correctly
    foundROM:
        LD   A, 'r'          ; ROM found, so print 'r'
        JR   printCh
    foundRAM:
        LD   A, 'W'          ; RAM found, so print 'W'
    printCh:
        OUT  (sioDat), A     ; write char to display
```

Finally, go to the next page until all 32 pages have been evaluated.

```
        ADD  HL, DE          ; point to next memory page
        DJNZ rtLoop          ; and loop until done
        RET
```

Burn it to ROM and start your system. You should see something like this:

```
    rrrrrrrrrrrrrrrrWWWWWWWWWWWWWWWW
```

Granted, it's not fancy output, but seeing the ROM/RAM distribution on your device is instructive. This mapper confirms that the MMU initializer did, in fact, create a 32K block of ROM followed by a 32K block of RAM.

Now that we've shown we can burn ROMs with our own SIO and MMU initializers, let's return to life under the RomWBW monitor.

The remaining examples assume that you are using your RC2014's debug monitor. Replace your ROM with the RomWBW chip. Then try "mapper2.asm" for the output format we initially wanted, and "mapper3.asm" for a splash of color.

Running an assembly language program under CP/M

CP/M wants to load and run programs at $100. So, if you've specified a $100 origin in your programs, things will go very smoothly. Return to the print string program, printstr.asm. Compile it as before, but with the "-g3" option:

```
tasm -80 -g3 printstr.asm
```

The compiler creates the object file, printstr.obj, as usual. The -g3 option causes the compiler to format the output as binary code, instead of using the Intel Hex format.

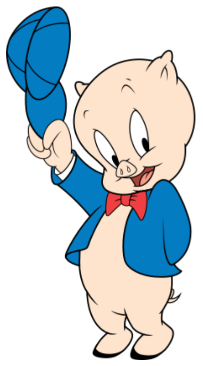
CP/M can run the binary object file directly. All we need to do is place it in the CP/M filesystem. I will use the ROM version of CP/M for this example.

1. Start ROM-based CP/M from the monitor with C <enter>. CP/M 2.2 will start with ROM-based commands (including XM) on the B drive, and compact flashed-based files on the C,D,E.. drives.
2. From the B> prompt, type: "xm r c:printstr.com".
3. From Tera Term menu, select File > Tranfer > XMODEM > Send.
4. Choose the file "printstr.obj" that you compiled (using the -g3 option) as above.
5. After the file has transferred, type "dir c:" to confirm the PRINTSTR.COM is present.
6. Run the program by typing "c:printstr". Do not include the filename extension ".com".

A few words are in order. In CP/M 2.2, .com files contain nothing more than machine code. The .com extension tells CP/M to "load this file at address $100 and run it". The command "xm r c:printstr.com" runs the XMODEM file transfer program. If you'd like to see a summary of its options, type "xm" at the CP/M B> prompt. The 'r' option tells it be in receive mode and place received data in the file c:printstr.com. Note that by loading printstr.obj on the PC and receiving printstr.com on the RC2014, we have not only transferred the code but renamed it, giving our code the proper CP/M extension for an executable file.

So, what's the deal with the A: drive? On startup the A: drive contains nothing. It is a RAM drive, which means that its contents do not persist on power-down. It is useful for temporary storage. You can transfer programs to the A: drive, especially if you just want to try them out. You may even reset the RC2014 and reboot CP/M; the files will still be there. But once you power-down, they are gone.

That's all, folks!



We come to this hobby with varying interests. Of all Z80 enthusiasts, only a fraction uses the RC2014 platform. And of the RC2014 users, a minority run RomWBW. And of those, a smaller number yet want to explore assembly. If you happen to be one of those intrepid souls, I hope that you found something useful here.

Bruce.

Last updated: 9 October 2023

Appendix

Converting Intel Hex to Binary format

An easy way to convert an Intel Hex file into binary is via the HxD Hex Editor.

1. Start the HxD hex editor
2. From the menu, select: File > Import > Intel Hex.
3. Choose the Intel Hex file. Its contents will be displayed onscreen.
4. Note that if your Hex file had an origin offset, the display will show it as a one-line block like this: "00000000-000041FF".
    a. If you keep this block intact, it will result in a large bin file with addresses 0 through the origin filled with zeros.
    b. If you select and delete the one-line block, it will create the smaller .bin file that you probably want. Just remember that the smaller bin file will not preserve the starting location of your program; you will need to remember where the program should be loaded.
5. From the menu, select: File > Save. Give the new filename a ".bin" extension.

Converting Binary to Intel Hex format

The HxD hex editor lets us do this, too:

1. Start the HxD hex editor.
2. From the menu, select File > Open and choose the binary file to open. It will load the file contents at offset $0000.
3. If the desired program origin is not $0000, insert bytes to the beginning of the file so that the program begins at the desired origin. For example, to set the origin at $0200:
    a. Put the cursor at the first byte (0000).
    b. From the menu, Edit > Insert Bytes
    c. Choose 200H bytes of 00
4. Select the program block by highlighting it. Important: if you fail to highlight the program block, the exported file will include all preceding zeroes. This is probably NOT what you want.
5. From the menu, select File > Export > 16 bit Intel Hex. Give the filename a ".hex" extension.