

# CDP-1802 Development System





# CDP-1802 Development System



## Overview

The 1802 Development System is a custom-built microprocessor tool that allows rapid prototyping of 1802-based systems or other systems that use the type 2716 EPROM.

Many of the connections to the internal 1802 microprocessor are brought to the front panel. There are LEDs that indicate the state of many of the lines, and the EFn lines each have a momentary pushbutton connected to it.

## Serial Interface

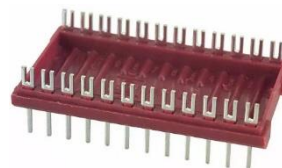
There is a “CRT” DB-25 connector for RS-232 serial interface. The Development System is controlled primarily using a terminal emulator connected to this port. **The port specs are: 9600 baud, 7-bit, odd parity, 1 stop bit, XON/XOFF flow control.**



There is a connector marked “PRINTER” which is not used.

## ROMulator

There is a connector marked “ROMULATOR” which allows access to 2Kx8 RAM inside the unit. A cable can be made to connect from the ROMULATOR port to a 24-pin header that will appear to be a type 2716 EPROM to any circuit it is connected to.



The cable wiring from the DB-25 male to the “2716” 24-pin header is:

2716:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
DB-25:	1	14	2	15	3	16	4	17	5	18	6	19	25	12	24	11	23	10	22	9	21	8	20	7

## ROM Socket

There is a “ROM 2716” ZIF (Zero Insertion Force) socket on the front panel. A 2716 may be placed into this socket and “burned” from internal RAM. It can be written-to directly, like RAM, albeit slowly and only zero-bits.

## Reset Button

There is a large “RESET” button that is used to return control to the terminal.

## Memory

The memory of the system is organized as:

0x0000 - 0x07FF: System ROM  
0x0800 - 0x08FF: RAM (system scratch pad)  
0x4000 - 0x47FF: ROM Socket on Front Panel: Read/Write  
0x5000 - 0x57FF: 2K RAM/ROMulator Connector

## Getting Started

Connect a terminal or terminal emulator program to the “CRT” connector. You may want to use a 25-to-9 pin converter and a USB-to-serial adapter cable. The Windows utility “HyperTerminal” will work. In the terminal emulator, select the COM port of the USB adapter cable. Set the parameters to:

- 9600 Baud
- 7 bits
- Odd Parity
- 1 Stop Bit
- Flow Control: XON/XOFF

Press the RESET button. You should see:

```
Reset
```

```
*
```

The asterisk is the command prompt. If you don’t see this, your serial interconnect and/or its settings are probably not right.

## Commands Overview

The control system supports six commands: **List**, **Edit**, **Move**, **Go**, **Rom Empty?** and **Upload**.

To list the first page (256 bytes) of the RAM, type “L 5000” and hit enter:

```
*L 5000
```

```
5000 BB 36 9E 06 F6 76 9A 66 9A B6 C2 66 9F 66 12 46
5010 9D F2 B9 76 99 16 8A 66 CB 06 18 46 98 46 0A 76
5020 97 C6 1A 66 DA 66 D6 66 99 8E 32 76 8A E6 1A 66
5030 DB 96 99 A6 99 E6 D9 46 99 4D 89 A5 19 66 99 2E
5040 9D 96 D2 66 9A 86 92 66 99 66 99 26 99 CE F2 66
5050 BB 36 9E 06 F6 76 9A 66 9A B6 C2 66 9F 66 12 46
5060 99 77 D5 06 99 46 54 66 95 D3 B9 06 91 F6 80 E6
5070 5B 74 BA 76 1A 46 89 26 7B 6E 9D 7E 99 E6 49 66
5080 8B 82 89 66 95 66 98 66 99 42 98 B6 99 D2 18 66
5090 99 86 DE C6 99 46 D2 E6 9A 36 84 46 8A 66 B4 66
50A0 96 86 9E 36 96 A6 47 66 9F B6 DE 6E 9C 76 92 66
50B0 99 2E 98 D6 99 D6 F8 E6 9B 25 9B 66 99 D7 32 36
50C0 9F 82 85 46 91 A6 D6 46 9F F6 96 66 93 26 02 66
50D0 9F 17 94 46 B7 66 95 64 99 B5 91 76 99 86 F1 66
50E0 99 CA 90 26 91 76 0A 66 D9 52 99 36 98 16 98 66
50F0 9A F6 96 E6 8D A6 26 66 19 B9 D9 E4 9B FB 09 24 *
```

It is random data at startup.

With no ROM in the ROM socket, list the first page of ROM:

```
*L 4000
```

```
4000 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4030 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4040 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4050 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4060 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4070 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4080 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
4090 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40A0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40B0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40C0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40D0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40E0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF *
```

Because there is no device in the socket, it sees FF. Let's copy 256 FFs to the RAM. This will make it easier to see our changes. Copying is done with the **Move** command. We want to copy the data starting at 0x4000 through location 0x4FF, to RAM starting at 0x5000.

Type:

```
*M 4000,40FF,5000
Done
```

This takes a fraction of a second. Verify by listing from 0x5000 again:

```
*L 5000

5000  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5010  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5020  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5030  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5040  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5050  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5060  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5070  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5080  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5090  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50A0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50B0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50C0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50D0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50E0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50F0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF *
```

Note that the Move command handles overlapping moves correctly, although this example was not an overlapping move. An overlapping move lets you shift a block of data down or up by less than the size of the block: M 5000,5004,5001 or M 5001,5005,5000.

Let's enter a simple program at 0x5000 that will make the Q LED blink on and off. Here is the program in hex:

```
7A F8 0F BF 2F 9F 3A 04
31 00 7B 30 01 00 00 00
```

We use the **Edit** command to enter the data:

```
*E 5000
5000  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
```

Type in the hex codes. Use backspace to go back, spacebar to skip forward. Use the ">" and "<" keys to move in 16-byte jumps. To exit the Edit mode, hit Enter or CTRL-C.

```
5000  7A F8 0F BF 2F 9F 3A 04 31 00 7B 30 01 00 00 00
```

We can list it:

```
*L 5000
```

```
5000  7A F8 0F BF  2F 9F 3A 04  31 00 7B 30  01 00 00 00
5010  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5020  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5030  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5040  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5050  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5060  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5070  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5080  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
5090  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50A0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50B0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50C0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50D0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50E0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
50F0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF *
```

Now let's run it. Use the Go command, at 0x5000:

```
*G 5000
```

The Q LED should blink on and off. To end the execution and regain control, preset the RESET button.

```
Reset
```

```
*
```

The LED blinks pretty fast. The time constant is at location 0x5002. Let's use the E command again to change it to 42. This time we'll Edit directly at 0x5002 instead of 0x5000 and type '42':

```
*E 5002
5000  7A F8 42 BF  2F 9F 3A 04  31 00 7B 30  01 00 00 00
```

Leave the E mode using enter. Run it again using G:

```
*G 5000
```

The Q LED blinks more slowly.

## Burning data to the ROM

To burn a ROM, you need a blank (erased) 2716 EPROM. Insert a blank 2716 into the ROM socket on the front panel. Verify that it is erased using the “RE” (Rom Empty) command:

```
*RE
Empty
```

(It will say, “Not Empty” if any byte is not FF.)

Copy (“Move”) the 16 bytes of data (5000-500F) from the RAM to the ROM:

```
*M 5000,500F,4000
```

Verify the copy by listing the ROM:

```
*L 4000

4000  7A F8 42 BF  2F 9F 3A 04  31 00 7B 30  01 00 00 00
4010  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
4020  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
4030  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
4040  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
4050  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
4060  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
4070  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
4080  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
4090  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
40A0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
40B0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
40C0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
40D0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
40E0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
40F0  FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF *
```



## Uploading hex text data

If you already have a document with the hex machine codes, and you don't want to manually type them in, you can cut and paste them to the system using the **Upload** command. The hex codes should be pairs of digits with whitespace (spaces, tabs, newlines) between them. The hex data should be preceded by at least one address in the form of the '@' symbol and a 4-digit hex address. You may want to put the data in a text editor first and tweak it to have the @0000 and a "Z" at the end:

```
@0000  
  
7A  
F8 42 BF  
2F 9F 3A 04  
31 00  
7B  
30 01  
  
Z
```

The last character should be a "Z" – this tells the Upload command to stop uploading. The Z is not required, but without it you will need to stop the upload manually by typing a Z at the keyboard.

From your document, select the text to be uploaded and copy it into the clipboard. Switch over to the terminal emulator. Start the upload to 0x5000 with:

```
*U 0000
```

(Note that the upload location '0000' loads to 0x5000.) Now the system is waiting for hex data. Paste the clipboard to the terminal (in HyperTerminal select "Edit/Paste to Host"). If all goes well, you will see the prompt reappear:

```
*
```

You can verify that the data uploaded using the List or Edit commands.

## Using the Macro Assembler

The Assembler (source code at the end of this document) lets you create programs using mnemonic commands instead of hex codes, and it keeps track of addresses for you, so you can branch to a named location, instead of a number. Also, this assembler allows you to define macros, which lets you create a sort of language that makes correct code easier to write and more readable.

*Note: the assembler uses a slightly modified set of 1802 mnemonic codes. A few of the original codes could be mis-interpreted as hex constants, like "B3" (renamed B\_EF3\_L) and "BDF" (renamed B\_DF). See the listing tables.h at the end of this document for the modified mnemonics.*

Using a text editor, enter the following text:

```
@0000

$start:
SEQ

$restart:
LDI 42
PHI_F
$loop:
DEC_F
GHI_F
BNZ $loop.0
BQ $start.0
REQ
BR $restart.0
```

Save it in a file with the extension .asm, like "blink.asm". Run the assembler:

```
> asm blink.asm
```

This produces a file called blink.cos:

```
@0000

$start: {0000}
7B
$restart: {0001}
F8 42 BF
$loop: {0004}
2F 9F 3A 04 31 00 7A 30 01

Z
```

You can upload this text using the Upload cut and paste technique described above. The Uploader ignores the \$LABELS: and the {COMMENTS} and white space.

## Macros

You can define macros to make inserting common patterns of code easier, or just to make the code easier to read. You can create your own kind of language using macros. Macros are code templates. Macros perform text substitution according to a defined template. Macros are not subroutines.

Define macros in your assembly code using the keyword 'macro'. Each macro is terminated with a forward slash '/':

```
macro loop_begin( count, name, register )
    LDI #1d<count> PLO<register> $<name>: /

macro loop_end( name, register )
    DEC_<register> GLO_<register> BNZ $<name>.L /

macro goto BR /
```

Now you can write a program like this:

```
{ Define some macros: }
macro loop_begin( count, name, register )
    LDI #1d<count> PLO_<register> $<name>: /

macro loop_end( name, register )
    DEC_<register> GLO_<register> BNZ $<name>.L /

macro goto BR /

{ Code starts here: }
@0000

$start:
SEQ

loop_begin( 42, loop1, F )
    { this is an empty loop just for delay }
loop_end( loop1, F )

REQ

loop_begin( 42, loop2, F )
    { this is an empty loop just for delay }
loop_end( loop2, F )

goto $start.0
```

Which assembles to this, which can be uploaded using the Upload command:

```
@0000
```

```
$start: {0000}
```

```
7B F8 2A AF
```

```
$loop1: {0004}
```

```
2F 8F 3A 04 7A F8 2A AF
```

```
$loop2: {000C}
```

```
2F 8F 3A 0C 30 00
```

```
Z
```

## Using the system subroutines

There are a number of useful subroutines in the system ROM that you can use. These will help you make use of the RS-232 serial terminal interface. Here are the entry points for them:

@0701	\$_STANDARD_CALL:
@0716	\$_STANDARD_RETURN:
@0750	\$Print_string:
@0744	\$Print_char:
@076C	\$Get_char:
@0776	\$Print_hex:
@07AD	\$Ascii2hex:
@06CB	\$Find_space:
@06E0	\$Address_to_R9:
@003E	\$RESTART:

Call them using this simple “call” macro:

```
macro call(subroutine_name) SEP_4 $<subroutine_name> /
```

Here are short descriptions of these calls:

**Print\_string:** Send a 0x00-terminated ASCII sequence to the serial output. The ASCII sequence follows the call. In this example, the Carriage Return, and Line Feed (0D 0A) are appended explicitly. The 00 terminates the string (without it, Print\_string will continue to output whatever codes it finds until it encounters a 00, and then continue execution with whatever follows the 00.)

```
call( Print_string ) “Hello, World!” 0D 0A 00
```

**Get\_char:** Wait for a character to be typed on the serial interface. The character is put in mR7 (memory location pointed to by Register 7). This is a blocking call: it won’t return until the character appears.

**Print\_hex:** Outputs to the serial port two ASCII characters representing the byte at mR7.

**Ascii2hex:** Converts a pair of ASCII codes (at mR7 and mR7+1) to a single data byte at mR7.

**Print\_char:** Outputs one ASCII character at mR7 to the serial interface.

**Find\_space:** Move the serial input buffer pointer to the character just after the last space.

**Address\_to\_R9:** Convert four characters in the serial input buffer to two bytes in register 9.

# Full Assembler Documentation

File: asm.c

Usage: asm sourcefile[.asm] [outfile[.cos]] [symfile[.sym]]

Author: Norm Hurst

Date: January 21, 1993

This is a simple macro assembler for the RCA COSMAC 1802 microprocessor. The syntax is as follows:

Tokens are delimited by whitespace: any number of spaces, newlines, or tabs.

Tokens are case-sensitive.

## Hexadecimal Constants

Hex constants (or machine instructions) may be inserted directly; no directive is used: F8 08 B8 ... Multi-byte constants may be inserted without delimiters: 08B9C7, but there must be an even number of digits (leading zeros must be supplied).

## Decimal constants

Decimal or hex constants may be specified as

#[N]Mnnnn[.B]

where N is the number of bytes to use to represent the number (LSByte first), M is the mode, either D or d for decimal, or H or h for hex. "nnnn" represents the constant itself, and B (preceded by a dot . and in the range 0-7) is the particular byte of the converted constant to use. B and N are optional, but if B is not supplied N must be supplied so that the number of bytes to insert is not ambiguous. If B is supplied, N may be supplied or not. Note that if B is supplied, the number of bytes is 1, regardless of N.

Examples:	#2D-2	==>	FE FF (-2 as short int)
	#D-2.1	==>	FE (-2 as one byte)
	#d543210.2	==>	08 (byte 2 of 543210)
	#4d728092570	==>	9A CF 65 2B (LSByte is first)

## Mnemonics

A modified mnemonic table is used. (It has been modified so that none of the mnemonics looks like a hex constant.) The table is defined in tables.h.

## Comments

Comments may appear enclosed within {braces}. They may extend over multiple lines. There is currently a restriction that the opening brace must be preceded by whitespace, e.g., this is not legal: LDI{BNZ} because there is no whitespace before the {.

Absolute address is established at any time using the @ sign followed by a hex address:

@0700

You may optionally follow this with a colon:

@0700:

### Address Symbols

Address symbols begin with a dollar-sign: \$start. If the symbol is followed by a colon, it defines the symbol. Symbols have a scope of the entire sourcefile. The following code fragment demonstrates symbol syntax:

```
$SETUP:      LDI 02 PHI_5 LDI 54 PLO_5
$LOOP:      GHI_5 BNZ $LOOP.L GLO_5 BNZ $LOOP.L
```

The .L side-effect indicates that only the low-byte of the address of \$LOOP should be used. The .H side-effect means only the high-byte of the address should be used. Here is a fragment that sets register 2 to point at a location called \$stack:

```
LDI $stack.H PHI_2
LDI $stack.L PLO_2
```

Address symbols may include a decimal offset. Here register 6 is set to point at a location 10 bytes beyond \$RAM:

```
LDI $RAM+10.H PHI_6
LDI $RAM.L+10 PLO_6
```

Note that either the dot-operator or the offset operator may appear first. Negative offsets are also allowed.

Note that .1 may be used for .H and .0 may be used for .L, which makes numerical constants and addresses have parallel syntax, allowing a macro to be written such as Set\_reg(reg, val), and value could be passed as either a constant like #2D723 or and \$address label.

### Strings

Strings surrounded by "double-quotes" will be inserted as inline ascii codes. The string "Hello!" will generate 48 65 6C 6C 6F 21. Note that there is no null-termination, nor is there an escape mechanism like \n for inserting non-printing characters; these must be inserted manually as hex codes. Strings may contain spaces.

### Macros

Macros may be defined as follows:

```
macro macro_name([arg1,arg2,arg3,...]) macro_text...  /
```

The definition begins with the keyword 'macro' and ends with a forward slash. The macro can extend over multiple lines.

The keyword 'macro' indicates that a macro definition follows, which is the macro's name with an optional dummy argument list in parentheses. After this prototype definition comes the text of the macro, which may contain mnemonic code, hex constants, strings, embedded dummy arguments and/or macro calls. Dummy arguments must be enclosed in angle brackets.

For example, here is a macro to set a register to the value contained in an address symbol:

```
macro Set_reg(reg_num,addr_symbol)
LDI <addr_symbol>.H PHI_<reg_num>
LDI <addr_symbol>.L PLO_<reg_num>
/
```

It could be invoked like this:

```
Set_reg(6,$RAM+10)
```

Macros must be defined before they are used. Macros may contain address symbol definitions like \$LOOP:, but this will produce a multiply-defined label if the macro is invoked more than once. However, you can pass a unique name for an address as an argument. Here are examples of a pair of macros that implement loops. The 'a' argument is the address label that allows the macro to be used uniquely more than once:

```
{ n is loop count, a is an address label, r is a register number }
macro loop16(n,a,r) LDI #2d<n>.1 PHI_<r> LDI #2d<n>.0 PLO_<r> $<a>: /
macro end_loop16(a,r) DEC_<r> GHI_<r> BNZ $<a>.0 GLO_<r> BNZ $<a>.0 /

macro goto BR /
```

You can use these like this:

```
@0000:

$START:
SEQ

loop16(20000,blink,A) {'blink' is unique to this loop}
    { empty delay loop: 20,000 loops }
end_loop16(blink,A)
```



REQ

```
loop16(5000,blink2,A) {'blink2' is unique to this loop}
    { empty delay loop, 5,000 loops }
end_loop16(blink2,A)
```

```
goto $START.0
```

Results in this machine code:

@0000

```
$TOP: {0000}
      7B F8 4E BA F8 20 AA
$blink: {0007}
      2A 9A 3A 07 8A 3A 07 7A F8 13 BA F8 88 AA
$blink2: {0015}
      2A 9A 3A 15 8A 3A 15 30 00
Z
```

Macro calls may be nested – macros can call macros:

```
macro call(addr) SEP_4 $<addr> /
macro print(string) call(prn_str) <string> 0D 0A 00 / {Note CRLF NULL}
```

### **Include files**

Files may be included in the source code using the keyword include followed by the filename to include:

```
include my_macros.mac
```

Includes may be nested up to five levels.

### **Input file**

The assembler is invoked from the command line:

```
$ asm my_file.asm
```

You can leave off the .asm extension from my\_file; the assembler will look for my\_file.asm.

### **Output files**

The assembler will create a file called my\_file.cos that contains the assembled machine code. It will also generate my\_file.sym, the symbol table. It may have been compiled with TRACE\_ON defined; if so, it will output asmtrace.txt.

You can force the assembler to output to specific file names:

```
$ asm my_file.asm assembledHexCode.dat symbol_table.txt
```

Here is an example assembly code output file:

```
@00FA

D4 01 0D 48 65 6C 6C 6F 21
0D 0A 00
48 65 6C 6C 6F 21
D5
F8 7B 7A 30 0C

@0800

00
00
00
00

@8FF

00

Z
```

The format @XXXX indicates the starting address for the following data. Data consist of pairs of hex digits. The last character in the file is a Z (followed by a newline). Characters other than hex digits, the @, and the Z should be ignored by a downloading program.

### Symbol Table

The assembler also outputs a list of address symbols and their values. The form is such that it could be pasted into another file and thus used as a crude form of linking. Here is an example symbol table:

```
@00FA $test:
@010D $print:
@0800 $RAM:
@08FF $Stack:
```

### More on macros

Macros may be used simply to make code more readable. Here is a set of macros to make subroutines easier to deal with:

```
macro subroutine(name) SEP_5 $<name>: /  
macro exit(name) BR $<name>.L-1 /  
macro call(name) SEP_4 $<name> /
```

Then subroutines look like this:

```
{  
send1 - send a single byte over the UART  
Sends the byte found at mR6.  
}  
subroutine(send1)  
    B_EF1_H $send1.0    {wait for UART buf to be empty}  
    SEX_6 OUT_6 DEC_6    {send what you find in mR6}  
    SEX_2  
exit(send1)
```

And can be invoked like this:

```
call(send1) { send the byte in mR6 to the UART }
```

## Assmblar Source Code

The assembler is written in C and consists of two files: asm.c and tables.h. To build it, put the two files in the same directory and compile, e.g.

```
$ gcc asm.c -o asm.exe
```

### asm.c

```
/*
How the code works:

This is a very simple 2-pass assembler. The program makes 2 scans, or "passes"
over the sourcefile. The first pass is to find symbolic address definitions
(e.g. $LOOP:) to build the symbol table (symtab), and macro definitions to
build the macro table (mactab). There is a third, fixed table that contains
the mnemonics (mntab). These tables are declared in tables.h

There are several global variables. The three tables mentioned above, the
lengths of the two variable tables (symtab_len and mactab_len), the current
location counter current_loc, the sourcefile line number line_num (used for
error reporting), and the pass number pass_num (either 1 or 2).

The code is somewhat modular. Here is a summary:

main(argc, argv)
    Invoked by the command line.

fgettok(fp, delims, token, term)
    Gets the next token from a file. Has special modes for "strings" and
    {comments}.

sgettok(string, delims, token, term, pos, skip_leading_delims)
    Like fgettok, but gets the next token from a string.

Parse_a_token
    Processes one token for either pass. This is the
    main parsing routine.

parse_addr(token, &addr_byte_hi, &addr_byte_lo)
    Parses tokens that start with $

Save_macro(infile)
    Parses one macro definition. Called when the token macro is found on
    Pass1.

Expand_macro(token, maci, infile, outfile)
    Expands macros when they are encountered on pass 1 or 2.
    Calls Pass1 or Pass2 for each expanded token. If the token
    is another macro call, the Parse_a_token routine will call Expand_macro
    again. This allows nested macros.

Read_until(fp, term_char, string, max_chars)
    Reads characters from the file fp into string until
    term_char is encountered or max_chars are read. Used
    in Pass2 for skipping over macro definitions.

error(error_code, line_num, bad_token)
    Prints the line number, a message, and the offending token
    in the event of an error.
```

```

*/
/**/
#define MAX_SYMBOL_LENGTH 22
#define MAX_MACRO_ARGS 8
#define MAX_MACRO_TEXT 512
#define MAX_MACRO_CALL 100
#define MAX_TOKEN_LENGTH 128
#define MAX_INCLUDE_NESTING 5

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tables.h"

FILE *trace_fp; /* file pointer for trace output file when TRACE > 0 */

enum error_names
{
    MULDEFLAB=1,
    UNDEFLAB,
    UNKSYMBOL,
    HEXODDDIG,
    INVPASNUM,
    INVDOTARG,
    NOCLOSPAREN,
    MACNOTTERM,
    NOCLOSANGLE,
    UNKDUMARG,
    TOOMANYARGS,
    WRONGNUMARGS,
    MACTOOBIG,
    TOOLONG,
    MACALLOCFAIL,
    INCLNEST2DEEP,
    CUDNOTOPENINCL,
    BADBYTESEL,
    BADCONSTMODE,
    NOSEMIORLEN
};

int error_cnt;
char *err[] =
{
    "*** ILLEGAL ERROR CODE = 0 ***",
    "Multiply-defined label",
    "Undefined label",
    "Unknown symbol",
    "Hex constant has odd-number of digits",
    "Invalid pass number",
    "Invalid dot argument in label",
    "Missing ) in macro prototype",
    "Macro not terminated with a /",
    "No close angle (>) in macro definition",
    "Unknown dummy argument in macro definition",
    "Too many arguments supplied to macro",
    "Wrong number of arguments supplied to macro",
    "Body of macro too long",
    "Exceeded character limit during Read_until",
    "Memory allocation for macro failed",
    "Include files nested too deeply",
    "Could not open include file",
    "Byte selector in constant must be 0-7",

```

```

"Constant type must be H or D",
"Must specify number of bytes or which byte in constant"
};

```

```

char *valid_hex = "0123456789ABCDEF";
// char *main_delims = " \n\t";
char *main_delims = " \n\t\r";

```

```

// #define and &&
// #define or ||
// #define xor ^
// #define not !
#define is_even(a) !(a%2)
#define is_odd(a)  a%2
#define LOOP(i,n) for(i=0; i<n; i++)
#define MIN(a,b)  ((a)<(b)) ? (a) : (b)
#define MAX(a,b)  ((a)>(b)) ? (a) : (b)

#define last_char(a) *(a+strlen(a)-1)
// #define CHAR_UPCASE(c) c = ( (c) >= 'a' && (c) <= 'z' ) ? (c) -= ('a' - 'A') : (c)
#define CHAR_UPCASE(c) ( (c) >= 'a' && (c) <= 'z' ) ? (c) -= ('a' - 'A') : (c)
#define is_all_hex(a) strspn(a,valid_hex)==strlen(a)
#define LOW_BYTE(a)  ((a) & 0xFF)
#define HIGH_BYTE(a) (((a) >> 8) & 0xFF)

```

```

char
charUppcase( char *c )
{
    // is it lower case?
    if ( *c >= 'a' && *c <= 'z' )
        return *c - ( 'a' - 'A' );
    else
        return *c;
}

```

```

/*
_____*/
void
error(error_code,line_num,bad_token)
int line_num, error_code;
char *bad_token;
{
    error_cnt++;
    fprintf(stderr, "Line %d: %s: [%s]\n", line_num, err[error_code], bad_token);
    if (TRACE > 0) fprintf(trace_fp, "Line %d: %s: [%s]\n", line_num, err[error_code], bad_token);
}

```

```

/*

```

---

```

parse_addr

```

token is the pointer to the string to be parsed.  
This function assumes that the \$ prefix is still  
the first character of the token.

sym\_tab is a pointer to an array of structures;  
it is the current symbol table.

symtab\_len is the number of defined symbols  
in sym\_tab.

loc is the value of the address counter  
maintained by the assembler.

pass is either 1 or 2, and tells this function which pass the assembler is on so that it does not do unnecessary parsing, such as calculating the first and second byte values, `addr_byte_hi` and `addr_byte_lo`, of the address.

`addr_byte_hi` and `addr_byte_lo` are the first and second byte values that are parsed on the second pass and returned to the assembler.

The token is either of the form

`$name[+|-n][.H|L]`

or the form

`$name:`

The first character must be a \$, followed by the symbol name. If the next (and last) character is a :, this indicates a definition of a symbol, which causes a new entry in the symbol table on the first pass, and no action on the second pass.

If the last character is not a :, then the token reflects the reference of a symbol within the code. There may optionally follow an offset of + or - and a decimal integer, but without any spaces.

There may also be a byte select operator (the .) followed by an H or an L (must be upper case), meaning to return only the high or low byte of the address.

For example

`$start:`

defines the label "start".

`$start-1.H`

is the high byte of the location of start-1.

\*/

/\*\*/

int

`parse_addr( char *token, int *addr_byte_hi, int *addr_byte_lo )`

{

```
    int    dot = 0;           /* position of the . in the token */
    int     plus = 0;          /* position of the + in the token */
    int     minus = 0;         /* position of the - in the token */
    int     toklen = 0;         /* length of the token */
    int     offset = 0;         /* address offset caused by + or - */
    int     address = 0;        /* absolute address of symbol */
    int     first_side_effect = 0; /* position of first of (+ - .) */
    int     sym_num = 0;        /* position of token in symbol table */
```

```
    int     i;                 /* a loop counter */
```

```
    char    which_byte = '\0'; /* either 'H' or 'L' or '\0' */
```

```

// the symbol name stripped of $ and other side effects (+, -, .)
char sym_name[MAX_SYMBOL_LENGTH];

// IS IT A LABEL DEF? I.E. LAST CHAR IS A :?
if ( last_char(token) == ':' ) { /*it's a label def --can't have side effects */

switch ( pass_num )
{
    /* label def's in pass 2 are skipped */
    case 1: /* pass 1: check table to be sure it's not a duplicate, then save it with its address
*/
        last_char( token ) = '\0'; /* lose the ':' */

        // scan table...
        for ( i = 0; i < symtab_len && strcmp(symtab[i].name,token+1) != 0; i++)
            ;
        if ( i < symtab_len )
        {
            /* error: already exists! */
            return MULDEFLAB;
        }
        else
        {
            /* new symbol... */
            symtab[symtab_len].loc = current_loc;
            strcpy(symtab[symtab_len].name, token+1);
            symtab_len++;
        }
        return 0;

        case 2: *addr_byte_hi =*addr_byte_lo =0x100; return 0; break;

        default: return INVPASNUM; /* INVALID PASS NUMBER! */

    } /* END OF switch (pass_num) */
} /* END OF if (last_char(token) is ':') */

/*
IT'S NOT A LABEL DEF, IT'S LABEL USE...
*/

/*
FIND SIDE EFFECTS (+, -, .)
*/
dot = (int) strcspn(token,".");
plus = (int) strcspn(token,"+");
minus = (int) strcspn(token,"-");
toklen = (int) strlen(token);

first_side_effect = MIN(MIN(dot,plus),minus);

// INCREMENT THE LOCATION COUNTER APPROPRIATELY
if (dot != toklen)
    current_loc++; /* A DOT MEANS ONLY ONE BYTE */
else
    current_loc += 2; /* NO DOT MEANS BOTH BYTES */

// EXTRACT THE SYMBOL'S NAME BY ITSELF
strcpy(sym_name, token+1); /* COPY WITHOUT THE $ */
if (first_side_effect != toklen) /* TRIM OFF SIDE EFFECTS */
    *( sym_name + first_side_effect - 1 ) = '\0';

```



```

if ( dot != toklen)
{
    switch (*(token+dot+1))
    {
        case 'H': case '1': which_byte = 'H'; break;
        case 'L': case '0': which_byte = 'L'; break;
        default: return INVDOTARG;
    }
}

switch (pass_num)
{
    case 1:          /* PASS 1 -that's all!*/
        return 0;
        break;

    case 2:          /* PASS 2 -wait! There's more!*/

// FIND SYMBOL ENTRY IN TABLE
        for ( sym_num = 0; sym_num < symtab_len
            && strcmp(symtab[sym_num].name,sym_name) != 0; sym_num++ ) ; /* scanning... */

        if (sym_num == symtab_len)
            return UNDEFLAB; /* UNDEFINED LABEL! */

// CALCULATE ADDRESS WITH POSSIBLE OFFSET
        address = symtab[sym_num].loc;

        if ( plus != toklen || minus != toklen )
        {
            strcpy( sym_name,token+(MIN(plus,minus)) );
            *(sym_name+strlen(sym_name)) = '\0';
            offset = atoi( sym_name );
            address += offset;
        }

// FIGURE BYTE VALUES TO RETURN
// SETS BIT 8 IF BYTE SHOULD NOT BE USED.

        *addr_byte_hi = HIGH_BYTE(address);
        *addr_byte_lo = LOW_BYTE(address);
        if ( which_byte == 'H' )
            *addr_byte_lo |= 0x100;
        else if ( which_byte == 'L' )
            *addr_byte_hi |= 0x100;

        return 0;
        break;

    default: return INVNUM;

} /* END OF switch (pass_num) */

} /* END OF parse_addr */

/*
_____*/
int
Read_until( FILE *fp, char term_char, char *string, int max_chars )
{
    int i = 0;

```

```

char c;

i = 0;
fread( &c, 1, 1, fp );
while ( !feof(fp) && i < max_chars-1 && c != term_char )
{
    string[i] = c;
    i++;
    if ( c == '\n' )
        line_num++;

    fread( &c, 1, 1, fp );
}

if ( c == term_char || i == max_chars-1 )
    string[i] = '\0'; /* terminate the string */

if ( feof(fp) )
    return feof(fp);

if ( i == max_chars-1 )
    return TOOLONG;
else
    return 0;
} /* end of Read_until */

```

/\*

---

Parses constants like

#2d-27

which means -27 described as 2 bytes (FFE5)

#d-27;1

which means the the second byte (FF)

General form is: #[N]Mnnnn[;B]

N - single decimal digit - number of bytes

M - mode: X, D, x, or d - x is hex, D is decimal. Must be given.

nnnn... - the value string

B - single decimal digit 0-7 selecting one byte of converted number

NOTE: If ;B is not given, N must be given. If N is given, .B may be specified as well.

Returns the converted value, the number of bytes to insert in the code,  
and which\_byte to insert as a single byte.

\*/

int

parse\_const( char \*token, int \*value, int \*num\_bytes, int \*which\_byte)

{

int dot = 0; /\* position of the ; in the token \*/

int toklen = 0; /\* length of the token \*/

char mode = '\0'; /\* either H or D \*/

// START

token++; /\* lose the leading #-sign \*/

\*which\_byte = 0; /\* if no dot, this will remain zero \*/

// FIND dot

dot = (int) strchr(token, "."); /\* semi is OK, too \*/

toklen = strlen(token);

/\*

FORM IS: #[N]Mnnnn[.B]

N - single decimal digit - number of bytes

M - mode: H, D, h, or d - H is hex, D is decimal. Must be given.

nnnnn... - the value string

B - single decimal digit 0-7 selecting one byte of converted number

NOTE: If .B is not given, N must be given. If N is given, .B may be specified as well.

```
*/
if ( *token >= '0' && *token <= '9' )
{ // N is given...
    *num_bytes = *token - '0';
    *( token + 1 ) = charUppcase( token + 1 );
    if ( *( token + 1 ) == 'H' || *( token + 1 ) == 'D' )
    {
        mode = *( token + 1 );
        if ( dot != toklen )
        {
            *num_bytes = 1;
            *which_byte = *( token + dot + 1 ) - '0';
            if ( *which_byte > 7 || *which_byte < 0 )
                return BADBYTESEL; /* out of range 0-7 */

            *( token + dot ) = '\0'; /* trim off the dot */
        }

        token += 2; /* point at 1st char of value string */
    }
    else
    {
        return BADCONSTMODE;
    }
}
else
{
    // not a digit - it had better be a D or H, and there must be a dot.
    *num_bytes = 1; /* remember, there must be a dot! */
    *token = charUppcase( token );
    if ( *token == 'D' || *token == 'H' )
    {
        if ( dot == toklen )
            return NOSEMIORLEN; /* ambiguous number of bytes */

        mode = *token;
        *which_byte = *( token + dot + 1 ) - '0';

        if ( *which_byte > 7 || *which_byte < 0 )
            return BADBYTESEL; /* out of range 0-7 */

        *( token + dot ) = '\0'; /* trim off the dot */
        token++; /* point at 1st char of value string */
    }
    else
    {
        return BADCONSTMODE; /* was not D or H */
    }
}

// CALCULATE THE VALUE...
if ( mode == 'D' )
    sscanf( token, "%d", value );
else
    sscanf( token, "%x", value );
```

```

        // INCREMENT THE LOCATION COUNTER
        current_loc += *num_bytes;

        return 0;

} /* END OF parse_const */


/*
_____*/
int
fgettok( FILE *fp, char *delims, char *token, char *term, int *line_num )
/* line_num gets incremented for every \n that is found */
{
    int ndelims;
    int i,count;
    char c;

    ndelims = strlen( delims );
    if ( feof(fp) )
        return EOF; /* Jump away immediately if already at EOF */

    /* CHEW UP LEADING DELIMITERS. { AND " ARE NOT DELIMITERS HERE. */
    fread( &c, 1, 1, fp );
    while( !feof(fp) && !ferror(fp) )
    {
        if ( c == '\n' )
            line_num++;

        for( i = 0; ( i < ndelims ) && ( c != *(delims+i) ); i++ )
            ;

        if ( i == ndelims )
            break;

        fread( &c, 1, 1, fp );
    }

    if (feof(fp)) return EOF;
    if (ferror(fp)) return ferror(fp);

    switch (c)
    { /* could be a { or a " or else it's just a regular token */
        case '{': /* it's a comment: scan until you find '}'. */
            *term = '{'; /* this means "it was a comment!" */

            // another way to say it was just a comment.
            *token = '{';
            *(token+1) = '\0';

            fread( &c, 1, 1, fp );
            while(!feof(fp) && !ferror(fp) && c!='}')
            {
                if ( c=='\n' )
                    line_num++;

                fread( &c, 1,1, fp );
            }
    }

```

```

        if (c==}')')
            return 0;

        if (feof(fp))
            return EOF;

        if (ferror(fp))
            return ferror(fp);

case '": /* it's a quoted string: scan until you another '"'. */
    *token = '"'; /* return 1st char as " */
    fread( &c, 1, 1, fp );
    count = 1;
    while(!feof(fp) && !ferror(fp) && c!="'")
    {
        if ( c == '\n' )
            line_num++;

        *( token + count ) = c;
        count++;

        fread( &c, 1, 1, fp );
    }

    if ( c == '"' )
    {
        *(token+count) = '"'; /* return final quote */
        *(token+count+1) = '\0';

        // set term to "; this means "it was a string!"
        *term = '"';
        return 0;
    }

    if (feof(fp)) return EOF;
    if (ferror(fp)) return ferror(fp);

default: /* it's a normal token */

    count = 0;
    *(token + count) = c; /* 1st char has already been found... */
    count++;

    fread( &c, 1, 1, fp );
    while( !feof(fp) && !ferror(fp) )
    {
        if ( c == '\n' )
            line_num++;

        for( i = 0; ( i < ndelims ) && ( c != *( delims + i ) ); i++ )
            ;

        if ( i == ndelims )
        {
            *(token+count) = c;
            count++;
        }
        else
        {
            *(token+count) = '\0';
            *term = c;
            return 0;
        }
    }

```

```

        fread( &c, 1, 1, fp );
    }

    if (feof(fp)) return 0; /* must return 0 (not EOF) so last token can be used */
    if (ferror(fp)) return ferror(fp);

} /* end of switch(c) */

printf( "fgettok: WARNING! Should not get here!\n" );
return 0;

} /*end of fgettok */

/* _____ */
int
sgettok(string, delims, token, term, pos, skip_leading_delims)

char *string;
char *delims;
char *token;
char *term;
int *pos;
int skip_leading_delims;

{
int ndelims;
int i, count;
char c;

ndelims = strlen(delims);

/*
CHEW UP LEADING DELIMITERS.
{ AND " ARE NOT DELIMITERS HERE.
*/

c = string[*pos];

if (skip_leading_delims) {
    while(c != '\0') {
        for(i=0; (i<ndelims) && (c!=*(delims+i)); i++)
            ;

        if (i==ndelims) break;
        c = string[++(*pos)];
    }
}

if (c=='\0') return EOF;

switch (c) { /* could be a { or a " or else it's just a regular token */

case '{': /* it's a comment: scan until you find '}'. */
    *term = '{'; /* this means "it was a comment!" */
    *token = '{'; *(token+1) = '\0'; /* another way to say it was just a comment. */
    c = string[++(*pos)]; /* fread(&c, 1,1, fp); */
    while(c != '\0' && c!='}') {
        c = string[++(*pos)]; /* fread(&c, 1,1, fp); */
    }
    if (c==}') {
        ++(*pos);

```

```

        return 0;
    }
    if (c=='\0')    return EOF;

case '': /* it's a quoted string: scan until you another ''. */
    *token = ''; /* return 1st char as " */
    c = string[++(*pos)];
    count=1;
    while(c != '\0' && c!='') {
        *(token+count) = c;
        count++;
        c = string[++(*pos)];
    }
    if (c=='') {
        *(token+count) = '';
        *(token+count+1) = '\0';
        *term = ''; /* set term to "; this means "it was a string!" */
        ++(*pos);
        return 0;
    }
    if (c=='\0')    return EOF;

default: /* it's a normal token */
    count=0;
    if (skip_leading_delims) {
        *(token + count) = c; /* 1st char has already been found... */
        count++;
        c = string[++(*pos)];
    }

    while(c != '\0') {
        for(i=0; (i<ndelims) && (c!=(delims+i)); i++) ; /* scan... */

        if (i==ndelims) { /* not a delimiter... */
            *(token+count) = c;
            count++;
        } else { /* delimiter... */
            *(token+count) = '\0';
            *term = c;
            ++(*pos);
            return 0;
        }

        c = string[++(*pos)];
    }

    if (c=='\0') {
        *(token+count) = '\0'; /* terminate the token string when found at end */
        ++(*pos);
        return 0;
    }

} /* end of switch(c) */

    printf( "sgettok: WARNING! Should not get here!\n" );
    return 0;

} /*end of sgettok */

```

```

/*
_____*_/
void
strupcase( char *s )
{
    int i=0;
    for (i=0; s[i] != '\0'; i++)
        *(s+i) = charUppcase( s+i );
    // s[i] = ( s[i] >= 'a' && s[i] <= 'z' ) ? s[i] -= ('a' - 'A') : s[i];
}

/*
_____*_/
Save_macro - Called for macro definitions in pass 1.
             Remembers the name, parses the argument list
             reads macro's text (up to the terminating /)
             into a buffer, then allocates storage for the text
             and points the structure's pointer at it.

*/
int
Save_macro( FILE *infile )
{
    int error_code = '\0';
    char string[MAX_MACRO_TEXT], other_half[MAX_MACRO_CALL];
    char *token; /* token is a pointer to hold the return of strtok */
    char term_char='\0';
    int i,text_len;

    /*
    GET THE NEXT TOKEN, WHICH MUST BE THE MACRO
    NAME AND PARAMETER LIST. SEARCH FOR THE
    TERMINATING ")"
    */

    // error_code = fgettok( infile, ")" ", string, &term_char, &line_num );
    error_code = fgettok( infile, ")" \r", string, &term_char, &line_num );
    if (strchr(string, '(') && strchr(string,')') == '\0' && term_char == ' ')
    {
        strcat(string, " "); /* add in the space */
        Read_until( infile, ')', other_half, MAX_MACRO_CALL );
        strcat( string, other_half );
        strcat( string, ")" );
    }

    /*
    RETURNED STRING IS OF THE FORM
        name(arg1,arg2,arg3
    OR
        name
    WITH NO FINAL ). ONLY COMMAS
    ARE VALID ARGUMENT DELIMITERS.
    */

    i = strcspn( string, "(" ); /* Find the length of the name */
    strncpy( mactab[mactab_len].name, string, i );
    *( mactab[mactab_len].name + i ) = '\0'; /* make sure it's terminated */
    if ( TRACE > 2 )
        fprintf(trace_fp, "    >>>Save_macro: name %s/\n",mactab[mactab_len].name);

    /*
    WALK THRU ARG LIST, SAVING ARG NAMES.
    */

```



```

        if (i != strlen(string)) { /* there is an argument list */

            token = strtok( string + i + 1, ", )"); /* point at 1st char beyond ( */
            for ( i = 0; token != '\0' && i < MAX_MACRO_ARGS; i++)
            {
                strcpy(mactab[mactab_len].argv[i], token);
                if ( TRACE > 2 )
                    fprintf( trace_fp, "Save_macro: ...arg%d /%s/\n", i,
mactab[mactab_len].argv[i] );

                token = strtok('\0', ", )");
            }
            mactab[mactab_len].argc = i;
        } else {

        }

        /*
        NOW GET THE BODY OF THE MACRO.
        IT IS TERMINATED BY A /.
        */
        error_code = Read_until(infile, '/', string, MAX_MACRO_TEXT);
        if (TRACE > 2) fprintf(trace_fp, "Save_macro: name /%s/ body: /%s/\n",mactab[mactab_len].name,
string);
        if (error_code != '\0') return MACTOOBIG;

        /*
        MEASURE TEXT LENGTH
        AND ALLOCATE STORAGE FOR IT
        */
        text_len = strlen(string);
        if (text_len > MAX_MACRO_TEXT) return MACTOOBIG;
        mactab[mactab_len].text = malloc(sizeof(char)*text_len +1); /* +1 is for the '\0' */
        if (mactab[mactab_len].text == '\0') {
            fprintf(stderr, "Macro allocation failure\n");
            fprintf(stderr, "Name: %s\nText: %s\n",mactab[mactab_len].name, string);
            return MACALLOCFAIL;
        } else {
            strcpy(mactab[mactab_len].text, string);
        }
        /*
        INCREMENT mactab_len TO POINT AT
        NEXT AVAILABLE SLOT.
        */
        mactab_len++;

        return 0;

    } /* END OF Save_macro */
    /*

```

---

```

Expand_macro - Called for macro uses in pass 1.
                Expands a single macro, replacing
                <dummy_arg> found in its text with
                values supplied in the argument list.
                Token points to the entire call; maci
                is the index of the macro that was called
                in the source code.

```

```

    /*

    /* maci is the table index of the selected called macro */
    int

```

```

Expand_macro(char *token, int    maci, FILE *infile, FILE *outfile )
{

    int error_code = '\0';
    char string[MAX_MACRO_TEXT]; /* = 0x200 */
    char term_char='\0';
    char arg_list[MAX_MACRO_ARGS][MAX_SYMBOL_LENGTH]; /* 17*8 = 136 = 0x88 */
    char subtok[MAX_TOKEN_LENGTH], subsubtok[MAX_TOKEN_LENGTH], expanded_tok[MAX_TOKEN_LENGTH];
    char term='\0';
    char *char_ptr;
    int i, pos1,pos2, sgettok_stat='\0'; //, num_args=0;
    int Parse_a_token();

    if ( TRACE > 1 )
        fprintf( trace_fp, "Expand Macro: [%s]\n", token );

    /*
    EXTRACT JUST THE ARGUMENT LIST INSIDE
    THE ()'s.
    */

    *string = '\0';
    if ( ( i = strcspn( token, "(" ) ) != strlen( token ) )
    { /* there are ()'s */

        strcpy( string, token + i + 1 ); /* copy argument list only */

        char_ptr = strchr( string, ')' ); /* this will be null if no ()'s */
        if ( char_ptr != '\0' )
            *char_ptr = '\0'; // remove closing param
    }

    /*
    RETURNED STRING IS OF THE FORM
        arg1,arg2,arg3
    ONLY COMMAS ARE VALID ARGUMENT DELIMITERS.
    STRING IS NULL IF THERE ARE NO ARGS OR NO ()'s.
    */

    /*
    WALK THRU COMMA-SEP ARG LIST, SAVING PARAM NAMES
    AND COUNTING PARAMS.
    */
    if ( mactab[maci].argc != 0 )
    {
        token = strtok(string, ",");
        for ( i=0; token != '\0' && i < MAX_MACRO_ARGS; i++ )
        {
            strcpy( arg_list[i], token );
            if ( TRACE > 3 )
                fprintf( trace_fp, "    param%d: [%s]", i, arg_list[i] );

            token = strtok( '\0', ", " );
        }

        if ( TRACE > 3 )
            fprintf( trace_fp, "\n" );

        if ( i == MAX_MACRO_ARGS )
            return TOOMANYARGS;

        if ( i != mactab[maci].argc )
            return WRONGNUMARGS;
    }
}

```

```

/*
PARSE THE_MACRO's TEXT FOR <args>
*/
strcpy( string, mactab[maci].text );
if ( TRACE > 3 )
{
    fprintf( trace_fp, "Macro body: %s\n", string );
    fflush( trace_fp );
}

pos1 = 0;
// (added \r to delim list so windows-style newlines with CR (\r) aren't an issue.)
while ( sgettok( string, " \r\t\n", subtok, &term, &pos1, 1 ) != EOF )
{ /* subtok form: xxx<ooo>xx<o>xxx... */
    if ( TRACE > 4 )
    {
        fprintf( trace_fp, " subtok to expand: %s/\n", subtok );
        fflush( trace_fp );
    }

    *expanded_tok = '\0'; /* start building expanded token*/
    pos2 = 0;
    sgettok_stat = sgettok( subtok, "<", subsubtok, &term, &pos2, 0 );
    while ( sgettok_stat == '\0' )
    {
        strcat(expanded_tok, subsubtok);

        if (TRACE > 4)
        {
            fprintf( trace_fp, " A: expanded_tok: %s subsubtok: %s term: %s
pos2: %d\n", expanded_tok, subsubtok, term, pos2);
            fflush( trace_fp );
        }

        if ( term == '<' )
        { /* find closing > and process dummy arg */

            /* This is a dummy arg. Which one? */
            sgettok_stat = sgettok( subtok, ">", subsubtok, &term, &pos2,
0 );

            if ( term != '>' )
                return NOCLOSANGLE;

            for ( i = 0; i < mactab[maci].argc
                && strcmp(subsubtok, mactab[maci].argv[i]) != 0;
i++ )

                ; // scan...

            if ( TRACE > 4 )
            {
                fprintf( trace_fp, " Dummy arg: [%s], param%d: [%s]\n",
subsubtok,i,arg_list[i] );
                fflush( trace_fp );
            }

            if ( i == mactab[maci].argc)
                return UNKDUMARG; /* Didn't find a match! */

            strcat( expanded_tok, arg_list[i] );

            if ( TRACE > 4 )
            {

```

```

                                                                    fprintf( trace_fp, " B: expanded_tok: %s subsubtok: %s
term: %c pos2: %d\n",
                                                                    expanded_tok, subsubtok, term, pos2);
                                                                    fflush( trace_fp );
                                                                    }
                                                                    }
                                                                    else if ( term == '' )
                                                                    {
                                                                    }
                                                                    else
                                                                    {
                                                                    break;
                                                                    }

                                                                    sgettok_stat = sgettok( subtok, "<", subsubtok, &term, &pos2, 0 );
                                                                    }

/*
EXPANDED TOKEN IS COMPLETE.
SEND IT TO Parse_a_token
*/
                                                                    if ( TRACE > 0 )
                                                                    {
                                                                    fprintf( trace_fp, " ---> <%s>...\n", expanded_tok );
                                                                    fflush( trace_fp );
                                                                    }

                                                                    error_code = Parse_a_token( expanded_tok, infile, outfile );

                                                                    if ( error_code != '\0' )
                                                                    error(error_code, line_num, expanded_tok);

                                                                    if ( pass_num == 2 && term_char == '\n' )
                                                                    fprintf( outfile, "\n" );

                                                                    } /* end of while(sgettok...) */

                                                                    return 0;

} /* END OF Expand_macro1 */

/*
-----
Parse_a_token - Performs all the parsing on a single token for either pass.
Calls Expand_macro(), which in turn calls Parse_a_token to
parse the tokens from that expansion. Thus, macros
may be recursively nested.
*/
int
Parse_a_token( char *token, FILE *infile, FILE *outfile )
{
    int      addr_byte_hi=0x100;
    int      addr_byte_lo=0x100;
    int      error_code='\0';
    int      i;
    int      value;          /* returned by parse_const */
    int      nbytes;         /* number of bytes to insert */
    int      byte_select;    /* which one byte to insert */

    static int prev_1st_digit = -1;
    char term_char;

```

```

char    string[MAX_MACRO_TEXT];
char    *char_ptr;

/*
CHECK FOR KEYWORDS macro include
*/
if ( strcmp( token,"macro" ) == 0 )
{
    switch ( pass_num )
    {
        case 1:
            if((error_code=Save_macro(infile)) != '\0')
                return error_code;

            break;

        case 2:
            // Move fp past macro definition
            Read_until( infile, '/', string, MAX_MACRO_TEXT );
            break;

        default: return INVPASNUM;
    }
}
else if ( strcmp( token, "include" ) == 0 )
{
    // (Added \r to support windows-style CRLF newlines.)
    fgettok( infile, " \n\t\r", token, &term_char, &line_num );
    if ( term_char == '"' )
    { /* DISCARD QUOTES IF PRESENT */
        token++;
        if ( last_char( token ) == '"' )
            last_char(token) = '\0';
    }
    include_level++;

    if ( include_level > MAX_INCLUDE_NESTING )
    {
        include_level--;
        return INCLNEST2DEEP;
    }

    if ( ( infile_stack[include_level] = fopen( token, "r" ) ) == '\0' )
    {
        include_level--;
        return CUDNOTOPENINCL;
    }
    return 0;
}
else
{
    /*
TEST 1st CHAR FOR @"$
*/
    switch ( *token )
    {
        case '@':
            if ( last_char(token) == ':' )
                last_char(token) = '\0'; /* colon is optional */

            sscanf( token + 1, "%x", &current_loc );

            if ( pass_num == 2 )
                fprintf(outfile, "%s\n", token);
    }
}

```

visually

```
break;

case '{':
if ( pass_num == 2 )
    fprintf(outfile, "\n\t\t"); // Print a CR to help break up code

    break;

case '"':
if ( last_char( token ) == '"' )
    last_char(token) = '\\0';

current_loc += strlen(token) -1; /* 1st char was the " */

if ( pass_num == 2 )
{
    LOOP( i, strlen( token ) - 1 )
        fprintf(outfile, "%02X ", (int) *( token + 1 + i ) );

        /* fprintf(outfile, "\n"); */
}
break;

case '#':
error_code = parse_const( token, &value, &nbytes, &byte_select);
if ( error_code != '\\0' )
    return error_code;

if ( pass_num == 2 )
{
    if ( nbytes == 1 )
    {
        value = 0x00FF & (value >> 8*byte_select);
        fprintf(outfile, "%02X ", value);
    }
    else
    {
        LOOP ( i, nbytes ) /* lowest byte is first !!! */
            fprintf(outfile, "%02X ", 0x00FF & (value >>
8*i));
    }
}

break;

case '$':
error_code = parse_addr( token, &addr_byte_hi, &addr_byte_lo );
if ( error_code != '\\0' )
    return error_code;

if ( pass_num == 2 )
{
    if ( addr_byte_hi < 0x100 )
        fprintf( outfile, "%02X ", addr_byte_hi );

    if ( addr_byte_lo < 0x100 )
        fprintf( outfile, "%02X ", addr_byte_lo );

    /* CHECK FOR CROSS-PAGE SHORT BRANCH */
    if ( addr_byte_hi > 0xFF
        && addr_byte_lo < 0x100
        && prev_1st_digit == 3
        && HIGH_BYTE( current_loc ) !=
LOW_BYTE( addr_byte_hi ) )
```

```

        {
            printf("Line %d: WARNING: Possible cross-page short
branch: <%s>\n",line_num,token);
        }

        /* CHECK FOR 2-BYTE SHORT BRANCH ARGUMENT*/
        if ( addr_byte_hi < 0x100
            && addr_byte_lo < 0x100
            && prev_1st_digit == 3 )
        {
            printf("Line %d: WARNING: Possible 2-byte short
branch argument: <%s>\n",line_num,token);
        }

        /* CHECK FOR 1-BYTE LONG BRANCH ARGUMENT*/
        if ( ( (addr_byte_hi > 0xFF) ^ (addr_byte_lo > 0xFF) )
            && prev_1st_digit == 0xC)
        {
            printf("Line %d: WARNING: Possible 1-byte long branch
argument: <%s>\n",line_num,token);
        }

        if (( addr_byte_hi & addr_byte_lo & 0x100) &&
LABELS_IN_OUTFILE )
            fprintf( outfile, "\n%s {%04X}\n\t\t", token,
current_loc );
    }
    break;

default:
    /* IS IT A MACRO? */
    strcpy( string, token );
    if ( (char_ptr = strchr( string, '(' ) ) != '\0' )
        *char_ptr = '\0';          /* KILL THE ( TO ISOLATE THE NAME */

    // SCAN THE MACRO TABLE:
    for (i=0; strcmp( string, mactab[i].name ) != 0
        && i < mactab_len; i++) ;

    if ( i < mactab_len )
    { /* It's a macro! */
        error_code = Expand_macro( token, i, infile, outfile );
        if (error_code)
            return error_code;
    }
    else if ( is_all_hex( token ) )
    {
        // hex constants must have an even number of hex digits
        if ( is_even( strlen( token ) ) )
        {
            current_loc += strlen(token) / 2;
            if ( pass_num == 2 )
                fprintf(outfile, "%s ",token);

            prev_1st_digit = -1; /* ?? it's a mess to set it right,
so at least Re-set it */
        }
        else
        { // odd num of hex digits is an error
            error(HXODDIG, line_num, token);
        }
    }
    else

```

```

        { /* scan the mnemonic table... */
            for ( i = 0; strcmp( token, mntab[i].name ) != 0 &&
                  *(mntab[i].name) != '#'; i++ ); /* scanning... */

            // '#' is a special mnemonic to mark the end of the table:
            if (*(mntab[i].name) != '#')
            {
                if ( pass_num == 2 )
                    fprintf( outfile, "%02X ", (int)mntab[i].code );

                current_loc++;
                prev_1st_digit = mntab[i].code >> 4;
            }
            else
            {
                error( UNSYMBOL, line_num, token );
            }
        }

        } /* END OF switch */
        return error_code;
    } /* END OF if macro, if include, else switch... */

    // printf( "Parse_a_token: WARNING! Should not get here!\n" );
    // ?? not sure if getting here is valid... '\0' means no error.
    return 0;
} /* END OF Parse_a_token */
/*
_____*/

```

```

int
main( int argc, char *argv[] )
{

    FILE *infile, *outfile,*symfile;

    char    token[MAX_TOKEN_LENGTH];
    char    other_half[MAX_TOKEN_LENGTH];
    char    delims[257];
    char    term_char='\0';

    int i,j;
    int error_code;
    char    infilename[32];
    char    outfilename[32];
    char    symfilename[32];

    *infilename = *outfilename = *symfilename = '\0';

    /*
    IF NO ARGUMENTS, PRINT USAGE
    */
    if (argc < 2)
    {
        fprintf(stderr, "\n Usage: asm sourcefile[.asm] [outfile[.cos]] [symfile[.sym]]\n\n");
        return 1;
    }

    /*
    DOES SOURCEFILE HAVE AN EXTENSION?
    */
    strcat( infilename, argv[1] );
    i = strcspn( infilename, "." );

```



```

if ( i == strlen( infilename ) )
{ /* NO: try adding .asm... */
    strcat( infilename, ".asm" );
}
else
{
    *(argv[1] + i) = '\0'; /* remove extension from original argv[1] to use below */
}

// argv[1] contains the basename without extension.

/*
OPEN THE OUTPUT FILE
*/
if ( argc < 3 )
{
    strcat( outfile, argv[1] ); /* start with base */
    strcat( outfile, ".cos" );
}
else
{
    strcat( outfile, argv[2] );
}

outfile = fopen( outfile, "w" );
if ( outfile == '\0' )
{
    fprintf( stderr, "Couldn't open %s\n", outfile );
    return 1;
}

/*
OPEN THE SYMBOL TABLE FILE
*/
if ( argc < 4 )
{
    strcat( symfilename, argv[1] ); /* start with base */
    strcat( symfilename, ".sym" );
}
else
{
    strcat( symfilename, argv[3] );
}

symfile = fopen( symfilename, "w" );
if ( symfile == '\0' )
{
    fprintf( stderr, "Couldn't open %s\n", symfilename );
    return 1;
}

/*
OPEN THE TRACE FILE
*/
if ( TRACE > 0 )
{
    trace_fp = fopen( "asmtrace.txt", "w" );
    if ( trace_fp == '\0' )
    {
        fprintf( stderr, "Couldn't open %s\n", "asmtrace" );
        return 1;
    }
}

```

[illegible]

[illegible]

```

        if ( TRACE > 1 )
            fprintf( trace_fp, "Include_level: %d.\n", include_level );

        while (fgettok(infile, delims, token, &term_char, &line_num) != EOF)
        {
            if ( strchr( token, '(' ) && strchr( token, ')' ) == '\0'
                && term_char == ' ' )
            {
                strcat( token, " " ); /* add in the space */
                Read_until( infile, ')', other_half, MAX_TOKEN_LENGTH );

                strcat( token, other_half );
                strcat( token, ")" );
            }

            if ( TRACE > 0 )
            {
                fprintf( trace_fp, "2==> %s\n", token );
                fflush( trace_fp );
            }

            error_code = Parse_a_token(token,infile,outfile);
            infile = infile_stack[include_level];

            if ( error_code != '\0' )
                error(error_code, line_num, token);

            if (term_char == '\n')
                fprintf( outfile, "\n\t\t" );

        } /* end of while(fgettok(...)) != EOF */

        fclose( infile );
        include_level--;

    } while ( include_level >= 0 );

    // print a final "Z". The 1802 Dev Sys Upload (U) command
    // looks for a Z to know when it can stop accepting data.
    fprintf( outfile, "\nZ\n" );

    return 0;

}
/*
_____/
*/

```

#### Project Notes:

Sept 5:

Added "strings" and {comments}. For now, strings must be entirely on one line of the source file. Comments are contained in braces and may span any length. Strings could be handled like comments, but why would you want more than one line of text? Perhaps to embed CR's...?

The problem with strings is that you lose when you detect the first "<>", you've already lost the delimiter, which was probably a space, but it could have been a <;> or a <,> or a <\t> because strtok replaces it with \0. What to do????

..

Sept 28:

After much frustration with strtok being ALMOST suitable, I wrote fgettok

that gets characters one-at-a-time from a file and returns tokens. It has special cases for { and " to handle comments and quoted strings. And it works. Finally! Now for macros...

Comments are in and have no restrictions! Except that the initial brace { must be preceded by a delimiter (<CR>, <TAB>, or <space>). They are Pascal-style, with open and close braces like these.}

Short branch/Long branch: If the 1st digit of the code preceding a symbolic address was a '3', then only the lower byte of the address is used. Otherwise, a symbolic address is replaced by two bytes.

Dec 21: Ported this code to MPW on the Mac, but it would not compile. MPW was broken. Fixed it, but code would not run right. Bug was that I declared \*token, but did not reserve memory for the token. Changed the declaration to token[MAX\_TOKEN\_LENGTH].

Also needed:

Address offset constants: \$FOO+1. This will be handled in PASS 2. Or \$foo+4.L would be the low byte of the value of foo+4.

Decimal constants would be nice... hmmm... how to express the syntax? 745.d2 means 745 expressed in 2 hex bytes: 02E9 745.d2.h means the high byte: 02

Macros would be nice:

```
macro set_reg(n,value), LDI value.h PHI(n) LDI value.L PLO(n) /endm
.
.
.
$G0:    set_reg(A,$STACK1)
```

Jan. 5, 1993 - Finally got macros working. And they are nestable! There are still some bugs, like the detection of too many or too few macro arguments doesn't work.  
\*/

## tables.h

/\* TABLES.H - This creates arrays of structures that describe macros }, symbols, and mnemonics. The mnemonic table is constant, the other tables are filled at run time according to the file being assembled.

The macro table has three members. name is the name of the macro, text is the body of the macro, and arg is an array of 8 strings of up to 16+\0 characters that contains up to 8 arguments, each argument may be 16 chars long. For example, let's say we want a macro to set register 'n' to the value contained in a symbolic address:

```
#MACRO SET_REG(N,$1)
    LDI $1H PHI(N) LDI $1L PLO(N)
```

```
#ENDM
```

.name would be "SET\_REG", .argv[0] would be a pointer to "N", .argv[1] would be a pointer to "\$1", .argc would be 2, and .text would be "LDI \$1H PHI(N) LDI \$1L PLO(N)".

\$1H means the high byte of the address given by the symbol in the second arg. () means substitute hex digits here, so (N) means substitute one hex digit }, the register number in this case.

The other tables are simpler. The mnemonic table consists of .name, the name of the instruction, and .code, the hex code (unsigned char) of the instruction.

The symbol table is similar: .name is the name of the symbol (minus the \$ identifier: \$start would have the name start), and .loc is the memory location (unsigned int) that the symbol represents.

```
*/
```

```
typedef struct {
    char name[MAX_SYMBOL_LENGTH];
    char argv[MAX_MACRO_ARGS][MAX_SYMBOL_LENGTH];
    int argc;
    char *text;
} Macro_table;
```

```
typedef struct {
    char name[MAX_SYMBOL_LENGTH];
    unsigned int loc;
} Symbol_table;
```

```
typedef struct {
    char name[17];
    unsigned char code;
} Mnemonic_table;
```

```
/*
```

```
DECLARE GLOBAL PARAMETERS
```

```
*/
```

```
Symbol_table symtab[256]; /* typedef Symbol_table defined in tables.h */
Macro_table mactab[64];  /* typedef Macro_table defined in tables.h */
```

```

int symtab_len = 0;
int mactab_len = 0;          /* points at the next avail. mactab entry. */
int current_loc = 0;
int line_num=0;
int pass_num=1;
FILE *infile_stack[MAX_INCLUDE_NESTING];
int include_level=0;

Mnemonic_table mntab[] = {
{ "IDL",0x00 },
{ "LD_1",0x01 },
{ "LD_2",0x02 },
{ "LD_3",0x03 },
{ "LD_4",0x04 },
{ "LD_5",0x05 },
{ "LD_6",0x06 },
{ "LD_7",0x07 },
{ "LD_8",0x08 },
{ "LD_9",0x09 },
{ "LD_A",0x0A },
{ "LD_B",0x0B },
{ "LD_C",0x0C },
{ "LD_D",0x0D },
{ "LD_E",0x0E },
{ "LD_F",0x0F },
{ "INC_0",0x10 },
{ "INC_1",0x11 },
{ "INC_2",0x12 },
{ "INC_3",0x13 },
{ "INC_4",0x14 },
{ "INC_5",0x15 },
{ "INC_6",0x16 },
{ "INC_7",0x17 },
{ "INC_8",0x18 },
{ "INC_9",0x19 },
{ "INC_A",0x1A },
{ "INC_B",0x1B },
{ "INC_C",0x1C },
{ "INC_D",0x1D },
{ "INC_E",0x1E },
{ "INC_F",0x1F },
{ "DEC_0",0x20 },
{ "DEC_1",0x21 },
{ "DEC_2",0x22 },
{ "DEC_3",0x23 },
{ "DEC_4",0x24 },
{ "DEC_5",0x25 },
{ "DEC_6",0x26 },
{ "DEC_7",0x27 },
{ "DEC_8",0x28 },
{ "DEC_9",0x29 },
{ "DEC_A",0x2A },
{ "DEC_B",0x2B },
{ "DEC_C",0x2C },
{ "DEC_D",0x2D },
{ "DEC_E",0x2E },
{ "DEC_F",0x2F },

```

```
{ "BR",0x30 },
{ "BQ",0x31 },
{ "BZ",0x32 },
{ "B_DF",0x33 },
{ "BPZ",0x33 },
{ "BGE",0x33 },
{ "B_EF1_L",0x34 },
{ "B_EF2_L",0x35 },
{ "B_EF3_L",0x36 },
{ "B_EF4_L",0x37 },
{ "SKP",0x38 },
{ "BNQ",0x39 },
{ "BNZ",0x3A },
{ "BNF",0x3B },
{ "BM",0x3B },
{ "BL",0x3B },
{ "B_EF1_H",0x3C },
{ "B_EF2_H",0x3D },
{ "B_EF3_H",0x3E },
{ "B_EF4_H",0x3F },
{ "LDA_0",0x40 },
{ "LDA_1",0x41 },
{ "LDA_2",0x42 },
{ "LDA_3",0x43 },
{ "LDA_4",0x44 },
{ "LDA_5",0x45 },
{ "LDA_6",0x46 },
{ "LDA_7",0x47 },
{ "LDA_8",0x48 },
{ "LDA_9",0x49 },
{ "LDA_A",0x4A },
{ "LDA_B",0x4B },
{ "LDA_C",0x4C },
{ "LDA_D",0x4D },
{ "LDA_E",0x4E },
{ "LDA_F",0x4F },
{ "STR_0",0x50 },
{ "STR_1",0x51 },
{ "STR_2",0x52 },
{ "STR_3",0x53 },
{ "STR_4",0x54 },
{ "STR_5",0x55 },
{ "STR_6",0x56 },
{ "STR_7",0x57 },
{ "STR_8",0x58 },
{ "STR_9",0x59 },
{ "STR_A",0x5A },
{ "STR_B",0x5B },
{ "STR_C",0x5C },
{ "STR_D",0x5D },
{ "STR_E",0x5E },
{ "STR_F",0x5F },
{ "IRX",0x60 },
{ "OUT_1",0x61 },
{ "OUT_2",0x62 },
{ "OUT_3",0x63 },
{ "OUT_4",0x64 },
```



```
{ "OUT_5",0x65 },
{ "OUT_6",0x66 },
{ "OUT_7",0x67 },
{ "INP_1",0x69 },
{ "INP_2",0x6A },
{ "INP_3",0x6B },
{ "INP_4",0x6C },
{ "INP_5",0x6D },
{ "INP_6",0x6E },
{ "INP_7",0x6F },
{ "RET",0x70 },
{ "DIS",0x71 },
{ "LDXA",0x72 },
{ "STXD",0x73 },
{ "ADDXC",0x74 },
{ "SDB",0x75 },
{ "SHRC",0x76 },
{ "RSHR",0x76 },
{ "SMB",0x77 },
{ "SAV",0x78 },
{ "MARK",0x79 },
{ "REQ",0x7A },
{ "SEQ",0x7B },
{ "ADDCI",0x7C },
{ "SDBI",0x7D },
{ "SHLC",0x7E },
{ "RSHL",0x7E },
{ "SMBI",0x7F },
{ "GLO_0",0x80 },
{ "GLO_1",0x81 },
{ "GLO_2",0x82 },
{ "GLO_3",0x83 },
{ "GLO_4",0x84 },
{ "GLO_5",0x85 },
{ "GLO_6",0x86 },
{ "GLO_7",0x87 },
{ "GLO_8",0x88 },
{ "GLO_9",0x89 },
{ "GLO_A",0x8A },
{ "GLO_B",0x8B },
{ "GLO_C",0x8C },
{ "GLO_D",0x8D },
{ "GLO_E",0x8E },
{ "GLO_F",0x8F },
{ "GHI_0",0x90 },
{ "GHI_1",0x91 },
{ "GHI_2",0x92 },
{ "GHI_3",0x93 },
{ "GHI_4",0x94 },
{ "GHI_5",0x95 },
{ "GHI_6",0x96 },
{ "GHI_7",0x97 },
{ "GHI_8",0x98 },
{ "GHI_9",0x99 },
{ "GHI_A",0x9A },
{ "GHI_B",0x9B },
{ "GHI_C",0x9C },
```

```
{ "GHI_D", 0x9D },
{ "GHI_E", 0x9E },
{ "GHI_F", 0x9F },
{ "PLO_0", 0xA0 },
{ "PLO_1", 0xA1 },
{ "PLO_2", 0xA2 },
{ "PLO_3", 0xA3 },
{ "PLO_4", 0xA4 },
{ "PLO_5", 0xA5 },
{ "PLO_6", 0xA6 },
{ "PLO_7", 0xA7 },
{ "PLO_8", 0xA8 },
{ "PLO_9", 0xA9 },
{ "PLO_A", 0xAA },
{ "PLO_B", 0xAB },
{ "PLO_C", 0xAC },
{ "PLO_D", 0xAD },
{ "PLO_E", 0xAE },
{ "PLO_F", 0xAF },
{ "PHI_0", 0xB0 },
{ "PHI_1", 0xB1 },
{ "PHI_2", 0xB2 },
{ "PHI_3", 0xB3 },
{ "PHI_4", 0xB4 },
{ "PHI_5", 0xB5 },
{ "PHI_6", 0xB6 },
{ "PHI_7", 0xB7 },
{ "PHI_8", 0xB8 },
{ "PHI_9", 0xB9 },
{ "PHI_A", 0xBA },
{ "PHI_B", 0xBB },
{ "PHI_C", 0xBC },
{ "PHI_D", 0xBD },
{ "PHI_E", 0xBE },
{ "PHI_F", 0xBF },
{ "LBR", 0xC0 },
{ "LBQ", 0xC1 },
{ "LBZ", 0xC2 },
{ "LBDF", 0xC3 },
{ "NOP", 0xC4 },
{ "LSNQ", 0xC5 },
{ "LSNZ", 0xC6 },
{ "LSNF", 0xC7 },
{ "LSKP", 0xC8 },
{ "LBNQ", 0xC9 },
{ "LBNZ", 0xCA },
{ "LBNF", 0xCB },
{ "LSIE", 0xCC },
{ "LSQ", 0xCD },
{ "LSZ", 0xCE },
{ "LSDF", 0xCF },
{ "SEP_0", 0xD0 },
{ "SEP_1", 0xD1 },
{ "SEP_2", 0xD2 },
{ "SEP_3", 0xD3 },
{ "SEP_4", 0xD4 },
{ "SEP_5", 0xD5 },
```

```

{ "SEP_6",0xD6 },
{ "SEP_7",0xD7 },
{ "SEP_8",0xD8 },
{ "SEP_9",0xD9 },
{ "SEP_A",0xDA },
{ "SEP_B",0xDB },
{ "SEP_C",0xDC },
{ "SEP_D",0xDD },
{ "SEP_E",0xDE },
{ "SEP_F",0xDF },
{ "SEX_0",0xE0 },
{ "SEX_1",0xE1 },
{ "SEX_2",0xE2 },
{ "SEX_3",0xE3 },
{ "SEX_4",0xE4 },
{ "SEX_5",0xE5 },
{ "SEX_6",0xE6 },
{ "SEX_7",0xE7 },
{ "SEX_8",0xE8 },
{ "SEX_9",0xE9 },
{ "SEX_A",0xEA },
{ "SEX_B",0xEB },
{ "SEX_C",0xEC },
{ "SEX_D",0xED },
{ "SEX_E",0xEE },
{ "SEX_F",0xEF },
{ "LDX",0xF0 },
{ "OR",0xF1 },
{ "AND",0xF2 },
{ "XOR",0xF3 },
{ "ADDX",0xF4 },
{ "SD",0xF5 },
{ "SHR",0xF6 },
{ "SM",0xF7 },
{ "LDI",0xF8 },
{ "ORI",0xF9 },
{ "ANDI",0xFA },
{ "XRI",0xFB },
{ "ADDI",0xFC },
{ "SDI",0xFD },
{ "SHL",0xFE },
{ "SMI",0xFF },
{ "#",0x00 } /* This is the "end of table" symbol */
};

```

## System ROM Dump

Here is a hex dump of the control system code.

```
0000  71 00 F8 08 B2 B7 BF F8 FF A2 F8 C0 A7 F8 07 B4
0010  B5 B6 F8 01 A4 F8 16 A5 F8 EE A6 F8 FF B1 F8 FF
0020  A1 E2 C4 C4 F8 00 B3 F8 2B A3 D3 C0 06 00 D4 07
0030  50 0D 0A 52 65 73 65 74 0D 0A 00 C4 C4 C4 D4 07
0040  50 0D 0A 2A 00 F8 08 B8 F8 80 A8 D4 07 6C 07 FF
0050  03 32 3E FF 05 3A 5F 88 FD 80 32 4B 28 30 68 07
0060  58 18 88 FD BF C2 00 DF D4 07 44 07 FD 0D 3A 4B
0070  D4 07 50 0A 00 F8 08 B8 F8 80 A8 08 FF 0D 32 3E
0080  FF 35 C2 07 9E FF 01 C2 07 9E FF 02 C2 03 00 FF
0090  01 C2 07 9E FF 01 C2 02 1A FF 01 C2 07 9E FF 04
00A0  C2 01 00 FF 01 C2 01 53 FF 03 C2 07 9E FF 02 C2
00B0  00 C4 FF 01 C2 07 9E FF 01 C2 07 9E FF 01 C2 05
00C0  00 C0 00 DF 18 08 FF 0D C2 00 DF FF 38 C2 02 30
00D0  FF 04 C2 07 9E FF 09 C2 07 9E FF 05 C2 07 9E D4
00E0  07 50 0D 0A 57 68 61 74 3F 07 0D 0A 00 C0 00 3E
00F0  FF FF FF FF FF FF FF FF FF FF FF FF FF AA
```

```
0100  D4 06 CB D4 06 E0 89 FA F0 A9 F8 00 AA 89 FA 0F
0110  3A 32 D4 07 50 0D 0A 00 6A FA 01 32 23 6B FF 03
0120  C2 00 3E 99 57 D4 07 76 89 57 D4 07 76 D4 07 50
0130  20 00 89 FA 03 3A 3C D4 07 50 20 00 09 57 D4 07
0140  76 D4 07 50 20 00 19 1A 8A 3A 0D D4 07 50 2A 00
0150  C0 00 45 D4 06 CB D4 06 E0 89 AA 99 BA 18 D4 06
0160  E0 89 AB 99 BB E7 8A 57 8B F7 AC 9A 57 9B 77 BC
0170  E2 1C CB 07 9E 18 D4 06 E0 89 52 8A F7 99 52 9A
0180  77 3B A0 0A 59 E9 F5 E7 32 8F E2 D4 01 DA E2 6A
0190  FA 01 CA 01 F8 1A 19 2C 8C 3A 83 9C 3A 83 30 CB
01A0  2C 8C 52 89 F4 AA 9C 52 99 74 BA 1C 8A A9 9A B9
01B0  0B 59 E9 F5 E7 32 BC E2 D4 01 DA E2 6A FA 01 CA
01C0  01 F8 29 2B 2C 8C 3A B0 9C 3A B0 E2 D4 07 50 44
01D0  6F 6E 65 0D 0A 00 C0 00 3E D5 D4 07 50 57 2E 46
01E0  2E 20 40 20 07 00 99 57 D4 07 76 89 57 D4 07 76
01F0  D4 07 50 0D 0A 00 30 D9 D4 07 50 41 62 6F 72 74
```

```
0200  65 64 20 40 20 00 99 57 D4 07 76 89 57 D4 07 76
0210  D4 07 50 0D 0A 00 6B C0 00 3E D4 06 CB D4 06 E0
0220  F8 02 BF F8 27 AF DF 99 B3 89 A3 D3 FF FF FF FF
0230  F8 40 BD F8 00 AD 0D FF FF 3A 50 1D 9D FF 48 3A
0240  36 D4 07 50 45 6D 70 74 79 2E 0D 0A 00 C0 00 3E
0250  D4 07 50 4E 6F 74 20 00 30 41 FF FF FF FF FF FF
0260  FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0270  FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0280  FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

0290	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
02A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
02B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
02C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
02D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
02E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
02F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

0300	F8	00	AB	D4	06	CB	D4	06	E0	89	AA	FA	F0	A9	D4	07
0310	50	07	00	99	57	D4	07	76	89	57	D4	07	76	D4	07	50
0320	20	00	89	FA	03	3A	2C	D4	07	50	20	00	09	57	D4	07
0330	76	D4	07	50	20	00	19	89	FA	0F	3A	22	D4	07	50	20
0340	00	29	D4	07	50	08	08	08	00	89	FA	03	FF	03	3A	55
0350	D4	07	50	08	00	89	57	8A	E7	F5	E2	3A	41	D4	07	50
0360	07	00	D4	07	6C	07	FF	03	C2	00	3E	FF	05	32	9E	FF
0370	05	C2	00	3E	8B	3A	8E	07	FF	5B	3A	7F	1B	30	62	07
0380	FF	20	32	B3	FF	1C	32	D1	FF	02	32	DB	30	97	07	FF
0390	5D	CA	04	19	2B	30	62	D4	07	44	07	BA	30	E5	29	89
03A0	FA	0F	FF	0F	3A	AF	D4	07	50	0D	0A	00	89	30	0A	89
03B0	AA	30	42	19	89	FA	0F	32	A6	29	09	57	D4	07	76	D4
03C0	07	50	20	00	19	89	FA	03	3A	62	D4	07	50	20	00	30
03D0	62	F8	10	AF	29	2F	8F	3A	D4	30	A6	F8	10	AF	19	2F
03E0	8F	3A	DE	30	A6	D4	07	6C	07	FF	03	C2	00	3E	FF	05
03F0	C2	04	11	07	17	57	27	9A	57	D4	07	AD	07	59	E9	F5

0400	E2	C2	04	09	D4	07	50	07	00	D4	07	50	08	00	C0	03
0410	B3	D4	07	50	08	00	C0	03	62	D4	07	50	5B	00	C0	03
0420	FC	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0430	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0440	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0450	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0460	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0470	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0480	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0490	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
04A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
04B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
04C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
04D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
04E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
04F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

{ upload "U" }

0500	F8	08	BB	F8	00	AB	F8	00	5B	1B	F8	50	5B	2B	F8	08
0510	BC	F8	02	AC	D4	06	CB	D4	06	E0	89	5C	1C	99	5C	2C

0520	D4 07 6C 07	FF 40 32 75	07 FF 24 32	61 07 FF 7B
0530	32 6B 07 FF	5A C2 00 3E	07 FF 03 C2	00 3E 07 FF
0540	30 3B 20 07	FD 46 3B 20	07 FF 41 33	54 07 FD 39
0550	3B 20 30 54	17 D4 07 6C	27 D4 07 AD	07 59 19 30
0560	20 D4 07 6C	07 FF 3A 3A	61 30 20 D4	07 6C 07 FF
0570	7D 3A 6B 30	20 F8 08 B8	F8 80 A8 F8	04 AA D4 07
0580	6C 07 58 18	2A 8A 3A 7E	F8 08 B8 F8	7F A8 D4 06
0590	E0 EC 89 F7	A9 1C 99 77	B9 2C E2 EB	89 F4 A9 1B
05A0	99 74 B9 2B	E2 30 20 FF	FF FF FF FF	FF FF FF FF
05B0	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
05C0	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
05D0	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
05E0	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
05F0	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF

0600	F8 08 BF F8	DD AF F8 55	5F 1F F8 08	5F D4 06 A1
0610	C0 00 2E FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
0620	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
0630	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
0640	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
0650	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
0660	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
0670	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
0680	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
0690	FF FF FF FF	FF FF FF FF	FF FF FF FF	FF FF FF FF
06A0	D5 F8 08 BF	AE F8 DD AF	4F 52 02 FE	73 7B CF 7A
06B0	C4 64 2E 8E	3A AA 0F 52	F8 08 AE 02	FE 73 7B CF
06C0	7A C4 64 2E	8E 3A BB 30	A0 FF D5 18	08 FF 0D 3A
06D0	D6 12 12 C0	07 9E FF 13	3A CB 30 CA	C4 C4 C4 D5
06E0	18 08 57 17	18 08 57 27	D4 07 AD 07	B9 18 08 57
06F0	17 18 08 57	27 D4 07 AD	07 A9 30 DF	FF FF FF AA

0700	D3 43 B5 43	A5 93 73 83	73 95 B3 85	A3 F8 07 B5
0710	F8 16 A5 30	00 D3 60 72	A3 F0 B3 30	15 6A FA 01
0720	3A 29 6A FA	02 32 22 30	49 6B FF 03	C2 00 02 FF
0730	10 3A 22 6A	FA 01 32 33	6B FF 03 C2	00 02 FF 0E
0740	3A 33 FF D5	30 1D C4 C4	C4 E7 61 27	E2 30 43 D5
0750	12 72 A7 F0	B7 07 32 5E	D4 07 44 17	30 55 17 97
0760	73 87 73 F8	08 B7 F8 C0	A7 30 4F D5	6A FA 01 32
0770	6C 6B 57 30	6B D5 07 FA	F0 F6 F6 F6	F6 FF 0A 33
0780	84 FC 3A C8	FC 41 17 57	D4 07 44 27	07 FA 0F FF
0790	0A 33 96 FC	3A C8 FC 41	57 D4 07 44	30 75 D4 07
07A0	50 48 75 68	3F 07 0D 0A	00 C0 00 3E	D5 07 FA F0
07B0	F6 F6 F6 F6	FF 03 32 C6	07 FA 0F FC	09 FE FE FE
07C0	FE FA F0 57	30 D0 07 FA	0F FE FE FE	FE FA F0 57
07D0	17 07 FA F0	F6 F6 F6 F6	FF 03 32 E3	07 FA 0F FC

07E0	09	30	E6	07	FA	0F	27	E7	F1	57	E2	30	AC	D3	17	07
07F0	27	FF	01	C4	C4	3A	F1	30	ED	FF	FF	FF	FF	FF	FF	FF