R H U G

The Official Newsletter of the

RICHMOND HEATH USERS GROUP

Volume II Issue 6

JUNE 1984

****

Subscriptions: $12.00 per calendar year.
Editor: Dave Harrington, 10813 Lunswood Rd, Chester VA, 23831
Next editorial deadline: July 2, 1984.
Note to other HUGs: If you want to trade newsletters with us, please mail yours to Harold Lanna, 600 Watch Hill Rd, Midlothian, VA 23113

****

(c) 1984, Richmond Heath Users Group. Contents may be reproduced only with proper credit to both the individual author and the Richmond Heath Users Group, but are not to be reproduced for commercial purpose without the express consent of RHUG.

****

MEETING NOTICE

The next meeting will be Monday, June 21, at 7:30. The meeting location is at Harold Lanna's Office, TDFB, Innsbrook Corporate Center, Innsbrook Rd, Off W Broad St, 'way out by beautiful downtown Short Pump, VA. Please call Harold at his office (346-0440) or home 794-0856, for instructions.

Everyone is welcome!

****

MINUTES (Meeting of May 21, 1984)

The minutes were not available for inclusion here, and will be presented at our next meeting.

****

THOUGHTS WHILE HACKING - Ed.

I recently purchased a copy of a program MXPRINT from a software house MICRO-INOVATIONS. The program is supposed to turn the MX-80/100 into a near letter-quality printer thru software. My first attempts, last February, resulted in no output at all. I made a few attempts at fixing the switch settings on my printer, but the result was still nothing. Since I was then still bogged down in too many other things, I just shelved the program until a later date.

I was pleasantly surprised when I received a notice from them advising me of a software change to be implemented thru the DDT Utility of CP/M. I finally got around to installing the patch, and cranked up the set. The results can best be described as better, but I still can't use the program. For some reason that I can't explain, the program doesn't like something, and wants to do wierd things with paging, line spacing and a few characters that seem to be a combination of free graphics. It did print out about 90% of my text in a readible format, and the resultant characters were indeed close appoximations of letter-quality print.

I sent them a letter with a sample of my output, about three weeks ago, but I haven't heard from them as yet. More later, if there is anything to report.

I am glad to have another article from Jim Scott for this newsletter. His informative articles are very educational. I would always welcome some more articles from others of you out there.

****

ASSEMBLY LANGUAGE PROGRAMMING - PART 9

by Jim Scott

INTRODUCTION

This is the ninth and last of a series of articles which parallel and summarize the discussions about assembly language at our meetings. The purpose of the discussions and the articles is to present enough information about assembly language programming so that someone who knows how to program in a higher-level language, and is willing to use the proper manuals for reference, will at least have some idea how to get started at programming in assembly language.

This article provides a brief introduction to the interfaces designed into the operating systems (CP/M and HDOS) for use by assembly language programs.

But first, here is an even briefer summary of what the preceding articles in this series have covered. Part 1 introduced the topic, and pointed out that the CPU under consideration is the 8080. Part 2 described the logical structure of the 8080 CPU: its memory and registers. Parts 3 through 7 described in detail the machine language of the 8080, and Part 8 discussed assembler directives.

The machine operations described in parts 3-7 are translated by the assembler from their source form into machine language. Each machine operation in the source program will become a one- to three-byte machine language instruction in the executable program (COM or ABS file), also known as the object program. The topic of part 8, the assembler directive (also known as a pseudo operation) is a source instruction that does not get translated into a machine language instruction. It is a way of telling the assembler something about how we want it to do its job. One way of looking at it is this - an assembler directive is executed at assembly time (when the assembler is actually being run to translate the source program into the executable program), whereas a machine operation will not be executed until object time (when the resulting COM or ABS file, or object program, is run, after the process of assembling is finished).

Now we learn how to write assembly language code that can ask the operating system to do things. In the case of HDOS, we will see that a strange instruction (SCALL) is involved which is a combination of machine operation and assembler directive.

OPERATING SYSTEM INTERFACES

The operating system you run and the assembly language program you write will be in the computer's memory at the same time, so they will have to deal with each other. The first thing that happens (once you have assembled your program and produced the COM or ABS file) is that you tell the operating system to run your program. You type in the name of the program, and the operating system loads the object file and starts it executing.

Once the program is running, it can run pretty much on its own if it wants to. (It may get temporarily interrupted by the operating system, for example when you are also running a spooler; but the operating system takes care of this automatically.) It can do arithmetic and move data from memory to registers and back again. But eventually it will want to communicate with the outside world; i.e., it will need to do input or output through a peripheral device such as a terminal, disk drive, or printer. In almost every case, the program will have to make a request to the operating system to do the I/O for it.

If you remember the medieval days of the H8 with cassette tape I/O, you may realize that back then there was essentially no operating system. You could code your own instructions to do terminal or tape I/O, using the IN and OUT instructions (or using subroutines in ROM). There are several reasons why you don't usually do this now: 1. Disk I/O is more complicated than tape I/O. 2. If you do your own I/O, you could confuse the operating system. 3. The operating system interface makes I/O easier.

CP/M I/O CALLS

Now let's look at some of the input/output operations that CP/M users can ask their operating system to do for them.

All CP/M I/O operations involve using the CALL instruction to transfer control temporarily to a subroutine known as BDOS, or Basic Disk Operating System. The address of this subroutine is memory location 5. (Actually, at memory location 5 is a JMP instruction, which jumps to the real location of BDOS.)

All of these operations also involve putting data into certain registers before executing the CALL instruction, to tell BDOS what we want it to do for us. Into register C goes a function number, which designates the operation to be done. For example, 1 means Read Console, or "read a character from the keyboard"; 2 means Write Console, or "write a character to the screen"; and 21 means Write Next Record, or "write a record to disk". Register pair DE, or sometimes just register E, will often contain other input data for the operation. After BDOS finishes, register A, or sometimes register pair HL, usually contains data representing the result of the operation.

The usual procedure is for an assembly language program to contain EQU assembler directives to equate easy-to-remember names with some of these

numbers. For example, the example program printed in part 1 of this series contained the following EQUs:

```
BASE    EQU     Ø       ;BASE ADDRESS OF CP/M.
BDOS    EQU     BASE+5H ;BDOS ENTRY POINT.
READC   EQU     1       ;READ CONSOLE INTO (A).
TYPEC   EQU     2       ;WRITE TO CONSOLE FROM (E).
```

Then the programming for the various I/O operations amounts to moving the correct input values into the registers, and doing the call. For example:

```
        MVI     C,READC
        CALL    BDOS    ;READ A CHARACTER FROM THE KEYBOARD, INTO (A).
        STA     CHAR    ;SAVE THE CHARACTER IN MEMORY.
        .
        .
        .
CHAR    DS      1       ;RESERVE A BYTE FOR CHARACTER FROM KEYBOARD.
```

Or:

```
        MVI     E,'G'   ;PUT LETTER 'G' INTO REGISTER (E).
        MVI     C,TYPEC
        CALL    BDOS    ;SEND IT TO THE SCREEN.
```

Note how these calls work. The CALL instruction, as described in part 6, does two things: first, it puts onto the stack the address of the instruction following the CALL instruction; then, it branches to the address (BDOS, or memory location 5, in this case) specified in the CALL instruction. This is the way to transfer control to a subroutine, so that the subroutine can eventually transfer control back when it is finished. In other words, the CALL branches to BDOS, which does what we want it to do; when BDOS is finished, it executes a RET instruction to return to the address at the top of the stack, which, if everything is working properly, will be the address of the instruction following the CALL instruction.

The way that BDOS knows what we want it to do is by what it finds in the registers. First it looks at the value in register C. If it finds a 1, it knows that we want it to read a character from the keyboard (and it doesn't need any other information from any other registers). If it finds a 2, it knows that we want it to write to the screen the character we have put into register E.

Also note that BDOS often does more for us than meets the eye. For example, when it reads a character from the console keyboard, it waits until a key has actually been pressed, so it will have something to hand back via (A).

Now, here is a section of code from the CP/M example in part 1:

```
; DISPLAY ON CONSOLE THE CHARACTER IN (A).

DISPLAY EQU     $
        PUSH    H       ;SAVE REGISTERS.
        PUSH    D
        PUSH    B
        MOV     E,A     ;PUT CHARACTER INTO (E) FOR BDOS.
        MVI     C,TYPEC
        CALL    BDOS    ;SEND IT TO THE SCREEN.
```

```
        POP     B       ;RESTORE REGISTERS.
        POP     D
        POP     H
        RET             ;RETURN TO CALLER.
```

This code is written as a subroutine that can be invoked with a CALL instruction. It has an advantage over the mere use of the three instructions shown previously; it saves and restores the register pairs BC, DE, and HL, which can be modified by the setup for the call, and by BDOS itself.

Another section of code from the example program is this, which calls the DISPLAY subroutine just listed:

```
; DISPLAY MESSAGE ON CONSOLE.
; MESSAGE TEXT FOLLOWS CALL STATEMENT, AND ENDS WITH
; A BYTE OF Ø.
; RETURN IS TO INSTRUCTION FOLLOWING Ø BYTE.
; REGISTERS H,L,A ARE DESTROYED.

TYPTX   EQU     $
        POP     H       ;ADDRESS OF MSG IN H,L.
TYPTX1  EQU     $
        MOV     A,M     ;PUT NEXT CHAR INTO (A).
        CPI     Ø       ;IS IT ZERO?
        JZ      TYPTX2  ;YES, QUIT.
        CALL    DISPLAY ;CALL ROUTINE TO SEND IT TO SCREEN.
        INX     H       ;ADD 1 TO H,L (NEXT CHAR).
        JMP     TYPTX1  ;LOOP TO SEND NEXT CHAR.
TYPTX2  EQU     $
        INX     H       ;ADD 1 TO H,L (BYTE AFTER Ø).
        PCHL            ;RETURN TO BYTE AFTER Ø.
```

This is a handy subroutine to send an entire message (as opposed to a single character) to the console screen. It is a little unusual in that it uses PCHL rather than RET to return; instead of returning to the point immediately after the instruction that calls it, it returns to the byte after the message. Thus the CALL TYPTX instruction can be followed by the message to be sent. The end of the message is marked by a byte containing a binary Ø (not a character zero).

The code to call TYPTX to send the message 'This Is The Title' to the screen would be as follows:

```
        CALL    TYPTX   ;DISPLAY THE FOLLOWING MESSAGE.
        DB      'This Is The Title',Ø
```

(CP/M, or BDOS, can display a message without needing either of the two routines DISPLAY or TYPTX. It works like this:

```
TYPMSG  EQU     9       ;DISPLAY MESSAGE.
        .
        .
        .
        LXI     D,PMSG  ;(DE) POINTS TO MESSAGE.
        MVI     C,TYPMSG
        CALL    BDOS    ;DISPLAY THE MESSAGE, UP TO '$'.
        .
        .
        .
```

```
PMSG    DB      'This Is The Title$'
```

The disadvantage of this code is that it requires the text of the message to be located somewhere other than right next to the CALL instruction that makes it get displayed; this makes the program source code harder to read.)

It's beyond the scope of this article to discuss in detail how to do disk I/O in an assembly language program. However, here is a brief list of some of the function codes used in calling BDOS to do disk I/O:

```
15  Open a disk file
16  Close a disk file
19  Delete a disk file
20  Read the next record of an open disk file
21  Write the next record of an open disk file
22  Create a disk file (create the directory entry)
23  Rename a disk file
```

The setup for most of these functions is as follows:

```
Load the address of the FCB into register pair DE
Load the function number into register C
CALL BDOS
Check register (A) for a code indicating success or failure.
```

The FCB, or File Control Block, is a data area that contains information on the disk file, including the file name. It is usually located at memory location 5C (hex), and is 36 bytes long.

The buffer, which contains the record to be written to disk, or the record just read from disk, is usually located at memory location 80 (hex), and is 128 bytes long.

HDOS I/O CALLS

Now let's see how HDOS does some of these same I/O operations.

All HDOS I/O operations involve using the SCALL instruction to transfer control temporarily to HDOS. This is the instruction I mentioned near the beginning of the article, which is a cross between a machine operation and assembler directive. The assembler translates SCALL as if it were an RST 7 machine operation followed by a DB assembler directive. The RST, or Restart, instruction puts the address of the next instruction onto the stack, then branches to an address in low memory equal to 8 times the operand of the RST. In this case, it branches to 8 x 7 = 56 = 070 octal, which is the address of a JMP instruction which jumps to HDOS. When HDOS is finished, it returns to the instruction following the byte reserved by the implicit DB; this is the address put on the stack, plus one.

HDOS, like CP/M, has function numbers that are passed to the operating system to specify which function is to be done. For example, 1 is SCIN or "read a character from the console keyboard"; and 2 is SCOUT, or "send a character to the console screen". (The fact that these two mean pretty much the same thing in HDOS as in CP/M is mostly coincidence.)

So an assembly language program to be run under HDOS can read a character from the keyboard by using the following code:

```
SCIN    EQU     1       ;CONSOLE INPUT.
        .
        .
        .
        SCALL   SCIN    ;READ A CHARACTER FROM THE KEYBOARD, INTO (A).
        STA     CHAR    ;SAVE THE CHARACTER IN MEMORY.
        .
        .
        .
CHAR    DS      1       ;RESERVE A BYTE FOR CHARACTER FROM KEYBOARD.
```

Looks a lot like CP/M, doesn't it? The main difference is that HDOS uses

```
        SCALL   SCIN
```

where CP/M uses

```
        MVI     C,READC
        CALL    BDOS
```

The SCALL SCIN assembles as if it were

```
        RST     7
        DB      SCIN
```

In other words, in CP/M the function number is passed in register C, and in HDOS it is passed in the byte following the RST instruction.

Note that SCALL is assembled in this way by the HDOS assembler only. The CP/M assembler would choke on it.

Similarly, an HDOS assembly language program can display a character on the console screen as follows:

```
SCOUT   EQU     2       ;CONSOLE OUTPUT.
        .
        .
        .
        MVI     A,'X'   ;PUT CHARACTER INTO (A).
        SCALL   SCOUT   ;SEND IT TO THE SCREEN.
```

CONCLUSION

This article completes this series on assembly language programming. For more information on CP/M CALLs or HDOS SCALLs, see any of various manuals for these operating systems. Even more important, examine any sample assembly language programs you can find, in magazines or anywhere else.

There have been other articles and series of articles on assembly language programming. The main object of this series has been to start more near the beginning than most, and to explain some of the fine points that usually are taken for granted. I hope this approach has contributed to your understanding of programming. But the coverage has been elementary. Your best next move would be to check out some of these other series of articles, to continue your learning. I recommend, for example, the series "Getting Started With Assembly

Language", by Pat Swayne, beginning in the April 1983 REMark.

You probably won't do most of your programming in assembly language. Maybe you'll never use it to write a non-trivial program. But if you don't at least understand assembly language, you won't fully understand your computer.

Good luck with your continued learning!