6502 Intermediate Programming

by Ted Kosan

Part of The Professor And Pat series
(professorandpat.org)

Table of Contents

Complexity And Subroutines

After I finished my lunch, I looked out a window so see what kind of a day it was. The sky was a dark sheet of gray and there was a steady rain falling. "What a great day for working inside!" I thought. I grabbed an umbrella and opened it as I stepped out of the back door of my house and walked towards the workshop. I was deep in thought as I rounded the corner of the shop and so I was startled when I saw Pat standing in the rain under a huge black umbrella.

Pat and I stared at each other for a few moments. Then Pat said "Complexity..."

I blinked and said "What!?"

"Complexity, Professor." Said Pat. "I have been going nuts trying to deal with all the complexity in the 6502 programming exercises you gave me to work through. I am having a hard time keeping all the parts of a program straight in my mind. There is too much detail to keep track of all at the same time."

"Oh that." I said. "There are techniques that have been developed which help with that problem. Lets go inside the shop and I will show them to you."

I unlocked and opened the door to my shop and we then made our way to the electronics room. How long were you standing there in the rain?" I asked.

"About 30 minutes." replied Pat "You know I don't like to bother you when you are in your house."

We sat down in front of the computer and as it booted up Pat asked "So, how do programmers deal with the complexity in a program?"

I thought for a while then asked "Do you like pirate movies?"

"Sure," said Pat "I think most people like pirate movies."

"On a pirate ship," I said "who makes all of the important decisions, like where the ship should go, what direction to point the ship in during a storm, and when to 'batten down the hatches'"?

"The captain does." replied Pat.

"Can the captain of a large sailing ship do all of the numerous tasks that need to be done to sail a ship without assistance?" I asked.

"Of course not!" said Pat. "There are too many things that need to be done, like raising and maintaining the sails, turning the rudder, plotting the course, and keeping a lookout. There is simply too much to deal with for one person to be able to handle it all by themselves."

"How is the captain able to control the ship, then, if there are so many tasks to handle?" I asked.

"The captain is not alone, though." said Pat "There is a crew on the ship and they handle most of the tasks that need to be done. The captain tells the crew what to do and they do it."

"Yes," I said. "The captain **calls** to the crew members to tell them what to do, and they do it. This is similar to one technique that is used to handle the complexity in a program. With this technique, a program is divided into one **main** part and one or more **helper** parts. The **main** part of a program is similar to the **captain** of a sailing ship, and the **helper** parts are similar to the **crew**. The main part of a program is often called 'main' and the 'crew' parts are each given a unique name, just like each crew member on a ship has a unique name.

The helper parts of a program are generally called **subroutines**, but they are also called **functions**, **methods**, **procedures**, and **subprograms** (depending on what computer language is being used). The helper parts of an assembly language program are usually called **subroutines**. When a program is executed, the code in the main part of the program is executed first and then the main part of the program **calls** the subroutines as needed. The subroutines can also call each other if they need work done that another subroutine is able to do."

"Can you show me a program that uses a subroutine?" Asked Pat. "I want to see how they work."

"Yes, I can do that." I said. I then created the following program and assembled it:

```
                000001 |;Program Name: addnums.asm.
                000002 |;
                000003 |;Version: 1.0.
                000004 |;
                000005 |;Description: Use a subroutine to add 2 numbers
                000006 |; All communications between the main routine and
                000007 |; the subroutine are handeled with registers.
                000008 |;
                000009 |;Assumptions: When added, the numbers will not be
                000010 |; greater than 255.
                000011 |
                000012 |
                000013 |;************************************
                000014 |;      Program entry point.
                000015 |;************************************
0200            000016 |     org 0200h
                000017 |
0200            000018 |Main *
0200 A2 01      000019 |     ldx #1d
0202 A0 02      000020 |     ldy #2d
0204 20 0B 02   000021 |     jsr AddNums
0207 8D 15 02   000022 |     sta answer
                000023 |
```

```
                      000024 |;Exit the program.
020A 00               000025 |     brk
                      000026 |
                      000027 |
                      000028 |;*************************************
                      000029 |;        Subroutines area.
                      000030 |;*************************************
                      000031 |
                      000032 |;*************************************
                      000033 |;AddNums subroutine.
                      000034 |;
                      000035 |;Information passed in:
                      000036 |;X and Y hold the two numbers to be added.
                      000037 |;
                      000038 |;Information returned:
                      000039 |;The result is returned in the 'A' register.
                      000040 |;*************************************
020B                  000041 |AddNums *
020B 8A               000042 |     txa
020C 8C 14 02         000043 |     sty temp
020F 18               000044 |     clc
0210 6D 14 02         000045 |     adc temp
0213 60               000046 |     rts
                      000047 |
                      000048 |
                      000049 |;*************************************
                      000050 |;        Variables area.
                      000051 |;*************************************
0214 00               000052 |temp  dbt 0d
0215 00               000053 |answer     dbt 0d
                      000054 |
                      000055 |     end
                      000056 |
```

"In this program," I said "execution begins in the main part of the program and notice how I placed a label called "Main" at the entry point so it is easier to find. The subroutine is called **AddNums** and it begins at address **020Bh**. The way a subroutine is called is with the **JSR** instruction, which stands for **Jump SubRoutine**. It works similar to the JMP instruction in that it changes the Program Counter to the address of the subroutine, which in this case is 020Bh. What makes it different from the JMP instruction, however, is that is also provides a way for the Program Counter to be pointed back to the next instruction below the JSR when the subroutine is finished executing."

"How does it do that?" asked Pat.

"We will cover how this is done in a moment," I replied "but for now, can you figure out how the main program tells the subroutine which 2 numbers to add together?"

Pat studied the program then said "It looks like the main program is placing the numbers to be added into the X and Y registers before it calls the subroutine."

"Yes," I said "and when the JSR instruction sends the Program Counter to the subroutine, all the subroutine needs to do is to obtain the numbers to be added from these registers. The TXA instruction transfers the number that is in the X register to the 'A' register and the STY instruction stores the number in the Y register into a variable called **temp**. The ADC instruction then adds the contents of the 'A' register to the contents of **temp** and the sum is placed back into the 'A' register.

The **RTS** command stands for **ReTurn from Subroutine** and it will sent the Program Counter to the address of the instruction that is immediately below the JSR command that issued the call. The result of the calculation is returned to the caller in the 'A' register and the last thing the main program does before exiting is to store the result into a variable called **answer**."

I then loaded the program into the emulator, unassembled it, and traced it so that Pat could see how it worked:

```
-u 0200

0200  A2 01     LDX #01h
0202  A0 02     LDY #02h
0204  20 0B 02  JSR 020Bh
0207  8D 15 02  STA 0215h
020A  00        BRK
020B  8A        TXA
020C  8C 14 02  STY 0214h
020F  18        CLC
0210  6D 14 02  ADC 0214h
0213  60        RTS
0214  00        BRK

-t 0200

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202          00         01        00         FD      00010100

0202  A0 02     LDY #02h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0204          00         01        02         FF      00010100

0204  20 0B 02  JSR 020Bh

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   020B          00         01        02         FD      00010100

020B  8A        TXA

-t
```

```
PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   020C          01          01         02          FB        00010100

020C   8C 14 02   STY 0214h

-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   020F          01          01         02          FB        00010100

020F   18         CLC

-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0210          01          01         02          FB        00010100

0210   6D 14 02   ADC 0214h

-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0213          03          01         02          FD        00010100

0213   60         RTS

-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0207          03          01         02          FF        00010100

0207   8D 15 02   STA 0215h

-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   020A          03          01         02          FD        00010100

020A   00         BRK

-d 0215

0215   03 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................
```

"Notice how the program counter is changed from 0204h to 020Bh when the JSR command is executed, and then how it is changed to 0207h (the address of the instruction under the JSR instruction) by the RTS instruction." I said.

"Thats cool!" said Pat. "But how does the RTS command know the address of the instruction that is immediately below the JSR command that called its subroutine?"

"Look closely at the trace output again," I replied "and tell me if you notice any values changing before and after JSR and before and after RTS."

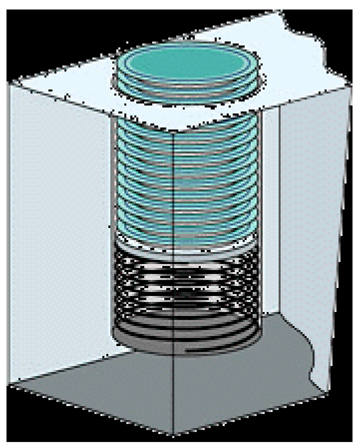
"Pat studied the trace output again then said "Hey! The **register** labeled **StkPtr** changes from **FF** to **FD** when the JSR is executed, and then from **FD** back to **FF** when the RTS is executed! What's the purpose of that register?"

The Stack Pointer Register

"That register is called the **Stack Pointer** and its purpose is to keep track of information like the addresses that subroutines need to return to when they are finished." I said.

"How does it do that?" asked Pat.

"Have you ever gone to a restaurant that had a plate stack machine next to the salad bar?" I asked. "They look something like this." I then found an image of a plate stack machine on the Internet:



"Sure," said Pat.

"How do they work?" I asked.

"Well, the restaurant workers **push** the plates onto the stack and the customers **pull** them off when they go to the salad bar." replied Pat.

"Is the first plate that is pushed onto the stack the first one that is pulled off?" I asked.

Pat thought about this question for a while then said "No, the first plate that is pushed onto the stack is the last plate that is pulled off."

"Correct." I said. "Most modern CPUs have a stack mechanism built into them, but it is implemented in a **data structure** in memory instead of in a mechanical device. Stacks are a type of data structure called a **LIFO** or **Last In First Out** data structure. The 6502's stack starts at **01FFh** in memory and grows downward as bytes are pushed onto it."

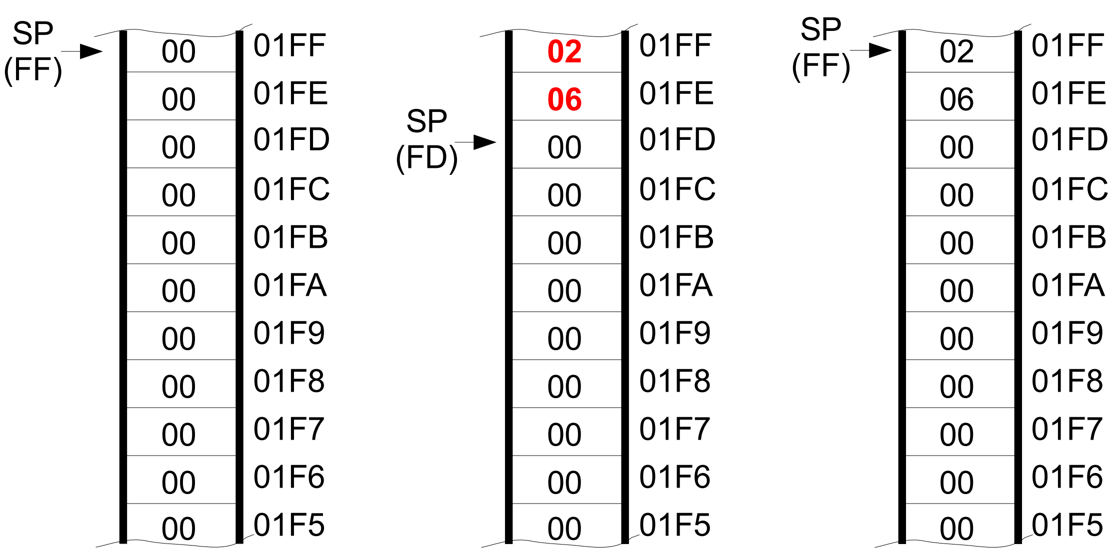
"What's a data structure?" asked Pat.

"A data structure is an organized way to store data in memory so that it can be easily accessed." I replied. "The lists of numbers between 0 and 255 we used in earlier programs were examples of data structures."

"Okay," said Pat "but how is a stack used to allow an RTS to return to the instruction that is underneath the JSR that called its subroutine?"

"When the JSR instruction is executed," I said "it calculates the address of the 3rd byte in the JSR instruction and pushes this address onto the top of the stack. The address of the 3rd byte of the JSR instruction in this program is **0206h**, so this is the address that gets pushed on the stack at the position of the stack pointer. When the RTS instruction is executed, the return address is pulled off the stack, 1 is added to it, and then this number is placed into the Program counter. The number 0206h + 1 is **0207h** and this is indeed the address of the next instruction after the JSR, which is the STA instruction." I then drew a diagram of the stack in memory on the whiteboard. (See Fig. 1)

Figure 1



Before executing the JSR instruction. After executing the JSR instruction. After executing the RTS instruction.

After Pat had looked at the diagram for a while, I traced through the program again, but this time I dumped the top part of the stack before and after executing the JSR instruction so Pat could see it in operation. (Note: The monitor currently has a bug in it that shows the stack pointer value 2 less than it should be.)

```
-d 01f0
01F0  00 00 BF 00 00 04 00 07 - 00 15 B7 17 4A 11 65 10  ............J.e.

<JSR instruction is executed here>

-d 01f0
01F0  BF 00 00 02 00 05 00 14 - AF 17 4A 11 65 10 06 02  ..........J.e...
```

Subroutines In The Monitor

"I think I am am starting to understand how subroutines can make handling the complexity in a program easier," said Pat "but can you show me a larger program that uses subroutines so I can get a better feel for how they are used?"

"Sure," I said "I have a wonderful program you can look at! The source code for the umon65 monitor is in the umon65 directory in the download file for the emulator and it contains a large number of subroutines. The file is called **umon65.asm** and I would like you to copy it into the directory you have been using to assemble your programs and then assemble it. Then, edit the .lst file that is generated so we can look at it." (Note: you should do this now too using version 1.04 or higher of the emulator download file.)

Pat assembled the monitor and then brought the .lst file up in an editor. As we looked through the monitor's list file from top to bottom, I recorded the names of all the subroutines:

```
Get Line From Serial Port
Parse Input Buffer
Check for Valid Command
Maskable Interrupt Service
Break Service
Assemble Command
Operator Scan
Address Mode Scan
Address Mode Table Search
Operand Scan
Scan For Hex Digit
Breakpoint Command
Compare Breakpoint Address
Dump command
Enter Command
Get List
Fill Command
Go Command
```

```
Help Command
Load Command
Process Record Length and Address
Get Code Byte Without Loading into Memory
Get Code Byte and Load it into Memory
Check Checksum
Get Number
Accumulate Checksum
ASCII Digit to Binary Number
Process Header Record
Process Code Record
Process Termination Record
Move Command
Register Command
Search Command
Trace Command
Scan for Valid Opcode
Unassemble Command
Print Mnemonic
Increment Pointer A
Get Address
Output a Colon Prompt
Out Spaces
Covert ASCII character to lower case
Covert ASCII character to upper case
Ascii to Binary
Initialize Variables
Print Message
Get Character (Don't Wait) From Serial Channel
Get Character (Wait) From Serial Channel
Output Chacacter to Serial Channel
Delay
```

"There are 50 subroutines in the monitor!" cried Pat "In fact, almost the whole program is subroutines!"

"If the monitor did not use subroutines, it would have been too complex to create and debug, and maintaining it would have been nearly impossible. Almost all program use subroutines for this reason, regardless of what language they were written in."

Utility Subroutines In The Monitor

Pat studied the monitor's .lst file for a while then pointed to a section of the program and said "What's a jump table?" Here is the section of code that Pat was pointing to:

```
               |;************************************************************
               |;              Monitor Utility Subroutine Jump Table.
               |;************************************************************
E003 4C B5 F3 | jmp OutChar   ;Output byte in A register to serial port.
               |
E006 4C 74 F3 | jmp GetChar   ;Get a byte from the serial port.
               |
E009 4C 97 F3 | jmp GetCharW  ;Wait and get a byte from the serial port.
```

```
                  |
E00C 4C 57 F3 | jmp PrntMess  ;Print a message to the serial port.
                  |
E00F 4C A7 F2 | jmp OutSpace  ;Output spaces to the serial port.
                  |
E012 4C 03 F3 | jmp OutHex    ;Output a HEX number to the serial port.
                  |
E015 4C 41 EA | jmp DgtToBin  ;Convert an ASCII digit into binary.
                  |
E018 4C 70 E0 | jmp GetLine   ;Input a line from the serial port.
```

"I was wondering if you were going to notice that." I said. "A jump table usually contains a series of JMP instructions that jump to subroutines that may be useful outside of a program. In this case, the subroutines listed in this jump table may be useful to programs that are run with the monitor. After all, the monitor program is in memory just like our programs are, and our programs can access the monitor's code as easily as the monitor itself can."

"Do you mean the programs we write are able to call these subroutines?" asked Pat.

"Yes," I replied "the monitor uses these subroutines to print messages to the user's screen and take input from the user's keyboard."

"I didn't know our programs could communicate with the user!" cried Pat "Lets write some programs that use these subroutines so I can see how they work. But first, why do we need to use a jump table to access these routines? Why can't we just call the utility subroutines directly by their addresses which are listed in the .lst file?"

"We could," I said "and these addresses would work as long as the monitor's code was unchanged. But if the monitor was edited and reassembled, all the address of the subroutines under edited code would be changed and this would break our program. If we call these subroutines indirectly through the jump table, however, the jump table automatically points to the new subroutine addresses when the monitor is reassembled. As long as we don't move the jump table itself, none of the programs that use it will break."

"I see," said Pat "that makes sense."

Strings

"I will now create some example programs that use the monitor's utility subroutines so you can see how they work." I said. I then created the following program called Hello, assembled it, loaded it into the monitor, and executed it:

Program 1: Hello

```
                000001 |;Program Name: hello.asm.
                000002 |;
                000003 |;Version: 1.02.
```

```
                     000004 |;
                     000005 |;Description: Print all characters in Mess using OutChar.
                     000006 |
                     000007 |;**************************************
                     000008 |;      Program entry point.
                     000009 |;**************************************
0200                 000010 |      org 0200h
                     000011 |
0200                 000012 |Main *
                     000013 |
                     000014 |;Point X to first character of Mess.
0200 A2 00           000015 |      ldx #0d
0202                 000016 |LoopTop *
                     000017 |;Grab a character from Mess.
0202 BD 11 02        000018 |      lda Mess,x
                     000019 |
                     000020 |;If the character is the 0 which is at the end
                     000021 |; of Mess, then exit.
0205 C9 00           000022 |      cmp #0d
0207 F0 07           000023 |      beq DonePrint
                     000024 |
                     000025 |;Call the OutChar monitor utility subroutine.
0209 20 03 E0        000026 |      jsr E003h
                     000027 |
                     000028 |;Point X to the next character in Mess and loop back.
020C E8              000029 |      inx
020D 4C 02 02        000030 |      jmp LoopTop
                     000031 |
0210                 000032 |DonePrint *
                     000033 |
                     000034 |;Exit the program.
0210 00              000035 |      brk
                     000036 |
                     000037 |;**************************************
                     000038 |;        Variables area.
                     000039 |;**************************************
0211 48              000040 |Mess  dbt "Hello"
0212 65 6C 6C
0215 6F
0216 00              000041 |      dbt 0d
                     000042 |
                     000043 |      end
                     000044 |
```

"The purpose of this program is to send a message to the user's screen." I said. "The message is held in the program's **variables** area and it consists of ASCII character that are placed next to each other in memory. See if you can find the message and tell me what it says."

Pat looked at the variables area of the program then said "The message says 'Hello'".

"And what are the values of the ASCII characters that are placed next to each other in order to form this message?" I asked.

"**48**, **65**, **6C**, **6C**, and **6F**." Replied Pat.

"Very good." I said. "A sequence of ASCII characters that are placed next to each other in memory are called a **string** and therefore the word **Hello** in this program is a string. The idea behind a string is that it represents a sequence of ASCII characters that are 'strung' together. Now, what variable is the string Hello assigned to?"

Pat looked at the program again then said "The string **Hello** is assigned to the variable **Mess**. Does this mean that the variable Mess holds the complete string?"

"No." I replied. "If you look closely at the variable Mess, you will notice that it only represents the address of the **first** character of the string, which is a capital letter '**H**'."

"I don't understand how Mess can refer to a string when it can only point it the string's first character." said Pat.

"Lets walk through the program, then, so you can see how strings work." I said. "This program uses the **X register** and a **loop** to point to each of the characters in the string, one after the another. The characters are sent to the monitor's **OutChar** subroutine one by one and the **OutChar** subroutine is responsible for displaying them on the user's screen.

The **ldx #0d** instruction on line **000015** sets the X register to offset 0 into the string The **lda Mess,x** instruction on line **000018** copies the character that is at offset x into the string Mess into the 'A' register. The first time through the loop it will copy the letter '**H**' into the 'A' register, the second time through the loop it will copy the letter '**e**' into the 'A' register, and so on. The **inx** instruction at line **000029** increments the X register each time through the loop so that it points to the next character in the string."

"How does the loop know when to stop looping?" asked Pat "I mean, how does it know where the string ends?"

"Look at the memory location that is immediately after the last character in the string." I said. "What value does it contain?"

Pat looked at the program then said "You placed a **dbt 0** immediately after the string," said Pat "so a **zero** has been placed after the lower case 'o' in the string. Why did you do that?"

"Look at the code that is on lines **000022** and **000023** and see if you can answer your own question." I replied.

After a while Pat said "Oh, I get it! The zero that was placed after the string is being used as an **end-of-string marker**. The **cmp #0d** on line **000022** is looking for this marker and if it is found, the **beq DonePrint** instruction on line **000023** exits the loop."

"Correct!" I said.

Passing An Address To A Subroutine

"Instead of sending a string one character at a time to the OutChar subroutine in order to display it," I said "we can use the monitor's **PrntMess** subroutine. All we have to do is to send the address of the first character of the string to the PrntMess routine, and it will display all of the string's characters using a loop which is similar to the loop in the Hello program. We have a problem, though."

"What's that?" asked Pat.

"How many bits wide are addresses in the 6502?" I asked.

"16 bits." replied Pat.

"And how many bits wide are the 6502's registers?" I asked.

"8 bits... oh I see the problem now." said Pat. "If addresses are 16 bits wide, but registers are only 8 bits wide, we can't send an address to the subroutine in a register. Hmmm, could we break the address in half then send the **upper half** in a register and send the **lower half** in another register."

"Yes," I replied "and this is exactly what the next example program does. When a 16 bit address is cut in half, the upper half is 8 bits wide and the lower half is 8 bits wide. Since 8 bits is a byte, the upper part of the address is called the **upper byte** and the lower part is called the **lower byte**. The upper byte is also called the **Most Significant Byte** or **MSB**, and the lower byte is called the **Least Significant Byte** or **LSB**."

"Why are they called this?" asked Pat.

"Because if bits in the LSB are changed, it changes the overall value of the address less than if bits in the MSB are changed." I replied.

I then created a program called Hello2:

Program 2: Hello2

```
000001 |;Program Name: hello2.asm.
000002 |;
000003 |;Version: 1.02.
000004 |;
000005 |;Description: Print all of the characters in Mess
```

```
using
                    000006 |; PrntMess
                    000007 |
                    000008 |;*************************************
                    000009 |;      Program entry point.
                    000010 |;*************************************
0200                000011 |     org 0200h
                    000012 |
0200                000013 |Main *
                    000014 |;Load the low byte of address of Mess into X.
0200 A2 08          000015 |     ldx #Mess<
                    000016 |
                    000017 |;Load the high byte of address of Mess into Y.
0202 A0 02          000018 |     ldy #Mess>
                    000019 |
                    000020 |;Call PrntMess monitor utility subroutine.
0204 20 0C E0       000021 |     jsr E00ch
                    000022 |
                    000023 |;Exit the program.
0207 00             000024 |     brk
                    000025 |
                    000026 |;*************************************
                    000027 |;        Variables area.
                    000028 |;*************************************
0208 48             000029 |Mess dbt "Hello2"
0209 65 6C 6C
020C 6F 32
020E 00             000030 |     dbt 0d
                    000031 |
                    000032 |     end
                    000033 |
```

"If you look at the address of the string variable Mess," I said "its low byte is **08** and its high byte is **02**. The assembler has special syntax which is used to extract either the low byte or the high byte from a variable or a label. The instruction **ldx #Mess<** places the low byte of the address of the variable Mess into the x register and the instruction **ldy #Mess>** places the high byte of the address of the variable Mess into the y register. The less than sign < is used to indicate the low byte and the greater than sign > is used to indicate the high byte."

The equ Assembler Directive

"Requiring the programmer to remember the address in the jump table of each of the utility subroutines is not as efficient as it could be." I said. "Therefore, most assemblers have an **equ** directive (or something similar to it) which helps with problems like this. The **equ** directive tells the assembler to take the string of characters to its left and replace it with the string of characters to its right, at each point where the string on its left is used in the program. I will now create a program called Hello3 which demonstrates how the **equ** directive can be used." I then created the following program:

Program 3: Hello3

```
                 000001 |;Program Name: hello3.asm.
                 000002 |;
                 000003 |;Version: 1.02.
                 000004 |;
                 000005 |;Description: Print all characters in Mess using
                 000006 |; PrntMess and equs.
                 000007 |;
                 000008 |;Assumptions: When added, the numbers will not be
                 000009 |; greater than 255.
                 000010 |
                 000011 |
;**************************************************************
                 000012 |     ;Monitor Utility Subroutine Jump Table.
                 000013 |
;**************************************************************
0000             000014 |OutChar   equ E003h ;Output byte in reg A to the user.
                 000015 |
0000             000016 |GetChar   equ E006h ;Get a byte from the serial port.
                 000017 |
0000             000018 |GetCharW  equ E009h ;Wait and get a byte from the user.
                 000019 |
0000             000020 |PrntMess  equ E00Ch ;Print a message to the user.
                 000021 |
0000             000022 |OutSpace  equ E00Fh ;Output spaces to the serial port.
                 000023 |
0000             000024 |OutHex    equ E012h ;Output a HEX number to the user.
                 000025 |
0000             000026 |DgtToBin  equ E015h ;Convert an ASCII digit to binary.
                 000027 |
0000             000028 |GetLine   equ E018h ;Input a line from the serial port.
                 000029 |
                 000030 |
                 000031 |
                 000032 |;*************************************
                 000033 |;      Program entry point.
                 000034 |;*************************************
0200             000035 | org 0200h
                 000036 |
0200             000037 |Main *
0200 A2 08       000038 | ldx #mess<
0202 A0 02       000039 | ldy #mess>
0204 20 0C 10    000040 | jsr PrntMess
                 000041 |
                 000042 |;Exit the program.
0207 00          000043 | brk
                 000044 |
                 000045 |;*************************************
                 000046 |;        Variables area.
                 000047 |;*************************************
0208 48          000048 |mess   dbt "Hello3"
0209 65 6C 6C
020C 6F 33
020E 00          000049 | dbt 0d
                 000050 |
```

```
                000051 | end
                000052 |
```

"In this program," I said "**equ** directives are used to associate the name of each monitor utility subroutine with its address in the jump table. Notice how this allows us to use **jsr PrntMess** on line **000040** instead of **jsr 100ch** in Hello2. The same machine code is generated in both cases but **jsr PrntMess** is easier to remember."

A Final Program For Pat To Study

"I am going to give you a final example program, Pat." I said. "This program demonstrates how to use the monitor utility subroutines to interact with the user and I want you to try to figure out how it works on your own."

"Okay," said Pat.

Program 4: addinput

```
;Program Name: addinput.asm.
;
;Version: 1.02.
;
;Description: Input 2 single digit numbers from the user, add
; them together, and then output the answer..

     ;*********************************************************
     ;              Monitor Utility Subroutine Jump Table.
     ;*********************************************************
OutChar   equ E003h ;Output byte in reg A to the user.

GetChar   equ E006h ;Get a byte from the serial port.

GetCharW  equ E009h ;Wait and get a byte from the user.

PrntMess  equ E00Ch ;Print a message to the user.

OutSpace  equ E00Fh ;Output spaces to the serial port.

OutHex    equ E012h ;Output a HEX number to the user.

DgtToBin  equ E015h ;Convert an ASCII digit to binary.

GetLine   equ E018h ;Input a line from the serial port.


;**************************************
;      Program entry point.
;**************************************
      org 0200h

Main *
;Ask user to enter the first number.
      ldx #InMess1<
```

```
       ldy #InMess1>
       jsr PrntMess

;Obtain the first number from the user, convert it from ASCII
;to binary, and then store it in num1.
       jsr GetCharW
       jsr DgtToBin
       sta num1

;Ask user to enter the second number.
       ldx #InMess2<
       ldy #InMess2>
       jsr PrntMess

;Obtain the second number from the user, convert it from ASCII
;to binary, and then store it in num2.
       jsr GetCharW
       jsr DgtToBin
       sta num2

;Add the numbers together and store the answer in sum.
       clc
       lda num1
       adc num2
       sta sum

;Inform the user that the answer is being printed.
       ldx #OutMess<
       ldy #OutMess>
       jsr PrntMess

;Print the answer.
       lda sum
       jsr OutHex

Exit *
;Exit the program.
       brk

;*************************************
;        Variables area.
;*************************************
InMess1 dbt "Enter number 1:"
        dbt 0d
InMess2 dbt "Enter number 2:"
        dbt 0d
OutMess dbt "The sum is:"
        dbt 0d

num1    dbt 0d
num2    dbt 0d
sum     dbt 0d

       end
```

Exercises

1) Enter programs 1-4 into the emulator and execute them to see how they work.

2) Create a program that contains a subroutine that adds 1 to the contents of the 'A' register when it is called. Have the main program call the subroutine 3 times with different values in 'A'.

3) Create a program that prints "You entered a one" if the user enters a 1, "You entered a two" if the user enters a 2, and "You entered a three" if the user enters a 3.