

6502 Intermediate Programming

by Ted Kosan

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

Copyright © 2008 by Ted Kosan

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/>

Table of Contents

6502 Intermediate Programming.....	1
Complexity And Subroutines.....	3
The Stack Pointer Register.....	8
Subroutines In The Monitor.....	11
Utility Subroutines In The Monitor.....	12
Strings.....	13
Program 1: Hello.....	13
Passing An Address To A Subroutine.....	16
Program 2: Hello2.....	17
The equ Assembler Directive.....	18
Program 3: Hello3.....	18
A Final Program For Pat To Study.....	19
Program 4: addinput.....	19
Exercises.....	21

1 **Complexity And Subroutines**

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

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

8 I blinked and said "What!?"

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

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

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

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

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

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

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

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

25 "The captain does." replied Pat.

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

28 "Of course not!" said Pat. "There are too many things that need to be done, like raising and
29 maintaining the sails, turning the rudder, plotting the course, and keeping a lookout. There is

30 simply too much to deal with for one person to be able to handle it all by themselves."

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

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

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

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

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

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

```

48          000001 |;Program Name: addnums.asm.
49          000002 |;
50          000003 |;Version: 1.0.
51          000004 |;
52          000005 |;Description: Use a subroutine to add 2 numbers
53          000006 |; All communications between the main routine and
54          000007 |; the subroutine are handled with registers.
55          000008 |;
56          000009 |;Assumptions: When added, the numbers will not be
57          000010 |; greater than 255.
58          000011 |
59          000012 |
60          000013 |;*****
61          000014 |;          Program entry point.
62          000015 |;*****
63 0200      000016 |          org 0200h
64          000017 |
65 0200      000018 |Main *
66 0200 A2 01  000019 |          ldx #1d
67 0202 A0 02  000020 |          ld y #2d
68 0204 20 0B 02 000021 |          jsr AddNums
69 0207 8D 15 02 000022 |          sta answer
70          000023 |

```

```

71          000024 |;Exit the program.
72 020A 00      000025 |      brk
73          000026 |
74          000027 |
75          000028 |;*****
76          000029 |;      Subroutines area.
77          000030 |;*****
78          000031 |
79          000032 |;*****
80          000033 |;AddNums subroutine.
81          000034 |;
82          000035 |;Information passed in:
83          000036 |;X and Y hold the two numbers to be added.
84          000037 |;
85          000038 |;Information returned:
86          000039 |;The result is returned in the 'A' register.
87          000040 |;*****
88 020B          000041 |AddNums *
89 020B 8A      000042 |      txa
90 020C 8C 14 02 000043 |      sty temp
91 020F 18      000044 |      clc
92 0210 6D 14 02 000045 |      adc temp
93 0213 60      000046 |      rts
94          000047 |
95          000048 |
96          000049 |;*****
97          000050 |;      Variables area.
98          000051 |;*****
99 0214 00      000052 |temp dbt 0d
100 0215 00     000053 |answer      dbt 0d
101          000054 |
102          000055 |      end
103          000056 |

```

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

112 "How does it do that?" asked Pat.

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

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

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

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

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

129 -u 0200

```

130 0200 A2 01      LDX #01h
131 0202 A0 02      LDY #02h
132 0204 20 0B 02  JSR 020Bh
133 0207 8D 15 02  STA 0215h
134 020A 00        BRK
135 020B 8A        TXA
136 020C 8C 14 02  STY 0214h
137 020F 18        CLC
138 0210 6D 14 02  ADC 0214h
139 0213 60        RTS
140 0214 00        BRK

```

141 -t 0200

	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
142	0202	00	01	00	FD	00010100

```

144 0202 A0 02      LDY #02h

```

145 -t

	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
146	0204	00	01	02	FF	00010100

```

148 0204 20 0B 02  JSR 020Bh

```

149 -t

	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
150	020B	00	01	02	FD	00010100

```

152 020B 8A        TXA

```

153 -t

```

154 PgmCntr (PC)  Accum (AC)  XReg (XR)  YReg (YR)  StkPtr (SP)  NV-BDIZC (SR)
155      020C          01        01        02        FB        00010100

156 020C  8C 14 02  STY 0214h

157 -t

158 PgmCntr (PC)  Accum (AC)  XReg (XR)  YReg (YR)  StkPtr (SP)  NV-BDIZC (SR)
159      020F          01        01        02        FB        00010100

160 020F  18          CLC

161 -t

162 PgmCntr (PC)  Accum (AC)  XReg (XR)  YReg (YR)  StkPtr (SP)  NV-BDIZC (SR)
163      0210          01        01        02        FB        00010100

164 0210  6D 14 02  ADC 0214h

165 -t

166 PgmCntr (PC)  Accum (AC)  XReg (XR)  YReg (YR)  StkPtr (SP)  NV-BDIZC (SR)
167      0213          03        01        02        FD        00010100

168 0213  60          RTS

169 -t

170 PgmCntr (PC)  Accum (AC)  XReg (XR)  YReg (YR)  StkPtr (SP)  NV-BDIZC (SR)
171      0207          03        01        02        FF        00010100

172 0207  8D 15 02  STA 0215h

173 -t

174 PgmCntr (PC)  Accum (AC)  XReg (XR)  YReg (YR)  StkPtr (SP)  NV-BDIZC (SR)
175      020A          03        01        02        FD        00010100

176 020A  00          BRK

177 -d 0215

178 0215  03 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....

```

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

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

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

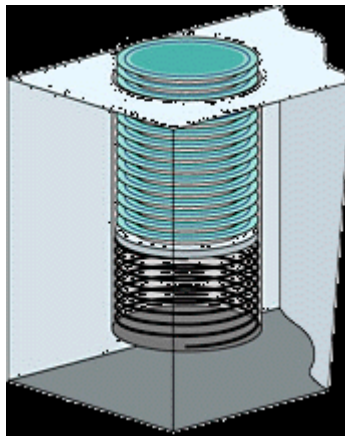
186 "Pat studied the trace output again then said "Hey! The **register** labeled **StkPtr** changes from
187 **FF** to **FD** when the JSR is executed, and then from **FD** back to **FF** when the RTS is executed!
188 What's the purpose of that register?"

189 **The Stack Pointer Register**

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

192 "How does it do that?" asked Pat.

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



196 "Sure," said Pat.

197 "How do they work?" I asked.

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

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

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

203 "Correct." I said. "Most modern CPUs have a stack mechanism built into them, but it is
 204 implemented in a **data structure** in memory instead of in a mechanical device. Stacks are a type
 205 of data structure called a **LIFO** or **Last In First Out** data structure. The 6502's stack starts at
 206 **01FFh** in memory and grows downward as bytes are pushed onto it."

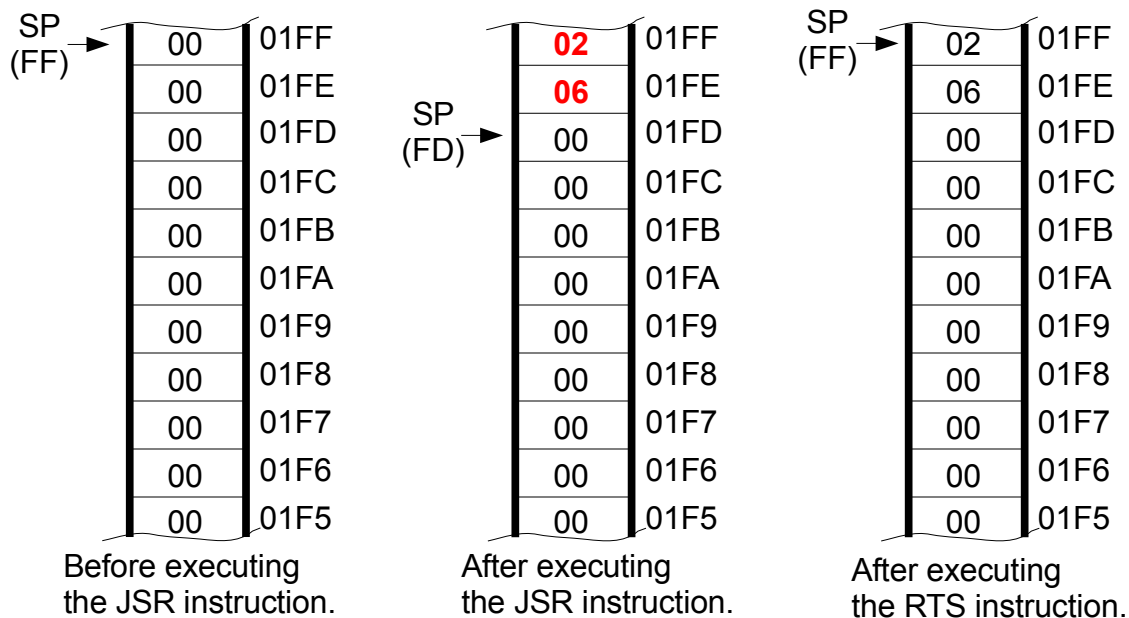
207 "What's a data structure?" asked Pat.

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

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

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

Figure 1



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

```
225 -d 01f0
226 01F0 00 00 BF 00 00 04 00 07 - 00 15 B7 17 4A 11 65 10 .....J.e.
```

227 **<JSR instruction is executed here>**

```
228 -d 01f0
229 01F0 BF 00 00 02 00 05 00 14 - AF 17 4A 11 65 10 06 02 .....J.e...
```

230 Subroutines In The Monitor

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

234 "Sure," I said "I have a wonderful program you can look at! The source code for the umon65
235 monitor is in the umon65 directory in the download file for the emulator and it contains a large
236 number of subroutines. The file is called **umon65.asm** and I would like you to copy it into the
237 directory you have been using to assemble your programs and then assemble it. Then, edit the
238 .lst file that is generated so we can look at it." (Note: you should do this now too using version
239 1.04 or higher of the emulator download file.)

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

```
242 Get Line From Serial Port
243 Parse Input Buffer
244 Check for Valid Command
245 Maskable Interrupt Service
246 Break Service
247 Assemble Command
248 Operator Scan
249 Address Mode Scan
250 Address Mode Table Search
251 Operand Scan
252 Scan For Hex Digit
253 Breakpoint Command
254 Compare Breakpoint Address
255 Dump command
256 Enter Command
257 Get List
258 Fill Command
259 Go Command
```

```

260 Help Command
261 Load Command
262 Process Record Length and Address
263 Get Code Byte Without Loading into Memory
264 Get Code Byte and Load it into Memory
265 Check Checksum
266 Get Number
267 Accumulate Checksum
268 ASCII Digit to Binary Number
269 Process Header Record
270 Process Code Record
271 Process Termination Record
272 Move Command
273 Register Command
274 Search Command
275 Trace Command
276 Scan for Valid Opcode
277 Unassemble Command
278 Print Mnemonic
279 Increment Pointer A
280 Get Address
281 Output a Colon Prompt
282 Out Spaces
283 Covert ASCII character to lower case
284 Covert ASCII character to upper case
285 Ascii to Binary
286 Initialize Variables
287 Print Message
288 Get Character (Don't Wait) From Serial Channel
289 Get Character (Wait) From Serial Channel
290 Output Chacacter to Serial Channel
291 Delay

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

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

```

297 Utility Subroutines In The Monitor

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

```

300 |;*****
301 |; Monitor Utility Subroutine Jump Table.
302 |;*****
303 E003 4C B5 F3 | jmp OutChar ;Output byte in A register to serial port.
304 |
305 E006 4C 74 F3 | jmp GetChar ;Get a byte from the serial port.
306 |
307 E009 4C 97 F3 | jmp GetCharW ;Wait and get a byte from the serial port.

```

```

308      |
309  E00C 4C 57 F3 | jmp PrntMess ;Print a message to the serial port.
310      |
311  E00F 4C A7 F2 | jmp OutSpace ;Output spaces to the serial port.
312      |
313  E012 4C 03 F3 | jmp OutHex   ;Output a HEX number to the serial port.
314      |
315  E015 4C 41 EA | jmp DgtToBin ;Convert an ASCII digit into binary.
316      |
317  E018 4C 70 E0 | jmp GetLine  ;Input a line from the serial port.

```

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

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

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

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

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

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

337 **Strings**

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

341 **Program 1: Hello**

```

342      000001 |;Program Name: hello.asm.
343      000002 |;
344      000003 |;Version: 1.02.

```

```

345          000004 |;
346          000005 |;Description: Print all characters in Mess using
347 OutChar.
348          000006 |
349          000007 |;*****
350          000008 |;      Program entry point.
351          000009 |;*****
352 0200      000010 |      org 0200h
353          000011 |
354 0200      000012 |Main *
355          000013 |
356          000014 |;Point X to first character of Mess.
357 0200 A2 00 000015 |      ldx #0d
358 0202      000016 |LoopTop *
359          000017 |;Grab a character from Mess.
360 0202 BD 11 02 000018 |      lda Mess,x
361          000019 |
362          000020 |;If the character is the 0 which is at the end
363          000021 |; of Mess, then exit.
364 0205 C9 00 000022 |      cmp #0d
365 0207 F0 07 000023 |      beq DonePrint
366          000024 |
367          000025 |;Call the OutChar monitor utility subroutine.
368 0209 20 03 E0 000026 |      jsr E003h
369          000027 |
370          000028 |;Point X to the next character in Mess and loop back.
371 020C E8      000029 |      inx
372 020D 4C 02 02 000030 |      jmp LoopTop
373          000031 |
374 0210      000032 |DonePrint *
375          000033 |
376          000034 |;Exit the program.
377 0210 00      000035 |      brk
378          000036 |
379          000037 |;*****
380          000038 |;      Variables area.
381          000039 |;*****
382 0211 48      000040 |Mess dbt "Hello"
383 0212 65 6C 6C
384 0215 6F
385 0216 00      000041 |      dbt 0d
386          000042 |
387          000043 |      end
388          000044 |

```

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

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

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

395 "48, 65, 6C, 6C, and 6F." Replied Pat.

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

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

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

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

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

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

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

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

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

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

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

426 the **beq DonePrint** instruction on line **000023** exits the loop."

427 "Correct!" I said.

428 **Passing An Address To A Subroutine**

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

434 "What's that?" asked Pat.

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

436 "16 bits." replied Pat.

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

438 "8 bits... oh I see the problem now." said Pat. "If addresses are 16 bits wide, but registers are
439 only 8 bits wide, we can't send an address to the subroutine in a register. Hmmmm, could we break
440 the address in half then send the **upper half** in a register and send the **lower half** in another
441 register."

442 "Yes," I replied "and this is exactly what the next example program does. When a 16 bit address
443 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,
444 the upper part of the address is called the **upper byte** and the lower part is called the **lower byte**.
445 The upper byte is also called the **Most Significant Byte** or **MSB**, and the lower byte is called the
446 **Least Significant Byte** or **LSB**."

447 "Why are they called this?" asked Pat.

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

450 I then created a program called Hello2:

451 **Program 2: Hello2**

```
452          000001 |;Program Name: hello2.asm.  
453          000002 |;  
454          000003 |;Version: 1.02.  
455          000004 |;  
456          000005 |;Description: Print all of the characters in Mess
```

```

457 using
458         000006 |; PrntMess
459         000007 |
460         000008 |;*****
461         000009 |;         Program entry point.
462         000010 |;*****
463 0200     000011 |         org 0200h
464         000012 |
465 0200     000013 |Main *
466         000014 |;Load the low byte of address of Mess into X.
467 0200 A2 08    000015 |         ldx #Mess<
468         000016 |
469         000017 |;Load the high byte of address of Mess into Y.
470 0202 A0 02    000018 |         ldy #Mess>
471         000019 |
472         000020 |;Call PrntMess monitor utility subroutine.
473 0204 20 0C E0 000021 |         jsr E00ch
474         000022 |
475         000023 |;Exit the program.
476 0207 00       000024 |         brk
477         000025 |
478         000026 |;*****
479         000027 |;         Variables area.
480         000028 |;*****
481 0208 48       000029 |Mess dbt "Hello2"
482 0209 65 6C 6C
483 020C 6F 32
484 020E 00       000030 |         dbt 0d
485         000031 |
486         000032 |         end
487         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."

488 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:

489 **Program 3: Hello3**

```

490          000001 |;Program Name: hello3.asm.
491          000002 |;
492          000003 |;Version: 1.02.
493          000004 |;
494          000005 |;Description: Print all characters in Mess using
495          000006 |; PrntMess and equs.
496          000007 |;
497          000008 |;Assumptions: When added, the numbers will not be
498          000009 |; greater than 255.
499          000010 |
500          000011 |
501 ;*****
502          000012 |          ;Monitor Utility Subroutine Jump Table.
503          000013 |
504 ;*****
505 0000          000014 |OutChar   equ E003h ;Output byte in reg A to the user.
506          000015 |
507 0000          000016 |GetChar   equ E006h ;Get a byte from the serial port.
508          000017 |
509 0000          000018 |GetCharW   equ E009h ;Wait and get a byte from the user.
510          000019 |
511 0000          000020 |PrntMess   equ E00Ch ;Print a message to the user.
512          000021 |
513 0000          000022 |OutSpace   equ E00Fh ;Output spaces to the serial port.
514          000023 |
515 0000          000024 |OutHex      equ E012h ;Output a HEX number to the user.
516          000025 |
517 0000          000026 |DgtToBin    equ E015h ;Convert an ASCII digit to binary.
518          000027 |
519 0000          000028 |GetLine     equ E018h ;Input a line from the serial port.
520          000029 |
521          000030 |
522          000031 |
523          000032 |;*****
524          000033 |;          Program entry point.
525          000034 |;*****
526 0200          000035 | org 0200h
527          000036 |
528 0200          000037 |Main *
529 0200 A2 08      000038 | ldx #mess<
530 0202 A0 02      000039 | ldy #mess>
531 0204 20 0C 10   000040 | jsr PrntMess
532          000041 |
533          000042 |;Exit the program.
534 0207 00          000043 | brk
535          000044 |
536          000045 |;*****
537          000046 |;          Variables area.
538          000047 |;*****
539 0208 48          000048 |mess      dbt "Hello3"
540 0209 65 6C 6C
541 020C 6F 33
542 020E 00          000049 | dbt 0d
543          000050 |

```

```

544         000051 | end
545         000052 |

```

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

550 A Final Program For Pat To Study

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

554 "Okay," said Pat.

555 Program 4: addinput

```

556 ;Program Name: addinput.asm.
557 ;
558 ;Version: 1.02.
559 ;
560 ;Description: Input 2 single digit numbers from the user, add
561 ; them together, and then output the answer..

562 ;*****
563 ;               Monitor Utility Subroutine Jump Table.
564 ;*****
565 OutChar    equ E003h ;Output byte in reg A to the user.
566
567 GetChar    equ E006h ;Get a byte from the serial port.
568
569 GetCharW   equ E009h ;Wait and get a byte from the user.
570
571 PrntMess   equ E00Ch ;Print a message to the user.
572
573 OutSpace   equ E00Fh ;Output spaces to the serial port.
574
575 OutHex     equ E012h ;Output a HEX number to the user.
576
577 DgtToBin   equ E015h ;Convert an ASCII digit to binary.
578
579 GetLine    equ E018h ;Input a line from the serial port.

580 ;*****
581 ;               Program entry point.
582 ;*****
583         org 0200h

584 Main *
585 ;Ask user to enter the first number.
586         ldx #InMess1<

```

```
587         ldy #InMess1>
588         jsr PrntMess

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

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

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

603 ;Add the numbers together and store the answer in sum.
604         clc
605         lda num1
606         adc num2
607         sta sum

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

612 ;Print the answer.
613         lda sum
614         jsr OutHex

615 Exit *
616 ;Exit the program.
617         brk

618 ;*****
619 ;      Variables area.
620 ;*****
621 InMess1 dbt "Enter number 1:"
622         dbt 0d
623 InMess2 dbt "Enter number 2:"
624         dbt 0d
625 OutMess dbt "The sum is:"
626         dbt 0d

627 num1    dbt 0d
628 num2    dbt 0d
629 sum     dbt 0d

630         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.