

6502 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

Pat Returns From A Visit.....	3
Algorithms.....	4
Computation.....	8
UML Activity Diagrams.....	12
Calculating The Sum Of The Numbers Between 1 And 10.....	12
Finding The Largest Number In A List Of 100 Numbers.....	17
Copy The Contents Of list1 To list2.....	27
Scan string1 And Determine How Many Capital And Lower Case A's It Contains.....	28
Place Capital A's In The First 10 Bytes Of alphaList, Capital B's In The Second 10 Bytes Of alphaList, And So On Until alphaList Is Filled With Letters.....	30
Calculate The Sum Of The 100 Bytes That Are In numbersList.....	31
Exercises.....	33

1 **Pat Returns From A Visit**

2 It had been almost a week since Pat had last stopped by. I was just
3 beginning to wonder if Pat had decided that learning how to program a
4 computer was too much effort when I heard a knock at my door. I opened
5 the door and there stood Pat, wearing a big frown.

6 “Come in Pat.” I said in a cheerful voice. “What are you so 'happy' about?”

7 “My family spent all last week visiting friends and I wasn't able to work with
8 assembly language at all!” said Pat. “And when I told people that I was
9 learning how a computer worked, they didn't seem to understand what I
10 was talking about when I tried to explain it to them. I felt like I was alone in
11 a strange way during this trip, even though I was surrounded by people. Do
12 you kind of know what I mean, professor?”

13 I smiled and nodded my head. “Oh yes, I know what you mean, Pat. Most
14 people that study technical subjects have this problem and the more you
15 learn, the worse it gets.”

16 After a few moments of thought I added “You could always stop learning
17 about computers and perhaps devote your time to learning something more
18 common, like a sport. This should give you plenty to talk about the next
19 time you go visiting.”

20 “No way!” said Pat with determination. “I'm never going to give up learning
21 about computers! Ever since I started learning about computers, it feels
22 like I've entered a whole new world which is full of wonder and unexpected
23 possibilities. I haven't told you this before, but sometimes I get little pangs
24 of fear about losing the knowledge of computers I have gained, like I
25 learned it all in a dream and when I wake up, the knowledge will be gone. I
26 wouldn't trade my knowledge of computers for anything in the world, not
27 even for a cure for the loneliness I felt.”

28 “As a technologist, you will indeed need to live with a certain amount of
29 loneliness,” I said “but only in certain situations like your visit.
30 Fortunately, there are millions of people in the world who are deep
31 technologists. They are very easy to communicate with through the
32 Internet and they will understand exactly what you are talking about. Over
33 time, you will find that the relationships you form with people on the
34 Internet who have the same interests as you do will more than make up for
35 the loneliness you experience in other areas of your life.”

36 “Thanks, professor!” Pat said with a smile. “That lifted my spirits a bit.”

37 I smiled too as Pat's face brightened.

38 “Do you know what would make me feel even better, though?” asked Pat.

39 “What?” I asked.

40 “If you told me more about how programming worked.” replied Pat. “When
41 I watch you create a program, you make it look so easy and I seem to
42 understand most of what you are doing. By the time I get home, though, my
43 understanding gets fuzzy and I become confused. That's why I want to talk
44 more about programming and how to do it properly.

45 I understand how most of the instructions we have been using work, but I
46 don't quite get how to arrange the instructions into sequences that can
47 solve problems. It seems to me that a lot of programming exists at a level
48 that is above the instructions themselves, but I just can't see it yet.”

49 Algorithms

50 “You are correct about a significant part of programming existing at a level
51 that is above the level of the instructions, Pat.” I said. “A computer
52 programmer certainly needs to know at least one programming language,
53 but when a programmer solves a problem, they do it at a level that is higher
54 in abstraction than even the more abstract computer languages.

55 After the problem is solved, then the solution is encoded into a
56 programming language. It is almost as if a programmer is actually two
57 people. The first person is the **problem solver** and the second person is
58 the **coder**. The reason that programming may seem confusing to you is
59 that, when you have watched me program you have only seen the coding
60 phase and not the problem solving phase which I have been doing in my
61 head.

62 I have been doing the problem solving phase in my head because the
63 programs we have developed up to this point have been simple. With more
64 complex programs, however, the problem solving phase and the coding
65 phase are more distinct. The solution to a problem is usually recorded and
66 then passed from the **problem solver** to the **coder** in the form of a
67 document.”

68 “Are you saying that the problem solver can solve a programming problem
69 without even turning on the computer?” asked Pat.

70 “Yes.” I replied “The first thing that a problem solver will do with a problem
71 is to **analyze** it. This is an extremely important step because if a problem is
72 not analyzed, then it can not be properly solved.”

73 “What does analyze mean?” asked Pat.

74 “To **analyze** something means to break it down into its component parts
75 and then these parts are studied to determine how they work.” I replied. “A
76 well known saying is '**divide and conquer**' and when a difficult problem is
77 analyzed, it is broken down into smaller problems which are each simpler to
78 solve than the overall problem. The **problem solver** then develops an
79 **algorithm** to solve each of the simpler problems and, when these
80 algorithms are combined, they form the solution to the overall problem.”

81 “An algo-what?” asked Pat.

82 “An al-gor-rhythm.” I said. “An **algorithm** is a sequence of instructions
83 which describe how to accomplish a given task. These instructions can be
84 expressed in various ways including writing them in natural languages (like
85 English), drawing diagrams of them, and encoding them in a programming
86 language.

87 The concept of an algorithm came from the various procedures that
88 mathematicians developed for solving mathematical problems, like
89 calculating the sum of 2 numbers or calculating their product.

90 Algorithms can also be used to solve more general problems. An example
91 would be the set of instructions that could be followed by a person who
92 needs to solve the garage sizing problem we discussed earlier.” I then
93 wrote the following algorithm on the whiteboard:

94 1) Measure the length and width of each item that will be placed into the
95 garage using metric units and record these measurements.

96 2) Divide the measurements from step 1 by 100 then cut out pieces of paper
97 that match these dimensions to serve as models of the original items.

98 3) Cut out a piece of paper which is 1.5 times as long as the model of the
99 largest car and 3 times wider than it to serve as a model of the garage floor.

100 4) Locate where the garage doors will be placed on the model of the garage
101 floor, mark the locations with a pencil, and place the models of both cars on
102 top of the model of the garage floor, just within the perimeter of the paper
103 and between the two pencil marks.

104 5) Place the models of the items on top of the model of the garage floor in
105 the empty space that is not being occupied by the models of the cars.

106 6) Move the models of the items into various positions within this empty
107 space to determine how well all the items will fit within this size garage.

108 7) If the fit is acceptable, go to step 10.

109 8) If there is not enough room in the garage, increase the length dimension,
110 the width dimension (or both dimensions) of the garage floor model by 10%,
111 create a new garage floor model, and go to step 4.

112 9) If there is too much room in the garage, decrease the length dimension,
113 the width dimension (or both dimensions) of the garage model by 10%, create
114 a new garage floor model, and go to step 4.

115 10) Measure the length and width dimensions of the garage floor model,
116 multiply these dimensions by 100, and then build the garage using these
117 larger dimensions.

118 "That's a lot of steps!" said Pat.

119 "Yes," I said "an algorithm needs to be detailed enough so that it leads to
120 the desired solution if it is followed exactly. This often results in an
121 algorithm having a significant number of steps. After the steps have been
122 developed and recorded in a document, however, they can be followed over
123 and over again by people who need to solve the given problem."

124 "This algorithm for the garage sizing problem is written in English
125 sentences." said Pat. "Are algorithms usually written using sentences
126 before being encoded into programs, or can they be encoded into a
127 programming language right from the start?"

128 "An algorithm is usually not expressed in a programming language while it
129 is being developed and this is why a computer does not need to be turned
130 on during algorithm development. After the algorithm is complete and
131 recorded in a document, it is then encoded into a programming language as
132 a separate step."

133 "What happens if the program that the algorithm was encoded into doesn't

134 run correctly when it is executed?" asked Pat.

135 "Assuming that the **syntax** of the program is correct," I replied "the
136 problem will usually be 1) an incorrect algorithm that was encoded
137 correctly, 2) a correct algorithm that was encoded incorrectly, or 3) an
138 incorrect algorithm that was encoded incorrectly. The programmer needs
139 to study both the algorithm and the way it was encoded into a program in
140 order to find the cause of the problem.

141 When either syntax errors or algorithm errors are present in a program, the
142 program is said to have '**bugs**' and the process of locating the bugs and
143 fixing them is known as **debugging** the program."

144 "Bugs can't get inside of computer chips so why are program errors called
145 bugs!?" asked Pat.

146 I smiled and said "Bugs can not crawl inside of today's computer chips, but
147 the early computers used larger electrical components that bugs could
148 crawl into." I then found the following information on the Internet which
149 explained where the concept of a computer bug came from:

150 Date: 23 Aug 1981 05:38:25-PDT
151 From: ARPAVAX.sjk at Berkeley
152 Subject: origin of bug

153 Ever wondered about the origins of the term "bugs" as applied to computer
154 technology? U.S. Navy Capt. Grace Murray Hopper has firsthand explanation.
155 The 74-year-old captain, who is still on active duty, was a pioneer in
156 computer technology during World War II. At the C.W. Post Center of
157 Long Island University, Hopper told a group of Long Island public school
158 administrators that the first computer "bug" was a real bug -- a moth.
159 At Harvard one August night in 1945, Hopper and her associates were working
160 on the "granddaddy" of modern computers, the Mark I. "Things were going
161 badly; there was something wrong in one of the circuits of the long
162 glass-enclosed computer," she said. "Finally, someone located the
163 trouble spot and, using ordinary tweezers, removed the problem, a two-inch
164 moth. From then on, when anything went wrong with a computer, we said it
165 had bugs in it." Hopper said that when the veracity of her story was
166 questioned recently, "I referred them to my 1945 log book, now in the
167 collection of Naval Surface Weapons Center, and they found the remains of
168 that moth taped to the page in question."

9/9

0800 Antan started
 1000 stopped - antan ✓ { 1.2700 9.037 847 025
 1300 (032) MP-MC 1.98264000 9.037 846 995 correct
 (033) PRO 2 2.130476415
 correct 2.130676415
 Relays 6-2 in 033 failed special speed test
 in relay 11.000 test.
 Relays changed
 1100 Started Cosine Tapc (Sine check)
 1525 Started Mult+ Adder Test.
 1545 Relay #70 Panel F
 (moth) in relay.
 First actual case of bug being found.
 1630 Antan started.
 1700 closed down.

Relay 2745
 Relay 337

169 "That's hilarious!" said Pat.

170 Computation

171 After a few moments of thought Pat said "I understand how a person can
 172 perform the steps in an algorithm, Professor, but I am having a hard time
 173 understanding how a computer can perform these steps when its CPU is
 174 only capable of executing simple machine language instructions."

175 "In order to understand how a CPU is able to perform the steps in an
 176 algorithm, one must first understand what **computation** (which is also
 177 known as **calculation**) is." I said. "Lets search for some good definitions of
 178 each of these words on the Internet and read what they have to say." I then
 179 performed the search and selected the following 2 definitions for
 180 **computation:**

181 1) The manipulation of numbers or symbols according to fixed rules. Usually
 182 applied to the operations of an automatic electronic computer, but by
 183 extension to some processes performed by minds or brains.
 184 (www.informatics.susx.ac.uk/books/computers-and-thought/gloss/node1.html)

185 2) A computation can be seen as a purely physical phenomenon occurring inside
 186 a closed physical system called a computer. Examples of such physical systems
 187 include digital computers, quantum computers, DNA computers, molecular
 188 computers, analog computers or wetware computers.

189 (www.informatics.susx.ac.uk/books/computers-and-thought/gloss/node1.html)

190 I had Pat read the definitions for computation out loud then I said “The 2
191 definitions for **computation** I selected indicate that it is the '**manipulation**
192 **of numbers or symbols according to fixed rules**' and that it '**can be**
193 **seen as a purely physical phenomenon occurring inside a closed**
194 **physical system called a computer.**' Both definitions indicate that the
195 machines we normally think of as computers are just **one type of**
196 **computer** and that other types of closed physical systems can also act as
197 computers. These other types of computers include DNA computers,
198 molecular computers, analog computers, and wetware computers (or
199 brains).”

200 “Are you saying that brains manipulate symbols using rules, just like normal
201 computers do?” asked Pat.

202 “Yes.” I replied.

203 “How can brains manipulate symbols?” asked Pat. “Do they implement the
204 Von Neumann architecture?”

205 “No,” I replied “brains do not implement the Von Neumann architecture,
206 but they do manipulate symbols using rules just like normal computers do.
207 Read aloud the 2 definitions for **calculation** I selected and lets see if these
208 definitions shed any light on the kind of rules that normal computers,
209 brains, and other types of computers use.” Pat then read the following
210 definitions:

211 1) A calculation is a deliberate process for transforming one or more inputs
212 into one or more results. (en.wikipedia.org/wiki/Calculation)

213 2) Calculation: the procedure of calculating; determining something by
214 mathematical or logical methods (wordnet.princeton.edu/perl/webwn)

215 “These definitions for calculation,” I said “indicate that it '**is a deliberate**
216 **process for transforming one or more inputs into one or more**
217 **results**' and that this is done '**by mathematical or logical methods**'. We
218 do not yet completely understand what mathematical and logical methods
219 brains use to perform calculations, but rapid progress is being made in this
220 area.”

221 “What is logic?” asked Pat.

222 “The concept of logic is used in a number of different contexts. I will find a
223 definition for **logic** on the Internet which fits our context.” I then located
224 the following definition and had Pat read it aloud:

225 The logic of a system is the whole structure of rules that must be used for
226 any reasoning within that system. Most of mathematics is based upon a well-
227 understood structure of rules and is considered to be highly logical. It is
228 always necessary to state, or otherwise have it understood, what rules are
229 being used before any logic can be applied. ([ddi.cs.uni-](http://ddi.cs.uni-potsdam.de/Lehre/TuringLectures/MathNotions.htm)
230 [potsdam.de/Lehre/TuringLectures/MathNotions.htm](http://ddi.cs.uni-potsdam.de/Lehre/TuringLectures/MathNotions.htm))

231 “**Reasoning**,” I said “is the process of using predefined rules to move from
232 one point in a system to another point in the system. For example, when a
233 person adds 2 numbers together on a piece of paper, they must follow the
234 rules of the addition algorithm in order to obtain a correct sum. The
235 addition algorithm's rules are its logic and, when someone applies these
236 rules during a calculation, they are **reasoning** with the rules.

237 Lets now apply these concepts to your question about how a computer can
238 perform the steps of an algorithm when its CPU is only capable of executing
239 simple machine language instructions. When a person develops an
240 algorithm, the steps in the algorithm are usually stated as high-level tasks
241 which do not contain all of the smaller steps that are necessary to perform
242 each task.

243 For example, a person might write a step that states 'Drive from New York
244 to San Francisco.' This large step can be broken down into smaller steps
245 that contain instructions such as 'turn left at the intersection, go west for 10
246 kilometers, etc.' If all of the smaller steps in a larger step are completed,
247 then the larger step is completed too.

248 A human that needs to perform this large driving step would usually be able
249 to figure out what smaller steps need to be performed in order accomplish
250 it. As we learned earlier, however, computers are very stupid and before
251 any algorithm can be executed on a computer, the algorithm's steps must be
252 broken down into smaller steps, and these smaller steps must be broken
253 down into even small steps, until the steps are simple enough to be
254 performed by the instruction set of a CPU.”

255 “How many smaller steps does a larger step usually need to be broken down
256 into before it can be implemented in machine language?” asked Pat.

257 “Sometimes only a few smaller steps are needed to implement a larger step,
258 but sometimes hundreds or even thousands of smaller steps are required.
259 Hundreds or thousands of smaller steps will translate into hundreds or
260 thousands of machine language instructions.”

261 “That's a lot of steps and instructions!” said Pat. “It almost seems like its
262 too much work to convert an algorithm into thousands of little steps and
263 then into maybe thousands of assembly language instructions needed to run
264 it on a computer.”

265 “You are right, Pat.” I said “If assembly language was the only language
266 that computers could be programmed in, then most algorithms would be too
267 large to be placed into a computer. It was for the purpose of solving this
268 problem that high-level languages were created.

269 With assembly language, a human needs to manually convert an algorithm
270 into assembly language instructions before a computer can execute the
271 algorithm. An algorithm that is encoded into a high-level language,
272 however, does not need to be broken down into as many smaller steps as
273 would be needed with assembly language. The hard work of breaking down
274 an algorithm that has been encoded into a high-level language is
275 automatically done by either a **compiler** or an **interpreter**. This is why
276 most of the time, programmers use a high-level language to develop in
277 instead of assembly language.”

278 “If high-level languages are so much easier to encode algorithms into than
279 assembly language,” said Pat “why are we studying assembly language?”

280 I smiled, looked at Pat, and said “For many years, I taught beginning
281 programming to students using the normal method of starting them with a
282 high-level language without covering assembly language at all. A beginning
283 programmer is definitely able to start writing programs quicker in a high-
284 level language than they could with assembly language and I (along with
285 many others) thought that quicker in this case meant better.

286 Over time, however, I noticed that most of these students were not able to
287 program very well. I eventually figured out that the main reason that most
288 of them could not program very well was that they did not understand how
289 a computer actually worked because much of this understanding is at the
290 assembly language level.

291 Even though most programmers do not program in assembly language,

292 somehow, having a solid understanding of how a computer works at its
293 lowest levels, and having had the experience of manually encoding
294 algorithms into assembly language, wires a programmer's brain in a way
295 that enables them to be much better high-level language programmers than
296 they otherwise would be.

297 I love assembly language, but even I prefer to program in a high-level
298 language whenever possible. The reason I am teaching assembly language
299 to you as your first computer language is because I want to give you the
300 foundation you will need to become an excellent high-level language
301 programmer."

302 UML Activity Diagrams

303 Pat said "I think I'm beginning to understand now how a programmer
304 becomes a **problem solver** when they develop an algorithm and then
305 becomes a **coder** when they encode the algorithm into a programming
306 language. I would like to talk some more about how the problem solver
307 develops an algorithm, though.

308 Earlier you said that an algorithm can be recorded in a natural language
309 like English but it can also be recorded using a diagram. Can you show me
310 how recording an algorithm into a diagram would work?"

311 "Yes." I replied. "You may be surprised to learn, however, that a whole
312 diagram-based language has been created which allows all aspects of a
313 program to be designed by 'problem solvers', including the algorithms that
314 a program uses. This language is call **UML** which stands for **Unified**
315 **Modeling Language**. Later on, when we begin studying high-level
316 languages, we will cover UML in more depth. For now, though, I will show
317 you how one of UML's diagrams, called an **Activity diagram**, works.

318 The purpose of an **Activity diagram** is to show the sequence of steps (or
319 activities) that are part of some piece of logic and this makes them well
320 suited for recording algorithms. I will now show you how activity diagrams
321 work by using them to develop algorithms which solve various problems. I
322 will also encode each diagram into a 6502 assembly language program so
323 you can see how the encoding process is done."

324 Calculating The Sum Of The Numbers Between 1 And 10

325 Before I began solving the first problem, I took some sheets of paper from
326 the printer and created 2 funny-looking hats with them using a scissors and

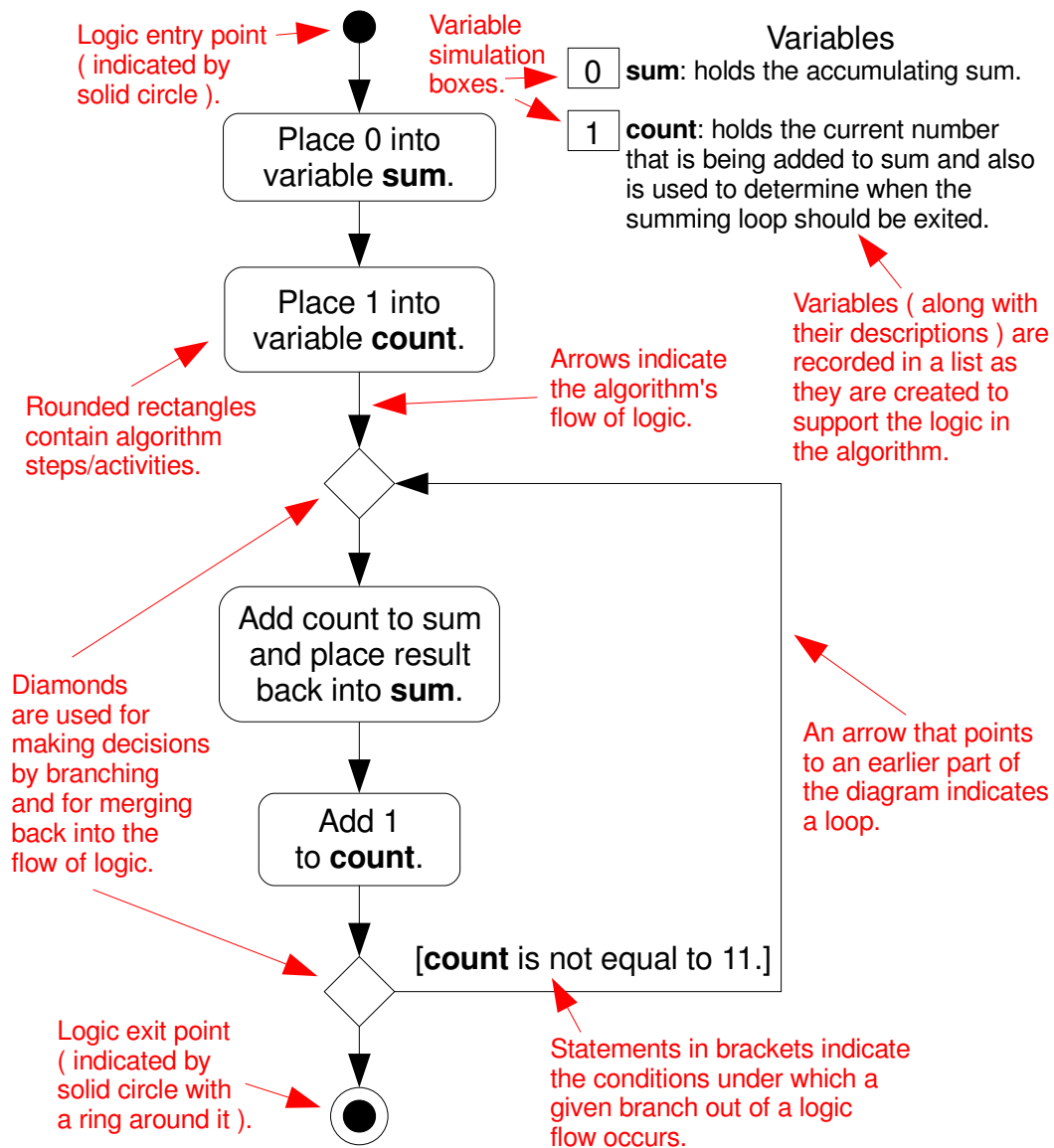
327 tape. I wrote 'Problem Solver' on the first hat and 'Coder' on the second hat
328 and Pat started laughing when I put on the 'Problem Solver' hat.

329 “The first thing that needs to be done with a problem before it can be
330 analyzed and solved,” I said “is to describe it clearly and accurately.” I then
331 wrote the following problem description on the whiteboard:

332 **Description:** In this problem, the sum of the numbers between 1 and 10
333 inclusive needs to be determined.

334 “Inclusive here means that the numbers 1 and 10 will be included in the
335 sum.” I said. “Since this is a fairly simple problem that is similar to one I
336 solved earlier with a program, we will not need to spend too much time
337 analyzing it.” I then developed an algorithm for solving the problem by
338 drawing an Activity diagram on the whiteboard. (see Fig. 1)

Figure 1



339 As I was developing the Activity diagram, I created variables as needed and
 340 recorded their names in a list along with their descriptions. I would also
 341 periodically start at the entry point and walk through the logic to make sure
 342 it was correct (I did this out loud so Pat could follow along too). I placed
 343 simulation boxes next to each variable so that I could record and update
 344 how the logic was changing the variable's values. During a walk-through, I
 345 would often find errors which I would then fix by moving flow arrows and
 346 adjusting the text that was inside the activity rectangles.

347 Eventually, I reached a point where I could not find any more errors in the

348 logic and then I said to Pat “It looks like the algorithm is complete now and
 349 I can stop being the **problem solver** and pass the algorithm over to the
 350 **coder** so it can be encoded into assembly language.” I then took off the
 351 'Problem Solver' hat and put on the 'Coder' hat and Pat started laughing
 352 again. I brought up MathRider and encoded the Activity diagram into the
 353 following 6502 assembly language program:

```

354          000001 |;Program Name: sum10.
355          000002 |;
356          000003 |;Version: 1.03
357          000004 |;
358          000005 |;Description: In this program, the sum of the
359          000006 |; numbers between 1 and 10 inclusive will be
360          000007 |; determined.
361          000008 |;
362          000009 |;*****
363          000010 |;Program entry point.
364 0200      000011 |      org 0200h
365          000012 |;
366          000013 |;Place 0 into variable sum.
367 0200 A9 00    000014 |      lda #0d
368 0202 8D 1F 02 000015 |      sta sum
369          000016 |;
370          000017 |;Place 1 into variable count.
371 0205 A9 01    000018 |      lda #1d
372 0207 8D 20 02 000019 |      sta count
373          000020 |;
374          000021 |;Top of summing loop.
375 020A      000022 |LoopTop *
376          000023 |;
377          000024 |;Add count to sum and place result back into sum.
378 020A AD 1F 02 000025 |      lda sum
379 020D 18      000026 |      clc
380 020E 6D 20 02 000027 |      adc count
381 0211 8D 1F 02 000028 |      sta sum
382          000029 |;
383          000030 |;Add 1 to count.
384 0214 EE 20 02 000031 |      inc count
385          000032 |;
386          000033 |;If count is not equal to 11 yet then branch back
387          000034 |; to LoopTop.
388 0217 AD 20 02 000035 |      lda count
389 021A C9 0B    000036 |      cmp #11d
390 021C D0 EC    000037 |      bne looptop
391          000038 |;
392          000039 |;Exit program.
393 021E 00      000040 |      brk
394          000041 |;
395          000042 |;*****

```

```

396          000043 |;Variables area.
397          000044 |
398          000045 |;Holds the accumulating sum.
399 021F 00    000046 |sum    dbt 0d
400          000047 |
401          000048 |;Holds the current number
402          000049 |; that is being added to sum and also
403          000050 |; is used to determine when the summing
404          000051 |; loop should be exited.
405 0220 00    000052 |count      dbt 0d
406          000053 |
407          000054 |      end

```

408 It did not take me very long to develop the program and when I was finished
 409 I took off the Coder hat.

410 Pat said "That was fast. In fact, it took you much longer to develop the
 411 algorithm than it did to code the program. Is that normal?"

412 "Yes," I replied "most of the work that is involved in developing a program
 413 happens in the **problem solving** stage. After the program's logic has been
 414 developed and recorded in a document like an Activity diagram, coding this
 415 logic is usually a straight-forward task.

416 I now want you to study both the Activity diagram and the program and
 417 then tell me if you notice anything interesting about the comments in the
 418 program.

419 Pat studied both the diagram and the program for a while then said "Hey,
 420 the comments in the program are taken directly from the diagram!"

421 "Indeed they are." I said. "You wanted to see how the steps in an algorithm
 422 were encoded into assembly language and you can see in this example that
 423 the encoding process consists of taking each step in the algorithm and
 424 creating a group of assembly language instructions that perform that step.
 425 All the assembly language instructions in a given group are the small steps
 426 that need to be performed in order to perform the larger step they are a
 427 part of.

428 The task of translating a step in the algorithm to a group of assembly
 429 language instructions that represent that step in a program is so direct that
 430 the text that describes the step in the algorithm can sometimes be used
 431 unchanged to describe what its analogous group of instructions does."

432 **Finding The Largest Number In A List Of 100 Numbers**

433 “Here is the next problem we are going to solve,” I said. I then wrote this
434 problem description on the whiteboard:

435 Description: Find the largest number in a list of 100 numbers.

436 “This problem is more sophisticated than the previous problem and so we
437 will need to spend more time analyzing it.” I said. “One of the strategies for
438 solving any problem is to **try to solve a simpler problem that is similar**
439 **to the original problem**. In this case, I propose that we try to solve the
440 problem for a list of 5 numbers first, then use the knowledge we gained to
441 solve the larger problem.”

442 “Okay.” Said Pat.

443 “What should the **range** be for the numbers in the list?” I asked.

444 “Range?” asked Pat.

445 “**Range** in this case means the numbers in the list can be no lower than a
446 certain number and no higher than some other number. Therefore, you
447 have a range of numbers between a lower limit number and an upper limit
448 number to choose from.” I said.

449 “Will the lower limit number and the upper limit number both be inclusive?”
450 asked Pat?

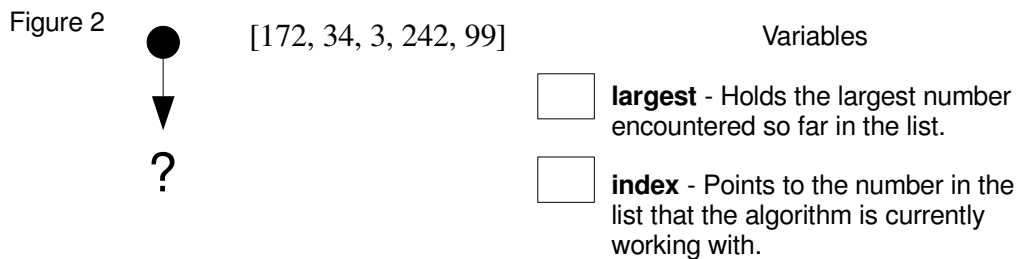
451 “Yes they will.” I replied. “Also, when a person is developing an algorithm
452 that will be encoded into a programming language, they must **always be**
453 **mindful of how many bits will be needed to represent the numbers in**
454 **the algorithm**. In this case, I recommend that we use **bytes** to represent
455 the numbers in the list and that the numbers be non-negative. With these
456 assumptions, what is the range of the numbers that can be in the list?”

457 “0 to 255.” said Pat.

458 “Very good.” I said. “Whenever assumptions are applied to an algorithm,
459 these should be recorded near the problem's description.” I then added the
460 following assumption section to the whiteboard:

461 Assumptions: The numbers in the list will be no lower than 0 and no higher
462 than 255.

463 “Now, give me a list of 5 numbers that meet these requirements.” I said.
464 Pat thought for a moment then said “172, 34, 3, 242, 99.”
465 I wrote these numbers on the whiteboard.
466 “What variables do you think our algorithm is going to need?” I asked.
467 Pat said “I think we are going to need a variable to hold the **largest**
468 number. Hmmm, I am not sure what other variables we will need, though.”
469 “How should we keep track of where we are in the list?” I asked.
470 “Oh yes, we will also need an **indexing** variable.” said Pat.
471 “Lets start with these variables, then, and begin developing the program's
472 logic.” I said. Then I put the information on the whiteboard (see Fig. 2)



473 “What should the algorithm do first, Pat?” I asked.
474 Pat studied the information I had placed on the whiteboard then said “I
475 think the first thing the algorithm should do is to set the variables to their
476 initial values.”
477 “I agree.” I said. “Setting variables to their initial values is referred to as
478 **initializing** the variables and they can be initialized either by algorithm
479 steps or by just placing the initial values into the variable's simulation
480 boxes. If the initial values are placed into the simulation boxes, these
481 variables will be initialized when the variables are defined in the program.
482 For example, in the 6502 assembly language we have been using, **largest**
483 **dbt 5d** would initialize the variable **largest** to 5 decimal. Since we
484 initialized the variables using logic in the last problem, we will initialize
485 them when they are defined in this problem.”

486 "Okay." said Pat.

487 "What should we initialize these variables to?" I asked.

488 "The variable **index** should be initialized to 0 but I'm not sure what **largest**
489 should be initialized to." said Pat after thinking about the question for some
490 time.

491 "If you are not sure, just come up with a number for now and we will
492 change it if we find a better number later.

493 "Set it to 75." said Pat. I then set **largest** to 75 and **index** to 0.

494 "Now pretend that you are a computer and describe how you would
495 determine the largest number in this list." I said. "Keep in mind that the
496 computer is not able to 'see' the whole list at one time like a human can.
497 The computer is only able to see one number in the list at any given time so
498 which number would you start with?"

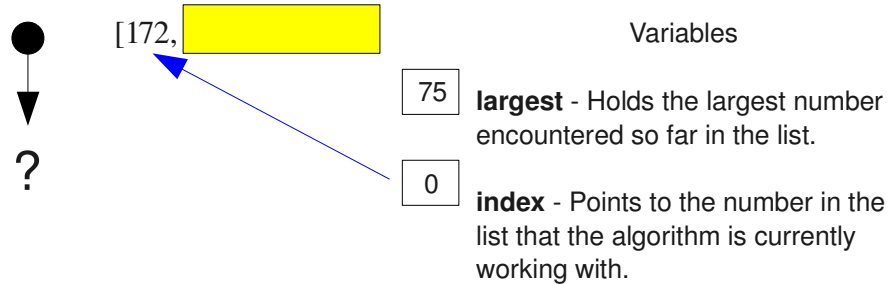
499 "I would probably start with the first number in the list." answered Pat.

500 "Which number is the first number in the list and what offset is it at?" I
501 asked.

502 "172 is the first number and it is at offset 0." replied Pat.

503 I then covered the other numbers in the list with a yellow piece of paper
504 and drew an arrow from **index's** simulation box to the number 172. (see
505 Fig. 3)

Figure 3

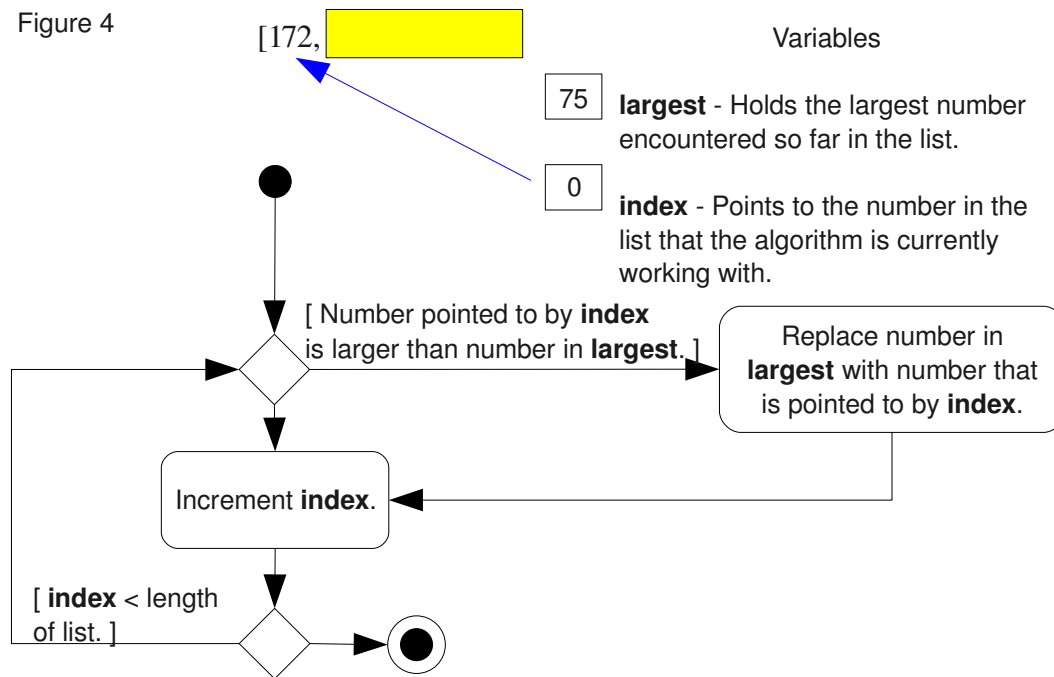


506 Then I said "What logic do you think should be performed in the algorithm,
507 Pat?"

508 "How about something like '**If** the number referenced by **index** is greater
509 than the number that is in **largest**, **then** replace the number in **largest**
510 with the number that is referenced by **index**. Increment **index** to point to
511 **each number in the list in turn** so that the logic can determine if it is the
512 largest so far.'" said Pat.

513 "That sounds pretty good!" I said. "Now lets place this logic into the
514 Activity diagram. Whenever you encounter '**if**' and '**then**' inside sentences
515 that describe logic, these words indicate that a **decision** is being made. In
516 this case, we will place a **diamond** immediately after the **entry point**
517 **symbol** in the Activity diagram and then attach an **activity rectangle** that
518 contains the 'number compare and replace logic' to it. The words '**in turn**'
519 indicate that a loop will be needed in the algorithm." I then added this logic
520 to the Activity diagram. (see Fig. 4)

Figure 4



521 When the Activity diagram was finished I said “Our next step is to pretend
522 we are a computer and perform a **dry run** of the logic in the algorithm.”

523 “A dry run?” asked Pat “What does that mean?”

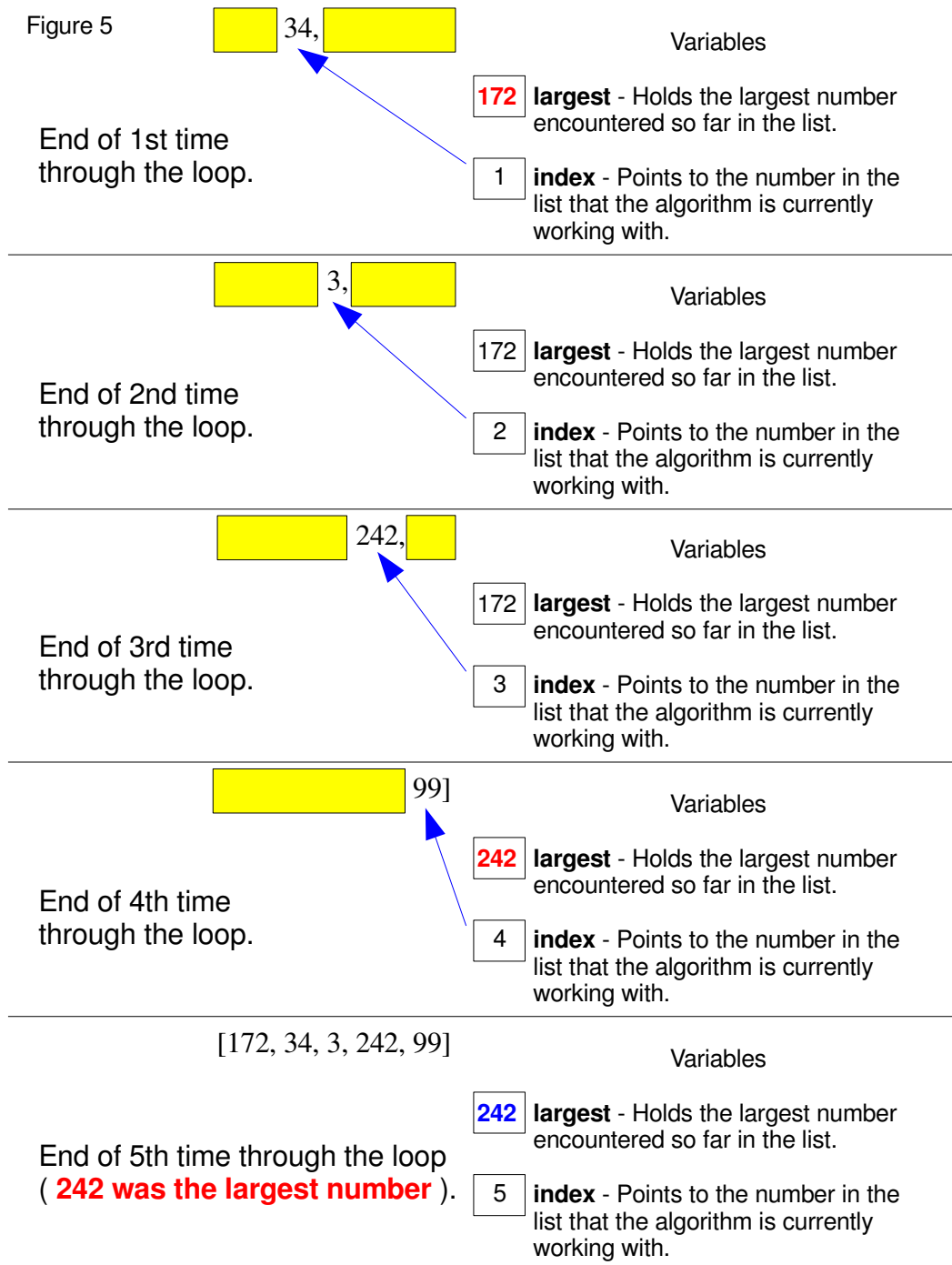
524 “In the context of computer programming,” I replied “it means to mentally
525 execute a program, or a program's algorithms, without using a computer. A
526 programmer should always dry run their algorithms and programs to make
527 sure they are correct.”

528 “Why are dry runs 'dry'?” asked Pat.

529 I laughed and said “My understanding is that the term **dry run** comes from
530 the firefighter service. When a firetruck is sent out on a rehearsal or test
531 run, water is not usually used and therefore the run is dry. The term dry
532 run is often used to mean any kind of a rehearsal or test that is performed
533 before doing it for real.”

534 Pat and I then did a dry run of the algorithm using the list of numbers that
535 Pat came up with which were 172, 34, 3, 242, and 99. (see Fig. 5)

Figure 5



536 “It worked!” cried Pat. “The algorithm figured out that the largest number
537 in the list is 242!”

538 “Yes it did, but we can not celebrate just yet.” I said.

539 “Why not?” asked Pat.

540 “Dry run the algorithm again, but this time use this list of numbers.”

541 I wrote the following list on the whiteboard:

542 {2, 9, 3, 5, 7}

543 Pat performed another dry run of the algorithm with the second list then
544 said “Hey, it didn't work this time! The algorithm says that the largest
545 number in the list is 75 even though the largest number is actually 9.”

546 “What do you think the problem is?” I asked.

547 Pat studied the algorithm for a while then said “Oh, I see the problem. I
548 had told you to initialize the variable **largest** to 75, but 75 is larger than all
549 of the numbers in the second list. Since none of the numbers in the list is
550 larger than 75, the 75 never gets replaced.”

551 “Is there a number we can initialize **largest** to so that the algorithm will
552 work correctly with any list that conforms to our stated assumptions?” I
553 asked.

554 Pat thought about this question for a while then said “I think we will need to
555 initialize **largest** with the lowest number in the range of numbers that are
556 allowed to be in the list. Since 0 is the lowest number in the range we
557 selected, this is what we should initialize largest to.”

558 “Very Good!” I said. “These are the type of errors that performing dry runs
559 of algorithms and programs are meant to find. I think our algorithm is
560 correct now and it should also be able to handle the 100 number list from
561 the original problem. I will now encode the algorithm into a program.” I
562 brought up an editor and encoded the logic that was in the Activity diagram
563 into the following program:

```
564          000001 |;Program Name: largest.  
565          000002 |;
```

```

566      000003 |;Version: 1.01.
567      000004 |;
568      000005 |;Description: Find the largest number in a list
569      000006 |; of 100 numbers.
570      000007 |;
571      000008 |;Assumptions: The numbers in the list will be no
572      000009 |; lower than 0 and no higher than 255.
573      000010 |
574      000011 |
575      000012 |;*****
576      000013 |;Program entry point.
577 0200      000014 |      org 0200h
578      000015 |
579      000016 |;This program will use the X register to point
580      000017 |; to the current number in the list
581      000018 |; instead of a variable called 'index'.
582      000019 |;
583      000020 |;Initialize register X to refer to offset 0 into
584      000021 |; the list of numbers starting at 'list'.
585 0200 A2 00      000022 |      ldx #0d
586      000023 |
587      000024 |;Compare the number that X is referring to in the
588      000025 |; list to the number that is in 'largest'. If the
589      000026 |; number being pointed to is larger than the contents
590      000027 |; of the variable 'largest', then fall through to
591      000028 |; the code that will replace the contents of 'largest'
592      000029 |; with the new number.
593 0202      000030 |LoopTop *
594 0202 BD 14 02      000031 |      lda list,x
595 0205 CD 13 02      000032 |      cmp largest
596 0208 90 03      000033 |      bcc NotLarger
597      000034 |
598 020A      000035 |IsLarger *
599      000036 |
600      000037 |;Replace the number in 'largest' with the current
601      000038 |; number from the list.
602 020A 8D 13 02      000039 |      sta largest
603      000040 |
604 020D      000041 |NotLarger *
605      000042 |
606      000043 |;Increment the X register so that it references the
607      000044 |; next number in the list.
608 020D E8      000045 |      inx
609      000046 |
610      000047 |;Check to see if the end of the list has been reached.
611      000048 |; If it has, then exit the program. If it has not,
612      000049 |; then branch back to the top of the loop.
613 020E      000050 |CheckListEnd *
614 020E E0 64      000051 |      cpx #100d
615 0210 D0 F0      000052 |      bne LoopTop

```



```

616      000053 |
617      000054 |;Exit the program.
618  0212 00    000055 |      brk
619      000056 |
620      000057 |
621      000058 |;*****
622      000059 |;Variables area.
623      000060 |;
624      000061 |;In this program, the variables area is being placed
625      000062 |; below the code because the variables take up
626      000063 |; a significant amount of memory. This makes
627      000064 |; it more difficult to determine where the variables
628      000065 |; area ends so that the start of the code area
629      000066 |; can be determined.
630      000067 |
631      000068 |
632      000069 |;Holds the largest number
633      000070 |; encountered so far in the list.
634      000071 |; ( Notice that variables can
635      000072 |; be initialized here instead of
636      000073 |; in the code as an option ).
637  0213 00    000074 |largest dbt 0d
638      000075 |
639      000076 |
640      000077 |;This list contains 100 numbers ranging from 0
641      000078 |; to 255.
642  0214 3B    000079 |list dbt 59d,61d,37d,128d,71d,150d,195d,130d,69d,84d
643  0215 3D
644  0216 25
645  0217 80
646  0218 47
647  0219 96
648  021A C3
649  021B 82
650  021C 45
651  021D 54
652  021E AB    000080 |      dbt 171d,227d,99d,214d,233d,136d,80d,253d,242d
653  021F E3
654  0220 63
655  0221 D6
656  0222 E9
657  0223 88
658  0224 50
659  0225 FD
660  0226 F2
661  0227 70    000081 |      dbt 112d,221d,151d,101d,117d,76d,226d,174d,205d
662  0228 DD
663  0229 97
664  022A 65
665  022B 75

```

666	022C	4C		
667	022D	E2		
668	022E	AE		
669	022F	CD		
670	0230	54	000082	dbt 84d,78d,139d,89d,195d,243d,69d,128d,217d,215d
671	0231	4E		
672	0232	8B		
673	0233	59		
674	0234	C3		
675	0235	F3		
676	0236	45		
677	0237	80		
678	0238	D9		
679	0239	D7		
680	023A	39	000083	dbt 57d,100d,227d,226d,233d,238d,229d,228d,135d
681	023B	64		
682	023C	E3		
683	023D	E2		
684	023E	E9		
685	023F	EE		
686	0240	E5		
687	0241	E4		
688	0242	87		
689	0243	8C	000084	dbt 140d,98d,211d,245d,120d,206d,198d,47d,191d
690	0244	62		
691	0245	D3		
692	0246	F5		
693	0247	78		
694	0248	CE		
695	0249	C6		
696	024A	2F		
697	024B	BF		
698	024C	EF	000085	dbt 239d,27d,236d,12d,242d,148d,98d,11d,38d,189d
699	024D	1B		
700	024E	EC		
701	024F	0C		
702	0250	F2		
703	0251	94		
704	0252	62		
705	0253	0B		
706	0254	26		
707	0255	BD		
708	0256	EE	000086	dbt 238d,225d,142d,214d,214d,21d,75d,17d,190d
709	0257	E1		
710	0258	8E		
711	0259	D6		
712	025A	D6		
713	025B	15		
714	025C	4B		
715	025D	11		

```
716 025E BE
717 025F B2      000087 |      dbt 178d,123d,125d,123d,10d,166d,123d,135d,220d
718 0260 7B
719 0261 7D
720 0262 7B
721 0263 0A
722 0264 A6
723 0265 7B
724 0266 87
725 0267 DC
726 0268 C1      000088 |      dbt 193d,46d,248d,222d,63d,206d,197d,101d,144d
727 0269 2E
728 026A F8
729 026B DE
730 026C 3F
731 026D CE
732 026E C5
733 026F 65
734 0270 90
735 0271 C9      000089 |      dbt 201d,233d,12d,241d,85d,180d,29d
736 0272 E9
737 0273 0C
738 0274 F1
739 0275 55
740 0276 B4
741 0277 1D
742      000090 |
743      000091 |      end
```

744 **Exercises**

745 These exercises use the programs that are included below them.

746 a) Load the copy, scan, alpha, and sumlist programs into the emulator and
747 **verify that they work correctly. You may need to trace them to**
748 **understand how they work.** Type at least one of these in by hand instead
749 of using copy and paste.

750 b) Create activity diagrams for copy, scan, alpha, and sumlist. Use any
751 drawing program that will work (<http://openoffice.org> is free and it's
752 drawing application works well for drawing diagrams.)

753 **Copy The Contents Of list1 To list2**

```
754 %uasm65

755 ;Program Name: copy.
756 ;
757 ;Version: 1.01.
758 ;
759 ;Description: Copy the contents of list1 to list2.
760 ;
761 ;Assumptions: The numbers in the list will be no
762 ; lower than 0 and no higher than 255.

763 ;*****
764 ;Program entry point.
765     org 0200h

766 ;The X register will be used to point to each number in both list1 and list2.
767     ldx #0d

768 ;Copy the 10 bytes from list1 to list2.
769 LoopTop *
770     lda list1,x
771     sta list2,x

772     inx

773     cpx #10d

774     bne LoopTop

775 ;Exit the program.
```

```
776      brk

777      ;*****
778      ;Variables area.

779      ;This list contains 10 bytes.
780      list1 dbt 41h,42h,43h,44h,45h,46h,47h,48h,49h,4ah

781      ;This byte is placed here to make it easier to see the contents of list1
782      ;and list2 when they are dumped in the monitor.
783      dbt 0d

784      ;Set aside 10 memory locations and initialize each one to 0d.
785      list2 dbt 10d(0d)

786      end

787      %/uasm65

788      Scan string1 And Determine How Many Capital And Lower Case A's
789      It Contains

790      %uasm65

791      ;Program Name: scan.
792      ;
793      ;Version: 1.01.
794      ;
795      ;Description: Scan list1 and determine how many capital A's and lower case
796      ; a's it contains.
797      ;
798      ;Assumptions: The numbers in the list will be no
799      ; lower than 0 and no higher than 255.

800      ;*****
801      ;Program entry point.
802      org 0200h

803      ;The X register will be used to point to each ASCII character in string1
804      ldx #0d

805      LoopTop *
806      lda string1,x

807      ;If we have reached the 0 that has been placed at the end of the string, then
808      ; exit the program.
```

```
809      cmp #0d
810      beq EndOfList

811  CheckLowerCase  *
812  ;If the current character is a lower case 'a', then increment LowerCaseCount.
813      cmp #'a'
814      bne CheckUpperCase
815      inc LowerCaseCount
816      jmp NextCharacter

817  CheckUpperCase  *
818  ;If the current character is an upper case 'a', then increment UpperCaseCount.
819      cmp #'A'
820      bne NextCharacter
821      inc UpperCaseCount

822  NextCharacter  *
823      inx
824      jmp LoopTop

825  EndOfList  *

826  ;Exit the program.
827      brk

828  ;*****
829  ;Variables area.

830  LowerCaseCount dbt 0d
831  UpperCaseCount dbt 0d

832  ;This list contains a string of ASCII characters.
833  string1 dbt "A bird in the hand is worth two in the bush.  Early to bed and "
834          dbt "early to rise makes a person healthy, wealthy, and wise."
835          dbt 0d

836      end

837  %/uasm65

838  Place Capital A's In The First 10 Bytes Of alphaList, Capital B's In
839  The Second 10 Bytes Of alphaList, And So On Until alphaList Is
840  Filled With Letters

841  %uasm65
```

```
842 ;Program Name: alpha.uasm.
843 ;
844 ;Version: 1.02.
845 ;
846 ;Description: Place capital A's in the first 10 bytes of alphaList, capital
847 B's
848 ; in the second 10 bytes of alphalist and so on until alphaList is filled
849 ; with letters.
850 ;
851 ;Assumptions: The numbers in the list will be no
852 ; lower than 0 and no higher than 255.

853 ;*****
854 ;Program entry point.
855     org 0200h

856 ;The X register will be used as a pointer.
857     ldx #0d

858     lda #10d
859     sta rowCount
860     sta columnCount

861 ;Initialize the 'A' register to be a capital 'A'.
862     lda #65d

863 LoopTop *
864 ;Place character in list at offset X.
865     sta alphalist,x

866 ;Point X to the next character position in the list.
867     inx

868 ;If we have not placed 10 characters in this row yet, then loop.
869     dec columnCount
870     bne LoopTop

871 ;Reset the column counter.
872     ldy #10d
873     sty columnCount

874 ;Increase to the next letter in the alphabet.
875     clc
876     adc #1d

877 ;If we have not filled 10 rows yet then loop.
878     dec rowCount
879     bne LoopTop
```

```
880 EndOfList  *
881 ;Exit the program.
882     brk

883 ;*****
884 ;Variables area.

885 rowCount    dbt 0d
886 columnCount dbt 0d
887 alphalist    dbt 100d(?)

888     end

889 %/uasm65

890 Calculate The Sum Of The 100 Bytes That Are In numbersList
891 %uasm65

892 ;Program Name: sumlist.uasm.
893 ;
894 ;Version: 1.01.
895 ;
896 ;Description: Calculate the sum of the numbers that are in numberlist.
897 ;
898 ;Assumptions: The numbers in the list will be no
899 ; lower than 0 and no higher than 255.

900 ;*****
901 ;Program entry point.
902     org 0200h

903 ;The X register will be used as a pointer.
904     ldx #0d

905 ;Initialize the 16 bit wide variable "sum" to 0.  Note: 8 bits is called a
906 ; "byte" and 16 bits is called a "word".  sum+0d accesses the high byte of sum
907 ; and sum+1d accesses the low byte of sum.
908     lda #0d
909     sta sum+0d
910     sta sum+1d

911 LoopTop  *
912 ;Obtain the next number from the list.
913     lda numberList,x
```



```
914 ;Add the number to the low byte of sum and then place the result back into
915 ; the low byte of sum.
916     clc
917     adc sum+1d
918     sta sum+1d

919 ;If the carry flag was set during the addition, this means that the addition
920 ; resulted in a value that was greater than 255 and the high byte of sum needs
921 ; to be incremented by 1. If the result was 255 or less, the carry would not
922 ; be set and we can branch around the increment of the high byte of sum.
923     bcc NoCarry
924 Carry *
925     inc sum+0d

926 NoCarry *
927 ;Point to the next number in the list and loop back if we have not reached
928 ; the end of the list.
929     inx
930     cpx #100d
931     bne LoopTop

932 ;Exit the program.
933     brk

934 ;*****
935 ;Variables area.

936 ;DWD stands for Define Word and it creates a variable that is 16 bits wide.
937 ; Sum needs to be 16 bits wide because the sum of the numbers in numberList
938 ; will be greater than 255 (which is the largest number that can be held in
939 ; an 8 bit byte).
940 sum    dwd ?

941 ;This list contains 100 numbers ranging from 0 to 255.
942 numberList *
943     dbt 59d,61d,37d,128d,71d,150d,195d,130d,69d,84d
944     dbt 171d,227d,99d,214d,233d,136d,80d,253d,242d
945     dbt 112d,221d,151d,101d,117d,76d,226d,174d,205d
946     dbt 84d,78d,139d,89d,195d,243d,69d,128d,217d,215d
947     dbt 57d,100d,227d,226d,233d,238d,229d,228d,135d
948     dbt 140d,98d,211d,245d,120d,206d,198d,47d,191d
949     dbt 239d,27d,236d,12d,242d,148d,98d,11d,38d,189d
950     dbt 238d,225d,142d,214d,214d,21d,75d,17d,190d
951     dbt 178d,123d,125d,123d,10d,166d,123d,135d,220d
952     dbt 193d,46d,248d,222d,63d,206d,197d,101d,144d
953     dbt 201d,233d,12d,241d,85d,180d,29d

954     end
```

955 %/uasm65