6502 Programming

by Ted Kosan

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

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

Pat Returns From A Visit

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

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

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

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

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

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

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

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

I smiled too as Pat's face brightened.

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

“What?” I asked.

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

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

Algorithms

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

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

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

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

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

“What does analyze mean?” asked Pat.

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

“An algo-what?” asked Pat.

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

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

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

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

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

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

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

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

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

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

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

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

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

“That's a lot of steps!” said Pat.

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

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

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

“What happens if the program that the algorithm was encoded into doesn't run correctly when it is executed?” asked Pat.

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

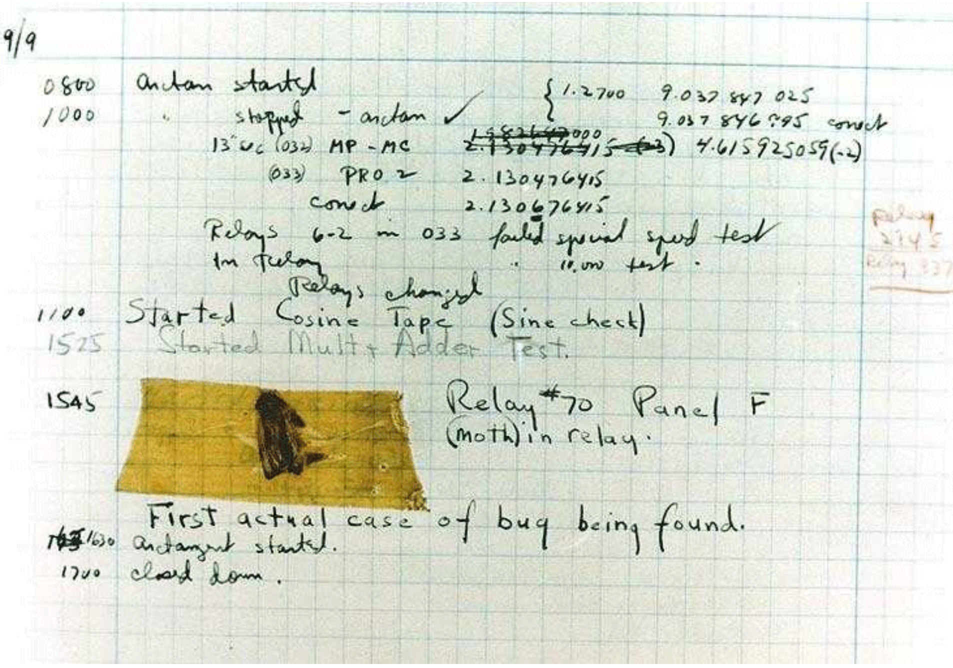
When either syntax errors or algorithm errors are present in a program, the program is said to have '**bugs**' and the process of locating the bugs and fixing them is known as **debugging** the program.”

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

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

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

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



“That's hilarious!” said Pat.

Computation

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

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

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

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

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

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

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

“Yes.” I replied.

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

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

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

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

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

“What is logic?” asked Pat.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Even though most programmers do not program in assembly language, somehow, having a solid understanding of how a computer works at its lowest levels, and having had the experience of manually encoding algorithms into assembly language, wires a programmer's brain in a way that enables them to be much better high-level language programmers than they otherwise would be.

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

UML Activity Diagrams

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

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

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

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

Calculating The Sum Of The Numbers Between 1 And 10

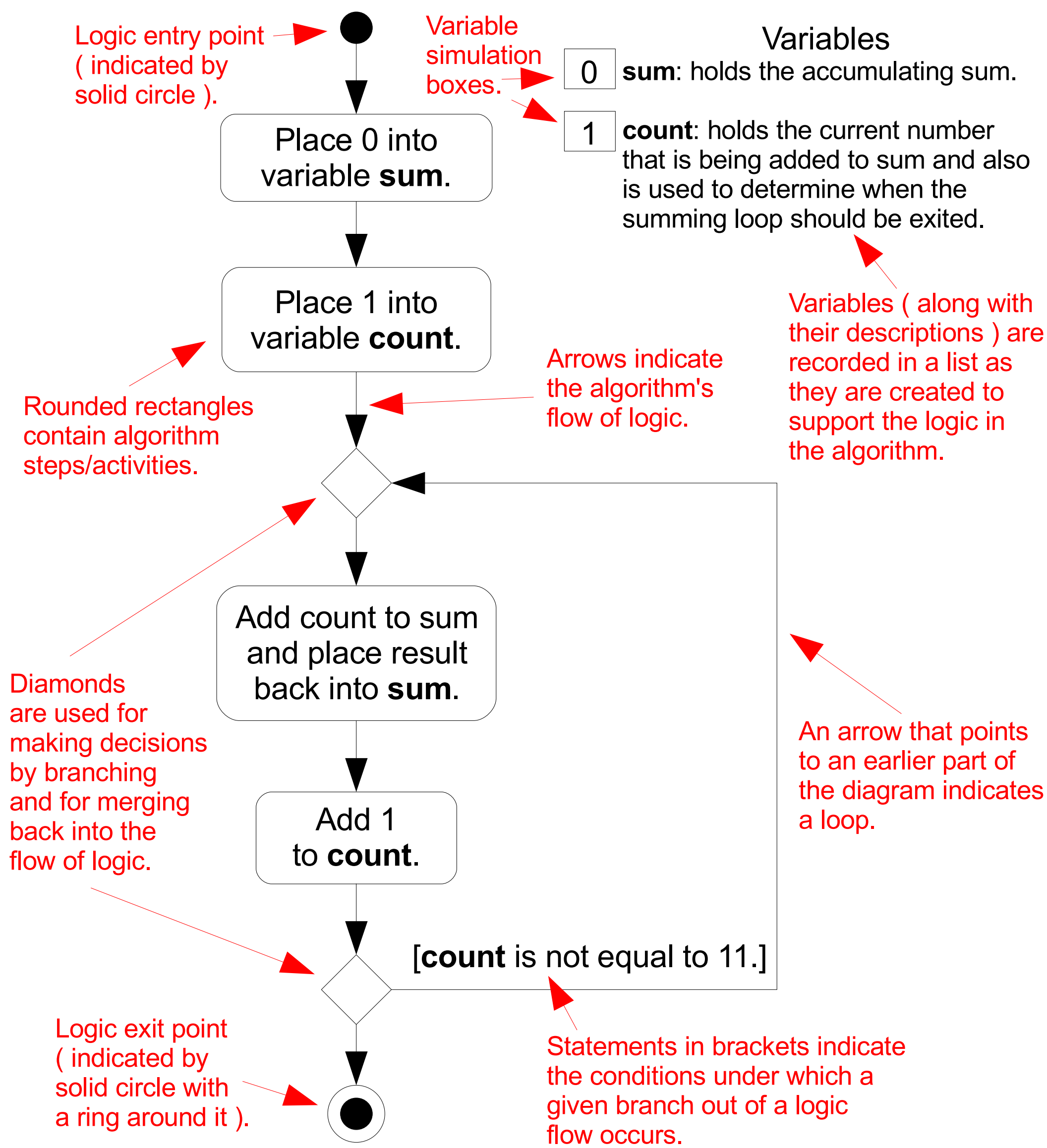
Before I began solving the first problem, I took some sheets of paper from the printer and created 2 funny-looking hats with them using a scissors and tape. I wrote 'Problem Solver' on the first hat and 'Coder' on the second hat and Pat started laughing when I put on the 'Problem Solver' hat.

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

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

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

Figure 1



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

Eventually, I reached a point where I could not find any more errors in the logic and then I said to Pat "It looks like the algorithm is complete now and I can stop being the **problem solver** and pass the algorithm over to the **coder** so it can be encoded into assembly language." I then took off the 'Problem Solver' hat and put on the 'Coder' hat and Pat started laughing again. I brought up an editor and encoded the Activity diagram into the following 6502 assembly language program:

```
                000001 |;Program Name: sum10.
                000002 |;
                000003 |;Version: 1.02
                000004 |;
                000005 |;Description: In this program, the sum of the
                000006 |; numbers between 1 and 10 inclusive will be
                000007 |; determined.
                000008 |
                000009 |;**************************************
                000010 |;Program entry point.
0200            000011 |     org 0200h
                000012 |
                000013 |;Place 0 into variable sum.
0200 A9 00      000014 |     lda #0d
0202 8D 1F 02   000015 |     sta sum
                000016 |
                000017 |;Place 1 into variable count.
0205 A9 01      000018 |     lda #1d
0207 8D 20 02   000019 |     sta count
                000020 |
                000021 |;Top of summing loop.
020A            000022 |LoopTop *
                000023 |
                000024 |;Add count to sum and place result back into sum.
020A AD 1F 02   000025 |     lda sum
020D 18         000026 |     clc
020E 6D 20 02   000027 |     adc count
0211 8D 1F 02   000028 |     sta sum
                000029 |
                000030 |;Add 1 to count.
0214 EE 20 02   000031 |     inc count
                000032 |
                000033 |;If count is not equal to 11 yet then branch back
                000034 |; to LoopTop.
0217 AD 20 02   000035 |     lda count
021A C9 0B      000036 |     cmp #11d
021C D0 EC      000037 |     bne looptop
                000038 |
                000039 |;Exit program.
021E 00         000040 |     brk
                000041 |
                000042 |;**************************************
                000043 |;Variables area.
                000044 |
                000045 |;Holds the accumulating sum.
021F 00         000046 |sum  dbt 0d
```

```
                 000047 |
                 000048 |;Holds the current number
                 000049 |; that is being added to sum and also
                 000050 |; is used to determine when the summing
                 000051 |; loop should be exited.
0220 00          000052 |count      dbt 0d
                 000053 |
                 000054 |      end
```

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

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

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

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

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

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

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

Finding The Largest Number In A List Of 100 Numbers

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

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

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

“Okay.” Said Pat.

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

“Range?” asked Pat.

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

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

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

“0 to 255.” said Pat.

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

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

“Now, give me a list of 5 numbers that meet these requirements.” I said.

Pat thought for a moment then said “172, 34, 3, 242, 99.”

I wrote these numbers on the whiteboard.

“What variables do you think our algorithm is going to need?” I asked.

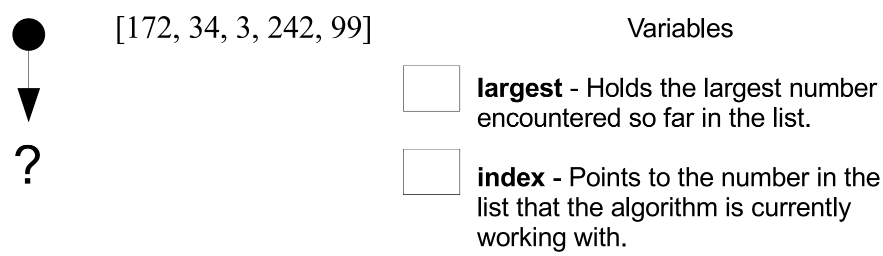
Pat said “I think we are going to need a variable to hold the **largest** number. Hmmm, I am not sure what other variables we will need, though.”

“How should we keep track of where we are in the list?” I asked.

“Oh yes, we will also need an **indexing** variable.” said Pat.

“Lets start with these variables, then, and begin developing the program's logic.” I said. Then I put the information on the whiteboard (see Fig. 2)

Figure 2



“What should the algorithm do first, Pat?” I asked.

Pat studied the information I had placed on the whiteboard then said “I think the first thing the algorithm should do is to set the variables to their initial values.”

“I agree.” I said. “Setting variables to their initial values is referred to as **initializing** the variables and they can be initialized either by algorithm steps or by just placing the initial values into the variable's simulation boxes. If the initial values are placed into the simulation boxes, these variables will be initialized when the variables are defined in the program. For example, in the 6502 assembly language we have been using, **largest dbt 5d** would initialize the variable **largest** to 5 decimal. Since we initialized the variables using logic in the last problem, we will initialize them when they are defined in this problem.”

“Okay.” said Pat.

“What should we initialize these variables to?” I asked.

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

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

“Set it to 75.” said Pat. I then set **largest** to 75 and **index** to 0.

“Now pretend that you are a computer and describe how you would determine the largest number in this list.” I said. “Keep in mind that the computer is not able to 'see' the whole list at one time like a human can. The computer is only able to see 1 number in the list at any given time so which number would you start with?”

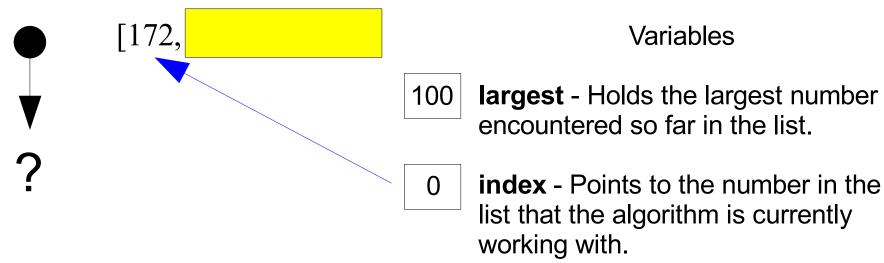
“I would probably start with the first number in the list.” answered Pat.

“Which number is the first number in the list and what offset is it at?” I asked.

“172 is the first number and it is at offset 0.” replied Pat.

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

Figure 3

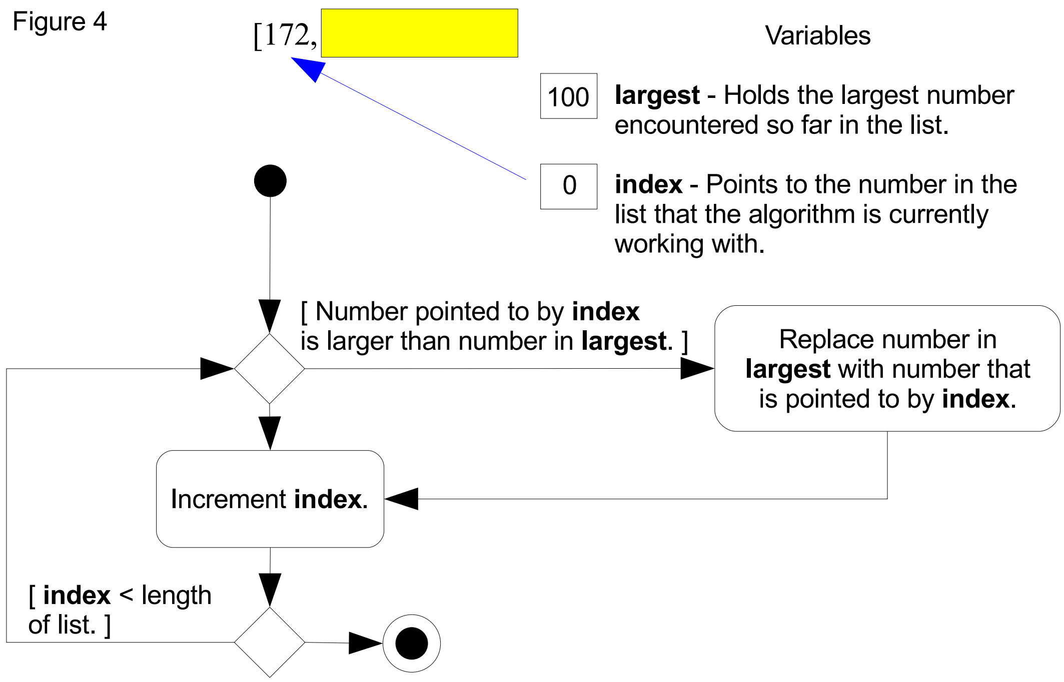


Then I said “What logic do you think should be performed in the algorithm, Pat?”

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

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

Figure 4



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

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

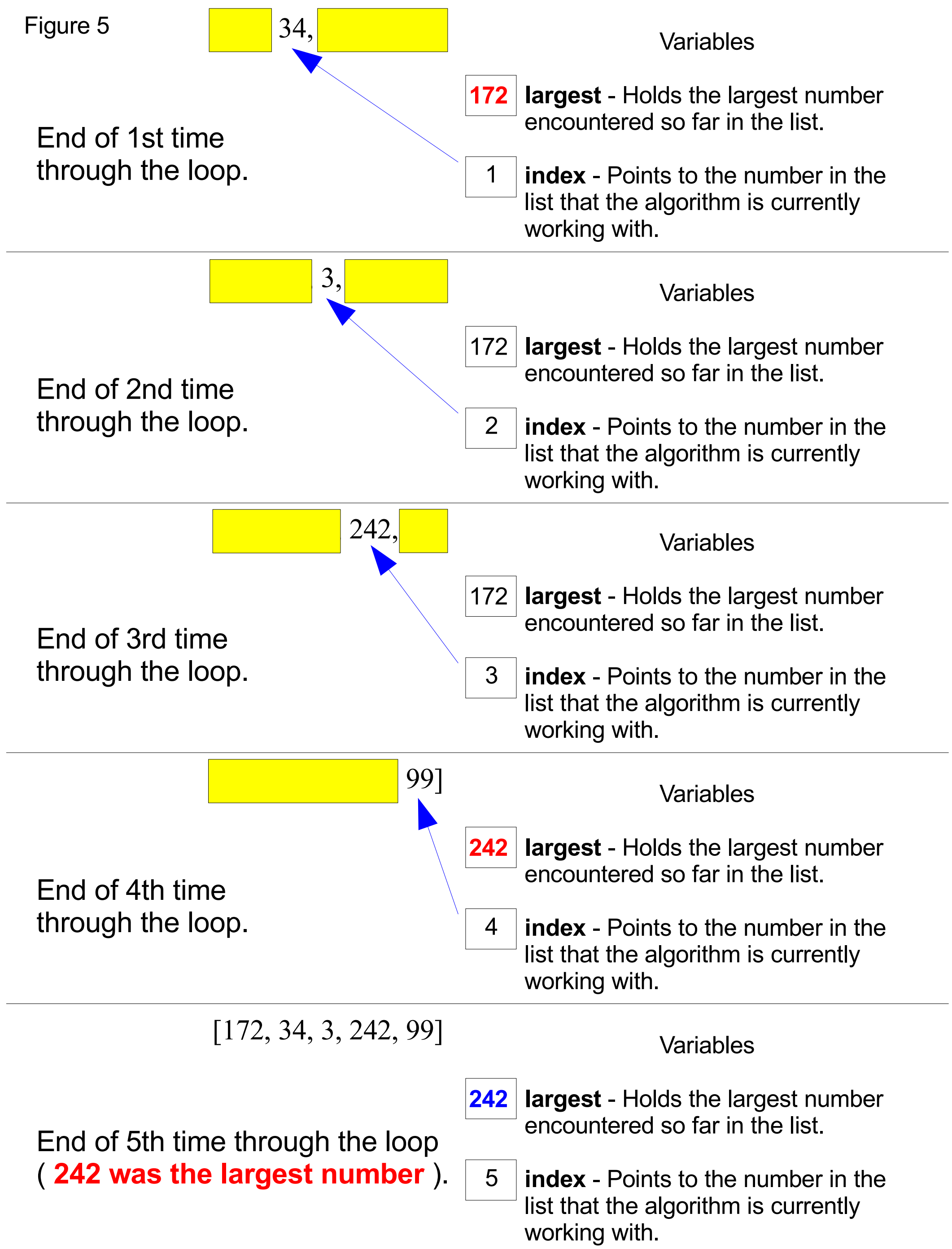
“In the context of computer programming,” I replied “it means to mentally execute a program, or a program's algorithms, without using a computer. A programmer should always dry run their algorithms and programs to make sure they are correct.”

“Why are dry runs 'dry'?” asked Pat.

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

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

Figure 5


34,
Variables
172
largest - Holds the largest number encountered so far in the list.
1
index - Points to the number in the list that the algorithm is currently working with.
End of 1st time through the loop.
3,
Variables
172
largest - Holds the largest number encountered so far in the list.
2
index - Points to the number in the list that the algorithm is currently working with.
End of 2nd time through the loop.
242,
Variables
172
largest - Holds the largest number encountered so far in the list.
3
index - Points to the number in the list that the algorithm is currently working with.
End of 3rd time through the loop.
99]
Variables
242
largest - Holds the largest number encountered so far in the list.
4
index - Points to the number in the list that the algorithm is currently working with.
End of 4th time through the loop.
[172, 34, 3, 242, 99]
Variables
242
largest - Holds the largest number encountered so far in the list.
5
index - Points to the number in the list that the algorithm is currently working with.
End of 5th time through the loop (242 was the largest number).

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

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

“Why not?” asked Pat.

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

I wrote the following list on the whiteboard:

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

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

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

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

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

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

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

```
;Program Name: largest.
;
;Version: 1.0.
```

```
                000004 |;
                000005 |;Description: Find the largest number in a list
                000006 |; of 100 numbers.
                000007 |;
                000008 |;Assumptions: The numbers in the list will be no
                000009 |; lower than 0 and no higher than 255.
                000010 |
                000011 |
                000012 |;**************************************
                000013 |;Program entry point.
0200            000014 |     org 0200h
                000015 |
                000016 |;This program will use the X register to point
                000017 |; to the current number in the list
                000018 |; instead of a variable called 'index'.
                000019 |;
                000020 |;Initialize register X to refer to offset 0 into
                000021 |; the list of numbers starting at 'list'.
0200 A2 00      000022 |     ldx #0d
                000023 |
                000024 |;Compare the number that X is referring to in the
                000025 |; list to the number that is in 'largest'.  If the
                000026 |; number being pointed to is larger than the contents
                000027 |; of the variable 'largest', then fall through to
                000028 |; the code that will replace the contents of 'largest'
                000029 |; with the new number.
0202            000030 |LoopTop  *
0202 BD 14 02   000031 |     lda list,x
0205 CD 13 02   000032 |     cmp largest
0208 90 03      000033 |     bcc NotLarger
                000034 |
020A            000035 |IsLarger  *
                000036 |
                000037 |;Replace the number in 'largest' with the current
                000038 |; number from the list.
020A 8D 13 02   000039 |     sta largest
                000040 |
020D            000041 |NotLarger  *
                000042 |
                000043 |;Increment the X register so that it references the
                000044 |; next number in the list.
020D E8         000045 |     inx
                000046 |
                000047 |;Check to see if the end of the list has been reached.
                000048 |; If it has, then exit the program.  If it has not,
                000049 |; then branch back to the top of the loop.
020E            000050 |CheckListEnd  *
020E E0 64      000051 |     cpx #100d
0210 D0 F0      000052 |     bne LoopTop
                000053 |
                000054 |;Exit the program.
0212 00         000055 |     brk
                000056 |
                000057 |
                000058 |;**************************************
                000059 |;Variables area.
```

```
                 000060 |;
                 000061 |;In this program, the variables area is being placed
                 000062 |; below the code because the variables take up
                 000063 |; a significant amount of memory.  This makes
                 000064 |; it more difficult to determine where the variables
                 000065 |; area ends so that the start of the code area
                 000066 |; can be determined.
                 000067 |
                 000068 |
                 000069 |;Holds the largest number
                 000070 |; encountered so far in the list.
                 000071 |; ( Notice that variables can
                 000072 |; be initialized here instead of
                 000073 |; in the code as an option ).
0213 00          000074 |largest dbt 0d
                 000075 |
                 000076 |
                 000077 |;This list contains 100 numbers ranging from 0
                 000078 |; to 255.
0214 3B          000079 |list  dbt 59d,61d,37d,128d,71d,150d,195d,130d,69d,84d
0215 3D
0216 25
0217 80
0218 47
0219 96
021A C3
021B 82
021C 45
021D 54
021E AB          000080 |      dbt 171d,227d,99d,214d,233d,136d,80d,253d,242d
021F E3
0220 63
0221 D6
0222 E9
0223 88
0224 50
0225 FD
0226 F2
0227 70          000081 |      dbt 112d,221d,151d,101d,117d,76d,226d,174d,205d
0228 DD
0229 97
022A 65
022B 75
022C 4C
022D E2
022E AE
022F CD
0230 54          000082 |      dbt
84d,78d,139d,89d,195d,243d,69d,128d,217d,215d
0231 4E
0232 8B
0233 59
0234 C3
0235 F3
0236 45
0237 80
```

```
0238 D9
0239 D7
023A 39         000083 |      dbt 57d,100d,227d,226d,233d,238d,229d,228d,135d
023B 64
023C E3
023D E2
023E E9
023F EE
0240 E5
0241 E4
0242 87
0243 8C         000084 |      dbt 140d,98d,211d,245d,120d,206d,198d,47d,191d
0244 62
0245 D3
0246 F5
0247 78
0248 CE
0249 C6
024A 2F
024B BF
024C EF         000085 |      dbt 239d,27d,236d,12d,242d,148d,98d,11d,38d,189d
024D 1B
024E EC
024F 0C
0250 F2
0251 94
0252 62
0253 0B
0254 26
0255 BD
0256 EE         000086 |      dbt 238d,225d,142d,214d,214d,21d,75d,17d,190d
0257 E1
0258 8E
0259 D6
025A D6
025B 15
025C 4B
025D 11
025E BE
025F B2         000087 |      dbt 178d,123d,125d,123d,10d,166d,123d,135d,220d
0260 7B
0261 7D
0262 7B
0263 0A
0264 A6
0265 7B
0266 87
0267 DC
0268 C1         000088 |      dbt 193d,46d,248d,222d,63d,206d,197d,101d,144d
0269 2E
026A F8
026B DE
026C 3F
026D CE
026E C5
026F 65
```

```
0270 90
0271 C9         000089 |      dbt 201d,233d,12d,241d,85d,180d,29d
0272 E9
0273 0C
0274 F1
0275 55
0276 B4
0277 1D
                000090 |
                000091 |      end
```

Exercises

These exercises use the programs that are included below them.

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

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

Copy The Contents Of list1 To list2

```
%uasm65

;Program Name: copy.
;
;Version: 1.0.
;
;Description: Copy the contents of list1 to list2.
;
;Assumptions: The numbers in the list will be no
; lower than 0 and no higher than 255.


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

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


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

      inx

      cpx #10d

      bne LoopTop


;Exit the program.
      brk


;*************************************
```

```
;Variables area.

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

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

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

      end

%/uasm65
```

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

```
%uasm65

;Program Name: scan.
;
;Version: 1.0.
;
;Description: Scan list1 and determine how many capital A's and lower case
; a's it contains.
;
;Assumptions: The numbers in the list will be no
; lower than 0 and no higher than 255.


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

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


LoopTop  *
      lda string1,x


;If we have reached the 0 that has been placed at the end of the string, then
; exit the program.
      cmp #0d
      beq EndOfList

CheckLowerCase  *
;If the current character is a lower case 'a', then increment LowerCaseCount.
      cmp #'a'
      bne CheckUpperCase
      inc LowerCaseCount
      jmp NextCharacter
```

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

NextCharacter  *
      inx
      jmp LoopTop

EndOfList  *

;Exit the program.
      brk


;*************************************
;Variables area.

LowerCaseCount dbt 0d
UpperCaseCount dbt 0d

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

      end

%/uasm65
```

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

```
%uasm65

;Program Name: alpha.uasm.
;
;Version: 1.01.
;
;Description: 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.
;
;Assumptions: The numbers in the list will be no
; lower than 0 and no higher than 255.


;*************************************
;Program entry point.
```

```
      org 0200h

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

      lda #10d
      sta rowCount
      sta columnCount


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

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

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

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

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

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

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


EndOfList  *

;Exit the program.
      brk


;*************************************
;Variables area.

rowCount    dbt 0d
columnCount dbt 0d
alphalist   dbt 100d(?)

      end

%/uasm65
```

Calculate The Sum Of The 100 Bytes That Are In numbersList

```
%uasm65

;Program Name: sumlist.uasm.
;
;Version: 1.0.
;
;Description: Calculate the sum of the numbers that are in numberlist.
;
;Assumptions: The numbers in the list will be no
; lower than 0 and no higher than 255.


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

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

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

LoopTop  *
;Obtain the next number from the list.
      lda numberList,x

;Add the number to the low byte of sum and then place the result back into
; the low byte of sum.
      clc
      adc sum+1d
      sta sum+1d

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

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

;Exit the program.
      brk


;**************************************
```

```
;Variables area.

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

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

      end

%/uasm65
```