Over the years Pete Stark has written several books on computers and computer programming (one of which we offer through the Kilobaud *books section), plus a number of magazine articles on computers and digital devices. A long-standing contributor to* 73 Magazine, *Pete wrote an article on the construction of a digital counter several years ago which became one of the most popular construction projects in the entire country. In 1967 (which is* ancient *history in this game), Pete wrote an article for* Electronics World *in which he described interfacing a Morse code keyer to a PDP-8S computer he was teaching with. The following excerpt from that article reveals quite a bit about his insight and vision regarding the "home computer" (remember . . . 1967).*

"The price of $10,000 for a mere automatic keyer may seem somewhat steep, even considering its features. But remember that it can also do other things. In its spare time, the computer can balance your checkbook, do the kids' homework, or play tic-tac-toe with your neighbor. During a dull party it makes an excellent conversation piece; it can even be taught to tell your guests things you would never dare tell them yourself!"

"Programming? It's simple!" not only presents the beginning programmer with an interesting (and practical) method of getting into programming . . . it also has a couple of neat application programs at the end. If we can collect enough such programs, we will reprint them as a book. Everyone who contributes an accepted program will not only get paid for the write-up but also receive a free copy of the book. Send 'em in! — John.

Programming?

It's Simple!

Peter A. Stark
PO Box 209
Mt. Kisco NY 10549

In case you hadn't noticed, there's a revolution going in personal computing. Years ago only a very large corporation (or government) could afford a computer; now there are thousands of computer hobbyists who have their very own personal computers purring away happily in their basements.

Even more people are interested in these small computers but are scared away by the prospect of having to *program* them. When you build or buy a computer, that's only half the job. In order to use it, you have to give it programs — the instructions that tell it what to do and when. And the common feeling is that programming must be very hard — after all, professional programmers can get upwards of $20,000 a year for doing it, so it must be hard — right?

Wrong! Once you get the hang of it, programming is a cinch. And it's great fun. When I first learned to program a real computer, I spent nights at the office programming and running the computer just for the sheer fun of it — on my own time. Now that I teach programming, I find many students whom I literally have to push out the door of the computer room at night when we close. I may have trouble getting a half page of *home*work out of them, but put them into the computer room and they don't want to go home. It's addictive.

In this article, I will try to introduce you to programming in a new way. And best of all, I will show you how, for less than $80, you can get your own "programming machine" to practice on and see whether you like programming well enough to get your own full-scale computer system going. Once you get your feet wet, I bet you won't be able to resist going deeper into computing. What I am going to do is to describe how to program one of the new programmable calculators and show you how this is similar to programming a real computer.

Programmable calculators are available from several manufacturers, including Hewlett-Packard, Texas Instruments, Novus (National Semiconductor), and Sinclair. The ones made by H-P and TI are the best known of these The H-P models range in price from about $125 up to several thousand dollars. The models of interest to the average individual, however, are in the range of about $125 to about $400. These include the HP-25 ($125) which can run fairly nice programs, but as soon as it is

turned off it forgets the program you have entered; if you want to run it again, you must key the program in again. The HP-25C (about $175) is identical, but has a CMOS memory which remains powered even with the power switch off. As a result it remembers programs and data even when turned off. The HP-67 (about $400) is substantially more powerful and has the added capability of storing and saving programs on small magnetic cards. With a stack of magnetic cards you can build up a library of progams and feed them back at any time.

The TI models are roughly comparable. The SR-56 model (about $80) is about as complex as the HP-25, while the SR-52 (about $200) has a magnetic card like the HP-67. Novus, Sinclair, and others have several much less sophisticated models in the $80-and-under range. For a nonmathematical beginner, the TI units are simplest to use, so I will describe programming with them in mind While the $200 SR-52 is more powerful, can run longer programs, and has the magnetic card which greatly simplifies use, the SR-56 at $80 (at the time of writing – by the time you read this it may be even less expensive) is perfectly adequate to explore programming and we will use this as our example.

As we talk about programming this calculator, we will use some of the words normally reserved only for the big computers. In this way you will learn some computer lingo. Whenever we introduce a new word it will be *italicized, like this.* Keep in mind also, that all numerical values will be given in decimal (like 5, 19 or 78.325), rather than in the binary numbers normally used by computers. This will help you master the fundamentals of programming without getting bogged down in irrelevancies. So, here goes.

First of all, a programmable calculator has a display like that of a regular calculator. This display shows the numbers the calculator is using. A computer usually has a control panel on which there are rows of lights which also indicate the numbers being worked on The difference is that on the calculator the display is the only *output device,* whereas a computer may have additional output devices such as a printer or tape punch. The readout display on a computer's control panel is generally used only rarely, whereas on the calculator it is used all the time.

Internally, the display is connected to a display register. The word *register* generally refers to a group of flip-flops which actually holds the number, while the display only indicates the contents of that register. In computer lingo, this register would be called something like *accumulator, accumulator register, AC register,* or perhaps just *A register.* Simple arithmetic such as addition or subtraction is done in this register, which is frequently just a temporary stopping place for numbers on their way from one place in the computer to another.

In addition to the accumulator register, programmable calculators and computers often have other registers which are used for temporary storage of numbers. In some machines they can be used only for such storage, while on other machines they may be able to do very simple operations such as addition. These are called auxiliary registers and are labeled with numbers or letters. In the 8008 microprocessor, for example, these registers are labeled A, B, C, D, E, L and H. The SR-56 has ten such registers, labeled Register 0 through Register 9. (Notice how the numbers range from 0 through 9; not 1 through 10. This is common in computers, and it brings up an old joke – How do you tell whether a person is a programmer or not? Ask him to count to five. If he starts with zero, he is; if he starts with 1, he's not.)

Programmable calculators and computers also have a *memory*. This memory is used to store program instructions, data to be used in the program, as well as results of calculations. (This is the one big drawback to programmable calculators: Their memory can be used to store only programs and data to be worked on – never results. Results can go only into registers.)

On the SR-56 calculator the memory is divided into 100 separate locations, each with its own *address.* On the SR-56, these addresses range from 00 to 99 (Notice that again we start with 0, not with 1). If we use this memory only for program instructions, then we can put in up to 100 instructions; on the other hand, if we use part of it for data to be used in calculations, there will be room for fewer instructions. In most cases we are limited to much shorter programs both because part of the memory is needed for data (and results in real-life computers) and because some instructions require more than one memory location.

Each calculator memory location provides room for the storage of a simple number. In the case of the SR-56, each location can store a two-digit number such as 02 or 95. For example, memory location 00 might have stored in it the number 33, while location 01 might hold the number 21, and so on. The *contents* of a memory location is called a *word* in computerese. Since each word is two digits long, we say that the *word length* is two digits and that this is a *fixed word length* memory. In this case, we are measuring word length in decimal digits; in computers we use *bi*nary digi*ts* (or *bits*), and so we talk about 8-bit word lengths and so on. Obviously the greater the word length, the more information you can store in each memory location. The programmable calculator is hampered by its very short word length and thus results of calculations cannot be stored in their main memory – there is just not enough room! Only the display and auxiliary registers are long enough (13-digit word length) to store big numbers.

Let's now jump ahead a bit and consider an actual program. Suppose we have a number stored in Register 1 which we wish to add to another number stored in Register 3, with the answer to be left in the display register. An actual program to do this would go something like this:

```
Number in Register 1
Add
Number in Register 3
= (Leave answer in display register.)
STOP
```

Since this "program" is written in English, it now must be translated into keystrokes on the keyboard, so that it can be stored in the calculator's memory. After examining the keys on the keyboard and reading the instruction book, we find that: One key is labeled RCL (Recall – which means get a number from a register); while there is no key labeled Add, there is one with a + sign; and finally there is a key labeled R/S, which stands for RUN/STOP. Using these keys, the program becomes

```
RCL 1
+
RCL 3
=
R/S
```

This is a program written using symbols. In computer lingo, we would say that it is written in *symbolic code* (often called *assembly language*, because it is then used by something called an *assembler*). A program written in such a language is simply a description of what we want done, cut up into little chunks small enough for the computer (or calculator) to handle one at a time. We simply have to memorize which symbols the computer or calculator will accept (learn a new language), and

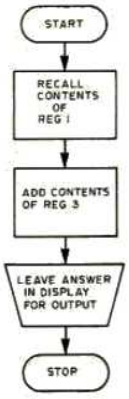


Fig. 1. Simple flowchart

then use them properly On a computer, all these symbols consist of letters (such as ADD, SUB, STOP, etc.), on calculators they may consist of letters or characters like +, -, or ÷.

If you have been reading all this carefully so far, you probably have a question or two. We said a short time ago that this program must be stored in the calculator's memory. But we also said that each memory location can only store a two-digit number. How then do we store something like RCL 1 or "="? The answer is simple — every time we press a key on the keyboard, the key generates a two-digit number. The RCL key is three keys down from the top and four keys in from the left, and so it is wired up to generate the number 34. The "=" key is in the ninth row from the top, four keys over,

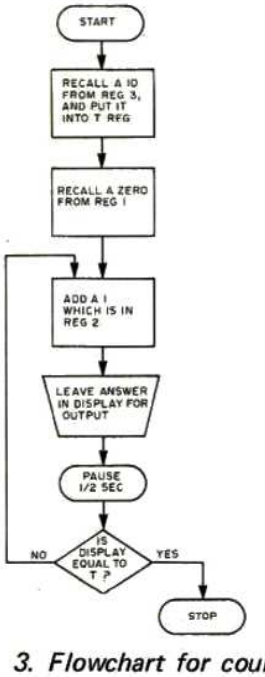


Fig. 3. Flowchart for counting to 10.

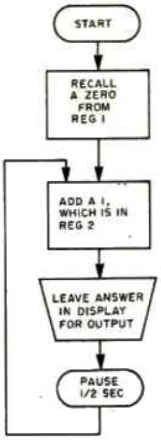


Fig. 2. Flowchart for counting 1, 2, 3, ...

and so it generates the code 94. The digit keys are treated a bit differently — the 1 translates into a code 01, and the 3 into 03. As soon as we press each key, the symbolic code gets translated into a numeric code:

Symbolic	Numeric
RCL 1	34 01
+	84
RCL 3	34 03
=	94
R/S	41

The numeric code, when used with computers, is actually called *machine language*. For each symbol in symbolic (assembly) language, there is a different numeric code. On a calculator, the translation is automatically done by the keyboard and associated circuits; in a computer the translation from assembly language to machine language is done by a program called an *assembler*, which uses a different procedure. Since different brands or models of calculators have different keys in different positions, they would have not only different symbolic languages but different machine languages as well The same applies to computers — learning to program one computer can teach us the basics of programming, but whenever we switch to a different model we have to look up the new codes (symbolic and machine) we need for that machine. When the very first computers were made, they were programmed entirely in machine language. Then, somewhere along the way, assemblers were invented and programmers switched to using assembly language. (The assembler is a fairly complex program which does the translation. Very small computers which do not have sufficient memory to run an assembler are therefore still programmed in machine language.) An even later development is a super-assembler called a *compiler* which can translate more useful and more easily learned languages, such as FORTRAN or BASIC, into machine language. But, since such a compiler uses even more memory than a ordinary assembler, you must already have a fairly large computer to be able to use it. Relatively few personal computers are big enough to run a FORTRAN or BASIC compiler. But enough of this detour — back to machine language.

We note that some instructions, like the R/S or stop instruction, were translated into a single number, like 41. This instruction is a very simple one and only consists of the *instruction* or *operation code* 41. On the other hand, the RCL 1 instruction was translated into 34 01. Obviously, the 34 stands for RCL, while the 01 specified which register. In this case the instruction consists of two parts — the operation code 34 and an *operand* 01. The operand specifies *where* or *to what* the operation applies.

If the calculator had a long enough word length, both parts of the RCL 1 instruction could be stored in the same memory location. In that case, each instruction would be stored in one memory location. Whenever a computer or calculator has a short word length, however, some instructions are too long to fit into one location. Depending on the type of instruction, we may have one word, two word, or even three word instructions mixed in the same program. This is one of the great differences between big computers and little ones — large computers tend to have long word lengths, and so relatively few words may be needed to make a reasonable program. Small computers on the other hand, usually have short word lengths, thus many words are required to produce the same program. For example, 8-bit microcomputers have 8-bit words, thus many of their instructions require two or three words. Likewise, the SR-56 has two and three word instructions, and so its 100-word memory limits us to programs of only a few dozen stops.

Let us now take the above program and actually put it into the SR-56 memory. Before putting it into memory, however, we must decide *where* in memory we shall put it. The usual convention is to pick a *starting address* and then put successive instructions into the following addresses. An easy choice is to start the program at location 00:

Memory Address	Memory Contents
00	34
01	01
02	84
03	34
04	03
05	94
06	41

A total of seven memory locations is needed for this program, out of the 100 locations available. (Note: The SR-52, 224-word memory permits longer programs.

Although the program's instructions are in some cases spread out over two (or even three) words, the calculator knows how many locations go with each operation code, and so it will perform the program properly. You may wonder why we needed the stop code 41 at the end of the program. Since the overall memory contains 100 locations, the other 93 unused locations might still have some other numbers in them from a previous use. If we did not put in the stop 41, the calculator would continue running past steps 05 and 06 into step 07, at which time it would probably start doing something else. It might never

stop, or it might output some really strange results.

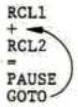
Knowing how theprogram is stored, now allows us to go back and rewrite it in symbolic terms (see program A).

LOCATION	CODE	COMMENTS
00	RCL	Recall register 1.
01	1	
02	+	Add contents of register 3.
03	RCL	
04	3	
05	=	Leave the sum in display.
06	R/S	Stop.

Program (A)

This is the form in which we would usually write a program for the calculator. We would use the symbolic code and let the calculator do its own conversion. Note especially the comments column — when you return to the program a day or a month later, good notes help a lot to remind you of what you were doing.

Another device which helps you to document a program and even to design it in the first place, is a *flowchart*. This is simply a map of the program which describes the order in which things get done. Fig. 1 is the flowchart of this simple program. Note that the shapes of the blocks differ, depending on what instruction is being carried out. In this particular case, the flowchart is so simple that it is almost a waste of time to write it. This is seldom the case. Often, there are *loops* in a program — repetitions of certain parts of program over and over. Fig. 2 shows the flowchart of a program which flashes the numbers 1, 2, 3, 4, 5 . . . on the display. This program is a loop, which starts with a zero (which we have previously put into register 1), adds to it a 1 (which we have previously put into register 2), puts the answer (1) into the display, and pauses for a moment to let us read it. Then it goes back to add another 1. This program will repeat the loop until we turn the calculator off, each time increasing the number in the display by one and displaying the result. To write this program we need two new instruction codes: PAUSE, which causes the program to pause for about ½ second before continuing; and GOTO, which causes the program to go to a different memory location for its next stop. If you remember our previous program, we started the program at location 00 and placed each successive part of the program into the next location. Thus the calculator (or computer) would normally start at 00, go to 01, then 02, and so on. The GOTO instruction breaks that sequence causing the program to take its next instruction from a different location. A simplified program to do the job outlined in Fig. 2 would be something like:



Assigning each part of this to a memory location, we note that PAUSE is a one-word instruction, while GOTO is a 3 word instruction. The program as actually entered into the machine is program B.

LOCATION	CODE	COMMENTS
00	RCL	Get a 0 from register 1.
01	1	
02	+	Add a 1 from register 2.
03	RCL	
04	2	
05	=	Leave answer in display.
06	PAUSE	Pause ½ second.
07	GOTO	
08	0	Go back to step 02 to add again.
09	2	

Program (B)

See why the GOTO is a three-word instruction? The next two words give the place to GO TO — in this case location 02, which is the location of the add instruction.

The above program is a good example of a loop, but it has the disadvantage that there is no way to get out of the loop. It will continue until you manually stop the calculator. In most cases you will wish to provide a built-in stop. Loops are usually performed for a fixed number of repetitions or until they provide a desired result. This is done by means of *conditional GOTO* instructions, which do a GOTO only if a certain condition is satisfied. The conditional GOTO *tests* some internal condition each time the loop occurs. In the case of the SR-56, there are several test instructions we could use, along with an extra register called a T (or Test) register. Let us modify the flowchart of Fig. 2 to count only up to 10 (see Fig. 3). This chart is similar to that of Fig. 2, but with important differences — near the top, we initialize the T register to contain the number 10, and later we compare the contents of the display register with the T register. If they are the same we stop; if they are different we go back to add another 1.

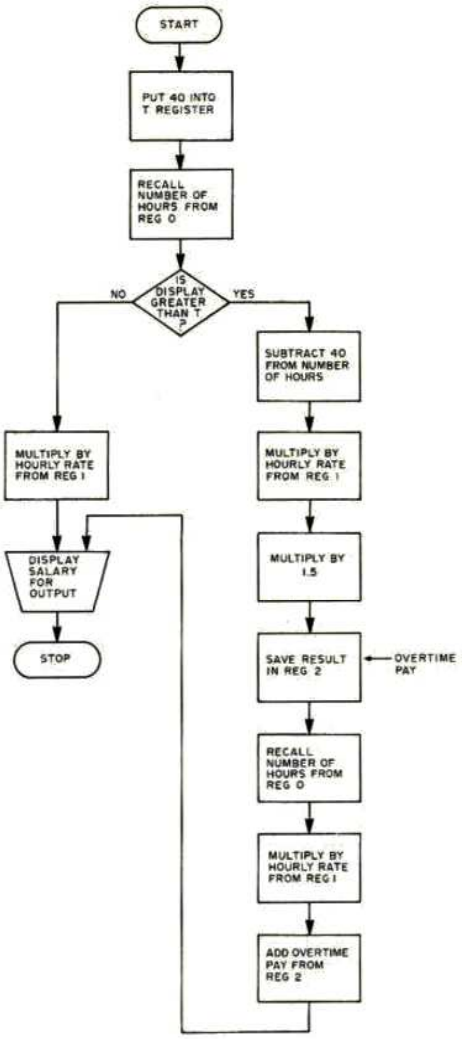


Fig. 4. Flowchart for computing weekly salary.

The flowchart is fairly easy to decipher if you start at the top and follow the arrows (making believe that you are the computer or calculator as you go along). Initially, we start with the display at 0, add 1, then pause with the number 1 showing. Then we check whether the 1 is the same as the 10 in the T register; since it isn't, we go back to add another 1, and next display a 2. We keep going around the loop like this until we display a 10, after which the diamond-shaped decision block shows us that we need to go to a stop.

In programming this problem, we need two more operation codes: X→T moves the number from the display register (called x on the SR-56) into the T register. "X=T?" checks whether they are the same. "X=T?" is actually a conditional GOTO

which does a GOTO if the condition is satisfied; it is a three-word instruction on the SR-56. (The complete program is shown in program C.)

shaped box) the program goes to the left and the result is simply the number of hours times the hourly rate.

If more than forty hours are worked, then the answer at the test box is a YES. As a result, the program computes straight pay for the first forty hours and then adds time and a half for overtime pay for the hours over forty.

Program (C)

LOCATION	CODE	COMMENTS
00	RCL	Get a 10 from register 3
01	3	
02	X-T	and put it into T register.
03	RCL	Get a 0 from register 1.
04	1	
05	+	Add a 1 from register 2.
06	RCL	
07	2	
08	=	Place answer in display.
09	PAUSE	Pause ½ second.
10	X=T?	
11	1	} If X=T GOTO location 16 to stop,
12	6	
13	GOTO	
14	0	} otherwise go back to step 05.
15	5	
16	R/S	Stop.

Conditional GOTOs can be used for jobs other than loops. For example, the conditional "X > T?" (is X greater than T?) would be used to make the decision in the overtime portion of the flowchart shown in Fig. 4. This program takes the number of hours a person works, together with the hourly rate, and computes the weekly salary (before deductions). If less than forty hours are worked (checked by the diamond

From some of the demonstration programs you have seen that really useful programs may take quite a few program steps. For this reason computers tend to have fairly sophisticated operation codes whose purpose is to do a lot with just one instruction. Each of the examples we have done so far could have been made shorter had we used some of the more complex instruction codes available. In addition to the simple operation codes we have touched on so far, the SR-56 includes many more. Some of these are more powerful than those of many large computers — for instance, the SR-56 can find square roots, sines, or logs in only one step, while actual computers generally need long programs to perform these same tasks. Thus even the 100-location memory of the SR-56 can be used to run quite useful programs.

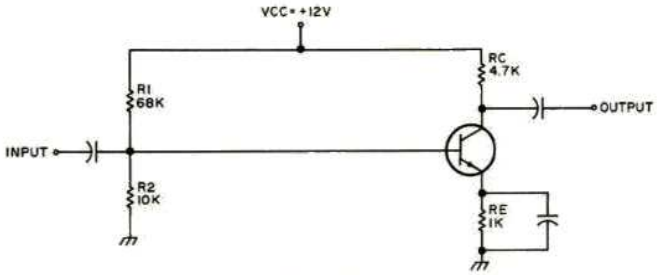


Fig. 5. Transistor amplifier.

For example, Fig. 5 shows a simple common-emitter amplifier. Given the values of the resistors and a few simple assumptions (such as that the transistor has a fairly good beta or that the emitter resistor is not too small), it is easy to calculate the voltages, currents and the gain. Fig. 6 shows the SR-56 program which does just that. Aside from a few new operation codes, such as STO which stores a number into an auxiliary register, this program demonstrates another computer trick.

Whenever a program needs a simple constant, such as 0.7 or 0.025, this constant could be stored in a register (or in a memory location of a large computer) and then recalled with an RCL instruction. But, if the constant is used only once in the entire program, this wastes a register which might be needed for something else. Instead, it is possible to put this constant directly into the program as shown in steps 16-17 or 42-45. In computers this is done by means of *immediate* instructions. For example, the 8008 microprocessor can load a number into the A register from memory by means of a LAM (load A from memory) instruction. But to do so requires that we also provide an address which goes with the instruction and tells the computer where to get the data to be loaded. It is much easier to use a LAI (load A immediate) instruction, and then put the number to be loaded into the very next location in memory after the LAI. We still have to put the number somewhere, but we avoid all the problems of addressing it. In a programmable calculator, this trick avoids the use of another register. ■

LOC	CODE	KEY	COMMENTS
00	34	RCL	Vcc
01	04	4	
02	64	X	X
03	34	RCL	R1
04	01	1	02
05	54		
06	52	(	
07	34	RCL	
08	00	0	(R1+R2)
09	84	+	
10	34	RCL	
11	01	1	
12	53	)	
13	94	=	=
14	41	R/S	Show VB
15	74	-	-
16	92	.	
17	07	7	.7
18	94	=	=
19	41	R/S	Show VE
20	54		
21	34	RCL	RE
22	03	3	
23	94	=	=
24	33	STO	Store

LOC	CODE	KEY	COMMENTS
25	05	5	
26	41	R/S	Show IE
27	64	X	X
28	34	RCL	
29	2	Rc	
30	93	+/-	Change
31	84	+	Sign
32	34	RCL	+
33	04	4	Vcc
34	94	=	=
35	41	R/S	Show Vc
36	34	RCL	Rc
37	02	2	
38	64	X	X
39	34	RCL	IE
40	05	5	
41	54		
42	92	.	
43	00	0	.025
44	02	2	
45	05	5	
46	94	=	=
47	41	R/S	Show Gain
48			
49			

REGISTERS	
0	R1
1	R2
2	RC
3	RE
4	Vcc
5	IE

NOTES	
Test case: if	
R1	= 68000
R2	= 10000
RC	= 4700
RE	= 1000
Vcc	= 12V
then	
VB	= 1.538V
VE	= .838V
IE	= .838ma
VC	= 8.059V
Gain	= 157.6

Program for finding base, emitter, and collector voltages, emitter current, and gain for the amplifier in Fig. 5.