Writing your own programs

D. M. de Boer

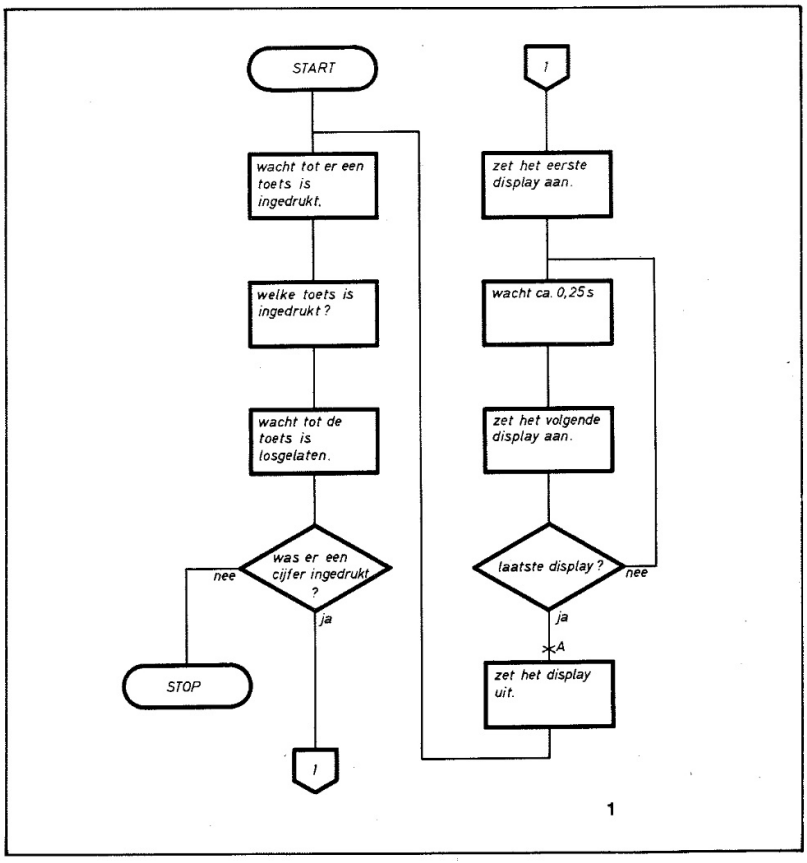
In our previous issue you were introduced to the KIM I microcomputer. Perhaps you have already built the Melodiant, but the most enjoyable work is of course writing your own programs, making the KIM I do exactly what you want. Now, programming is not that easy for someone who has never done it before. Especially at the beginning, the aspiring programmer works in such a disorganised way that in the end he no longer knows which instruction is at which address. It is therefore important to set up a program in a clear, orderly way and to write down every change. In this issue we will use an example to show how this is done. However, we leave the exact working-out to the reader, because here too it is certainly true: practice is the best teacher!!!

What do we want?
When we are going to write a program, we must first state clearly what exactly the KIM I is supposed to do.
As an example, let us say that a typed-in digit should appear on the leftmost display and then keep moving one place to the right until the digit has disappeared from view. A clear specification could look like this:
1 The typed-in digit may only become visible on the leftmost display when the key is released.
2 Next, the digit must move to the right at a fixed speed.
3 As long as the digit is on one of the displays, pressing a key must have no effect (except RS and ST, which always interrupt the program).
4 When the displays are off and a key without a digit on it is pressed, the program must be interrupted after the key is released.

Using this definition we will now show how a program comes about. Just to be clear: normally you decide for yourself, of course, what the KIM I should do. The definition above leads to a pointless program and therefore serves only as an example.

How are we going to do it?
In the next phase we ask ourselves how the program should be structured. At this stage we do not yet pay attention to the available instructions and registers. The idea is to split the complete program into small, but still understandable, separate tasks. The first task could be, for example: 'Wait until a key is pressed'. The second task is therefore only carried out once a button has actually been pressed.

This second task is then: 'See which key has been pressed'. It comes down to writing down all the actions we ourselves would perform one after another to obtain the desired effect. In this way the whole program is drawn up. We write the tasks in a flow chart. The flow chart of the example could look like the one drawn in fig. 1. The tasks are in rectangles, while decisions (conditional jumps) are written in a diamond. With this flow chart we can now see at a glance in which order the various parts of the program are executed. Before going any further, we must first check whether the flow chart meets the requirements. We see that the displays can only light up once the button has been released, so that is fine. It is also checked whether the pressed key was actually a digit. If it was not, the program is stopped. The task 'wait approx. 0.25 sec.' ensures that each display stays lit for a clearly visible time. If we look at the flow chart critically, however, we see that the last display is switched off again almost immediately, so it will not visibly light up.
To remedy this we can insert a task 'wait approx. 0.25 sec.' at point A in the flow chart, so that the last display also stays lit long enough. Another solution is to pretend that there are 7 displays instead of 6. The 6th display is now treated 'normally', while display 7 (which does not exist) is lit too briefly. In this way all displays will be lit for the same length of time.



1

Working out the flow chart

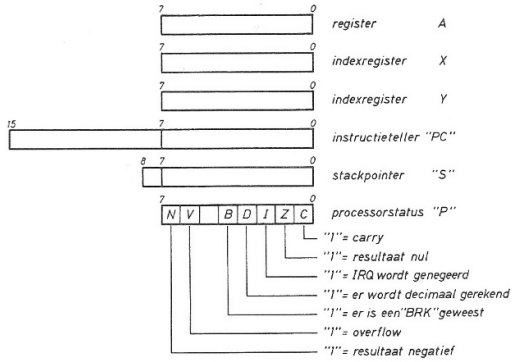
We have now set up the flow chart in broad outline, in separate tasks that we understand. The computer, however, does not understand any of it yet. That is why in this phase we must work out the flow chart into even smaller tasks that the computer can understand. Where possible, we can make use of subroutines that are already present in the ROM. This is usually the case for tasks that are also used by the monitor program. As an example we can

take the second task: 'Which key has been pressed?' It will be clear that the monitor program (which makes sure that typed-in numbers end up in the right locations) also has to determine repeatedly which key has been pressed. Studying the monitor program (found in the supplied KIM I user manual) shows that a ready-made subroutine for this task is located at address 1F6A.
Before we can work out the flow chart further, we need to know which instructions the microprocessor knows and how the registers in the microprocessor are organised. The instruction set consists of 56 instructions, which were already discussed in the previous issue (RB Aug. '77 p. 288). The 6502 microprocessor has 3 freely usable internal registers of 8 bits each. These are register A (accumulator) and the index registers X and Y. The microprocessor of course also contains the program counter (16 bits), the stack pointer (8 bits) and the status register (7 bits). These registers are shown once more in a clear overview in fig. 2.
As already mentioned, the idea is to split each block of the flow chart we now have into a number of blocks with very specific tasks that the microprocessor can understand. In this way a second flow chart is created. The number of blocks into which each block of fig. 1 has to be divided can vary considerably.

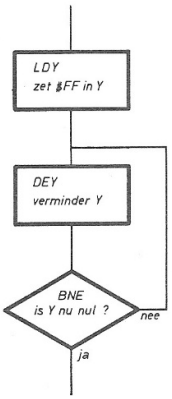
As an example we will work out one block. You can then try to work out the other blocks yourself. For this we take 'wait approx. 0.25 sec.'. How do we achieve this? It is not possible to stop the microprocessor; it keeps executing 200,000 . . . 500,000 instructions per second! The only way to make the microprocessor appear to wait is to have it run through a small piece of program a very large number of times. This is called a loop. Fig. 3 shows the flow chart of such a loop.

We start by loading index register Y with a certain value. Then we keep decreasing the contents of this register by 1 until the result is '0'. Each time the loop is run, the microprocessor is kept busy for 5 μs (2 μs for the decrement and 3 μs for the jump back). The maximum content of register Y is $FF, written in decimal 256. The total achievable delay is therefore 256 x 5 μs = 1280 μs. Longer times can be made by nesting 2 or 3 loops inside each other. With 2 loops we can get times of up to approx. 0.3 seconds, sufficient for our example. The working-out of 'wait approx. 0.25 sec.' now becomes as shown in fig. 4. Here the flow chart has been reduced to instruction level, i.e. each block of the flow chart represents one, and no more than one, instruction.

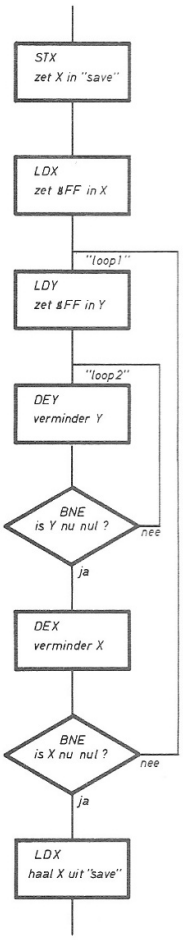
When using loops we must remember that the original contents of the index registers are lost. In the working-out (fig. 4) we assumed that index register X was already in use. The value of this register is now temporarily stored in memory.



2



3



4

adres	instructie			naam	OPcode	operand	verklaring
					STX	save	zet X zolang in het geheugen
					LDX	#$ FF	inhoud bepaalt
				loop 1	LDY	#$ FF	totale vertraging
				loop 2	DEY		
					BNE	loop 2	einde tweede loop
					DEX		
					BNE	loop 1	einde eerste loop
					LDX	save	haal X uit het geheugen

5

adres	instructie			naam	OPcode	operand	verklaring
0200	8E	23	01		STX	save	zet X zolang in het geheugen
0203	A2	FF			LDX	#$ FF	inhoud bepaalt
0205	A0	FF		loop 1	LDY	#$ FF	totale vertraging
0207	88			loop 2	DEY		
0208	D0	FD			BNE	loop 2	einde tweede loop
020A	CA				DEX		
020B	D0	F8			BNE	loop 1	einde eerste loop
020D	AE	23	01		LDX	save	haal X uit het geheugen
0210							

6

2 De interne registers van de 6502 microprocessor.
3 Een enkele loop, deze flow chart is tot het instructieniveau terug gebracht.
4 Een dubbele loop, hiermee zijn tijden tot ca. 0,3 sec. te realiseren. Ook deze flow chart is tot instructieniveau teruggebracht.
5 Het programma van het stukje flow chart uit afb. 4. De eerste 4 kolommen blijven voorlopig leeg.
6 Als het hele programma klaar is gaan we de machinecodes en de adressen bijschrijven. De instructie kan bestaan uit 1, 2 of 3 bytes.

Only when the whole flow chart has been brought down to instruction level (via one or more intermediate steps) do we start writing the program.

As an example we will limit ourselves here to the piece of flow chart in fig. 4. To begin with, we write a name (label) at every point where flow lines come together. In fig. 4 these are 'loop 1' and 'loop 2'. This is done so that these points can easily be found again later in the program, and so that for the jumps we can indicate where to jump to. Also, when data has to be stored somewhere temporarily, we indicate a memory location by a name, not by an address! In practice this turns out to be the handiest way of working, because memory locations can still easily be moved around afterwards. After all, we only need to define an address at the very last moment.

Writing the program

Once we have brought the whole flow chart to instruction level, we can start writing the program. It is easiest to work if we divide a sheet of paper into 8 columns, as shown in fig. 5. We leave the first 4 columns empty for now. In the 5th column we fill in the name of the memory location concerned, if any. The OP code goes in the 6th column, the operand in the 7th column. We use the 8th column to give a brief explanation of the program.

This program explanation will prove very useful if you want to look at the program again a month later. The 'operand' column always contains the name of the memory location. Only with immediate addressing does the operand itself appear here. The # sign indicates that immediate addressing is used, while $ is used to indicate that the number is written in hexadecimal.

Once we have written the whole program in this way, we fill in the first 4 columns. It is best to start with the 2nd . . . 4th columns. In these we write the machine codes belonging to the instructions. The first instruction (fig. 5) reads: 'STX save'. The contents of index register X must be placed for the time being in a memory location named 'save'. We have the choice between zero page addressing and absolute addressing. In the example we assume that memory location 'save' has been given the address $0123. So we must use absolute addressing. The instruction is coded as follows: first we look up the code for STX. For this code we find: 8E. This is the OP code, which goes in the first byte (fig. 6). In the 2nd and 3rd bytes we put the address of the operand. It may happen that this address is not yet known; in that case we simply leave these places open for now. Note that the last 2 digits of the address go in the 2nd byte and the first 2 digits in the 3rd byte! In this way we fill in all the codes. For instructions that are only 1 or 2 bytes long, we simply leave the last columns empty (fig. 6). Filling in the addresses will not cause any difficulties; in principle we can start at any address, provided the program stays within the available space. We can use all addresses between 0000 and 03FF; only addresses 00EF . . . 00FF are reserved for the monitor program, as are the highest addresses on page 1, which are reserved for the stack. (A space of 01E0 . . . 01FF is usually sufficient.)

Calculating the relative address for branch instructions

A final difficulty when coding the program is determining the relative addresses for the

branch instructions. At address 020B (fig. 6) we find such a branch instruction. In this case we must jump back to 'loop 1'. The microprocessor determines the new address by adding the relative address to the value of the program counter. When the microprocessor performs this operation, the program counter has already been incremented. In our case the program counter is therefore at 020D. After the jump we must end up at 'loop 1' at address 0205. So we must make a jump of 0205 - 020D = -08 ($D=13, 5-13=-8). If the addresses are not yet known, we can also simply count the number of bytes between the two instructions. The minus sign indicates that we must jump back. So in the second byte of the branch instruction we must fill in -8. For this we first have to calculate the two's complement. This is done as follows:

We first write the number in binary, invert it, and finally add '1'.

Example:

$08 -> 00001000
invert 11110111
add 1 11111000 -> $F8

The hexadecimal number -08 thus becomes $F8 in two's complement (the $ sign again stands for hexadecimal). For small negative jumps (smaller than $F) it is often quicker to simply subtract the desired jump from 16 (decimal). Example: a negative jump of 8: 16-8=8. The two's complement becomes $F8. A jump of 3: 16-3=13, or D in hexadecimal. The two's complement becomes $FD.

Driving the displays

If you want to try working out the example given, you still need to know how the displays are driven. Fig. 7 shows the connection diagram of the displays and the keypad. In the normal state, inputs PA 0 . . . 6 scan the keypad. Outputs 0 . . . 2 of IC 1 are made low in turn (all of this is done by the monitor program). When we want to drive the displays, we must turn inputs PA 0 . . . 6 into outputs. This happens as soon as we write a '1' for bits 0 . . . 6 in the direction register PADD2 (address 1741) ($7F). We must also select one of the six displays using PB 1 . . . 4 and IC 1. The table below shows which number must be written to PBD2 (address 1742) to select a particular display.

Display no.	number in PBD2
1	08
2	0A
3	0C
4	0E
5	10
6	12

Once the display has been selected and connections PA 0 . . . 6 have been programmed as outputs, we can put the display code on PAD2 (address 1740). In fig. 8 we see a number next to each segment. If we add up the numbers of the segments that must light

up, we get the display code.
The code to get the number 5 on the display is therefore 1+4+8+20+40 = 6D. When we have finished displaying, we must not forget to define the connections PA 0 . . . 1, now programmed as outputs, as inputs again ($00 in PADD2, address 1741).

Useful subroutines
As mentioned earlier, the monitor program makes regular use of subroutines that are already in the ROM. We can also make use of these subroutines in our own programs. Below is a list of interesting subroutines.

INIT 1 address 1E8C
This subroutine sets all direction registers to the correct state. The decimal mode flag is set to '0', the interrupt flag to '1'. The contents of index register X are lost.

AK address 1EFE
This subroutine determines whether a key has been pressed. The contents of register A become 0 if no key has been pressed, and non-zero if a key has been pressed. The contents of register A and index registers X and Y are lost.

GETKEY address 1F6A
This subroutine determines which key has been pressed. The typed-in number is placed in register A. The contents of register A and index registers X and Y are lost.

SCANDS address 1F1F
This subroutine makes the contents of addresses 00FB, 00FA and 00F9 appear on the display. Only hexadecimal digits can be shown; all 6 displays are always lit at the same time. The direction registers are also set correctly. The contents of index registers X and Y, as well as the contents of register A, are lost. At the end of this subroutine a jump is made to AK, so that it is immediately determined whether a key has been pressed.

CONVD address 1F48
This subroutine lights one of the 6 displays and can therefore be used in the example given. The direction registers are not set correctly by this subroutine. The number in register A is converted into a display code. The number in index register X determines which display is lit (see the table under 'Driving the displays'). The contents of index register X are automatically incremented by 2, so that this register now points to the next display. The contents of register A are lost.

Checking the program
You now have enough information to work out the example yourself. Once the whole program has been written, you can type it into the computer.
t is wise to first run through the program in 'single step' mode. To do this we must put '00' at address 17FA and '1C' at 17FB. Then we set the little slide switch to SST. It is particularly important to check with the branch instructions whether the jumps have

been calculated correctly. Between steps we can check the contents of the various registers by entering the following addresses:00F1 status register
00F2 stack pointer
00F3 register A
00F4 index register Y
00F5 index register X
If necessary we can also change the contents of these registers, which can be particularly useful with loops. When we want to continue with the program, we must first press PC, so that the program resumes from the point where we left off.
If we want to check the contents of the various registers and/or memory locations somewhere in the middle of the program, there is a quicker method: we can also insert stops. To do this we must first enter the 'stop address', i.e. the address to jump to when a stop is encountered. Usually this address will be 1C00 (monitor program). Again we enter '00' at address 17FE and '1C' at address 17FF. At the place where we want to check the registers and/or memory, we temporarily replace an instruction with BRK (code 00). The program will now run normally until the BRK command. At this point the microprocessor jumps to the monitor program, so that we can check the various registers.

Some common mistakes
Finally, we will list a few common mistakes, so that you can be on the lookout for them.
- Incorrectly calculated branch instructions. A wrong jump can cause an operand to be read as an OP code. This results in a completely different (and often very strange) program. Such an error can cause the program to overwrite itself, so that the whole program has to be checked again.
- Registers are often used twice. When subroutines are used, the contents of one of the registers are sometimes lost. In loops, the contents of the index registers change. So remember that in these cases the contents of various important registers must be temporarily stored in memory.
- Make sure you give both Push and Pull instructions. 'Push' causes the stack pointer to be decreased by 1; 'Pull' causes the stack pointer to be increased again. If only one of these instructions occurs in a loop, the stack pointer is continually increased (or decreased), which can cause all information on page 01 to be lost.

[Fig. 1: This is what the flow chart of the example could look like.]

[Fig. 2: The internal registers of the 6502 microprocessor.]

[Fig. 3: A single loop; this flow chart has been reduced to instruction level.]

[Fig. 4: A double loop, which makes times of up to approx. 0.3 sec. possible. This flow chart has also been reduced to instruction level.]

[Fig. 5: The program for the piece of flow chart from fig. 4. The first 4 columns remain empty for now.]

[Fig. 6: When the whole program is finished, we add the machine codes and the addresses. An instruction can consist of 1, 2 or 3 bytes.]

[Fig. 7: This is how the displays and the keypad are connected to output ports PAD2 (1740) and PBD2 (1742).]

[Fig. 8: The display code for any combination of segments is obtained by adding up the numbers shown.]