the practical introduction
to a powerful system

JUNIOR COMPUTER

Book 4

Elektor

The Elektor Junior Computer

the practical introduction to a powerful system

A. Nachtmann
G. H. Nachbar

Elektor Publishers Ltd.

The circuits published are for domestic use only. Patent protection may exist with respect to circuits, devices, components etc. described in this publication. The publishers do not accept responsibility for failing to identify such patent or other protection.

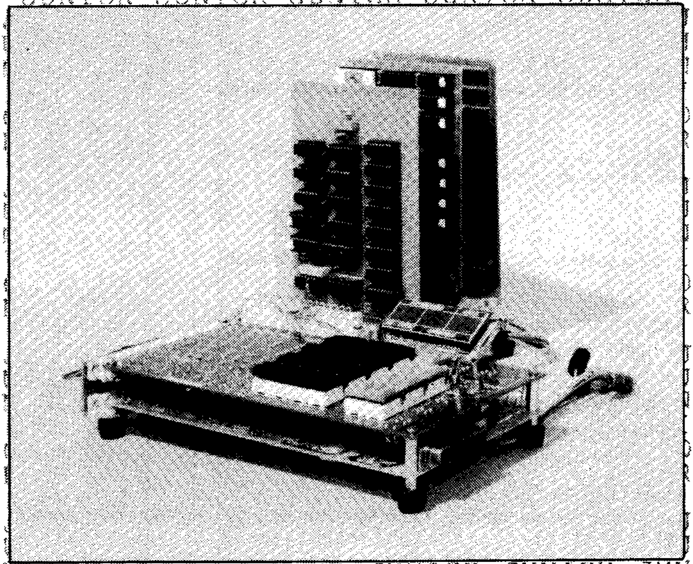
Printed in the Netherlands

ISBN 0 905705 11 4

Foreword

This, the last in the series of Junior Computer Books, describes all the software required to operate the complete system. Chapter 13 introduces an extended version of the Printer Monitor program (PME) which combines the advantages of the original editor program with those of the Printer Monitor. It selects the best of both worlds. As a result, program entry and assembly can be accomplished quickly and easily.
Chapter 14 provides a detailed description of the Printer Monitor program and the associated subroutines, while the extended version is fully described in chapter 15. The software required for the cassette interface, the Tape Monitor program, is explained in great detail in chapter 16.
As well as providing a complete run down on the various system routines, Book 4 also contains detailed program listings of the PME, the TM and PM software in the appendices at the back of the book. Finally, appendix 5 describes how BASIC may be run on the Junior Computer; a welcome facility that puts the machine in touch with the rest of the world.

Contents



13

Editing and assembling at high speed

The PM Editor

Writing a program is great fun . . . until it has to be entered into the computer. That is, unless the programmer has acquired a certain degree of expertise and has discovered all sorts of shortcuts to speed up the process. After all, the Junior Computer was designed to provide enthusiasts with an educational, yet enjoyable pastime – not forced labour!

The first step towards simple and efficient program entry was taken with the introduction of the editor and assembler in Book 2. Book 3 went a step further with full details of the cassette interface which enables programs to be stored on tape, so that they only have to be entered manually once. But, as this chapter will show, the entire process can be accelerated considerably with the aid of a high speed editor.

The computer's 'jump' from the original monitor program to the Printer Monitor was also described in Book 3. It is now time for yet another leap forward: from the original editor to the PM Editor (PME). As its name suggests, this is an editor program based on the Printer Monitor. In other words, it involves the use of the ASCII keyboard and the results can be monitored on a video screen or with the aid of a printer, instead of the hexadecimal keyboard and the seven segment displays. Nevertheless, the system will still make use of hexadecimal labels for efficient program entry.

Let us examine the number of new possibilities that are now open to us. For one thing, there are more keys available, some of which already serve a variety of 'new', time-saving functions. In addition, several instructions can now be displayed at the same time. What is more, the actual number of key operations can now be reduced by about 30% with respect to the original monitor routine, so that the programmer is less liable to make typing errors.

All this has come about with thanks to PME. Again, this very useful feature will be explained in detail and will be illustrated with practical programming examples wherever relevant. The programs will involve PM subroutines such as the ones described in chapter 12 of Book 3.

Reading this chapter and learning to work with PME is bound to take a certain amount of time and effort, but the programmer will discover that in the end it will save him/her precious hours . . .

Junior's gaining stature

The computer completed its physical growth in Book 3. In Book 4 the machine will be supplied with sophisticated software to update its mental education.

Both the EPROMs containing the Tape Monitor and the Printer Monitor have a vacancy for about ¾ kilobytes of memory. Obviously, something had to be done to 'fill the gap'. PME was developed to occupy these (so far) unused memory locations. Although the PM editor does not occupy a great deal of memory space, this 'investment' provides a lot more room: on the keyboard and especially on the video screen which now allows instruction sequences and various comments to be shown in an easy-to-read manner. Therefore, the new editing feature is a real 'eye-opener' in every sense of the word!

Of course, the statement that a new editor is required will make readers wonder what was wrong with the old one, or rather, what advantages does PME have over the original editor routines. The argument that there happens to be sufficient room for PME in an existing EPROM is hardly valid! After all, the memory space could have been filled with all sorts of equally useful routine and/or interesting information!

What does PME have to offer?

Well, what did we have to start with? As you will remember from Book 2, the original editor program had one very great advantage in that it enabled the use of hexadecimal labels. The alternative is to use labels consisting of words of up to six letters: labels similar to the ones found in the listings at the back of Book 2 and at the back of this book. The thing is, such listings are made with the aid of a large editor and assembler which occupy a great deal more memory space than the ¾k taken up by PME. So, apart from the extension memory (busboard EPROM), there just is not sufficient EPROM available for word programming. In any case, the majority of programmers would prefer to store programs on cassette tape rather than run to the expense of filling up yet another EPROM. However, there is another reason for not choosing a large extensive editor.

This would require a fully documented user program, so that all the labels are known and all the memory locations involved are assigned a name beforehand (such as POINTL or BYTES), in other words, they have to be 'declared'. The same thing refers to subroutines with a fixed address, tables, etc.

Now, supposing an operator has thought of a program at four o'clock in the afternoon and wishes to know by five whether or not it works (so as not to miss the train home!). The chances are, the program will need brushing up here and there. Then it should be taken into account that:

a. a large editor/assembler requires a considerable amount of preparation work beforehand;
b. more often than not, some of the preparations will be found to have been superfluous.

Seen from that angle, it is preferable to have an editor that works with hexadecimal labels like the original editor and PME of the Junior Computer. Granted, the method mentioned above would have the program entered by four thirty, but what is the point, if it takes another hour to 'clean it up'?

Undoubtedly, hexadecimal labels make programming a lot easier. During the 'creative phase' in particular, when a program is actually being developed, the hexadecimal system is indispensible. Furthermore, it saves having to write a new assembler, as the existing version described in chapter 9 of Book 2 is perfectly suited to its task.

There is only one (possible) snag in using hexadecimal labels: label numbers may not be identical to the low order byte (ADL) of an absolute address, in other words, an address which was established before the program was edited and assembled. However, there are matters which the operator will get to grips with soon enough, so it is hardly a disadvantage. In any case, only about 80 labels are actually available which should avoid confusion. Readers may think that this is a limited number, but they can be assured that, even when relatively long programs are involved, 80 is more than sufficient.

In spite of these advantages, the original editor routine needs to be replaced for the following important reasons:

- Firstly, the key functions on the main board of the Junior Computer are very restricted and so is the six digit display.
- **Error messages.** The standard editor only features one type of error message: 'EEEEEE' on the display. This is shown until the offending key is released. Statistically, therefore, it is quite likely to escape the operator's notice altogether. Since it is always useful to be able to learn from one's mistakes, it might be helpful if the computer were to be a little more explicit by providing

- **more error reports.** One large disadvantage in the original editor is the fact that **it gives no indication when the memory range defined by BEGAD on the one hand and by ENDAD on the other is full after an excessive number of instructions and labels have been entered.** This can be extremely annoying and is exactly what happens in the PLAY program given in Book 2 (page 41). The available memory range, ØØØØ . . . ØØED, is quite sufficient for the assembled version of the PLAY program (labels are deleted), but not for the non-assembled version where the labels are still included. As a result, the computer runs out of memory space, so that the EOF character, 77, will be stored in location ØØE3 (= BEGADH!) and the first instruction is overwritten. This can be remedied by dropping the label 9Ø (PLAY), which is quite feasible since the program does not actually jump to this label. Then there would be room for the whole program (including all other labels) . . . but there would not be enough locations left to store the first label to be detected during assembly (see figure 1). The solution is to store the PLAY program on page Ø2, where there is plenty of room.

By the way, the extended version of the Junior Computer provides consecutive RAM from Ø2ØØ . . . Ø7FF, amounting to 1½ kilobytes, so there is very little danger of running out of space.

- **Less fingerwork.** INPUT is by far the most commonly used key function, because more often than not an instruction has to be stored in the memory location following the one currently on display. Thus, most of the typing involves entering a consecutive series of labels and instructions. Fortunately, PME avoids the need to depress the INPUT key for every single instruction and label, as it will accept an automatic INPUT whenever numeric data (Ø . . . F) is entered, unless another key function was chosen which also requires numeric data to be typed in (such as INSERT or SEARCH). The automatic INPUT mechanism saves a considerable amount of typing.

Introducing PME

In the original editor program, the display alternates between showing which key was depressed and the reaction by the Junior Computer. What is missing is a 'bird's eye view' of what is going on. As readers will remember from Book 3, a video screen allows far more information to be packed into a single line and a total of sixteen lines to be displayed at the same time. Thus, the operator has a clear idea of the complete programming process. The 'hardware echo' feature inherent to the Elekterminal was also mentioned in Book 3. The echo enables any key that is depressed to be displayed on the screen or printed on paper automatically – without requiring a single byte of software.

Two columns can be printed on the video screen or via a printer. One represents the key operations by the user and the other expresses the reactions by the Junior Computer. Thus, one column tells the operator what he/she did and the other prints the computer's response.

The left-hand column is reserved for the reaction by the computer and the right-hand one for the action by the user, the latter being on the line above

the former. (As will be seen later on, labels form an exception to this rule as they are printed across the full width of the screen.)

Computer messages During the discussion on PM several computer messages such as 'JUNIOR' or 'WHAT?' were introduced. Similarly, PME reports back providing either general information or error indications. **The error reports are specified,** so that the user knows exactly what went-wrong.

The addresses are also shown.

Now that there is plenty of printing space, the computer might just as well go the whole way and print each instruction along with the address containing its opcode. This is a very informative addition, as it enables the user to keep track of how much memory has been used at any one time and (more importantly?) how much memory is still available for the rest of the program.

And much more . . . A faster and more convenient method of defining BEGAD and ENDAD, four start addresses instead of two, many more key functions, faster assembly, address information about labels etc. Sounds intriguing? Read on for a closer acquaintance with PME!

PME: a thorough survey

A. General points

The PME system program is stored inside the PM program EPROM. This is IC5 on the interface board. The PME program occupies memory locations 14F8 . . . 17FD. Use is made of several of the PM subroutines, which means that **PME will have to be started by way of PM (choice of four start addresses).** If it were to be activated by way of the monitor routine, the input/output (I/O) parameters would not be correctly defined.

B. Cold start entry

Start address: $ 15ØØ

As in the case of the original editor program, the cold start entry into PME occurs at the very beginning of the editing process. In other words, this does not refer to a user program that has already been edited once, and perhaps even assembled, and that needs to be edited again (there are other start adresses for this, which will be dealt with later).

The procedure is to start up PM first:

RST 1 Ø Ø Ø GO RUBOUT (= RES)

which brings us into PM. Now for the cold start entry into PME:

1 5 Ø Ø SP R (SP = SPace bar, not 'S' followed by 'R').

That starts the PME routine and the computer answers:

BEGAD, ENDAD

As readers may have already guessed, this indicates that the first address, BEGAD, and the last address, ENDAD, have to be entered to define the memory range in which the program is to be stored. A cold start entry always involves RAM.

Now the process is similar to that of the M key in PM (hex dump listing). First the address corresponding to BEGAD is entered, then the comma and then ENDAD. Finally, CR (Carriage Return) is pressed. Again, leading zeros do not have to be included. For example, the popular address range Ø2ØØ . . . Ø3FF becomes:

BEGAD, ENDAD: 2ØØ, 3FF CR
PM EDITOR
Ø2ØØ 77

When the text 'PM EDITOR' appears on the screen, this means that instructions and labels can now be entered. The '77' at BEGAD is identical to the one in the original editor. It is none other that the EOF character which moves up the address range as instructions and labels are typed in. Not surprisingly, the first instruction after the cold start entry has to be entered with the aid of the INSERT key: I (see point G, number 5).

All in all, the PME cold start entry is a lot faster than its original counterpart, as all the business concerning the high and low order bytes of the start and end addresses has been discarded.

N.B. If ENDAD turns out to be lower than BEGAD, the computer will not report the error, unlike the M function in PM. It stands to reason that ENDAD must be higher than BEGAD and therefore lower down in the memory map.

C. Warm start entry

Start address: $ 1533

The warm start entry procedure will also be familiar from the original editor. This allows the computer to return to the editor, during which the following address pointers will be defined in accordance with the initial editing process:

BEGAD: BEGin ADdress pointer
ENDAD: END ADdress pointer
CEND: Current END address pointer
CURAD: CURrent ADdress pointer. This is used to point to the address location containing the opcode of the instruction currently on display. In PME it points to the address location containing the opcode of the last instruction to be printed by PME in the left-hand column on the video screen or printer.

After the start address 1533 has been entered (via PM!) and the R key has been depressed, the computer reports back by way of PME:

PM EDITOR
XXXX YY YY YY

Here XXXX represents the contents of CURAD and YY YY YY the instruction or label whose opcode is situated at the CURAD address. Depending on the length of the instruction, only two, four, or all six characters will be displayed. As opposed to the cold start entry (see B) the EOF character 77 will not be stored at location BEGAD.

D. Lukewarm start entry

Start address: $ 1667

This is new. As its name suggests, it is half way between 'cold' and 'warm'. Rather like a cup of tea that has been standing for a few minutes. It is like the 'cold start' in that first BEGAD and ENDAD have to be entered and the first instruction (with its opcode situated at BEGAD) is displayed. On the other hand, it is also similar to the 'warm start', as the EOF character is not stored at BEGAD. During a cold start entry, the current end address pointer (CEND) initially points to an address location which is one higher than BEGAD. However, during a lukewarm start entry, the contents of the CEND pointer are made equal to those of ENDAD.
The procedure is as follows:

```
1 6 6 7 SP R
BEGAD, ENDAD: 1 C Ø Ø, 1 F F F CR
PM EDITOR
1CØØ  85
```

In the given example BEGAD was defined as 1CØØ and ENDAD as 1FFF. That's funny . . . isn't that the address range of the original monitor EPROM? It certainly is, for the lukewarm start entry is especially designed to examine ready-made programs, whether they are located in RAM or EPROM, and the key functions SP, Z, L, P and S (see point G) play a significant role in this process. Thus, this type of entry into the PM Editor is of vital importance during the learning process of the apprentice programmer, as he/she can now analyse his/her own programs, bought or borrowed software, system programs stored in EPROM, listings, etc.

E. The warm CEND start entry

Start address: $ 17C5

Chapter 11 of Book 3 discussed how user programs, including labels, could be stored on cassette tape to great advantage. One method was to use the TM program (which was designed specifically for this purpose), but there is another way involving PM. In the latter case, not only must an identifier (ID) be entered, but also a start address (SA) corresponding to BEGAD and an end address (EA) corresponding to CEND. The address pointed to by CEND can be found quite easily with PME, as this is equal to the address that is one location higher than the one containing the EOF character 77.
Right, imagine that a program is stored on cassette and we wish to 'add the finishing touches' with the aid of PME. First of all, the program will have to be re-entered from tape. This is done by way of PM (key G). Next, PME is activated by way of a warm start entry. This means that BEGAD, ENDAD and CEND will each have to have a value which corresponds to those of the program that has just been retrieved. In addition, the CURAD pointer should also have a suitable value. The complete procedure was described in chapter 11 (Book 3). Quite a number of addresses had to be noted down, etc., which proved to be rather long-winded.

The solution is to activate PME by way of a warm CEND start entry. Obviously, both ID and BEGAD will have to be known factors, but ENDAD can be readjusted to allow extra instructions and labels to be inserted into the program. The end address pointer also needs to be modified if several edited programs are to be entered from·tape as one continuous sequence (remove all EOFs) with the aid of the 'ID = FF trick'.
What happens after the warm CEND start entry of PME? Well, once the pointers BEGAD and ENDAD have been defined (as in the cold and lukewarm start entries), PME searches for the memory location containing the **pseudo-opcode,** 77. As soon as this has been found, the address containing this value is incremented by one and the CEND pointer is then readjusted so that its contents are equal to this new address. The computer reports:

PM EDITOR
XXXX 77

and CEND is pointing to address location XXXX + 1.
The warm CEND start entry can be implemented to re-edit programs (including labels) which were stored on cassette, where SA = BEGAD and EA = CEND. This is the prime objective of this method of entry into PME. However, it also serves other purposes. For one thing, **it can be used to check whether the EOF character 77 is still included in the program once it has been assembled.** After assembly, in other words, after all the labels have been removed, 77 will still act as the EOF character. It will now be held in the memory location immediately following the one containing the last instruction in the program. If for any reason an assembled program needs to be re-edited (labels can always be replaced if necessary!) the warm CEND start entry into PME can be employed, provided the final EOF character was included in the version stored on tape. Then it is merely a question of selecting a location for EA (End Address) that is one greater than the value of the address containing the last instruction in the program. This is accomplished automatically in TM when the SEF key is depressed.
For a warm CEND start entry into PME to take place, therefore, the program will have to include an EOF character. If not, the text 'PM EDITOR' will not appear on the screen and the program will 'crash'. If this does happen, depress the RST key, start up PM and choose another method of entry into PME. A practical example of this type of start entry is provided further on in this chapter (example 5).

F. The BREAK key

Interrupting the printout . . .

The BREAK key was mentioned during the discussion on PM. It was used to interrupt the printing process of fairly long texts. Thus, it acted as a kind of emergency brake. A similar thing occurs during PME as well. Whereas in PM it often proved necessary to interrupt the function provided by the M key, which prints out the hex dump of a program, it is the L key in the case of PME (see point G, number 8) which gives cause for such a measure to be taken. The L key allows the programmer to examine

a particular program at high speed. When a program section appears which requires closer scrutiny, the user simply depresses the BREAK key and then the P key once or twice (see point G, number 9). After this, the computer will report 'PM EDITOR'.
N.B. If PME is activated by way of a cold, lukewarm or warm CEND start entry, the BREAK jump vector will still be defined by PM. If the computer is interrupted while printing 'BEGAD, ENDAD' by way of the BREAK key, the text 'JUNIOR' will appear on the video screen.

G. The key functions of PME

1. SP (space bar) (SKIP)

The 'increment instruction' key

This key is well known to us from its use in the original editor program. Suppose that PME has printed an instruction in the left-hand column of the video screen (or the printer). This will include the address containing the opcode of the instruction. The cursor of the Elekterminal (or the carriage of the printer) will be 'pointing' to the first position of the right-hand column. It will be on the same line as the last instruction mentioned. By depressing the space bar, SP, nothing will appear in the right-hand column which is devoted to displaying key operations, as the SP key is an 'invisible' command. But something does happen, for the next instruction in the program, or the next label (FF XX ØØ) is printed on the following line in the left-hand column. The length of the previous instruction can therefore be calculated from the address of the opcode of the following instruction or label. This address will be one, two or three locations higher than its predecessor.
Once all the instructions in a program have been run through, that is to say, repeatedly depressing the SP key has brought us to the EOF character 77, SP is operated once more, thus:

```
XXXX 77  SP
DONE
YYYY (ZZ ZZ ZZ)
```

Before the new instruction is printed, the computer reports 'DONE' to indicate that the CEND pointer has been passed. The address YYYY is one location higher than XXXX, because the 'instruction' having an opcode 77 (the EOF character) is assigned a length of one byte. The instruction 'ZZ ZZ ZZ' is more than likely quite meaningless. It depends on what happened beforehand: the editor may well have been activated by means of a cold start entry. In any case, the instruction 'ZZ ZZ ZZ' is not part of the program that the user is currently working on. The number of 'Zs' depends on the length of the real/imaginary instruction. From the description of the subroutine OPLEN/LENACC (chapter 8 in Book 2) we know that any combination of two hexadecimal numbers is assigned an opcode.

N.B. If the space bar is depressed several times after skipping the EOF character, the following 'instruction' will be printed after the message

'DONE'. Thus the PME program does not 'disable' the process, but the repeated 'DONE' messages politely remind the user that it is unnecessary to continue.

2. Z (BACKSPACE)

The 'decrement instruction' key

One great advantage of the Printer Monitor as compared to the original monitor was the fact that it implemented a minus key in addition to the plus key. Similarly, as opposed to the original editor, PME has a DECREMENT instruction key in addition to the SKIP key (INCREMENT instruction key). As an example:

```
Ø2AC  FF  17  ØØ     Z       (label 17)
Ø2AB  CA             Z       (DEX)
Ø2A9  A9  FF         Z       (LDA # FF)
Ø2A6  2Ø  14  ØØ     etc.    (JSR-label 14)
```

If the Z key is depressed often enough, the instruction or label situated at the BEGAD address will be reached, and depressing the Z key once more will cause the first instruction or label to be printed again. After all, there can not be an address below BEGAD!

3. K (DELETE)

Remove the label or instruction currently on display

This is a familiar operation. Pressing K causes the last instruction printed to be deleted from memory, or rather, overwritten by the one immediately following it. In other words, depending on the length of that instruction, the program will be made shorter by one, two or three bytes. The following instruction is then printed. If the last instruction of a program is deleted, the address CEND minus one and its corresponding data (77) will be displayed. For example:

```
Ø2A6  2Ø  14  ØØ     SP
Ø2A9  A9  FF         K
Ø2A9  CA             SP
Ø2AA  FF  17  ØØ     SP
Ø2AD  77             SP
DONE
Ø2AE  XX  XX  XX
```

As can be seen, the instruction CA and the label FF 17 ØØ have moved down two locations in the address range and the instruction A9 FF has been removed from the program.

4. T (TOP OF FILE)

Back to the beginning

It can not be said that the operation performed by the T key is particularly spectacular, but it can come in very useful. When the T key is depressed the instruction or label contained at the first address of the program (BEGAD) is printed. This comes in handy when the program is to be checked via the SP, L or P keys. For example:

```
Ø2AE  XX XX XX    T
Ø2ØØ  FF 1Ø  ØØ
```

From this it can be seen that the contents of BEGAD are equal to Ø2ØØ in this example and that the program starts with the label 1Ø. With the original editor program the start adress would be reached by using either SEARCH FF 1Ø or by skipping through the program towards it. The PM Editor introduces yet another method in the form of 'S FF 1Ø ØØ' (see point G, numbers 6 and 7), but simply depressing the T key saves a lot of work – which is just fine for those of us who do not relish a lot of typing!

5. I (INSERT)

Inserting an instruction or a label

This key function is by no means new, as even the original editor makes use of this operation. Depressing the I key followed by the numeric data (belonging to the instruction or label to be inserted) causes the new instruction or label to be placed at a location in RAM which is immediately in front of the location(s) occupied by the previous instruction to be displayed. This only happens **after the required number of numeric keys are depressed that correspond to the length of the particular instruction to be inserted.** The previous instruction and its successor(s) are shifted higher up in the memory range accordingly.

If after depressing I and **before** the instruction was fully entered (= press two, four or six numeric keys), a different key is depressed (i.e. not Ø . . . 9 or A . . . F) the computer will report the error:

ILLEGAL KEY

and the 'last' instruction to be displayed is printed once again. The user may well have pressed a non-valid key quite by accident, but it could also have been deliberate. For instance, the programmer may have realised that the data entered was incorrect. The user must then start from scratch by pressing I, since it is the INSERT function that is involved – not the INPUT function.

Again, it is time for an example. Supposing the situation is like that in point G number 3 where the instruction A9 FF is to be deleted. The CA instruction will then be re-located at address Ø2A9. We wish to replace A9 FF by A9 ØØ and therefore delete A9 FF. This is accomplished by depressing K. Then the keys I, A, 9, Ø and Ø are pressed and the following happens:

```
Ø2A9  CA            I  A  9  Ø  Ø
Ø2A9  A9  ØØ        SP
Ø2AB  CA            SP
Ø2AC  FF  17  ØØ
```

As can be seen, all the instructions following the LDA immediate instruction have moved up two places in memory. Did you notice that PME ensures that the bytes in an instruction are separated by a space when they are displayed? This occurs in both the 'user action' column and the 'computer reaction' column in order to give a much clearer representation.

Together with the INPUT key function (see number 10), the INSERT function allows instructions and labels to be typed in to the computer. In both instances, the memory range reserved for the program will be extended. This ranges from the address corresponding to BEGAD up to and including the address containing the EOF character 77, in other words, from BEGAD to CEND.

Obviously, the available memory space will not be infinitely large. After all, an ENDAD has been defined for a specific purpose. The end address has no real value in the original editor mode, but it does have importance in the assembler and in PME. We are now going to discuss an error message which is displayed whenever the available (= specified) memory range is exceeded. This is quite a new feature.

As mentioned earlier, the assembler described in chapters 5 and 9 in Book 2 is also used for programs which are entered into the computer by way of PME. During the first phase of assembly each label (opcode = FF) that is found is stored on the symbol stack, along with the corresponding address, and then deleted from the program. Thus, for every label that is removed the program is shortened by three bytes. Before the first label can be deleted it will have to be overwritten. Provided there is sufficient room for it, there will automatically be room for the remaining labels in the program.

The first label must be shifted to location ENDAD and the two memory locations below that (lower address, but higher up in the memory map). These locations may not be used to store instructions or labels belonging to the program that is to be edited, nor for the EOF character, 77. This means that the position of the CEND pointer, as shown in figure 1, is the lowest possible one. CEND will then be pointing to the highest possible address, its contents being at a maximum level.

(By the way, the minimum number of vacant memory locations mentioned on page 134 and page 181 in Book 2 is too large. Only four locations have to be reserved: three for the first label and one for the EOF character. The address ENDAD is taken into account when the total number of memory locations is determined).

Back to the space check. As soon as an instruction or a label has been entered – by way of an INSERT or an INPUT operation – PME checks to see whether the corresponding rise of the CEND pointer (since the program is increasing in length) does not lead to a new CEND position which is one location higher than that shown in figure 1. If this is so, the computer reports back with the text 'FULL'. The last instruction to be entered is not stored in memory as there is no room for it. However, instructions can still be printed after the 'FULL' message. In the case of an INSERT operation, this will be the last instruction to be displayed before the new one was entered via the I key – and 'rejected'. In other words, the current address pointer, CURAD, remains unchanged. If, on the other hand, the INPUT function is used to enter an instruction and this is rejected, CURAD will rise by the length of the last instruction to be displayed. The full details of the difference between the INPUT and

1

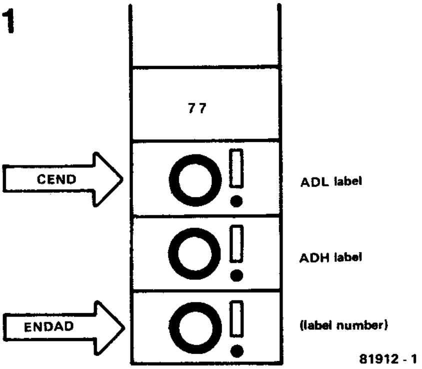


Figure 1. The 'lower' four memory locations – the top four addresses – of the memory range defined by the pointers BEGAD and ENDAD can not be used to edit programs. As soon as the contents of CEND are less than those of ENDAD minus two, PME will report 'FULL'. The traffic sign in the lower three memory locations only refers to the situation during the editing process. After the first phase of assembly the first label encountered is stored in these memory locations (in the order: label number, ADH and ADL).

INSERT functions are shown in table 2.
N.B. After an entered instruction has been rejected due to lack of space, the current end address pointer, CEND, will be raised according to the length of the rejected instruction. The EOF character, 77, will however, remain where it is!

6. S (SEARCH)

7. Y (YES)

As you will probably remember from the original editor program, the SEARCH function works as follows. Two bytes are typed in, which involves depressing four numeric keys. The two bytes may represent either a double-byte instruction or the first two bytes of a triple-byte instruction or label. When the computer is searching for a particular two-byte pattern, starting at BEGAD, it stops as soon as that pattern is found. In other words, there is no way of checking whether the same pattern is repeated anywhere else in the program. Imagine instructions such as A9 ØØ, for instance. Single-byte instructions could not be tracked down at all and triple-byte instructions only had a 50% chance of being found, since the computer only looks at the first two bytes.
It is high time this situation was remedied. The search function in the PM Editor has two important new features.
Firstly: rather than working to a double-byte pattern, any length of instruction (one, two or three bytes) can now be searched for. Thus, the user is able to examine a particular program for any instruction he/she likes.

Secondly: The operator can now check whether a specific instruction crops up more than once in the memory range defined by the pointers BEGAD and CEND. Not only can we track down any instruction or label, but also all the addresses at which the same instruction crops up on separate occasions (it is important to ensure that each label only appears once!).

What is the procedure?

a. Firstly, key S is depressed. This is printed in the right-hand column of the display.

b. Next, the instruction we wish to find is entered. This is done by depressing two, four or six hexadecimal keys. The number of keys depressed will, of course, correspond to the length of the instruction or label which is to be traced. Only after the complete instruction has been typed in will PME start looking for it. This is printed after the S in the right-hand column. The PM Editor produces spaces between the bytes. If a non-numeric key is depressed after the S key, the computer will report **ILLEGAL KEY** and the SEARCH operation will have to be resumed from scratch, starting with the depression of the S key. An illegal key may be operated by the user deliberately if he/she realises that they are entering the wrong instruction.

c. The instruction has now been entered and PME will begin to look for it starting at the lowest address in the memory range, BEGAD. One of two things may now happen: either the sought instruction is indeed present inside the memory range defined by BEGAD and CEND and the computer reacts by printing the instruction and its address on the left-hand side of the screen; **or**, the required instruction is not found in that particular memory range whereupon the computer will report:

DONE
XXXX ZZ ZZ ZZ

Here XXXX stands for the address reached after the CEND pointer contents have been passed and ZZ ZZ ZZ represents the instruction located at that address (this does not necessarily have to be three bytes long, as the number of Zs may suggest).

d. The following key operations are only relevant if an instruction was found during point c. This leads to two different possibilities:

I. The operator wishes to check whether the instruction appears elsewhere in the memory range being examined. **This involves depressing the Y key**. This is again displayed in the right-hand column. PME can now report back to the user in one of two ways. If the sought after instruction is in fact duplicated it will be displayed, along with its address, in the left-hand column. As to be expected, this address will be different from the one corresponding to the first 'discovery' (see point c); in fact it will be higher up the address range. Alternatively, the instruction concerned may well only appear once in the program, in which case the computer will report:

DONE
XXXX ZZ ZZ ZZ

(see point c).

II. The operator does **not** wish to check whether the instruction appears anywhere else. In this instance, any key on the ASCII keyboard which will generate an ASCII code, except for the Y key, can be depressed. The computer will once again reply with:

DONE

XXXX ZZ ZZ ZZ

where ZZ ZZ ZZ represents the instruction found in point c. and XXXX represents the address where it was found.

In practice, the SEARCH feature involves a few important considerations:

- The SEARCH for an instruction can be interrupted at any moment. Since the instruction concerned, together with its location (address), is printed after the 'DONE' report, we can see instantly whether or not we need to continue the SEARCH.
- The SEARCH routine will not be complete until both the 'DONE' report and the instruction concerned have been output by the computer.
- The Y key acquires importance only after the S key has been depressed and the instruction to be searched for has been entered in full. If the Y key is operated before the S key or before the instruction has been typed in, the computer will report 'ILLEGAL KEY'.
- When a SEARCH action is interrupted, all the keys used by the PM Editor, except the Y key, become fully operative once more and so no 'ILLEGAL KEY' message will appear on the screen. Thus the original functions of the keys (including 'S' but excluding 'Y') are ignored while a SEARCH operation is in progress. Other keys which normally have no function at all, and so automatically cause an 'ILLEGAL KEY' message, do have a purpose once the S key has been depressed and the instruction has been found in at least one position in memory.

All this does sound rather complicated, but will be explained in full detail in chapter 15, which is devoted to the description of the PME software.

It is now high time for a practical example. This will make use of the computer printout listed in table 1. As can be seen in table 1, the PM Editor has been activated by way of a lukewarm start entry. The address pointers BEGAD and ENDAD have been chosen to select the full memory range occupied by the original monitor EPROM on the main board of the Junior Computer. After the Carriage Return key (CR) has been operated (not shown in table 1), the computer outputs the text 'PM EDITOR' and the first address and the instruction contained therein is printed: 1CØØ 85 F3.

Let us see what happens when the instruction STAZ-POINTL is searched for (this is accomplished by depressing the keys: – S, 8, 5, F and A. Do not try to include spaces between the various characters – this must be left to PME, as otherwise the text 'ILLEGAL KEY' will be displayed and you will have to start all over again!

Well, the operator knows that this instruction must be contained in the program, because memory location ØØFA constitutes the much used display buffer POINTL. Not surprisingly, the computer comes across the required instruction almost immediately at address location 1CØ8.

By depressing the Y key eight times in succession the computer will output another seven address locations where the particular instruction can be found. Note that as we are moving up the address range, each

Table 1. An example of the warm start entry into PME and how the SEARCH function (key S) operates.

```
JUNIOR
1667
1667 2Ø R
BEGAD,ENDAD: 1CØØ,1FFF
PM EDITOR
1CØØ 85 F3         S85 FA
1CØ8 85 FA         Y
1C83 85 FA         Y
1CBØ 85 FA         Y
1CDD 85 FA         Y
1D3D 85 FA         Y
1E37 85 FA         Y
1FDA 85 FA         Y
1FEA 85 FA         Y
DONE
2ØØØ ØA            SC8
1CBF C8            Y
1CEA C8            Y
1D55 C8            Y
1EØE C8            Y
1E56 C8            Y
1F7Ø C8            Y
1FB6 C8            Y
1FBF C8            Y
1FC4 C8            Y
DONE
2ØØØ ØA            T
1CØØ 85 F3         P
1CØ2 68
1CØ3 85 F1
1CØ5 68
1CØ6 85 EF
1CØ8 85 FA
1CØA 68
1CØB 85 FØ
1CØD 85 FB
1CØF 84 F4
1C11 86 F5
1C13 BA
1C14 86 F2
1C16 A2 Ø1
1C18 86 FF
1C1A 4C 33 1C      S8D 83 1A
1C1F 8D 83 1A      Y
DONE
2ØØØ ØA            S8E 83 1A
DONE
2ØØØ ØA            S8C 83 1A
DONE
2ØØØ ØA
```

'discovery' will be situated at a higher address location. After the Y key has been depressed for the eighth time, all the locations containing the searched for instruction will have been found and the computer will acknowledge this fact by displaying the text 'DONE'. This is followed by an instruction appearing at address location 2ØØØ. This is because the lukewarm start entry made the contents of the CEND pointer equal to those of ENDAD (= 1FFF).

Next in table 1, the instruction C8 is searched for. This is the opcode for the instruction INY and appears no less than nine times in the monitor program. By the way, the address locations containing the above mentioned instructions (STAZ-POINTL and INY) can be checked with the aid of the source listing of the monitor program which can be found on pages 194 . . . 203 of Book 2.

Let us continue with table 1. After depressing the T key, address location 1CØØ is displayed along with the instruction contained therein. The effect of the P key is to print out a section of the contents of memory and will be elaborated on later (see point G, number 9).

We will end the description of table 1 with a small piece of research. We wish to know where and how the port B data direction register is affected inside the original monitor EPROM. The address of PBDD is $ 1A83. This register must be affected by a store instruction. There are only three possibilities:

```
8D 83 1A   STA-$ 1A83
8E 83 1A   STX-$ 1A83
8C 83 1A   STY-$ 1A83
```

First, we shall check to see whether the instruction STA-$ 1A83 appears anywhere in memory. As can be seen, it does, once, at location 1C1F. However, the other two instructions can not be found anywhere. Thus, the only place where PBDD is affected is at an address which is part of the RESET initialisation routine in the original monitor program.

Now for the next function utilised by the PM Editor.

8. L (LIST)

Printing the program

The Printer Monitor program introduced various print commands, such as the hex dump. Up until now, we have been concerned with only those PME functions which allow a single instruction to be printed, sometimes followed by a message on the part of the computer. Key functions L and P (see point G, number 9) on the other hand, enable several instructions to be printed in sequence.

Depressing the L key results in a printout of all the instructions and labels contained within the particular memory range defined by the pointers BEGAD and CEND, starting with the first instruction or label situated at BEGAD. They are listed in the left-hand column of the video screen or printer. As soon as all the instructions have been output (the current address pointer CURAD is updated as each new instruction is printed), the computer reports the situation with the message 'DONE'. Then the contents of the CURAD pointer (and the corresponding instruction) after it has 'passed' the value of CEND are output. The LIST function enables the programmer to give a particular program a quick 'once-over' to see whether everything is correct. If the user wishes to examine any section of program at leisure, the BREAK key should be depressed (the computer will report 'PM EDITOR') followed by the P key – for reasons we shall describe now.

9. P (PRINT)

Printing program blocks

This key performs a very similar operation to that of the L key. Supposing an instruction has just been displayed by PME. Exactly how is not relevant here. This instruction is displayed in the left-hand column and then the P key is operated. This appears on the same line, but in the right-hand column. Next, the following 15 instructions are output one below the other, again in the left-hand column. Therefore, a total of sixteen address locations and corresponding instructions, including the initial instruction, are displayed. In the case of the Elekterminal, this will exactly fill the video screen.

Obviously, the sixteen instructions must not contain the EOF character, 77, as this will cause the computer to halt the printing process as soon as it has been displayed.

N.B. An example of the operation performed by the P key is shown in table 1.

10. INPUT

Keys Ø . . . 9 and A . . . F

As mentioned earlier, the INPUT function is carried out automatically. In other words, no function key has to be depressed before inputting data by means of the numeric keys. The only other key functions which require the entry of numeric data are INSERT (I) and SEARCH (S). This means that as far as these operations are concerned, either the I key or the S key has to be depressed before the numeric data can be entered.

The PM Editor may well be waiting for data keys to be depressed after carrying out a particular function (this can be ascertained from the position of the cursor on the video screen or the position of the printer carriage. Both should be situated at the start of the right-hand-column. If, after this, the required numeric keys are depressed, the INPUT function will be activated automatically.

Let us briefly recap on the operation of the INPUT function. This function stores the instruction or label entered by the operator in the memory location(s) immediately following the one(s) containing the last instruction (printed in the right-hand column). In practice, the INPUT function is very similar to the INSERT function in some respects, but very different in others.

Again, the instruction or label is not stored in memory until its entry is complete. Whenever a non-numeric key is depressed, either by mistake or on purpose, the computer will report 'ILLEGAL KEY'. This means having to re-enter the complete instruction. As before, the instruction and its address are printed in the left-hand column of the display on the following line and again the PM Editor provides the spaces between the instruction bytes in both columns. Therefore, do not try to add spaces as this will only lead to an 'ILLEGAL KEY' message!

Since the INPUT function leads to a new instruction to be added to the existing sequence, the EOF character and the contents of the CEND pointer move up a higher address location. The number of locations they move depends, of course, on the length of the instruction added.

As the memory range defined by the pointers BEGAD and CEND is therefore extended, the chances are that the available memory will fill up after a while. This is illustrated in figure 1 and explained in the text describing the INSERT function. If there is no more room for any more instructions the computer will report:

FULL
XXXX ZZ ZZ ZZ

where the address XXXX and the instruction ZZ ZZ ZZ correspond to the contents of CURAD being incremented as a result of the 'rejected' instruction. This is different to the INSERT function, where the address XXXX and the instruction ZZ ZZ ZZ belong to the last instruction printed before the rejected instruction was entered.

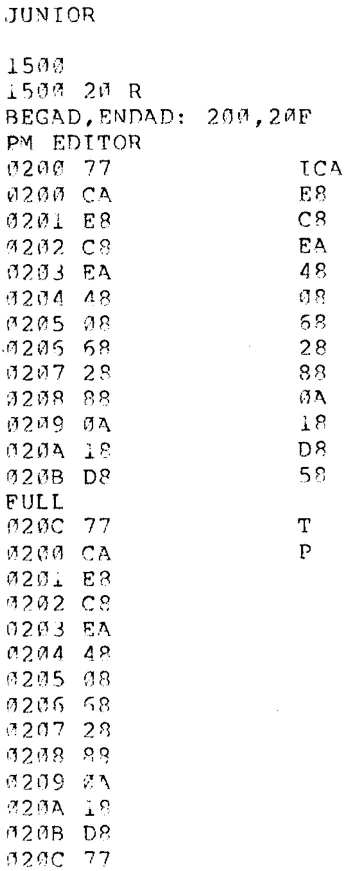
In spite of the fact that the instruction is rejected by the computer because of lack of memory, the current end address pointer (CEND) is incremented by the corresponding length of the rejected instruction. As before, the EOF character (77) remains where it is.

The two listings in table 2 provide a practical example of the difference between INSERT and INPUT. In both cases a range of 17 memory locations is defined (BEGAD, ENDAD) and an imaginary program is involved consisting of single byte instructions only. In the left-hand listing the instructions are entered by means of the INPUT function – apart from the initial instruction which has to be entered with the aid of the INSERT function. The right-hand listing in table 2 shows the same sequence of single-byte instructions, but these are entered by means of the INSERT function only.

In both instances the last instruction will be stored at location Ø2ØB. Neither 'program' has room for the instruction 58 (CLI). Therefore, the last available memory location is Ø2ØB, which proves the rule that the pointer ENDAD must contain a value four locations greater than the length of the program (since $ØF – $ØB is the same as 15 – 11 which equals 4). Address location Ø2ØC contains the EOF character, 77, which is also shown in table 2 following the instructions printed with the aid of the P key. The three locations Ø2ØD, Ø2ØE and Ø2ØF are reserved for the first label during the initial phase of the assembly procedure.

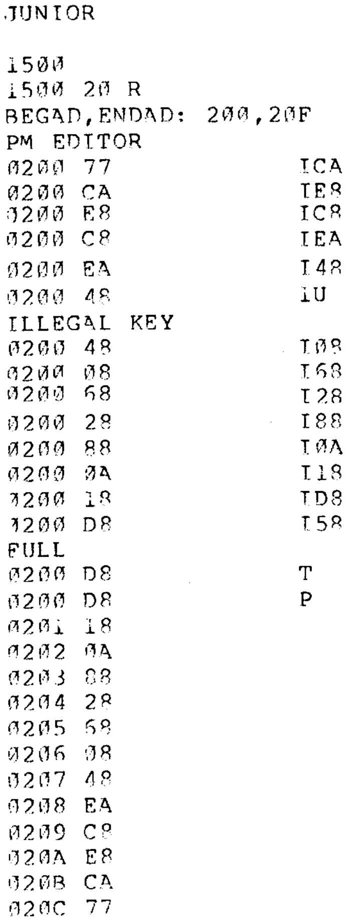
Table 2 also shows another clear difference between the INPUT and INSERT functions as far as the order in which the instructions are actually stored in memory is concerned. To put it in a nutshell: the INPUT and INSERT functions are not synonymous. In fact, the order is the exact opposite. Quite a different sequence of instructions is printed after the text 'FULL'.

Table 2. How to enter an imaginary program containing only single byte instructions using the INPUT function, on the left-hand side, and the INSERT function, on the right-hand side. Note the different order in which the instructions are stored in memory.



```
JUNIOR

15ØØ
15ØØ 2Ø R
BEGAD,ENDAD: 2ØØ,2ØF
PM EDITOR
Ø2ØØ 77           ICA
Ø2ØØ CA           E8
Ø2Ø1 E8           C8
Ø2Ø2 C8           EA
Ø2Ø3 EA           48
Ø2Ø4 48           Ø8
Ø2Ø5 Ø8           68
Ø2Ø6 68           28
Ø2Ø7 28           88
Ø2Ø8 88           ØA
Ø2Ø9 ØA           18
Ø2ØA 18           D8
Ø2ØB D8           58
FULL
Ø2ØC 77           T
Ø2ØØ CA           P
Ø2Ø1 E8
Ø2Ø2 C8
Ø2Ø3 EA
Ø2Ø4 48
Ø2Ø5 Ø8
Ø2Ø6 68
Ø2Ø7 28
Ø2Ø8 88
Ø2Ø9 ØA
Ø2ØA 18
Ø2ØB D8
Ø2ØC 77
```



```
JUNIOR

15ØØ
15ØØ 2Ø R
BEGAD,ENDAD: 2ØØ,2ØF
PM EDITOR
Ø2ØØ 77           ICA
Ø2ØØ CA           IE8
Ø2ØØ E8           IC8
Ø2ØØ C8           IEA
Ø2ØØ EA           I48
Ø2ØØ 48           1U
ILLEGAL KEY
Ø2ØØ 48           IØ8
Ø2ØØ Ø8           I68
Ø2ØØ 68           I28
Ø2ØØ 28           I88
Ø2ØØ 88           IØA
Ø2ØØ ØA           I18
Ø2ØØ 18           ID8
Ø2ØØ D8           I58
FULL
Ø2ØØ D8           T
Ø2ØØ D8           P
Ø2Ø1 18
Ø2Ø2 ØA
Ø2Ø3 88
Ø2Ø4 28
Ø2Ø5 68
Ø2Ø6 Ø8
Ø2Ø7 48
Ø2Ø8 EA
Ø2Ø9 C8
Ø2ØA E8
Ø2ØB CA
Ø2ØC 77
```

Readers should also note that the right-hand column in the second listing in table 2 deliberately called for the use of an 'ILLEGAL KEY'. When Ø8 was being entered a '1' was typed instead of an 'I'. This was remedied by depressing the 'U' key.

11. X (EXECUTE)

Assembly

Like the original editor program, PME operates on the basis of hexadecimal labels. This means that the assembler we already know from Book 2 can be used as it stands to assemble programs edited by PME. In other words, the labels can all be removed from the program once the corresponding address information and label number have been noted down in the label memory space which starts from ENDAD.

Another reason for keeping the original assembler is that it is a very 'discrete' program. During its execution no use is made of the display and the operator does not have to press any keys. It is not until after the assembly process that the display reports its completion.

The start address of the assembler remains the same as it was previously, namely $1F51. Once a program had been edited by means of the original editor program, the RST key had to be depressed, the start address of the assembler routine entered and then the GO key had to be operated. Alternatively, the NMI jump vector could be used to point to the start address of the assembler which could then be started by depressing the ST key. All this is now unnecessary as the PM Editor contains the X function.

Once the X key has been depressed, the computer will jump to the start of the assembler routine after a couple of preparatory measures have been taken. One of these steps involves loading the NMI jump vector with an address inside PME after which the program proceeds to the assembly phase. Some time after the X key has been depressed (exactly how long depends on the length of the program) the seven segment display will light. The program will then be completely assembled. If the ST key on the original keyboards is then depressed, a non-maskable interrupt is enabled which will return the computer to the PM Editor.

What happens next is very interesting indeed . . .

All the labels are output either on the video screen or on paper via the printer! This includes their label numbers and their corresponding addresses. So an imaginary program containing five labels could well look like this:

```
LAB $ 1Ø: $ Ø2ØØ LAB $ 12: Ø2Ø8 LAB $ 14: Ø2ØC LAB $ 13: Ø213
LAB $ 15: Ø22B
PM EDITOR
XXXX   ZZ ZZ ZZ
```

All this is possible as the labels are still stored in memory at the end of the assembly process. Each label requires three memory locations and the label memory range starts at ENDAD, each consecutive label having a lower address.

As can be seen, up to four labels can be printed on a single line. The labels are printed in the order they are discovered, noted down and deleted during the first phase of assembly. Thus, the addresses corresponding to the labels increase as they are printed even though the label numbers may jump around a bit. The reason for this was illustrated in chapter 9 of Book 2, in the section discussing the assembler software. It is entirely up to the programmer as to whether the label numbers and the addresses both increase in regular order. The programmer does not necessarily have to stick to any particular label number order, although it is advisable for the sake of clarity. In the example we have just given, the label numbers appear in the order 1Ø, 12, 14, 13, 15. There is no label number 11, perhaps because the program has been modified since it was first conceived.

Once all the labels have been printed (in this particular instance there are no columns) the computer will report with the text 'PM EDITOR' on a new line. Next, the first address and instruction of the assembled program is printed on the following line, so address XXXX represents the contents of the BEGAD pointer. After assembly and after all the labels have been stored, a jump was made to the warm start entry address of PME.

That covers all the key functions of the PM EDITOR . . . at last! It is now time to deal with the practical aspects of programming the extended version of the Junior Computer. Readers did get a glimpse of what is involved in chapter 12 of Book 3, where certain PM subroutines were experimented with.

Using the PM EDITOR

First, the operator (and the computer) should start by having a warming up session:

1. Eight bit hexadecimal-to-decimal conversion

Pages 9 . . . 25 in chapter 5 of Book 2 were devoted to putting the original editor program into practice with the aid of a concrete example. The example used a routine called DISPLAY (together with various subroutines) which displayed the decimal value of an eight bit hexadecimal number. The number was entered by depressing two hexadecimal keys. The program itself, or rather its algorithm, was not discussed in detail at the time and we do not intend to go into it here either, at least not until point 2, where an extended version will be described. What we are concerned with here is to point out the main differences between the procedures of the original editor program and the PM Editor.

Compare the listing on pages 23 and 24 of Book 2 with the one in table 3. This starts with PM (the computer reports 'JUNIOR'). Next, the PM Editor is activated by means of a cold start entry. The first 'instruction' is label 1Ø and is entered with the aid of the INSERT key. This is necessary because otherwise the EOF character, 77, will remain at address Ø2ØØ. It makes no difference to the current end address pointer whether the first instruction is entered by means of the INPUT or INSERT function; CEND will always move to a higher address. However, if the program is started with the 'opcode' 77 (Ø2ØØ is also the start address!) the operator is asking for trouble!

All the instructions and labels after this are entered by means of the INPUT function, which is merely a question of pressing numeric keys. A few lines further on in table 3 we find a 'K' in the right-hand column and the following instruction is entered by way of the I key. This is because

Table 3. Using the PM Editor to enter the DISPLAY program (see pages 9 . . . 25 in Book 2 and example 1 in this chapter).

```
JUNIOR

15ØØ
15ØØ 2Ø R
BEGAD,ENDAD: 2ØØ,3FF
PM EDITOR
Ø2ØØ 77            IFF 1Ø ØØ
Ø2ØØ FF 1Ø ØØ      A9 ØØ
Ø2Ø3 A9 ØØ         85 F9
Ø2Ø5 85 F9         85 FA
Ø2Ø7 85 FA         85 FB
Ø2Ø9 85 FB         FF 11 ØØ
Ø2ØB FF 11 ØØ      2Ø 6F 1D
Ø2ØE 2Ø 6F 1D      1Ø 1Ø
Ø211 1Ø 1Ø         85 F9
Ø213 85 F9         85 F7
Ø215 85 F7         K
Ø215 77            I85 D7
Ø215 85 D7         2Ø 12 ØØ
Ø217 2Ø 12 ØØ      4C 11 ØØ
Ø21A 4C 11 ØØ      FF 12 ØØ
Ø21D FF 12 ØØ      2Ø 14 ØØ
Ø22Ø 2Ø 14 ØØ      85 FA
Ø223 85 FA         84 D7
Ø225 84 D7         2Ø 14 ØØ
Ø227 2Ø 14 ØØ      A2 Ø4
Ø22A A2 Ø4         FF 13 ØØ
Ø22C FF 13 ØØ      ØA
Ø22F ØA            CA
Ø23Ø CA            DØ 13
Ø231 DØ 13         Ø5 FA
Ø233 Ø5 FA         85 FA
Ø235 85 FA         84 FB
Ø237 84 FB         6Ø
Ø239 6Ø            FF 14 ØØ
Ø23A FF 14 ØØ      AØ ØØ
Ø23D AØ ØØ         84 D8
Ø23F 84 D8         2Ø 15 ØØ
Ø241 2Ø 15 ØØ      18
Ø244 18            A5 D7
Ø245 A5 D7         69 ØA
Ø247 69 ØA         6Ø
Ø249 6Ø            FF 15 ØØ
Ø24A FF 15 ØØ      38
Ø24D 38            A5 D7
Ø24E A5 D7         E9 ØA
Ø25Ø E9 ØA         85 D7
Ø252 85 D7         A5 D8
```

```
0254 A5 D8         E9 00
0256 E9 00         30 16
0258 30 16         C8
025A C8            4C 15 00
025B 4C 15 00      FF 16 00
025E FF 16 00      60
0261 60            X
LAB $10: $0200 LAB $11: $0208 LAB $12: $0217 LAB $13: $0223
LAB $14: $022E LAB $15: $023B LAB $16: $024C
PM EDITOR
0200 A9 00         P
0202 85 F9
0204 85 FA
0206 85 FB
0208 20 6F 1D
020B 10 F3
020D 85 F9
020F 85 D7
0211 20 17 02
0214 4C 08 02
0217 20 2E 02
021A 85 FA
021C 84 D7
021E 20 2E 02
0221 A2 04
0223 0A            P
0224 CA
0225 D0 FC
0227 05 FA
0229 85 FA
022B 84 FB
022D 60
022E A0 00
0230 84 D8
0232 20 3B 02
0235 18
0236 A5 D7
0238 69 0A
023A 60
023B 38
023C A5 D7         P
023E E9 0A
0240 85 D7
0242 A5 D8
0244 E9 00
0246 30 04
0248 C8
0249 4C 3B 02
024C 60
024D 77
```

85 F7 was inadvertently entered instead of 85 D7, the operation of the K key corrects this.

It is a matter of minutes before the entire program is well and truly stored in memory. There is a much quicker way of checking for errors than previously (when using the original editor program). After all, a total of sixteen lines can be displayed at the same time on the Elekterminal and if a suitable printer is available the programmer can run his/her eyes over the complete listing.

The next phase in the operation involves depressing the X key to assemble the program. After this, the display will light and the ST key (STOP, NMI) on the original keyboard can be operated. As a result, PME will display all the labels before reporting 'PM EDITOR' and printing the first instruction of the assembled program.

To obtain a complete listing of the assembled program all that is required is to depress the P key three times in succession (remember to wait while the particular block of data is being printed before depressing the P key the second and third times!). Once the programmer acquires the knack of working with PME, a program such as that given in table 3 can be entered and running in about five minutes. Of course, entering pre-recorded programs from tape is even quicker.

2. Sixteen bit hexadecimal-to-decimal conversion

We now wish to write a program which will convert a sixteen bit hexadecimal value into its decimal equivalent. Each (up to) four-digit hexadecimal figure must be entered and printed at the beginning of a new line. The Junior Computer must be informed when the hexadecimal value has been completely entered by depressing the ':' key. The decimal equivalent must be printed on the same line after this colon. The computer must also be able to deal with 4, 8 or 12 bit hexadecimal numbers. If more than four numeric keys are depressed in a single row by mistake or on purpose, the last four keys to be depressed must be processed. Now let us translate the above into usable software.

The figure 1981 contains a single thousand. Thus, if we subtract two thousand from it we are left with a negative result. Similarly, 1981 contains nine hundreds, eight tens and one unit.

The highest possible 16 bit hexadecimal figure is \$FFFF. This corresponds to the decimal figure 65535. In other words, the highest contribution of the decimal version of the figure is made by the ten thousands. There are six of them. Thus:

$10{,}000_{10}$	corresponds to	\$271Ø
$1{,}000_{10}$	corresponds to	\$Ø3E8
100_{10}	corresponds to	\$ØØ64
10_{10}	corresponds to	\$ØØØA

The procedure is as follows. After a figure has been fully entered, the value \$271Ø (ten thousand) is subtracted from it until the result becomes negative. Then \$271Ø is added to this. The number of times that \$271Ø was subtracted without causing a negative result is recorded in the Y register and is then printed. It is possible, of course, that the number of ten thousands in the decimal figure will be zero.

Next, the same procedure is repeated. but this time the value $Ø3E8 is subtracted from the remainder until the number becomes negative. The new remainder is found by adding $Ø3E8 to the negative result. The number of hundreds and then tens is calculated and printed in the same way. At the end of the procedure, the remainder will be a value between 0 and 9 and is therefore printed without any need for further calculations.

Before describing the actual program, let us look at the PME subroutines which are implemented:

I. **CRLF** Address $11E8. Two consecutive commands to enable data to be printed at the start of a new line.

II. **RECCHA** Address $12AE. This waits for a key to be depressed and then stores the ASCII value in the accumulator.

III. **HEXNUM** Address $126F. This processes any hexadecimal key which is depressed. After the ASCII code is converted into a data nibble, the latter is shifted into the input buffer INL from right to left. The buffers INH and INL contain data corresponding to the last four hexadecimal keys to be depressed. A high order nibble is always 'older' than a low order nibble. If any non-hexadecimal keys are depressed (G . . . Z etc.) the computer reports the error with 'WHAT?'.

IV. **RESIN** Address $1268. This clears the contents of the INH and INL buffers.

V. **PRNIBL** Address $129B. This is the second half of the subroutine PRBYT. It prints the low order data nibble of the contents of the accumulator. For further details, readers are referred to chapter 14.

Now for the program. The main routine is called, appropriately, HEXDEC and is shown in figure 2. To start with, the CRLF routine is used to make sure that the new data is printed at the start of a fresh line and then the computer waits for a key to be depressed. Then the computer checks to see whether the depressed key was the colon or not. If so, the figure must have been entered and the conversion and printing procedure described earlier can begin.

First of all, however, let us see what happens if hexadecimal data is entered instead of the colon. The branch instruction (BNE) leads the processor to the label DATA and to the subroutine HEXNUM. If it was not a hexadecimal key, HEXNUM reports 'WHAT' and resets the Z flag. In that case, the following branch instruction will make the program proceed to the label NEW: the contents of the buffers INH and INL are cleared and the computer returns to the start of the program. We then have to start all over again by typing in a new hexadecimal figure.

If the computer does detect a hexadecimal key, the HEXNUM subroutine makes sure that the corresponding data nibble is stored in the low order nibble of INL. Buffer INH always contains the first two data nibbles to be entered and INL contains the last two of the four hexadecimal keys to be depressed. Thus a high order data nibble is always less recent than a low order one. Unused nibbles are always zero.

What happens when the processor encounters a colon? See the section of program following the left-hand BNE instruction in figure 2. Memory locations POWERL ($ØØØØ) and POWERH ($ØØØ1) are loaded with data related to checking the number of ten thousands, thousands, hundreds and

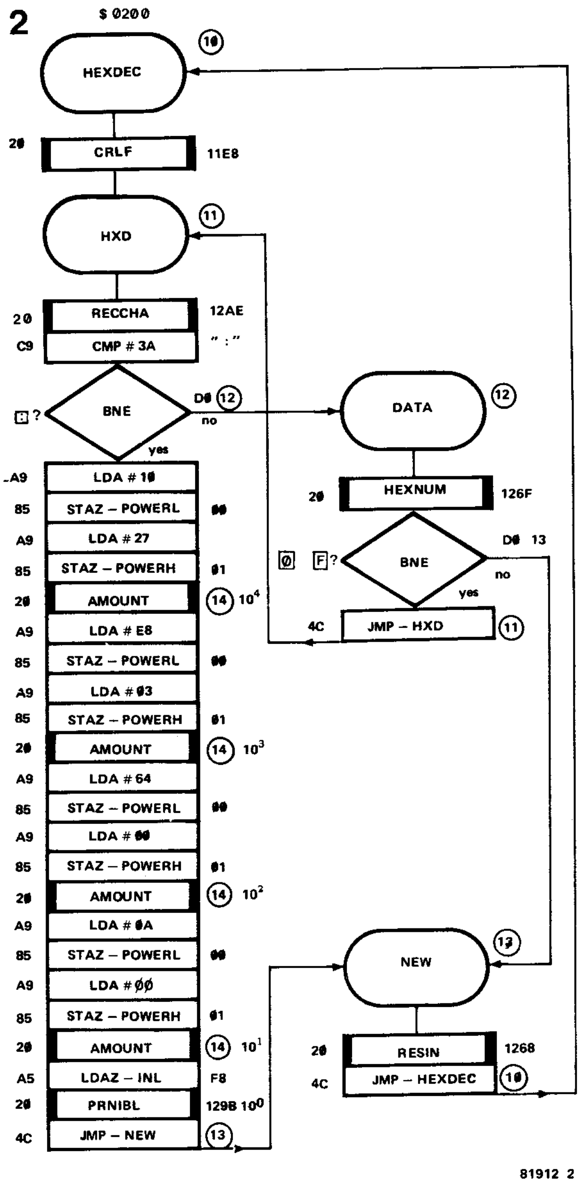


Figure 2. The main routine, HEXDEC, in the program which enables a four digit hexadecimal number to be converted into the corresponding decimal figure and the result to be displayed.

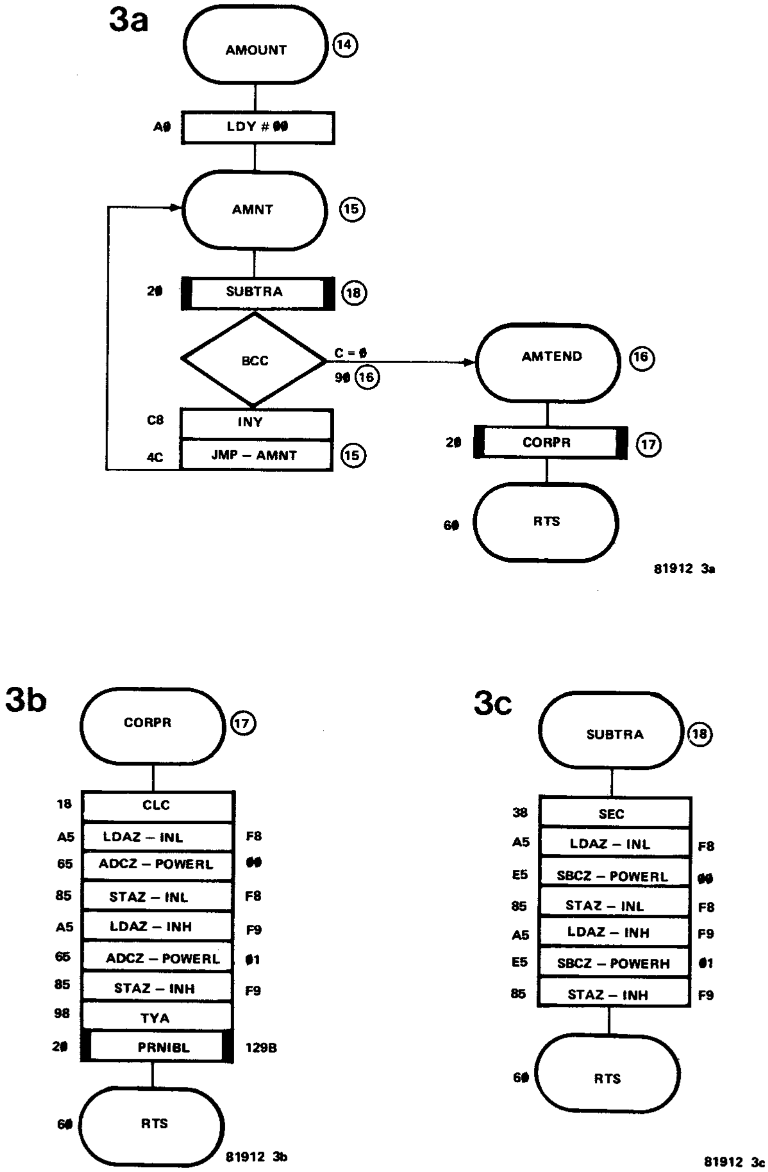


Figure 3. The subroutine AMOUNT in figure 3a determines how many times a certain power of ten appears in the entered hexadecimal number and also sees to it that this figure is printed. The subroutine makes use of two other subroutines, namely CORPR (figure 3b) and SUBTRA (figure 3c).

tens are contained in the input figure four times in succession. Subroutine AMOUNT is also called four times in succession to calculate the number of thousands etc. and to print this. After AMOUNT has been run four times, the various amounts are known and are printed and the operator is left with a remainder – a certain number of units. This remainder is stored in INL and is printed by means of the PRNIBL subroutine. The computer then jumps back to HEXDEC via NEW (clear buffers) to deal with the next hexadecimal figure.

A few more details about subroutine AMOUNT. This is shown in figure 3a. The subroutine begins by clearing the contents of the Y index register. Then subroutine SUBTRA is called after the label AMNT. This subroutine is shown in figure 3c. The value contained in locations POWERH and POWERL is subtracted from the hexadecimal number (INH, INL). If the result of the subtraction is either positive or zero, the carry flag will be logic one. A negative result, on the other hand, will make it go low. The carry business is very easy to remember:

carry = $\overline{\text{borrow}}$ and so borrow = $\overline{\text{carry}}$.

If the result is negative, an amount will have to be borrowed, so the borrow will be logic one and the carry logic zero. If the result is greater than or equal to zero, nothing will have to be borrowed.

If the result is positive, the contents of the Y index register will be incremented (INY) and another subtraction takes place – the program jumps back to the label AMNT. If this subtraction produces a negative result, the computer proceeds to the label AMTEND. The value contained in the Y register corresponds to the number of times the hexadecimal figure was decremented without leading to a negative result.

The rest of the AMOUNT subroutine can be explained in very few words. After the label AMTEND the subroutine CORPR is called (shown in figure 3b). Firstly, the contents of locations POWERH and POWERL are added to those of the buffers INH and INL, as a result of which the buffers contain a value corresponding to the situation where the next/last figure is to be calculated. The contents of the Y register are transferred to the accumulator and the subroutine PRNIBL is called to print the number that has just been calculated.

The routines in figures 2 and 3 are entered into the Junior Computer with the aid of the PM Editor. Table 4 provides a 'hard copy' version of all the key operations required and the corresponding reactions by PME. To start with, PM is activated and PME entered by way of a cold start entry then all the instructions and labels are typed in. The first label is entered with the aid of the INSERT function as always. After pressing the X key the program is assembled. Following this, the ST key on the main board of the Junior Computer is depressed causing all the labels and their addresses to be printed on the video screen or the printer. Then the text 'PM EDITOR' will appear on the screen followed by the start address of the program and the instruction held therein.

The program can now be executed, which means having to leave PME, as this was activated by means of a warm start entry after the business with the labels. Then RST, 1, Ø, Ø, Ø, GO and RUBOUT (=RES) are operated on the main keyboard, in that order, to start PM. This is necessary as the HEXDEC program makes use of certain PM routines. The I/O parameters

Table 4. Using the PM Editor to enter the HEXDEC program and its associated subroutines (see also figures 2 and 3 and example 2).

```
SIXTEEN BIT HEXADECIMAL TO DECIMAL CONVERSION
HEXDEC
JUNIOR

15ØØ
15ØØ 2Ø R
BEGAD,ENDAD: 2ØØ,3FF
PM EDITOR
Ø2ØØ 77                IFF 1Ø ØØ
Ø2ØØ FF 1Ø ØØ          2Ø E8 11
Ø2Ø3 2Ø E8 11          FF 11 ØØ
Ø2Ø6 FF 11 ØØ          2Ø AE 12
Ø2Ø9 2Ø AE 12          C9 3A
Ø2ØC C9 3A             DØ 12
Ø2ØE DØ 12             A9 1Ø
Ø21Ø A9 1Ø             85 ØØ
Ø212 85 ØØ             A9 27
Ø214 A9 27             85 Ø1
Ø216 85 Ø1             2Ø 14 ØØ
Ø218 2Ø 14 ØØ          A9 E8
Ø21B A9 E8             85 ØØ
Ø21D 85 ØØ             A9 Ø3
Ø21F A9 Ø3             85 Ø1
Ø221 85 Ø1             2Ø 14 ØØ
Ø223 2Ø 14 ØØ          A9 64
Ø226 A9 64             85 ØØ
Ø228 85 ØØ             A9 ØØ
Ø22A A9 ØØ             85 Ø1
Ø22C 85 Ø1             2Ø 14 ØØ
Ø22E 2Ø 14 ØØ          A9 ØA
Ø231 A9 ØA             85 ØØ
Ø233 85 ØØ             A9 ØØ
Ø235 A9 ØØ             85 Ø1
Ø237 85 Ø1             2Ø 14 ØØ
Ø239 2Ø 14 ØØ          A5 F8
Ø23C A5 F8             2Ø 9B 12
Ø23E 2Ø 9B 12          4C 13 ØØ
Ø241 4C 13 ØØ          FF 12 ØØ
Ø244 FF 12 ØØ          2Ø 6F 12
Ø247 2Ø 6F 12          DØ 13
Ø24A DØ 13             4C 11 ØØ
Ø24C 4C 11 ØØ          FF 13 ØØ
Ø24F FF 13 ØØ          2Ø 68 12
Ø252 2Ø 68 12          4C 1Ø ØØ
Ø255 4C 1Ø ØØ          FF 14 ØØ
Ø258 FF 14 ØØ          AØ ØØ
Ø25B AØ ØØ             FF 15 ØØ
Ø25D FF 15 ØØ          2Ø 18 ØØ
Ø26Ø 2Ø 18 ØØ          9Ø 16
```

```
Ø263 9Ø 16          C8
Ø265 C8             4C 15 ØØ
Ø266 4C 15 ØØ       FF 16 ØØ
Ø269 FF 16 ØØ       2Ø 17 ØØ
Ø26C 2Ø 17 ØØ       6Ø
Ø26F 6Ø             FF 17 ØØ
Ø27Ø FF 17 ØØ       18
Ø273 18             A5 F8
Ø274 A5 F8          65 ØØ
Ø276 65 ØØ          85 F8
Ø278 85 F8          A5 F9
Ø27A A5 F9          65 Ø1
Ø27C 65 Ø1          85 F9
Ø27E 85 F9          98
Ø28Ø 98             2Ø 9B 12
Ø281 2Ø 9B 12       6Ø
Ø284 6Ø             FF 18 ØØ
Ø285 FF 18 ØØ       38
Ø288 38             A5 F8
Ø289 A5 F8          E5 ØØ
Ø28B E5 ØØ          85 F8
Ø28D 85 F8          A5 F9
Ø28F A5 F9          E5 Ø1
Ø291 E5 Ø1          85 F9
Ø293 85 F9          6Ø
Ø295 6Ø
Ø296 77             X
LAB $1Ø: $Ø2ØØ LAB $11: $Ø2Ø3 LAB $12: $Ø23E LAB $13: $Ø246
LAB $14: $Ø24C LAB $15: $Ø24E LAB $16: $Ø257 LAB $17: $Ø25B
LAB $18: $Ø26D
PM EDITOR
Ø2ØØ 2Ø E8 11
JUNIOR

2ØØ
Ø2ØØ 2Ø R
FFFF:65535
FFF:Ø4Ø95
FF:ØØ255
F:ØØØ15
ABCD:43981
1ØØØ:Ø4Ø96
T
WHAT?

CF16:53Ø14
14F8:Ø5368
17FF:Ø6143
1Ø24:Ø4132
271Ø:1ØØØØ
Ø3E8:Ø1ØØØ
ØØ64:ØØ1ØØ
```

```
000A:00010
:00000

JUNIOR

153D
153D A0 R
PM EDITOR
0200 20 E8 11        P
0203 20 AE 12
0206 C9 3A
0208 D0 34
020A A9 10
020C 85 00
020E A9 27
0210 85 01
0212 20 4C 02
0215 A9 E8
0217 85 00
0219 A9 03
021B 85 01
021D 20 4C 02
0220 A9 64
0222 85 00           P
0224 A9 00
0226 85 01
0228 20 4C 02
022B A9 0A
022D 85 00
022F A9 00
0231 85 01
0233 20 4C 02
0236 A5 F8
0238 20 9B 12
023B 4C 46 02
023E 20 6F 12
0241 D0 03
0243 4C 03 02
0246 20 68 12        P
0249 4C 00 02
024C A0 00
024E 20 6D 02
0251 90 04
0253 C8
0254 4C 4E 02
0257 20 5B 02
025A 60
025B 18
025C A5 F8
025E 65 00
0260 85 F8
0262 A5 F9
```

```
0264 65 01
0266 85 F9           P
0268 98
0269 20 9B 12
026C 60
026D 38
026E A5 F8
0270 E5 00
0272 85 F8
0274 A5 F9
0276 E5 01
0278 85 F9
027A 60
027B 77
```

must be established accordingly, so the original monitor program must be dispensed with here!

After HEXDEC is started, a number of hexadecimal numbers are keyed in and their decimal equivalents are printed behind the closing colon. By operating the colon key right away, the value ØØØØØ will be printed.

So the program works correctly. To be absolutely sure that we know what is going on, we would like to see the program in its assembled form. This involves a warm start entry into the PM Editor. As table 4 shows, the start address used is 153D instead of 1533. This is feasible as the program section between 1533 and 153C is devoted to defining the BREAK jump vector and to resetting the stack pointer (see chapter 15). Address 153D can be used instead of 1533, because we do not intend to use the BREAK key anyway (as this would bring the text 'JUNIOR' on to the display, in other words, the computer would return to PM). This can also be seen in the last section of table 4, where HEXDEC and its three subroutines are listed in three and a bit blocks of instructions.

3. Decimal addition

The next example involves a program which is not particularly practical, especially if you happen to own a calculator, but the important thing here is to learn how to use PM and PME subroutines.

We wish to write a program which will add two decimal numbers together. The first number can have a maximum of eight digits (the result of a previous addition) while the second number (the one to be added) can only have up to six digits. Once the addition has been performed, another (up to) six digit number can be added to the cumultative total – up to a maximum value of 99,999,999.

The first six digit number is entered by depressing up to six decimal keys (0 . . . 9) followed by a full stop '.'. Next, the operator indicates a number of not more than six digits which has to be added to the previous one. This is the procedure: first enter the number followed by 'P' (for plus). The P key is used instead of '+' to avoid having to depress the shift key for each addition as this can be somewhat of a bind! Any keys other than '0 . . . 9', '.' and 'P' (therefore including the hexadecimal keys A . . . F) will automatically lead to the error message 'WHAT?'.

In practice, things will look like this:

```
123456.
    123456
654321P
+   654321
= ØØ777777
1P
+   ØØØØØ1
= ØØ777778
```

The reaction by the Junior Computer is printed in bold characters as opposed to the key operations performed by the operator which are shown in normal type. As a matter of fact, the operations performed by the user look a little different when they appear on the screen/printer as a line ending in '.' or 'P' is overwritten by the first reaction from the computer. This means that the results cannot be printed out on a printer as two lines will be printed one on top of the other. A video terminal such as the Elekterminal presents no problems as each line is erased prior to being overwritten on. If a paper printer is used, the hardware echo – which prints the user operations – can be suppressed by switching from the 'half-duplex' mode to the 'full-duplex' mode. The result is a clear survey of all the various figures neatly listed.

The program is called DECADD and is shown in figure 4. Various memory locations are involved, so we will discuss these first:

INL	address $ØØF8	entered figure 10^0 and 10^1
INH	address $ØØF9	entered figure 10^2 and 10^3
POINTL	address $ØØFA	entered figure 10^4 and 10^5
DECA	address $ØØDE	cumulative figure 10^0 and 10^1
DECB	address $ØØDF	cumulative figure 10^2 and 10^3
DECC	address $ØØEØ	cumulative figure 10^4 and 10^5
POINTH	address $ØØFB	cumulative figure 10^6 and 10^7

As mentioned previously, the maximum result of the addition (= cumulative figure) is 99,999,999.

The DECADD program and its associated subroutines make use of the following PME routines:

I. **CRLF, RECCHA:** see example 2.

II. **PRCHA** Address $1334. This prints a character or produces a control function provided the corresponding ASCII code is stored in the accumulator.

III. **PRSP** Address $11F3. This simply prints a space . . .

IV. **PRBYT** Address $128F. This prints the high order nibble of the accumulator contents followed by the low order nibble after converting them into their respective ASCII codes.

V. **MESSY** Address $11D6. Despite its name, this is quite an orderly subroutine, as it prints text obtained from a look-up table. The text is defined by the initial value contained in the Y register and by the position of the EOT character $Ø3 in the look-up table. More details about this will be provided in chapter 14.

The DECADD program starts by clearing the three buffers so that a six digit number can be entered. Then subroutine CRLF ensures that the entered data is printed at the start of a new line and the program waits during RECCHA for a key to be depressed. The keys '0 . . . 9', '.' and 'P' are the only ones which mean anything to the program, all the others are ignored.

Suppose that a key other than '.' or 'P' is depressed. The DECADD program includes a section of program labelled DATA and the subroutine DECNUM, which is very similar to the subroutine HEXNUM described earlier. Only the keys 0 . . . 9 are processed into valid data. In all other cases the computer responds with 'WHAT?' and resets the Z flag. Any key in the range 0 . . . 9 is processed into a nibble that is shifted into the INL buffer from right to left, as a result of which all the existing nibbles move up one place in the three buffers. Thus, POINTL, INH and INL contain data corresponding to the last six keys to be depressed. A high order nibble is always 'older' than a low order nibble. The original high order nibble in POINTL is lost. If less than six keys were depressed, that is if

4

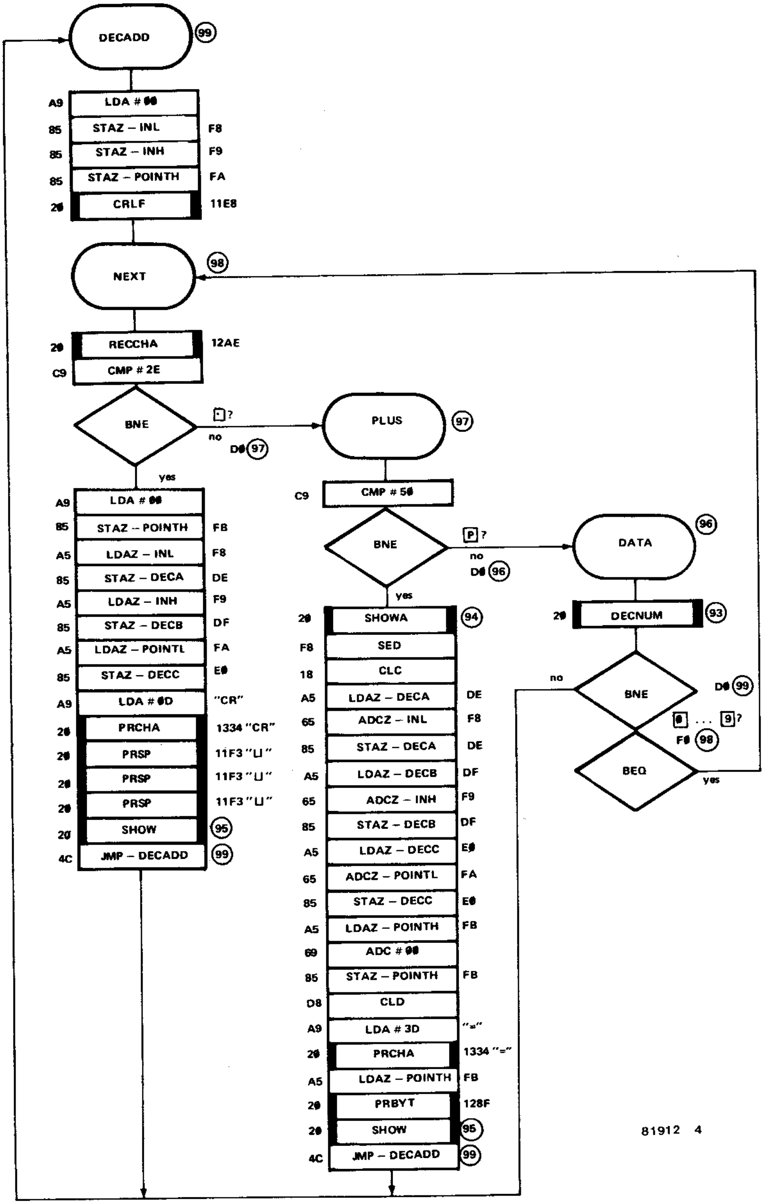


Figure 4. The main routine DECADD of the program which enables a six digit decimal number to be added to an eight digit decimal number: the cumulative figure.

the figure were less than six digits, the unused nibbles will be zero. After the DECNUM subroutine the state of the Z flag informs the computer as to whether the process should be repeated from scratch (after discovering an error) or whether the computer can expect another data key, in which case the processor jumps back to label NEXT.

When the plus key is operated, after the end of each figure entry, the processor clears the contents of buffer POINTH – just in case the result of the addition (cumulative figure) exceeds 999,999. Then the contents of buffers INL, INH and POINTL are transferred to buffers DECA, DECB and DECC respectively. Following this the program performs a few control functions. The carriage return (CR) command takes the cursor back to the beginning of the same line. Then three spaces are printed (three PRSP operations). The SHOW subroutine simply prints out the figure that has just been entered. After that, a return is made to the start of DECADD.

The subroutine SHOW is given in figure 5a and consists of three load operations and calls subroutine PRBYT three times. All very simple!

Now let us see what happens when the P key is depressed, that is to say, when the entered figure is to be added to the current cumulative figure. The procedure starts with SHOWA (see figure 5b). Again, this subroutine

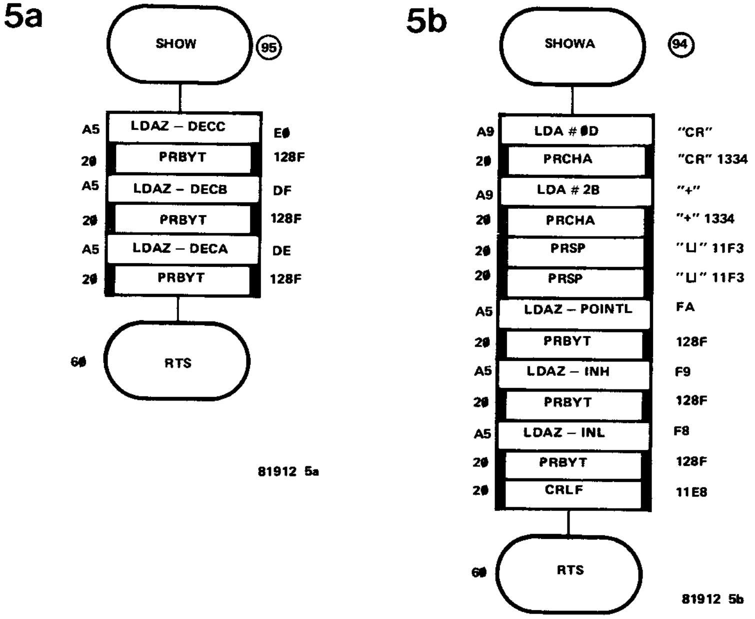


Figure 5. Subroutine SHOW (figure 5a) used by DECADD (figure 4) enables the six right-hand digits in the cumulative figure to be printed. Subroutine SHOWA (figure 5b) prints a plus sign followed by two spaces followed by the new six digit number to be added to the cumulative figure, thereby producing a new cumulative total.

does nothing but print characters. Starting at the beginning of the same line it prints a '+', two spaces and then the figure that has just been typed in. This is followed by another jump to the CRLF subroutine so that the cursor is placed at the beginning of the next line.

The figure just printed has to be added to the current cumulative figure and the result then acts as the new cumulative figure. The state of the carry flag determines whether the contents of the 'overflow' buffer, POINTH, are modified or not. The addition is carried out in the decimal mode; it starts with the instruction SED. Immediately after the addition the computer reverts to the hexadecimal mode (CLD). This is necessary because a number of PM subroutines will not function correctly if the D flag is set. This was discussed at length in chapter 12 of Book 3.

After the addition the 'equals' sign (=) is printed on the next line. Then the contents of POINTH are printed and the subroutine SHOW prints out the contents of the other three buffers. Finally, the program returns to the start of DECADD.

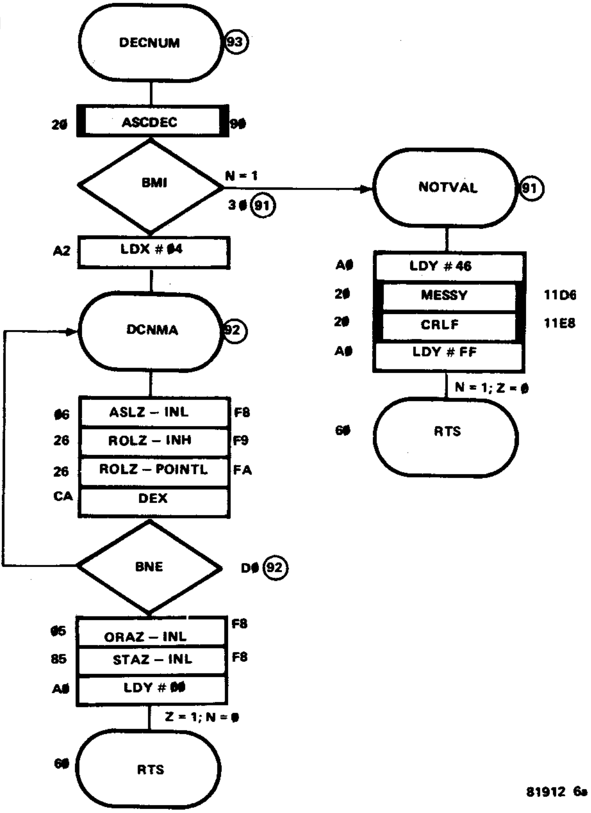
Now for subroutine DECNUM, which is shown in figure 6a. This starts off by calling yet another subroutine namely ASCDEC (see figure 6b). The ASCII code of the depressed key is stored in the accumulator. The ASCII codes 3Ø . . . 39 correspond to the numbers Ø . . . 9. The ASCDEC subroutine tests the validity of the depressed key. Essentially, the outcome of the routine is that the N flag will be set and the Z flag therefore reset if the depressed key was not valid. If a valid key is depressed, the instruction AND #ØF (after label VALID) converts the ASCII codes 3Ø . . . 39 into a data nibble with the value ØØ . . . Ø9.

Back to DECNUM. The BMI instruction after the jump to the ASCDEC subroutine leads the program to label NOTVAL if a non-valid key was depressed. After printing the text 'WHAT?' (MESSY, where the contents of the Y index register are 46 – see chapter 14) the N flag is set. This means that the Z flag will be reset. A valid key entry means that the contents of buffers INL, INH and POINTL have to be shifted four positions to the left and the data for the depressed key will have to be shifted into INL. All this is accomplished during the DECNUM subroutine after the label DCNMA. To complete the procedure the Z flag becomes set so the N flag becomes reset.

It is now time to edit and then assemble the DECADD program and all its subroutines. How this is done is shown in table 5. After the start of PM (text report = 'JUNIOR') and the cold start entry into PME, all the instructions involved in the program are entered into the computer, the first one by means of the INSERT function. After depressing the X key the program is assembled. Then the ST key on the main keyboard is operated and all the labels are listed and the program makes a warm start entry into PME. The assembled version of the DECADD program is then listed by depressing the P key the relevant number of times.

Since, as mentioned earlier, a user line is overwritten by the first reaction line on the part of the computer, no examples of the running of the DECADD program are shown in table 5. If required, user lines may be preserved by replacing the two instructions in the program which 'print' a carriage return by the instruction JSR-CRLF. It is a useful exercise for the user to know where these are and how to modify the not-yet-as-

6a



sembled program with the aid of PME. The procedure for this will be illustrated in example number 5 a few pages further on.

4. Decimal-to-hexadecimal conversion

Assuming that it is possible to write a program to convert hexadecimal numbers to decimal numbers (see figure 2 and example 2), it must surely be possible to write a program to convert decimal numbers to hexadecimal numbers. This is in fact no problem as can be seen from the DECHEX program given in figure 7. This looks remarkably similar to the program given in figure 2. A number is entered (after label DATA in figure 7) with the aid of the DECNUM subroutine (see figure 6a). This means that three figure buffers are available giving us the possibility of entering a six digit decimal number.

However, in this instance we are only interested in decimal numbers from zero up to and including 65,535 (= $FFFF). This is because the computer

6b

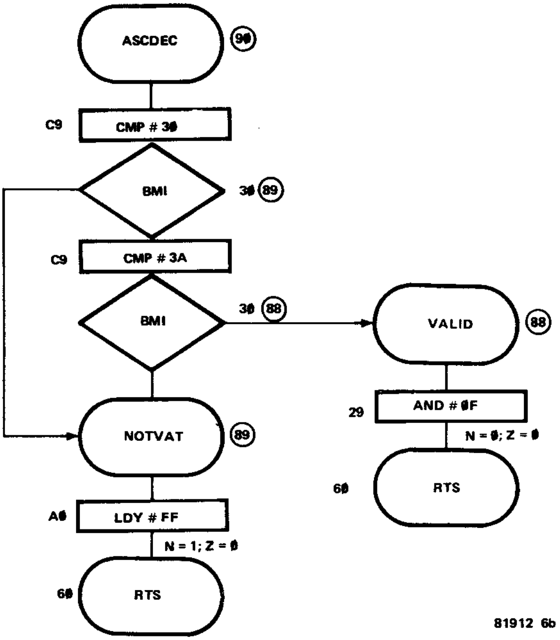


Figure 6. Subroutines DECNUM (figure 6a) and ASCDEC (figure 6b) used by the DECADD routine (figure 4) and used by the DECHEX routine (figure 7) process the valid decimal numbers 0 . . . 9 stored in the buffers POINTL, INH and INL.

is unable to deal with hexadecimal figures containing more than sixteen bits without increasing the complexity of the various subroutines.

The DECHEX program is virtually identical to the previously described HEXDEC program. The computer has to determine (in subroutine AMOUNT) how many times the figure 4096 (16^3 or $1ØØØ appears in the entered decimal number and then how many times the number 256 (16^2 or $1ØØ) appears and after that, how often 16 (16^1 or $1Ø) appears. And finally, the DECHEX program checks the number of units remaining in the decimal figure (16^0 = 1 = $1).

Any decimal figure which is equal to or greater than the value 65536 (16^4 = $1ØØØØ) will cause problems in DECHEX as the computer will be unable to detect the number of 65535 figures present in the entered number. Instead, a number of 4096 figures in excess of 15 will be found. Despite this warning, a preventive measure has been included in the routine to save any damage being caused by any nonsense (in the event where the entered number is greater than 65535). The high order nibble of

(continued on page 56 →)

Table 5. Using the PM Editor to enter the DECADD program and its subroutines (see also figures 4 . . . 6 and example 3).

```
SIX DIGIT DECIMAL ADDITION
UP TO 99,999,999
DECADD
JUNIOR

15ØØ
15ØØ 2Ø R
BEGAD,ENDAD: 2ØØ,3FF
PM EDITOR
Ø2ØØ 77            IFF 99 ØØ
Ø2ØØ FF 99 ØØ      A9 ØØ
Ø2Ø3 A9 ØØ         85 F8
Ø2Ø5 85 F8         85 F9
Ø2Ø7 85 F9         85 FA
Ø2Ø9 85 FA         2Ø E8 11
Ø2ØB 2Ø E8 11      FF 98 ØØ
Ø2ØE FF 98 ØØ      2Ø AE 12
Ø211 2Ø AE 12      C9 2E
Ø214 C9 2E         DØ 97
Ø216 DØ 97         A9 ØØ
Ø218 A9 ØØ         85 FB
Ø21A 85 FB         A5 F8
Ø21C A5 F8         85 DE
Ø21E 85 DE         A5 F9
Ø22Ø A5 F9         85 DF
Ø222 85 DF         A5 FA
Ø224 A5 FA         85 EØ
Ø226 85 EØ         A9 ØD
Ø228 A9 ØD         2Ø 34 13
Ø22A 2Ø 34 13      2Ø F3 11
Ø22D 2Ø F3 11      2Ø F3 11
Ø23Ø 2Ø F3 11      2Ø F3 11
Ø233 2Ø F3 11      2Ø 95 ØØ
Ø236 2Ø 95 ØØ      4C 99 ØØ
Ø239 4C 99 ØØ      FF 97 ØØ
Ø23C FF 97 ØØ      C9 5Ø
Ø23F C9 5Ø         DØ 96
Ø241 DØ 96         2Ø 94 ØØ
Ø243 2Ø 94 ØØ      F8
Ø246 F8            18
Ø247 18            A5 DE
Ø248 A5 DE         65 F8
Ø24A 65 F8         85 DE
Ø24C 85 DE         A5 DF
Ø24E A5 DF         65 F9
Ø25Ø 65 F9         85 DF
Ø252 85 DF         A5 EØ
Ø254 A5 EØ         65 FA
Ø256 65 FA         85 EØ
```

```
Ø258 85 EØ
Ø25A A5 FB
Ø25C 69 ØØ
Ø25E 85 FB
Ø26Ø D8
Ø261 A9 3D
Ø263 2Ø 34 13
Ø266 A5 FB
Ø268 2Ø 8F 12
Ø26B 2Ø 95 ØØ
Ø26E 4C 99 ØØ
Ø271 FF 96 ØØ
Ø274 2Ø 93 ØØ
Ø277 DØ 99
Ø279 FØ 98
Ø27B FF 95 ØØ
Ø27E A5 EØ
Ø28Ø 2Ø 8F 12
Ø283 A5 DF
Ø285 2Ø 8F 12
Ø288 A5 DE
Ø28A 2Ø 8F 12
Ø28D 6Ø
Ø28E FF 94 ØØ
Ø291 A9 ØD
Ø293 2Ø 34 13
Ø296 A9 2B
Ø298 2Ø 34 13
Ø29B 2Ø F3 11
Ø29E 2Ø F3 11
Ø2A1 A5 FA
Ø2A3 2Ø 8F 12
Ø2A6 A5 F9
Ø2A8 2Ø 8F 12
Ø2AB A5 F8
Ø2AD 2Ø 8F 12
Ø2BØ 2Ø E8 11
Ø2B3 6Ø
Ø2B4 FF 93 ØØ
Ø2B7 2Ø 9Ø ØØ
Ø2BA 3Ø 91
Ø2BC A2 Ø4
Ø2BE FF 92 ØØ
Ø2C1 Ø6 F8
Ø2C3 26 F9
Ø2C5 26 FA
Ø2C7 CA
Ø2C8 DØ 92
Ø2CA Ø5 F8
Ø2CC 85 F8
Ø2CE AØ ØØ
Ø2DØ 6Ø
```

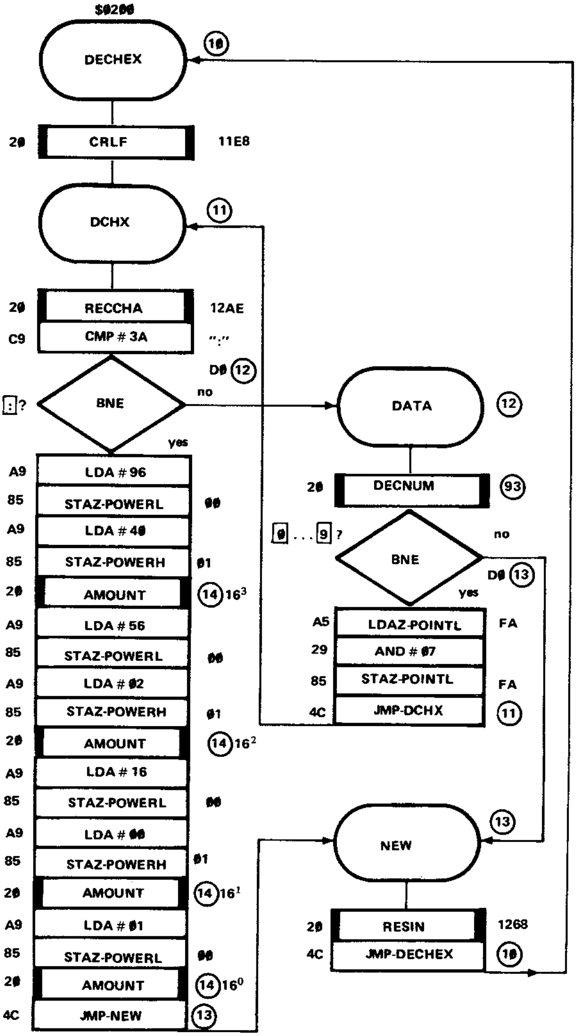
```
A5 FB
69 ØØ
85 FB
D8
A9 3D
2Ø 34 13
A5 FB
2Ø 8F 12
2Ø 95 ØØ
4C 99 ØØ
FF 96 ØØ
2Ø 93 ØØ
DØ 99
FØ 98
FF 95 ØØ
A5 EØ
2Ø 8F 12
A5 DF
2Ø 8F 12
A5 DE
2Ø 8F 12
6Ø
FF 94 ØØ
A9 ØD
2Ø 34 13
A9 2B
2Ø 34 13
2Ø F3 11
2Ø F3 11
A5 FA
2Ø 8F 12
A5 F9
2Ø 8F 12
A5 F8
2Ø 8F 12
2Ø E8 11
6Ø
FF 93 ØØ
2Ø 9Ø ØØ
3Ø 91
A2 Ø4
FF 92 ØØ
Ø6 F8
26 F9
26 FA
CA
DØ 92
Ø5 F8
85 F8
AØ ØØ
6Ø
FF 91 ØØ
```

```
Ø2D1 FF 91 ØØ      AØ 46
Ø2D4 AØ 46         2Ø D6 11
Ø2D6 2Ø D6 11      2Ø E8 11
Ø2D9 2Ø E8 11      AØ FF
Ø2DC AØ FF         6Ø
Ø2DE 6Ø            FF 9Ø ØØ
Ø2DF FF 9Ø ØØ      C9 3Ø
Ø2E2 C9 3Ø         3Ø 89
Ø2E4 3Ø 89         C9 3A
Ø2E6 C9 3A         3Ø 88
Ø2E8 3Ø 88         FF 89 ØØ
Ø2EA FF 89 ØØ      AØ FF
Ø2ED AØ FF         6Ø
Ø2EF 6Ø            FF 88 ØØ
Ø2FØ FF 88 ØØ      29 ØF
Ø2F3 29 ØF         6Ø
Ø2F5 6Ø
Ø2F6 77            X
LAB $99: $Ø2ØØ LAB $98: $Ø2ØB LAB $97: $Ø236 LAB $96: $Ø268
LAB $95: $Ø26F LAB $94: $Ø27F LAB $93: $Ø2A2 LAB $92: $Ø2A9
LAB $91: $Ø2B9 LAB $9Ø: $Ø2C4 LAB $89: $Ø2CC LAB $88: $Ø2CF
PM EDITOR
Ø2ØØ A9 ØØ         P
Ø2Ø2 85 F8
Ø2Ø4 85 F9
Ø2Ø6 85 FA
Ø2Ø8 2Ø E8 11
Ø2ØB 2Ø AE 12
Ø2ØE C9 2E
Ø21Ø DØ 24
Ø212 A9 ØØ
Ø214 85 FB
Ø216 A5 F8
Ø218 85 DE
Ø21A A5 F9
Ø21C 85 DF
Ø21E A5 FA
Ø22Ø 85 EØ         P
Ø222 A9 ØD
Ø224 2Ø 34 13
Ø227 2Ø F3 11
Ø22A 2Ø F3 11
Ø22D 2Ø F3 11
Ø23Ø 2Ø 6F Ø2
Ø233 4C ØØ Ø2
Ø236 C9 5Ø
Ø238 DØ 2E
Ø23A 2Ø 7F Ø2
Ø23D F8
Ø23E 18
Ø23F A5 DE
Ø241 65 F8
```

```
Ø243 85 DE          P
Ø245 A5 DF
Ø247 65 F9
Ø249 85 DF
Ø24B A5 EØ
Ø24D 65 FA
Ø24F 85 EØ
Ø251 A5 FB
Ø253 69 ØØ
Ø255 85 FB
Ø257 D8
Ø258 A9 3D
Ø25A 2Ø 34 13
Ø25D A5 FB
Ø25F 2Ø 8F 12
Ø262 2Ø 6F Ø2       P
Ø265 4C ØØ Ø2
Ø268 2Ø A2 Ø2
Ø26B DØ 93
Ø26D FØ 9C
Ø26F A5 EØ
Ø271 2Ø 8F 12
Ø274 A5 DF
Ø276 2Ø 8F 12
Ø279 A5 DE
Ø27B 2Ø 8F 12
Ø27E 6Ø
Ø27F A9 ØD
Ø281 2Ø 34 13
Ø284 A9 2B
Ø286 2Ø 34 13       P
Ø289 2Ø F3 11
Ø28C 2Ø F3 11
Ø28F A5 FA
Ø291 2Ø 8F 12
Ø294 A5 F9
Ø296 2Ø 8F 12
Ø299 A5 F8
Ø29B 2Ø 8F 12
Ø29E 2Ø E8 11
Ø2A1 6Ø
Ø2A2 2Ø C4 Ø2
Ø2A5 3Ø 12
Ø2A7 A2 Ø4
Ø2A9 Ø6 F8
Ø2AB 26 F9          P
Ø2AD 26 FA
Ø2AF CA
Ø2BØ DØ F7
Ø2B2 Ø5 F8
Ø2B4 85 F8
Ø2B6 AØ ØØ
```

```
Ø2B8 6Ø
Ø2B9 AØ 46
Ø2BB 2Ø D6 11
Ø2BE 2Ø E8 11
Ø2C1 AØ FF
Ø2C3 6Ø
Ø2C4 C9 3Ø
Ø2C6 3Ø Ø4
Ø2C8 C9 3A          P
Ø2CA 3Ø Ø3
Ø2CC AØ FF
Ø2CE 6Ø
Ø2CF 29 ØF
Ø2D1 6Ø
Ø2D2 77
```

7



figuur 81912 - 7

Figure 7. The main routine DECHEX converts any decimal number less than 65536 into its hexadecimal equivalent. It is very similar to the HEXDEC routine (figure 2). Subroutines DECNUM and ASCDEC are shown in figures 6a and 6b. Subroutine AMOUNT is shown in figure 3a. The subroutines CORPR and SUBTRA, however, will have to be modified slightly as shown in figure 8.

Table 6. Using the PM Editor to enter the DECHEX program and its subroutines (see also figures 3a, 6a, 6b, 8a and 8b and example 4).

```
JUNIOR

15ØØ
15ØØ 2Ø R
BEGAD,ENDAD: 2ØØ,3FF
PM EDITOR
Ø2ØØ 77                       IFF 1Ø ØØ
Ø2ØØ FF 1Ø ØØ                 2Ø E8 11
Ø2Ø3 2Ø E8 11                 FF 11 ØØ
Ø2Ø6 FF 11 ØØ                 2Ø AE 12
Ø2Ø9 2Ø AE 12                 C9 3A
Ø2ØC C9 3A                    DØ 12
Ø2ØE DØ 12                    A9 96
Ø21Ø A9 96                    85 ØØ
Ø212 85 ØØ                    A9 4Ø
Ø214 A9 4Ø                    85 Ø1
Ø216 85 Ø1                    2Ø 14 ØØ
Ø218 2Ø 14 ØØ                 A9 56
Ø21B A9 56                    85 ØØ
Ø21D 85 ØØ                    A9 Ø2
Ø21F A9 Ø2                    85 Ø1
Ø221 85 Ø1                    2Ø 14 ØØ
Ø223 2Ø 14 ØØ                 A9 16
Ø226 A9 16                    85 ØØ
Ø228 85 ØØ                    A9 ØØ
Ø22A A9 ØØ                    85 Ø1
Ø22C 85 Ø1                    2Ø 14 ØØ
Ø22E 2Ø 14 ØØ                 A9 Ø1
Ø231 A9 Ø1                    85 ØØ
Ø233 85 ØØ                    2Ø 14 ØØ
Ø235 2Ø 14 ØØ                 4C 13 ØØ
Ø238 4C 13 ØØ                 FF 12 ØØ
Ø23B FF 12 ØØ                 2Ø 93 ØØ
Ø23E 2Ø 93 ØØ                 DØ 13
Ø241 DØ 13                    A5 FA
Ø243 A5 FA                    29 Ø7
Ø245 29 Ø7                    85 FA
Ø247 85 FA                    4C 11 ØØ
Ø249 4C 11 ØØ                 FF 13 ØØ
Ø24C FF 13 ØØ                 2Ø 68 12
Ø24F 2Ø 68 12                 4C 1Ø ØØ
Ø252 4C 1Ø ØØ                 FF 14 ØØ
Ø255 FF 14 ØØ                 AØ ØØ
Ø258 AØ ØØ                    FF 15 ØØ
Ø25A FF 15 ØØ                 2Ø 18 ØØ
Ø25D 2Ø 18 ØØ                 9Ø 16
Ø26Ø 9Ø 16                    C8
Ø262 C8                       4C 15 ØØ
Ø263 4C 15 ØØ                 FF 16 ØØ
```

```
Ø266 FF 16 ØØ
Ø269 2Ø 17 ØØ
Ø26C 6Ø
Ø26D FF 17 ØØ
Ø27Ø F8
Ø271 18
Ø272 A5 F8
Ø274 65 ØØ
Ø276 85 F8
Ø278 A5 F9
Ø27A 65 Ø1
Ø27C 85 F9
Ø27E A5 FA
Ø28Ø 69 ØØ
Ø282 85 FA
Ø284 D8
Ø285 98
Ø286 2Ø 9B 12
Ø289 6Ø
Ø28A FF 18 ØØ
Ø28D F8
Ø28E 38
Ø28F A5 F8
Ø291 E5 ØØ
Ø293 85 F8
Ø295 A5 F9
Ø297 E5 Ø1
Ø299 85 F9
Ø29B A5 FA
Ø29D E9 ØØ
Ø29F 85 FA
Ø2A1 D8
Ø2A2 6Ø
Ø2A3 FF 93 ØØ
Ø2A6 2Ø 9Ø ØØ
Ø2A9 3Ø 91
Ø2AB A2 Ø4
Ø2AD FF 92 ØØ
Ø2BØ Ø6 F8
Ø2B2 26 F9
Ø2B4 26 FA
Ø2B6 CA
Ø2B7 DØ 92
Ø2B9 Ø5 F8
Ø2BB 85 F8
Ø2BD AØ ØØ
Ø2BF 6Ø
Ø2CØ FF 91 ØØ
Ø2C3 AØ 46
Ø2C5 2Ø D6 11
Ø2C8 2Ø E8 11
Ø2CB AØ FF
```

```
2Ø 17 ØØ
6Ø
FF 17 ØØ
F8
18
A5 F8
65 ØØ
85 F8
A5 F9
65 Ø1
85 F9
A5 FA
69 ØØ
85 FA
D8
98
2Ø 9B 12
6Ø
FF 18 ØØ
F8
38
A5 F8
E5 ØØ
85 F8
A5 F9
E5 Ø1
85 F9
A5 FA
E9 ØØ
85 FA
D8
6Ø
FF 93 ØØ
2Ø 9Ø ØØ
3Ø 91
A2 Ø4
FF 92 ØØ
Ø6 F8
26 F9
26 FA
CA
DØ 92
Ø5 F8
85 F8
AØ ØØ
6Ø
FF 91 ØØ
AØ 46
2Ø D6 11
2Ø E8 11
AØ FF
6Ø
```

```
Ø2CD 6Ø              FF 9Ø ØØ
Ø2CE FF 9Ø ØØ        C9 3Ø
Ø2D1 C9 3Ø           3Ø 89
Ø2D3 3Ø 89           C9 3A
Ø2D5 C9 3A           3Ø 88
Ø2D7 3Ø 88           FF 89 ØØ
Ø2D9 FF 89 ØØ        AØ FF
Ø2DC AØ FF           6Ø
Ø2DE 6Ø              FF 88 ØØ
Ø2DF FF 88 ØØ        29 ØF
Ø2E2 29 ØF           6Ø
Ø2E4 6Ø
Ø2E5 77              X
LAB $1Ø: $Ø2ØØ LAB $11: $Ø2Ø3 LAB $12: $Ø235 LAB $13: $Ø243
LAB $14: $Ø249 LAB $15: $Ø24B LAB $16: $Ø254 LAB $17: $Ø258
LAB $18: $Ø272 LAB $93: $Ø288 LAB $92: $Ø28F LAB $91: $Ø29F
LAB $9Ø: $Ø2AA LAB $89: $Ø2B2 LAB $88: $Ø2B5
PM EDITOR
Ø2ØØ 2Ø E8 11        P
Ø2Ø3 2Ø AE 12
Ø2Ø6 C9 3A
Ø2Ø8 DØ 2B
Ø2ØA A9 96
Ø2ØC 85 ØØ
Ø2ØE A9 4Ø
Ø21Ø 85 Ø1
Ø212 2Ø 49 Ø2
Ø215 A9 56
Ø217 85 ØØ
Ø219 A9 Ø2
Ø21B 85 Ø1
Ø21D 2Ø 49 Ø2
Ø22Ø A9 16
Ø222 85 ØØ           P
Ø224 A9 ØØ
Ø226 85 Ø1
Ø228 2Ø 49 Ø2
Ø22B A9 Ø1
Ø22D 85 ØØ
Ø22F 2Ø 49 Ø2
Ø232 4C 43 Ø2
Ø235 2Ø 88 Ø2
Ø238 DØ Ø9
Ø23A A5 FA
Ø23C 29 Ø7
Ø23E 85 FA
Ø24Ø 4C Ø3 Ø2
Ø243 2Ø 68 12
Ø246 4C ØØ Ø2        P
Ø249 AØ ØØ
Ø24B 2Ø 72 Ø2
Ø24E 9Ø Ø4
```

```
Ø25Ø C8
Ø251 4C 4B Ø2
Ø254 2Ø 58 Ø2
Ø257 6Ø
Ø258 F8
Ø259 18
Ø25A A5 F8
Ø25C 65 ØØ
Ø25E 85 F8
Ø26Ø A5 F9
Ø262 65 Ø1
Ø264 85 F9              P
Ø266 A5 FA
Ø268 69 ØØ
Ø26A 85 FA
Ø26C D8
Ø26D 98
Ø26E 2Ø 9B 12
Ø271 6Ø
Ø272 F8
Ø273 38
Ø274 A5 F8
Ø276 E5 ØØ
Ø278 85 F8
Ø27A A5 F9
Ø27C E5 Ø1
Ø27E 85 F9              P
Ø28Ø A5 FA
Ø282 E9 ØØ
Ø284 85 FA
Ø286 D8
Ø287 6Ø
Ø288 2Ø AA Ø2
Ø28B 3Ø 12
Ø28D A2 Ø4
Ø28F Ø6 F8
Ø291 26 F9
Ø293 26 FA
Ø295 CA
Ø296 DØ F7
Ø298 Ø5 F8
Ø29A 85 F8              P
Ø29C AØ ØØ
Ø29E 6Ø
Ø29F AØ 46
Ø2A1 2Ø D6 11
Ø2A4 2Ø E8 11
Ø2A7 AØ FF
Ø2A9 6Ø
Ø2AA C9 3Ø
Ø2AC 3Ø Ø4
Ø2AE C9 3A
```

```
Ø2BØ 3Ø Ø3
Ø2B2 AØ FF
Ø2B4 6Ø
Ø2B5 29 ØF
Ø2B7 6Ø                  P
Ø2B8 77
JUNIOR

2ØØ
Ø2ØØ 2Ø R
4Ø96:1ØØØ
256:Ø1ØØ
8192:2ØØØ
65535:FFFF
1ØØØØ:271Ø
1ØØØ:Ø3E8
1ØØ;
WHAT?

1ØØ:ØØ64
4Ø95:ØFFF
255:ØØFF
15:ØØØF
12345:3Ø39
```

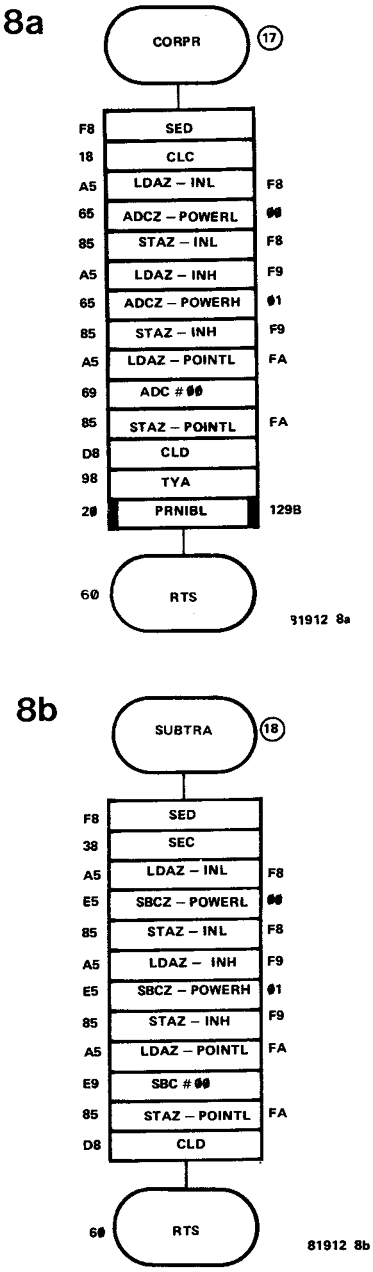


Figure 8. With respect to figures 3b and 3c, subroutines CORPR and SUBTRA are slightly modified for use in the DECHEX program.

buffer POINTL contains the number of hundred thousands contained in the decimal figure. The low order nibble stores the number of ten thousands. By masking the contents of POINTL with the value Ø7 after a key has been depressed, the computer makes sure that the decimal figure contains no hundred thousands and that the number of ten thousands does not exceed 7.

The AMOUNT subroutine, which is used four times during the DECHEX program, can be used in its original form here (see figure 3a). However, the subroutine CORPR and SUBTRA called by AMOUNT do have to be slightly modified for their task in DECHEX. This is why they look different in figures 8a and 8b. What is the reason for such changes? Well, a decimal addition (CORPR) and a decimal subtraction (SUBTRA) have to take place. Furthermore, the contents of POINTL also have to be modified, depending on the status of the carry flag (ADC #ØØ in CORPR; SCD #ØØ in SUBTRA).

How the DECHEX program is edited and assembled is illustrated in table 6. The various routines were entered in the following sequence:

DECHEX:	labels 1Ø . . . 13	(figure 7)
AMOUNT:	labels 14 . . . 16	(figure 3a)
CORPR:	label 17	(figure 8a)
SUBTRA:	label 18	(figure 8b)
DECNUM:	labels 93 . . . 91	(figure 6a)
ASCDEC:	labels 9Ø . . . 88	(figure 6b)

The rest of the procedure should be familiar to everyone by now: assemble (X), return to PME (ST) by printing out the labels, printing the assembled program (P) and start the program after the return to PM. Finally, execute the program for as long as you like. When the operator has had enough, the RST key can be depressed and a return can be effected by starting PM.

5. The warm CEND start entry into PME and the cassette

In chapter 11 of Book 3 we mentioned the fact that TM could be used to store a program on cassette before it has been assembled and tested. Especially where extensive programs are concerned, it is no trouble to reenter the unassembled program – if the program needs to be modified here and there – and then to restore it to its original state (before it was stored on cassette) by means of a warm ČEND start entry into PME. Next, the actual debugging can take place and the same procedure can be repeated ad infinitum, or rather, for as long as is necessary.

Here is an example to illustrate the coming procedures. It is the ASCII program first described in chapter 12 of Book 3 – see figure 9. This program is so short that it does not really need editing and assembling. Nevertheless, we wish to illustrate the principles involved with the aid of this program as it saves reams and reams of listings etc. What is more, it gives readers a clear insight into what is involved in a certain PME feature. Never mind if the example is rather like using a sledge-hammer to crack a nut!

The complete program is listed in table 7. Each step will be dealt with in turn:

① The system program PM is started.

9

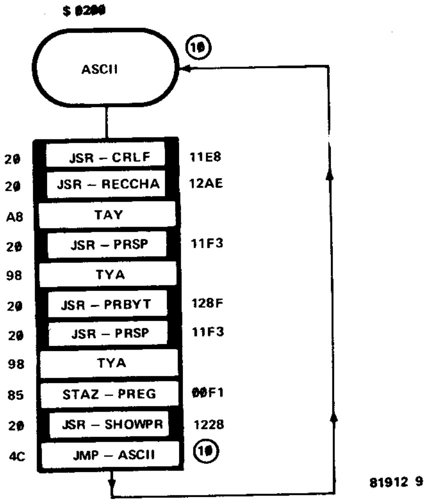


Figure 9. The ASCII program described in chapter 12 of Book 3 is used here to illustrate how the warm CEND start entry into PME operates. The start address (BEGAD) is now Ø2ØØ. The text explains (see example 5) why this address is assigned the label 1Ø.

② Now a cold start entry into PME is made. The start address of the ASCII program is $Ø2ØØ.

③ The instructions are entered, starting by 'inserting' label 1Ø – see figure 9.

④ All the instructions are now entered and the complete program is listed by depressing L.

⑤ The not yet assembled program has to be stored on cassette tape. For this to happen, a return has to be made to the PM program (report 'JUNIOR'), the cassette machine is prepared for recording and then keys 'S', '8', '8', ',' , '2', 'Ø', 'Ø', ',' , '2', '1' and 'E' are depressed in that order. After starting the machine, the CR key is depressed. The complete program is then stored and the computer reports with the text 'READY'. The cassette machine is stopped. The program number in this particular instance is 88. The contents of SA are the same as those of BEGAD and the contents of EA are the same as those of CEND, one address higher than the one containing the EOF character, 77 (see listing for point 4).

⑥ PME is then entered by means of a warm start entry.

⑦ The program is assembled (press keys X and ST). There is only one label in this case.

⑧ The assembled program is printed out (P).

⑨ A return is made to PM so that the program can be started and tested. After entering the start address and following the start, the ASCII code of key A needs to be known. This is not a problem, which means that the program must be working correctly . . . or is it? When we come to find out

Table 7. Elaborating the ASCII program while making use of a warm CEND start entry into PME (see also figure 9 and example 5).

```
① JUNIOR

② 15ØØ
  15ØØ 2Ø R
  BEGAD,ENDAD: 2ØØ,2FF
  PM EDITOR
③ Ø2ØØ 77                IFF 1Ø ØØ
  Ø2ØØ FF 1Ø ØØ          2Ø E8 11
  Ø2Ø3 2Ø E8 11          2Ø AE 12
  Ø2Ø6 2Ø AE 12          A8
  Ø2Ø9 A8                2Ø F3 11
  Ø2ØA 2Ø F3 11          98
  Ø2ØD 98                2Ø 8F 12
  Ø2ØE 2Ø 8F 12          2Ø F3 11
  Ø211 2Ø F3 11          98
  Ø214 98                85 F1
  Ø215 85 F1             2Ø 28 12
  Ø217 2Ø 28 12          4C 11 ØØ
④ Ø21A 4C 11 ØØ          L
  Ø2ØØ FF 1Ø ØØ
  Ø2Ø3 2Ø E8 11
  Ø2Ø6 2Ø AE 12
  Ø2Ø9 A8
  Ø2ØA 2Ø F3 11
  Ø2ØD 98
  Ø2ØE 2Ø 8F 12
  Ø211 2Ø F3 11
  Ø214 98
  Ø215 85 F1
  Ø217 2Ø 28 12
  Ø21A 4C 11 ØØ
  Ø21D 77
  DONE
  Ø21E E8
⑤ JUNIOR

  S88,2ØØ,21E
  READY
⑥ 1533
  1533 A9 R
  PM EDITOR
⑦ Ø21E E8                X
  LAB $1Ø: $Ø2ØØ
  PM EDITOR
⑧ Ø2ØØ 2Ø E8 11          P
  Ø2Ø3 2Ø AE 12
  Ø2Ø6 A8
  Ø2Ø7 2Ø F3 11
  Ø2ØA 98
  Ø2ØB 2Ø 8F 12
```

```
   Ø2ØE 2Ø F3 11
   Ø211 98
   Ø212 85 F1
   Ø214 2Ø 28 12
   Ø217 4C 11 ØØ
   Ø21A 77
⑨  JUNIOR

   2ØØ
   Ø2ØØ 2Ø R
   A 41 Ø1ØØØØØL6ABC
⑩  JUNIOR

   G88
   READY
⑪  17C5
   17C5 2Ø R
   BEGAD,ENDAD: 2ØØ,2FF
   PM EDITOR
⑫  Ø21D 77               S4C 11 ØØ
   Ø21A 4C 11 ØØ         K
   DONE
   Ø21A 4C 11 ØØ         K
   Ø21A 77               I4C 1Ø ØØ
   Ø21A 4C 1Ø ØØ
⑬  Ø21D 77               X
   LAB $1Ø: $Ø2ØØ
   PM EDITOR
⑭  Ø2ØØ 2Ø E8 11         P
   Ø2Ø3 2Ø AE 12
   Ø2Ø6 A8
   Ø2Ø7 2Ø F3 11
   Ø2ØA 98
   Ø2ØB 2Ø 8F 12
   Ø2ØE 2Ø F3 11
   Ø211 98
   Ø212 85 F1
   Ø214 2Ø 28 12
   Ø217 4C ØØ Ø2
   Ø21A 77
⑮  JUNIOR

   2ØØ
   Ø2ØØ 2Ø R
   A 41 Ø1ØØØØØ1
   B 42 Ø1ØØØØ1Ø
   C 43 Ø1ØØØØ11
   D 44 Ø1ØØØ1ØØ
   1 31 ØØ11ØØØ1
   2 32 ØØ11ØØ1Ø
   3 33 ØØ11ØØ11
   4 34 ØØ11Ø1ØØ
   K 4B Ø1ØØ1Ø11
   Z 5A Ø1Ø11Ø1Ø
```

the ASCII code for key 6, the 6 is not printed and the corresponding ASCII code does not appear at all. So, what is going on here? What about keys B and C? No, they do not produce the correct results either, there is definitely something wrong here!

Well that is rather funny, for the program worked fine in chapter 12. After a good look through the program once more, the problem is quite easy to find. The single ASCII label was entered as 4C 11 ØØ instead of 4C 1Ø ØØ. The label 11 was not entered and so after assembly and after running the program once, the computer jumped to address location ØØ11. The ASCII program then 'crashed'.

How do we amend the fault? Well, first the program is re-loaded from cassette, is modified and re-assembled etc.

⑩ After operating the RST key, PM is started. The ASCII program is re-entered from cassette in its unassembled form.

⑪ The PM Editor is then activated by means oт a warm CEND start entry. Locations BEGAD and ENDAD remain unchanged (ENDAD can be modified if the programmer wishes to add a few instructions). Once CEND has been changed back to its original value, that is before the program was stored on cassette, the editing process can be continued.

⑫ The instruction that is to be modified is located by using the SEARCH function and then the K key is depressed to delete that instruction from memory. The correct instruction (4C 1Ø ØØ) is then entered by means of the INSERT function.

⑬ The program is re-assembled.

⑭ The assembled program is listed.

⑮ The computer returns to PM and the ASCII program is restarted. This time it works correctly!

A few general remarks

- During a cold, lukewarm and warm CEND start entry, keep an eye on the following. During the initial phase, that is before the carriage return key is depressed, the BREAK jump vector will still be defined as for PM. If the BREAK key is operated while the text 'BEGAD, ENDAD', is being printed, the text 'JUNIOR' will appear as soon as the BREAK key is released and the processor returns to PM. If an error is made during the entry of BEGAD and ENDAD, the computer simply asks 'WHAT?' and then new data for these two pointers can be entered. The reasons for this are explained in full detail in chapters 14 and 15. As mentioned previously, there will be no error message if ENDAD is specified to be greater than BEGAD.
- Something about entering address operands which have to be assembled. In chapter 5 of Book 2, it was mentioned that the opcode (2Ø for a jump to subroutine; 4C for a jump) must be followed by a label number (to which the computer is to jump) and by a second operand byte, the limiter byte. The latter has been selected as ØØ throughout the Junior Computer Book series, however this need not be strictly adhered to as can be gathered from chapter 9 in Book 2.

- Never choose a label number which is the same as the low order byte of an absolute address (this does not need to be assembled). This should be checked before the program is edited.
- Do not note down an absolute displayed value where jump instructions are concerned, as this may well be a label number too – prior to assembly.
- Use each label number once only. Although it is not at all compulsory, it is advisable to arrange the label numbers in an ascending or descending order (leaving out any prohibited numbers) and in the order in which they are stored in memory during the editing process.

Most of this has, of course been mentioned earlier in chapter 5 of Book 2, but the application of the various points discussed as far as PME is concerned does take rather a lot of practice. The only aspect which is identical in both editor systems is the method of using hexadecimal labels. We hope that you will be able to benefit to the full from implementing PME and that it will help you to develop useful and versatile programs.

14

1.2 k bytes of PM software

The 'Junior on TV' program

Chapter 12 of Book 3 introduced the Printer Monitor system program. Now that we are familiar with its operation, it is time to describe the software involved in detail. Readers may wonder why this should be necessary, considering that the program can be used adequately without this knowledge. There are two arguments in favour of analysing the PM software:

1. **It will help the programmer to develop his/her own software.**
2. **Various PM subroutines (or parts of subroutines) can be incorporated into user programs – see the examples used in chapters 12 (Book 3) and 13 (Book 4).**

This does not mean, however, that we are going to dwell on each and every byte, for as can be gathered there are a total of 1268 bytes in all! In any case, after reading three (and a bit) books on the Junior Computer, most readers should be completely up-to-date with the 6502 machine language.

1 Any system program consists of a main routine as well as a number of subroutines. The main PM routine is based on a well-known principle: the computer waits for a key to be depressed and then checks to see which key was operated. Next, the corresponding key function is executed and the machine returns to a central 'cross-roads' in the program and waits for another key to be depressed, thereby completing a circuit, so to speak.
An initialisation routine is only run occasionally, usually at the start of a particular program, for instance. The Printer Monitor initialisation routine is shown in figures 1a and 1b. It is very similar to the RESET routine of the original monitor program and is run during the first and second starts of PM (see chapter 12 of Book 3). The section of program after the label LABJUN in figure 1b is also run after the BREAK key is depressed and when the text 'JUNIOR' is to be output as an error message or otherwise.
The STEP routine (figure 1b) is run whenever an instruction has been

1a

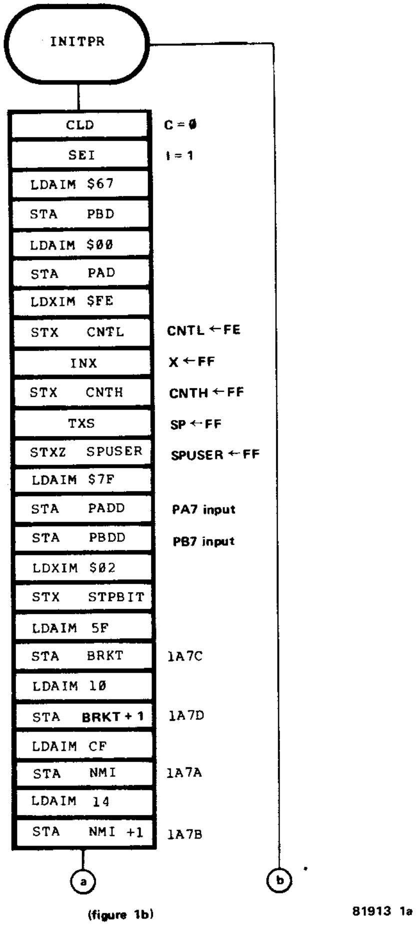


Figure 1a. The first section of the PM initialisation routine. This starts at label INITPR and ends at label STRTBT.

executed while the computer is in the STEP mode. Generally speaking, this follows a non-maskable interrupt as well.

The Initialisation routine INITPR.

As can be seen from figure 1a, this routine starts by dealing with a number of elementary matters. The computer is switched to the binary mode and interrupt requests (IRQ) are disabled (SEI). The input/output (I/O) parameters for the system are also established. By loading the data direction registers PADD and PBDD with the value $ 7F all the port lines except for PA7 and PB7 are programmed as outputs. In other words, port line PB7 will be set to receive data from the cassette recorder, PA7 is prepared to receive serial data from the keyboard and port line PBØ is ready to transmit serial data. Since the value ØØ has been stored in the port A data register (PAD) the outputs PAØ . . . PA6 will go low. This would cause all the segments of the six displays to light (see chapter 7 in Book 2), but this is prevented by the fact that the value $67 has been stored in port B data register (PBD). This means that port lines PB1 and PB2 are both logic one and that port lines PB3 and PB4 are both logic zero. In turn, this means that the unused output (3) of the decoder, IC7, on the Junior Computer main board will be enabled (see figure 9 on page 119 in Book 2). By making port lines PB5 and PB6 go high also, the two indication and relay control circuits involved in the operation of the cassette will be disabled (switched off – see the interface board circuit diagram in Book 3). In addition, by making the output PBØ go high the logic level of the RS 232 line will be low (inverted) whenever the Junior Computer is not transmitting any characters.

Quite a few more preparations are also needed. The computer and user stack pointers are loaded with the value Ø1FF. The memory location STPBIT is loaded with the value Ø2 so that every ASCII character transmitted by the computer ends with two stop bits. Further to this, the BREAK jump vector is loaded with the start address of LABJUN (figure 1b; 'JUNIOR' message) and the NMI vector is loaded with the start address of the STEP routine (also figure 1b). All that is left to be done in figure 1a is to load memory locations CNTL and CNTH with the values FE and FF respectively. The reasons for this are about to be described.

2 Let us turn to figure 1b. The initialisation routine in figure 1a has been run through completely from the first start of PM. The question now is when to go ahead with the second start? As readers will remember from chapter 12 (Book 3), the second start requires the RUBOUT (= RES) key to be depressed. As a result, the corresponding ASCII code, 7F, is transmitted – together with the start bit. What happens next is illustrated in figure 2. The section of program after the label STRTBT examines the serial input line (PA7) with the aid of a BIT instruction. This instruction performs an AND operation between a memory location and the accumulator, but does not store the result of the AND operation in the accumulator. The BIT instruction sets the N flag to the value of bit 7 of the memory location being tested, the V flag to the value of bit 6 and the Z flag is set if the result of the AND operation is zero. The BIT instruction permits the examination of an individual bit without disturbing the value in the accumulator.

In this particular instance the N flag is set to the value of bit 7 of the port A data register – PA7. The following BMI instruction then checks to see whether this value is logic one or logic zero. As soon as a start bit is detected (N = Ø) the processor jumps to subroutine COMTIM. This subroutine is shown in figure 3. During the COMTIM subroutine the computer waits for the start bit to pass, in other words, for port line PA7 to revert to a logic one level. This is because the serial data bits which are to follow are also logic one, since the computer is to check whether the

1b

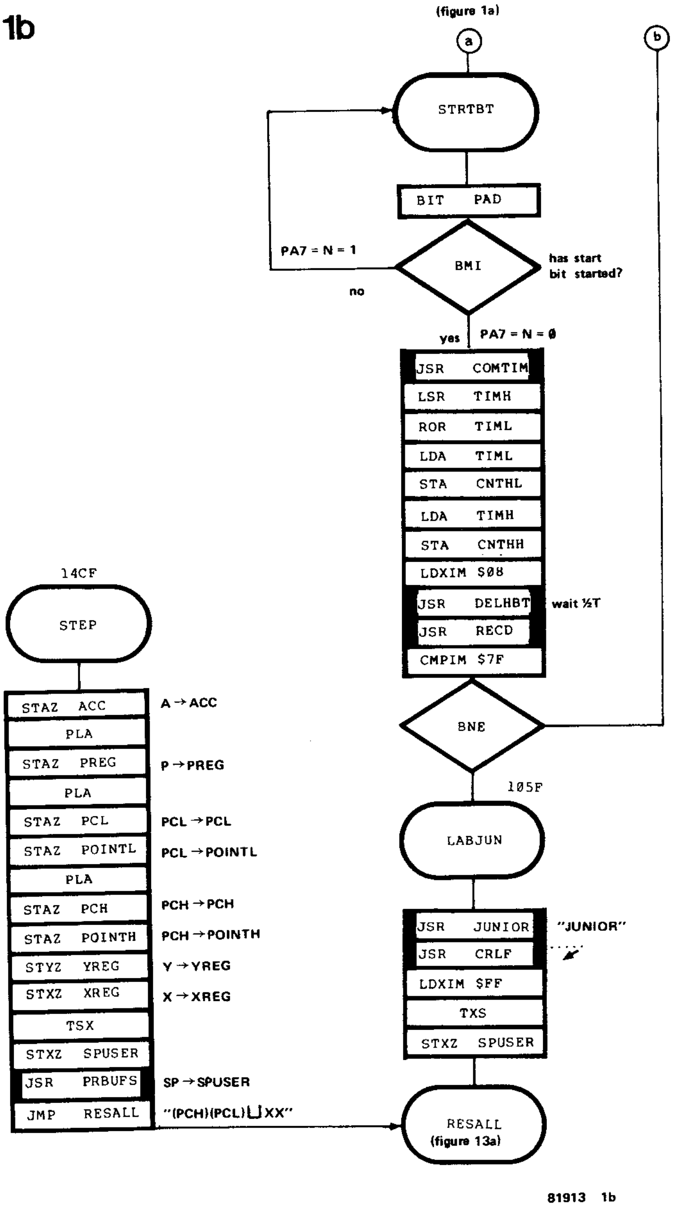


Figure 1b. The continuation of the initialisation routine shown in figure 1a and the STEP routine.

2

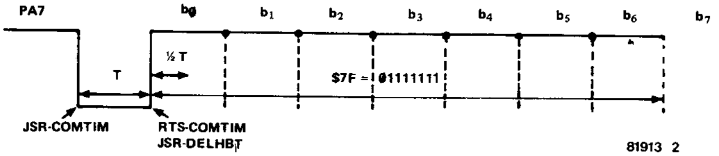


Figure 2. The pulse diagram of a serially transmitted ASCII character. By waiting half a bit period, the logic level of a data bit will always be detected in the middle of the bit period concerned.

3

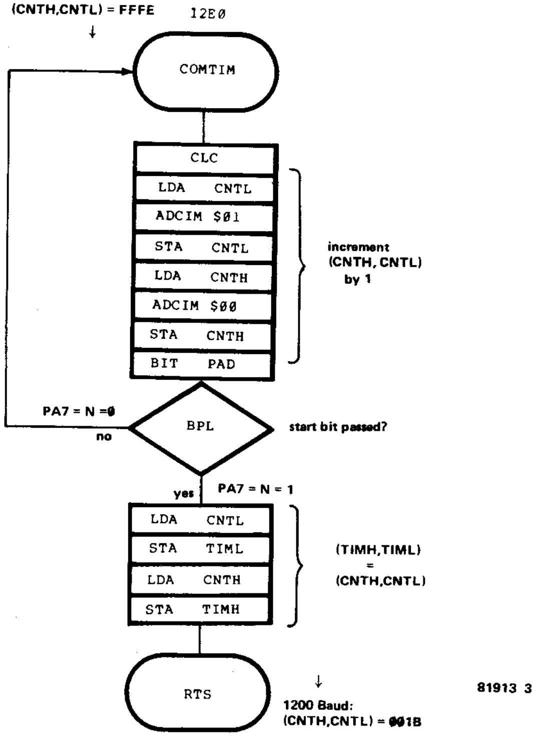


Figure 3. The COMTIM subroutine indirectly determines the length of time the start bit of a received ASCII character (7F) lasts. This establishes the bit period, T.

ASCII code 7F has arrived. This is also illustrated in figure 2.

While the start bit is passing, the 16 bit number contained in memory locations CNTH and CNTL is periodically incremented. This addition takes a certain finite time. The initial combined value of these locations is FFFE and is increased relative to the bit period T. The bit period is determined by the clock transmission speed, which is the Baud rate. The

Baud rate is produced by the clock generator and the UART situated inside the peripheral device being used. The purpose of these locations (CNTH and CNTL) is to allow the Junior Computer to transmit ASCII characters at roughly the same speed as they are received from the peripheral device.
The Baud rate switch has been removed from the Elekterminal, see chapter 12 in Book 3, as there is a fixed rate of 1200 Baud. This corresponds to a value of ØØ in location CNTH and a value of 1B in location CNTL. This can be verified by examining address locations 1AFA (CNTL) and 1AFB (CNTH) after starting PM. These two locations enable the Junior Computer to adapt itself to a wide range of Baud rates produced by the peripheral equipment connected to it. Everything will be just fine as long as the contents of CNTH do not exceed FF, in other words, as long as the Baud rate is sufficiently fast. At the end of the COMTIM subroutine the final value contained in locations CNTH and CNTL are copied into locations TIMH and TIML respectively.

3 Immediately after the COMTIM subroutine the contents of locations TIMH and TIML are altered. All the bits pertaining to the final count are shifted one place to the right. This is accomplished by shifting (LSR) all the bits in TIMH to the right and rotating (ROR) all the bits in TIML to the right. As a result, the figure then stored in locations CNTHH and CNTHL is approximately half the value contained in locations CNTH and CNTL. This is not quite correct where odd numbers are concerned. Locations CNTH and CNTL contain a figure corresponding to the **bit period,** T and the contents of CNTHH and CNTHL correspond to the **half bit period, ½T.** The reason for this is explained in the next point.
(N.B. to be continued in point 6).

4 Before we continue with the rest of the initialisation routine for the PM program – the rest of figure 1b – let us have a small BREAK and look at the subroutines **DELHBT** and **DELBIT** in figure 4. Assuming that the bit period situation is as stipulated in point 3, subroutines DELHBT and DELBIT ensure that the computer waits for either half a bit period (½T) or a full bit period (T). These delays are spent as follows:
Memory locations TIMH and TIML are loaded with either the contents of locations CNTHH and CNTHL or with the contents of CNTH and CNTL, depending on whether the computer is to delay half a bit period (DELHBT) or one full bit period (DELBIT) respectively. The section of program after label CNTDN (countdown) simply subtracts one from the value contained in locations TIMH and TIML until such time as this value becomes negative (FFFE). When this is the case the computer no longer branches back to the label CNTDN, but returns to the main routine from the delay subroutine.
The contents of locations TIMH and TIML are now the same as the initial contents of locations CNTH and CNTL (= FFFE) (see point 2). This means that if we make sure that the program section between label CNTDN and the BCS (branch on carry) instruction takes the same amount of time to execute as the program section between label COMTIM and the BPL (branch on result positive) instruction (figure 3), the computer will auto-

4

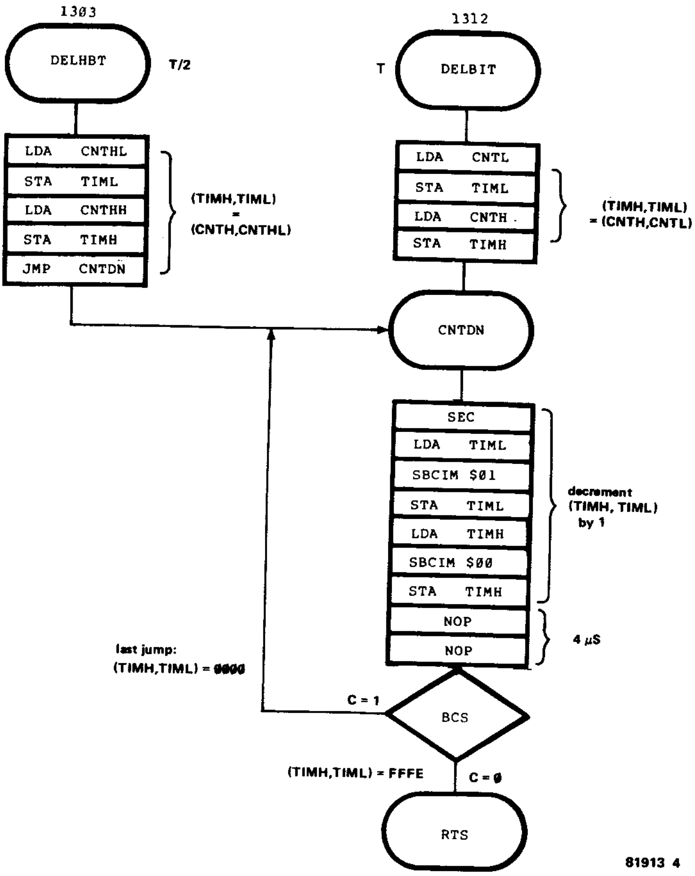


Figure 4. Subroutines DELHBT and DELBIT ensure that the computer waits for half a bit period or a full bit period respectively.

matically wait for a half bit period (label DELHBT) or a full bit period (label DELBIT).

The aforementioned section of the COMTIM subroutine takes 29 μs to execute once. Excluding the two NOP (no operation) instructions between the CNTDN label and the BCS instruction, this section of program takes 25 μs to execute once. If the two NOP instructions are included in the subroutine, the execution time is increased to 29 μs, which is why these seemingly superfluous instructions have been added to the program.

The final value of ØØ1B contained in locations CNTH and CNTL mentioned earlier (point 2) means that the contents of these locations have been incremented a total of 29 times after the initial value of FFFE. This took 29 x 29 = 841 μs. Calling subroutine COMTIM takes another 6 μs and the leading edge of a start bit is unlikely to be detected at exactly the same instant each time (the wait loop is interrupted by the BIT - PAD instruction after the label STRTBT (figure 1b). Thus, the fixed bit period will be around 850 μs. On the other hand, we have an incoming Baud rate of 1200, or 1200 bits per second. This corresponds to a bit period of 833 μs, which is very close to the mark. Fortunately, the PM program is not affected by a difference of a few microseconds.

5 Now for another BREAK, this time to discuss the subroutine **RECCHA** shown in figure 5. This subroutine detects the transmission of an ASCII character to the Junior Computer and stores the relative bit pattern in the correct order in the accumulator. Such a character will correspond to a certain software key operated on the ASCII keyboard. Since the computer features a hardware echo (see chapter 12 in Book 3), the character will also be displayed on the video screen or printed on paper at the same time.

The RECCHA subroutine starts with a wait loop consisting of the instructions BIT - PAD and BMI (branch on result minus). The computer will stay in this loop until such time as a start bit (PA7 = Ø) is detected. The subroutine continues by storing the contents of the X index register in location TEMPB. In the lower right-hand corner of figure 5, just before the RTS (return from subroutine) instruction, we can see that the process is reversed so that the X index register assumes its original contents. Thus, the contents of the X register are not lost during the RECCHA subroutine. During the subroutine the X index register is used as a bit counter. It is loaded with an initial value of Ø8, the number of bits in an ASCII character. After this, the computer waits for a half bit period before reaching label RECA where it waits for a further full bit period. Consequently, the computer will have reached the time when the first bit, bØ, of the ASCII character is received – see figure 2. The machine then checks via the instructions BIT - PAD and BPL to see whether the bit in question is logic one or logic zero and then acts accordingly. The value of the carry flag is made the same as the bit value, after which it is shifted from left to right in location CHA by way of the instruction ROR (rotate right). The contents of the X index register are then decremented and if this value is not equal to zero the computer branches back to label RECA. As the contents of location CHA are shifted one position to the right before the X register is decremented, this location assumes the following status:

When the contents of X = Ø8:	bØ	x	x	x	x	x	x	x
When the contents of X = Ø7:	b1	bØ	x	x	x	x	x	x
When the contents of X = Ø6:	b2	b1	bØ	x	x	x	x	x
When the contents of X = Ø5:	b3	b2	b1	bØ	x	x	x	x
When the contents of X = Ø4:	b4	b3	b2	b1	bØ	x	x	x
When the contents of X = Ø3:	b5	b4	b3	b2	b1	bØ	x	x
When the contents of X = Ø2:	b6	b5	b4	b3	b2	b1	bØ	x
When the contents of X = Ø1:	b7	b6	b5	b4	b3	b2	b1	bØ

All the bits are now correctly positioned in location CHA. Now for the finishing touch. After the label RECC the computer waits for a full bit period – the duration of the first stop bit. The contents of location CHA are then copied into the accumulator and masked by the value 7F. This clears the parity bit sent by the UART. As you will have gathered from

5

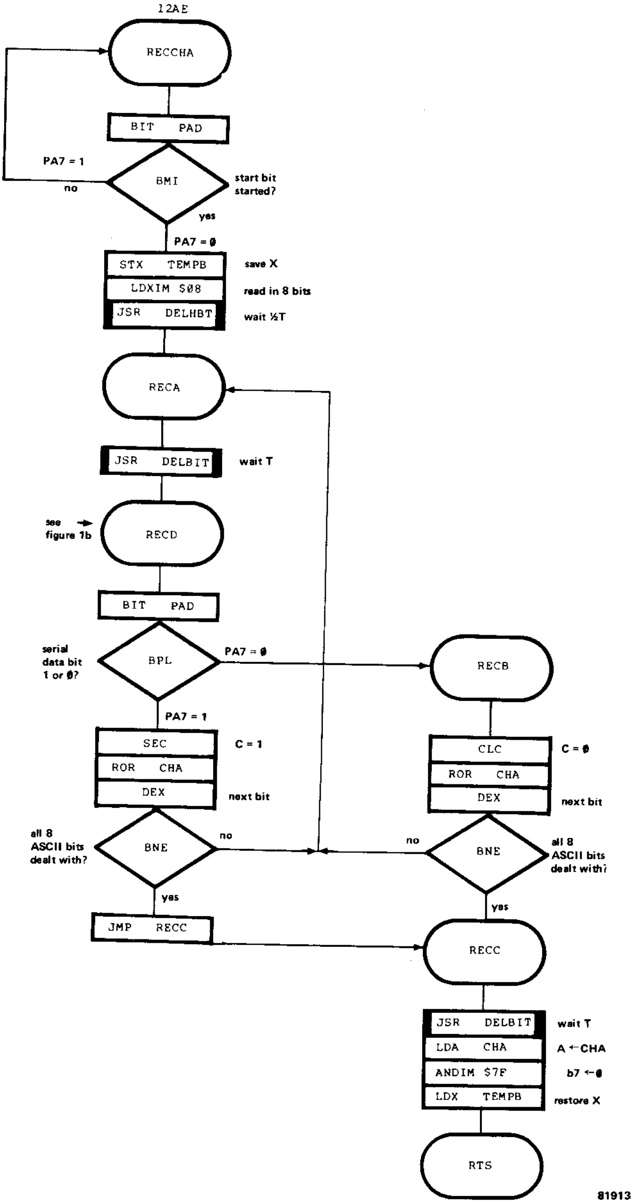


Figure 5. Subroutine RECCHA detects any ASCII character transmitted to the Junior Computer, in other words, any keys depressed on the ASCII keyboard.

chapter 12 in Book 3, we are not interested in this particular bit.
The label RECD in figure 5 may seem a little superfluous, but the computer jumps to this label during the initialisation routine (see figure 1b and point 6). When this occurs, the start bit has already been detected and the contents of the X index register are Ø8.

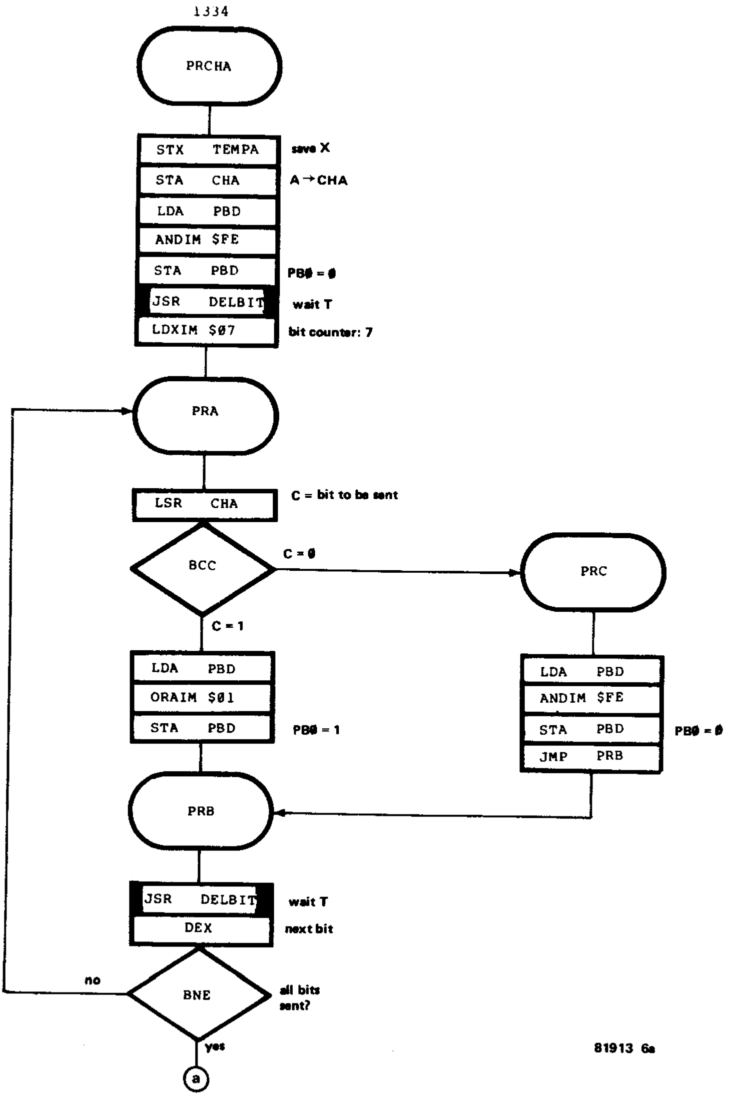
6 (continued from point 3) Now it is time to examine the remainder of the second half of the PM initialisation routine (figure 1b). After the start bit has passed and the various bit period locations have been filled with the correct data, the value Ø8 is stored in the X index register, the computer waits for a half bit period and then jumps to the section in the subroutine RECCHA which reads in all the data concerned with the ASCII character after the start bit (see figure 5 and point 5). Once that has all been taken care of, the computer checks to see whether the ASCII character received has the code 7F. In other words, if the key RUBOUT (= RES) was depressed to indicate that it is time for the second start of PM. If so, the computer continues from the label LABJUN. If not, it returns to the start of the initialisation procedure (INITPR - figure 1a).
Bringing the procedure to a close after the label LABJUN involves printing the message 'JUNIOR' and making sure that any following data is printed at the start of a new line. The two subroutines, JUNIOR and CRLF, will be discussed in detail later on (see points 11 and 12).
Considering the fact that the program also jumps to the label LABJUN when the BREAK key is depressed, the stack is quite likely to 'overflow', with disastrous consequences. For this reason the stack pointer and location SPUSER have to be kept under control (loaded with the value $FF). This then completes the discussion of the PM initialisation routine and brings us to label RESALL, which is the central point in the main routine of the PM program.

7 A second initialisation routine belonging to the PM program is the STEP routine shown in figure 1b. This is executed once the machine returns to PM after a single instruction has been processed in the STEP mode. Generally speaking, this will also be the case after a non-maskable interrupt. During the (INITPR) initialisation routine, the start address of the STEP routine is loaded into the memory location reserved for the NMI jump vector (see figure 1a).
The STEP routine is identical to the SAVE routine contained in the original monitor program. The contents of all the various registers are saved in the well-known memory locations (see chapter 3 in Book 1 and chapter 7 in Book 2). The routine ends by calling subroutine PRBUFS, which we will consider at length in point 10. For the moment it is sufficient to know that the contents of the program counter (= PCH, PCL) are printed at the start of a new line followed by a space and then the contents of the address location concerned. This will be found to be the current work address together with the data contained therein. When a program is executed in the STEP mode, that particular memory location will contain the opcode belonging to the next instruction to be executed. (This instruction is carried out by depressing the R key).

The STEP 'entry' into PM can also be used to jump to PM after the BRK instruction or after a hardware IRQ. In that case, the IRQ jump vector will have to be defined by the operator.

That just about covers the initialisation routines performed by the PM program. Several PM subroutines have also been dealt with 'en passant'. Before we go on to consider the main routine, the time has arrived to examine a few more PM subroutines. These all have something to do with the printing process.

6a



8 Whereas the subroutine RECCHA is the main ASCII receiving routine, the subroutine PRCHA is the main ASCII character transmission routine. The flowchart for the PRCHA subroutine is shown in figures 6a and 6b. This subroutine prints the ASCII character corresponding to the code contained in the accumulator.

To start with, the contents of the X index register are copied into location TEMPA. Again, just before the RTS instruction in figure 6b, the previous value of the X register is restored. Therefore, once again, the contents of the X register remain unaltered during this subroutine. As with the RECCHA subroutine, location CHA plays an important part in the procedure; the contents of the accumulator are copied into this location. Next, port line PBØ is made logic zero by ANDing the contents of PBD with $ FE. Then the computer waits for one full bit period – the start bit has now been transmitted. After this, the contents of the X index register, which acts as a bit counter, are made equal to Ø7.

6b

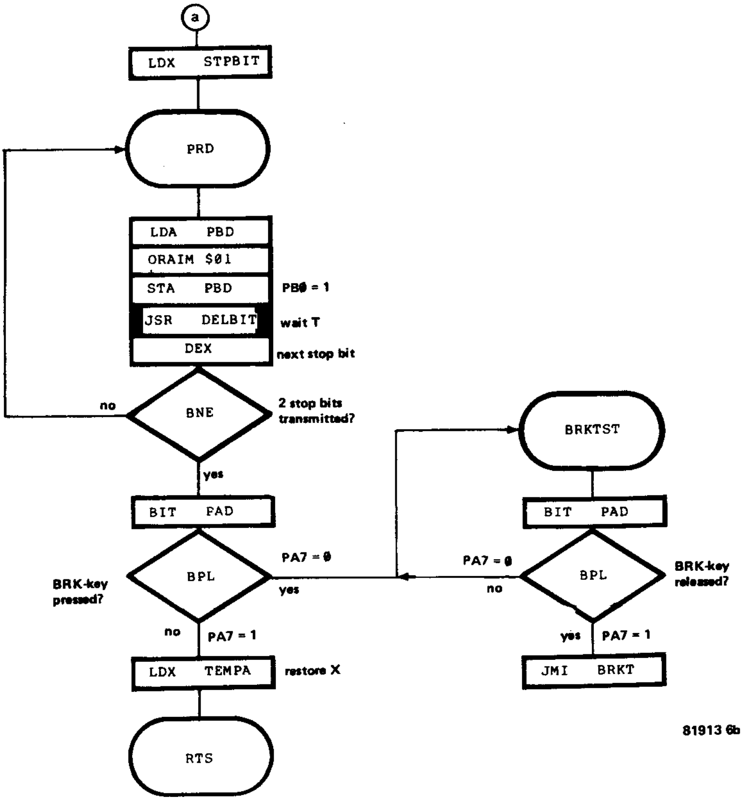


Figure 6. The PRCHA subroutine is devoted to the actual transmission of an ASCII character by the Junior Computer to the video terminal (or printer). The complete flowchart is shown in figures 6a and 6b.

Following this, the value of the carry flag is made to be the same as the value of the transmitted bit (after label PRA in figure 6a). This is accomplished by the instructions LSR - CHA and BCC (branch on carry clear). Port line PBØ is assigned the value of the carry bit; all other port lines, PB1 . . . PB7 remain unaltered.

The procedure is as follows:

Start of PRCHA:	Ø	b6	b5	b4	b3	b2	b1	bØ:	C = x
when X = Ø7	:	Ø	Ø	b6	b5	b4	b3	b2	b1:	C = bØ
when X = Ø6	:	Ø	Ø	Ø	b6	b5	b4	b3	b2:	C = b1
when X = Ø5	:	Ø	Ø	Ø	Ø	b6	b5	b4	b3:	C = b2
when X = Ø4	:	Ø	Ø	Ø	Ø	Ø	b6	b5	b4:	C = b3
when X = Ø3	:	Ø	Ø	Ø	Ø	Ø	Ø	b6	b5:	C = b4
when X = Ø2	:	Ø	Ø	Ø	Ø	Ø	Ø	Ø	b6:	C = b5
when X = Ø1	:	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø:	C = b6

As you can see, bit 7 is zero before the start of the data transmission and is therefore not sent by the computer. This is quite correct, for there is no parity bit involved in the PM program and only seven bits are used for the ASCII code.

This brings us to figure 6b, the transmission of two stop bits. The value Ø2 is held in the memory location STPBIT (see figure 1a) and is now transferred to the X index register. Next, the computer waits for two consecutive bit periods while port line PBØ is made logic zero.

The actual transmission of the ASCII character is now complete and the remainder of the routine in figure 6b affects the BREAK key. As readers know from chapter 12 (Book 3), the character output process performed by the Junior Computer can be halted by depressing the BREAK key. After the ASCII character being dealt with at that moment has been transmitted, the computer checks to see whether or not the BREAK key was operated. If not, the computer will return to the main routine after restoring the contents of the X index register. If the BREAK key was operated, however, the computer waits for it to be released and then performs an indirect jump (shown in figure 6b as the unofficial JMI instruction). The effective jump address is defined in locations $1A7C (BRKT) and $1A7D (BRKT + 1). This effective jump address is that of the label LABJUN (see figure 1a). After depressing (and releasing) the BREAK key, the computer reports 'JUNIOR'.

9 The **PRBYT** subroutine – prints out eight bit data held in the accumulator. This is shown in figure 7a. The data is transmitted in the form of two ASCII characters; one for the high order nibble and one for the low order nibble. This means that each nibble has to be translated into its corresponding ASCII code before transmission. (For instance, if the data 2Ø were not converted into the ASCII codes 32 and 3Ø, the computer would simply acknowledge it as the ASCII code for a space.) For this reason, the subroutine **NIBASC** shown in figure 7b must be executed.

The ASCII codes for numbers Ø . . . 9 (hexadecimal and decimal numeric keys) are $3Ø . . . $39. The ASCII codes for the characters A . . . F (hexadecimal numeric keys) are $41 . . . $46 (see Appendix 7 in Book 3). Let us suppose that the accumulator contains the value ØX, where X = Ø . . . F, prior to the start of NIBASC. Either the value $3Ø will have to be added

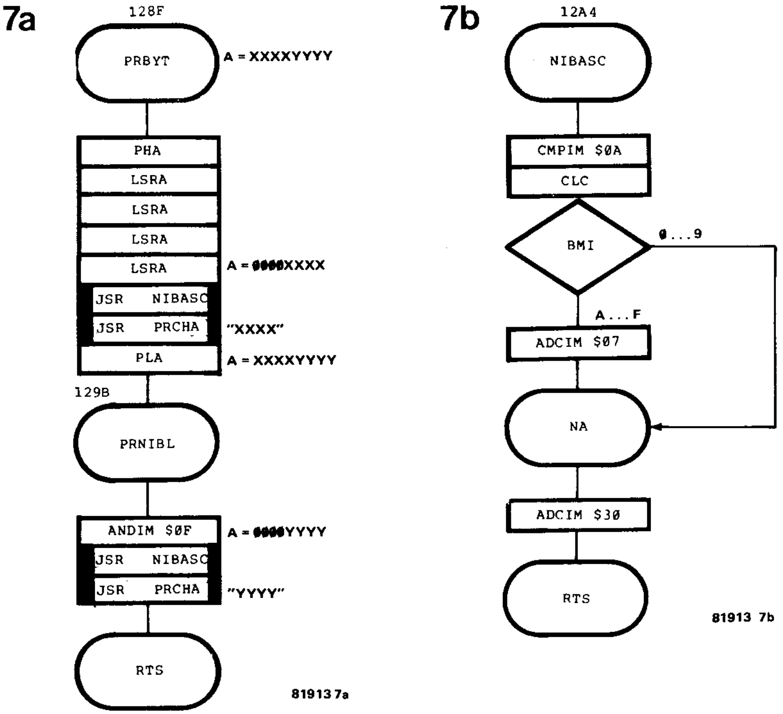


Figure 7. Subroutine PRBYT (figure 7a) prints a data byte in the form of two data nibbles. Each data nibble must first be converted into its corresponding ASCII code. This is accomplished by the subroutine NIBASC in figure 7b.

to the contents of the accumulator to produce the ASCII codes for the numbers Ø . . . 9, or the value $37 will have to be added to produce the ASCII codes for the letters A . . . F. This is exactly what happens during the NIBASC subroutine. The instructions CMP (compare) immediate and BMI (branch if minus) determine whether the contents of the accumulator are equal to or greater than $ØA. If the value in the accumulator is between ØØ . . . Ø9, the NIBASC routine adds $3Ø to it. If the value is between ØA . . . ØF, $Ø7 is added followed by $3Ø, making the required total of $37.

Back to the PRBYT subroutine in figure 7a. This starts by storing the contents of the accumulator in the stack (PHA). Then the contents of the accumulator are shifted to the right four times in succession. As a result, the previous low order nibble is replaced by the high order nibble and the new high order nibble becomes zero.

Now the program calls subroutine NIBASC (figure 7b); the original high order nibble is translated into its correct ASCII code. This is then displayed on the video screen or printer via the subroutine PRCHA (see point 8 and figures 6a and 6b). After this, the original contents of the accumulator are restored (PLA) and the program moves on to label PRNIBL to display the ASCII equivalent of the original low order nibble. This section of the routine can be used on its own to print a single nibble. By masking the high order nibble, the contents of the accumulator are prepared for processing by the subroutines NIBASC and PRCHA. After the low order nibble has been printed the computer returns to the main routine.

10 The subroutine **PRBUFS** (see figure 8) is devoted to printing the current 'work address'. This address information is held in the buffers POINTH (high order address byte) and POINTL (low order address byte). Not only is the actual work address printed, but also the contents of that address location. The PRBUFS subroutine can be explained in very few words. The computer always starts to print at the beginning of a fresh line (subroutine CRLF, see point 12). Then the high order address byte (ADH) of the current work location is printed by means of the subroutine PRBYT which we are already familiar with. The same procedure occurs for the low order address byte (ADL). Then a space is 'printed' via the subroutine PRSP (see point 12). The contents of the memory location defined by the buffers POINTH and POINTL are then loaded into the accumulator by means of post indexed indirect addressing (where the contents of the Y index register are ØØ). Finally, the two data nibbles are displayed via the PRBYT subroutine and another space is printed.

8

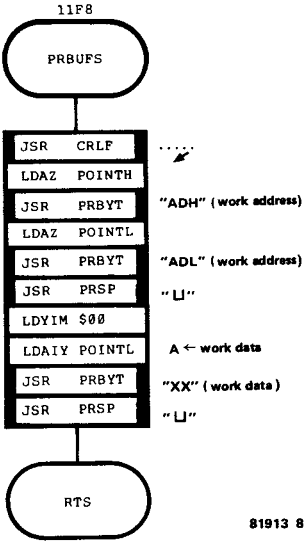
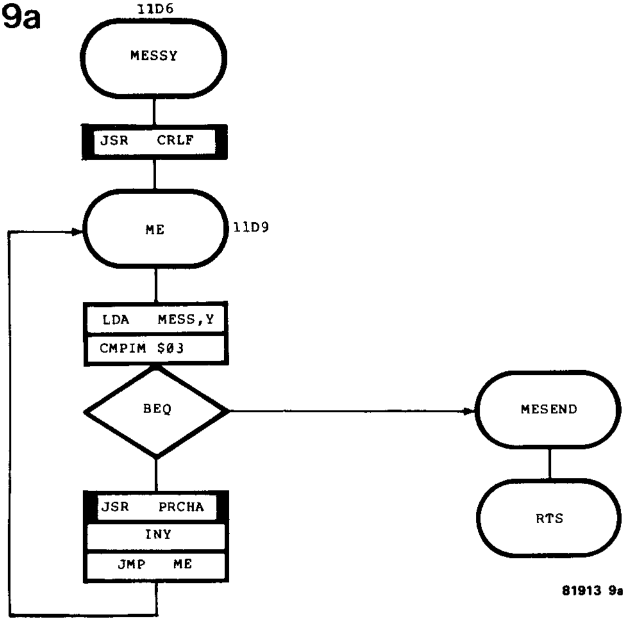


Figure 8. Subroutine PRBUFS is used to print the current work address and its corresponding work data.

9a



9b

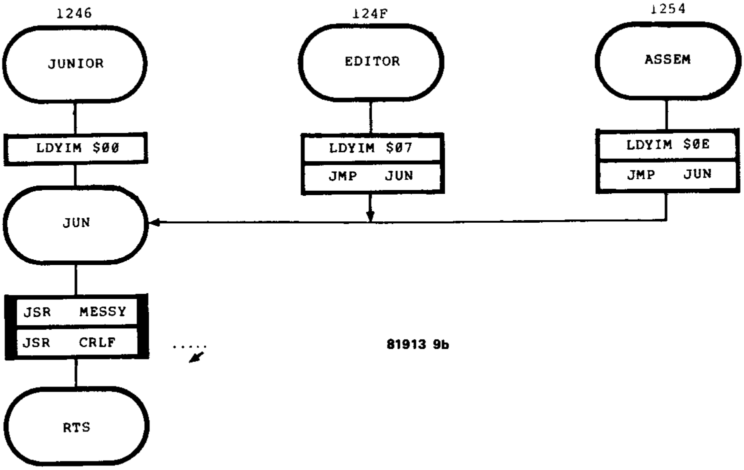


Figure 9. Subroutine MESSY (figure 9a) prints a computer message; in some instances this constitutes an error report. Subroutines JUNIOR, EDITOR and ASSEM in figure 9b use subroutine MESSY to report the texts 'JUNIOR', 'EDITOR' and 'ASSEMBLER', respectively. (The last two reports do not apply during the PM routine.)

11 The subroutine **MESSY**, shown in figure 9a, allows a 'string' of text to be displayed, in other words, a message such as 'JUNIOR' or 'WHAT?' etc. The text concerned is stored in ASCII coded form in a number of consecutive memory locations in the look-up table MESS, starting from address $13BD and ending at $141D (see the listing at the back of this book). One memory location is reserved for each ASCII character. The value contained in the Y index register determines how many locations from the start of the table (13BD) that the particular text string begins. A text string ends with the 'end-of-text' (EOT) character $Ø3. Effectively, what happens is that initially the contents of the Y index register correspond to the start address of a particular message and are then incremented until the EOT character is detected.

The subroutine MESSY (figure 9a) starts by calling the subroutine CRLF (see point 12). This positions the cursor (or printer carriage) to the start of a new line. Then the instruction LDA MESS, Y loads the ASCII value of the first/next character in the message into the accumulator. The initial value of the Y index register must be established before the jump to MESSY. The computer then checks to see whether the EOT character (Ø3) has been reached, for this would end the subroutine (RTS). If not, it continues to print the rest of the string. In some instances the rest of MESSY after the label ME is used as a subroutine. This means that the printing does not start at the beginning of a new line.

Here is an example of how MESSY is used: subroutines JUNIOR, EDITOR and ASSEM in figure 9b. Further on in the chapter a lot more examples are given. First, a particular value is loaded into the Y index register, then subroutine MESSY is called and the relevant text ('JUNIOR', 'EDITOR' or 'ASSEMBLER') is printed followed by a carriage return and line feed

10

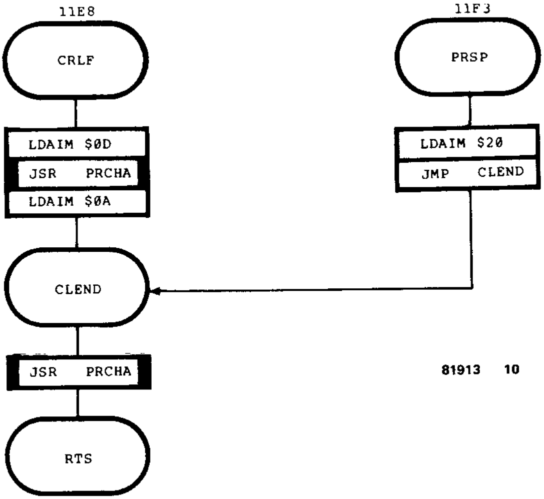


Figure 10. Subroutines CRLF (start afresh from the beginning of a new line) and PRSP (print a space).

(CRLF) to ensure that any further data etc. is printed on a new line.
N.B. Subroutines EDITOR and ASSEM are not used in PM.

12 Subroutines **CRLF** and **PRSP** (see figure 10) are used frequently to provide clarity and legibility. The subroutines are extremely straightforward. It is simply a question of loading the ASCII code of the required command into the accumulator and 'printing' it with the aid of the PRCHA subroutine. The carriage return (CR) command (= back to the start of the same line) has the ASCII code ØD and the line feed (LF) command (= move to the same position on a new line) has the ASCII code ØA. The space routine (PRSP) is also straightforward: load the ASCII value 2Ø into the accumulator and jump to label CLEND, leaving a space.

13 The subroutine **HEXNUM** (figure 11) plays an important role in determining whether an ASCII character received by the Junior Computer belongs to a hexadecimal key or not. If indeed a hexadecimal key was depressed, the data is added, after translation from the ASCII code, to the contents of the data buffers INH and INL. If not, the computer asks 'WHAT?' (short for 'what is going on?').
Before dealing with the subroutine HEXNUM in detail, let us consider this subroutine **ASHETT** (figure 12), which establishes whether hexadecimal data is involved and if so distills the data out of the received ASCII code.
Looking at figure 12, let us assume that the particular ASCII code is stored in the accumulator, codes 3Ø . . . 39 (keys Ø . . . 9) and codes 41 . . . 46 (keys A . . . F) are valid; all other codes (and keys) will be considered non-valid. The ASHETT subroutine contains four consecutive compare instructions, each followed by a branch instruction. These determine whether the incoming ASCII code is valid or not. The non-valid ones are filtered out, so to speak. Once a non-valid character is detected, the N flag is set and the Z flag is cleared before the computer returns to the HEXNUM subroutine.
The ASCII code of a valid character has to be translated into the corresponding data nibble before it can be used by the Junior Computer. This is accomplished after the label VALIT in the ASHETT subroutine. The ASCII characters 3Ø . . . 39 can be translated to ØØ . . . Ø9 simply by masking the high order nibble of the accumulator contents (AND #ØF). Where the ASCII characters 41 . . . 46 are concerned, however, the value Ø9 has to be added to the accumulator contents first, before the masking process. At the end of the ASHETT subroutine, the high order nibble of the accumulator contents will always be zero and (nine times out of ten!) the low order nibble will not be equal to zero. This means that both the N flag and the Z flag will be reset after the return.
Now to continue with the HEXNUM subroutine (figure 11). This starts by calling the ASHETT subroutine (figure 12) which we have just discussed. This is followed by a BMI instruction to test the status of the N flag. A non-valid ASCII character leads the computer to the label HNUB, which is where the text 'WHAT?' is dealt with. Firstly, the value 46 is loaded into the Y index register. Then subroutine MESSY is called (see point 11 and figure 9a). After the error message 'WHAT?' has been printed, the cursor is positioned at the beginning of a new line via the CRLF subroutine. Before

11

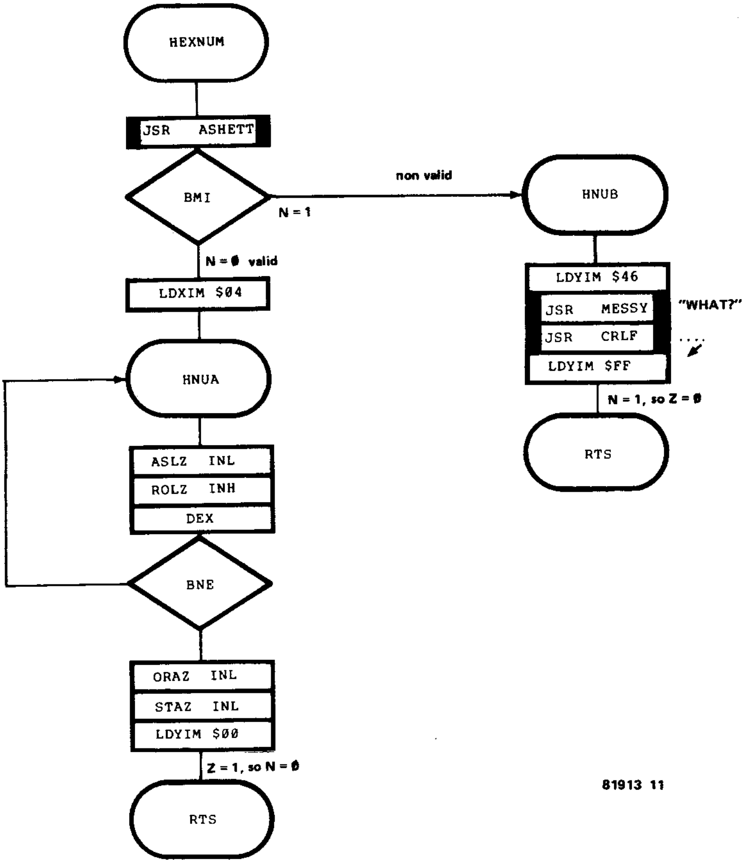


Figure 11. Subroutine HEXNUM processes the data nibble corresponding to a hexadecimal key. This data nibble is stored in the data buffers INH and INL.

returning to the main routine, the value FF is loaded into the Y index register. This means that the N flag will be set and the Z flag will be reset. If a valid hexadecimal ASCII character was received, the corresponding data nibble is 'set aside' in the data buffers INH and INL. This is done by shifting and rotating the bits in these buffers to the left four times. The way in which this is accomplished is identical to similar operations in the original monitor program (see Book 2). For this reason we only require a brief summary here:

- The low order nibble contained in buffer INL is the same as the data nibble just received.
- The high order nibble in buffer INL is the same as the previous low order INL nibble.

12

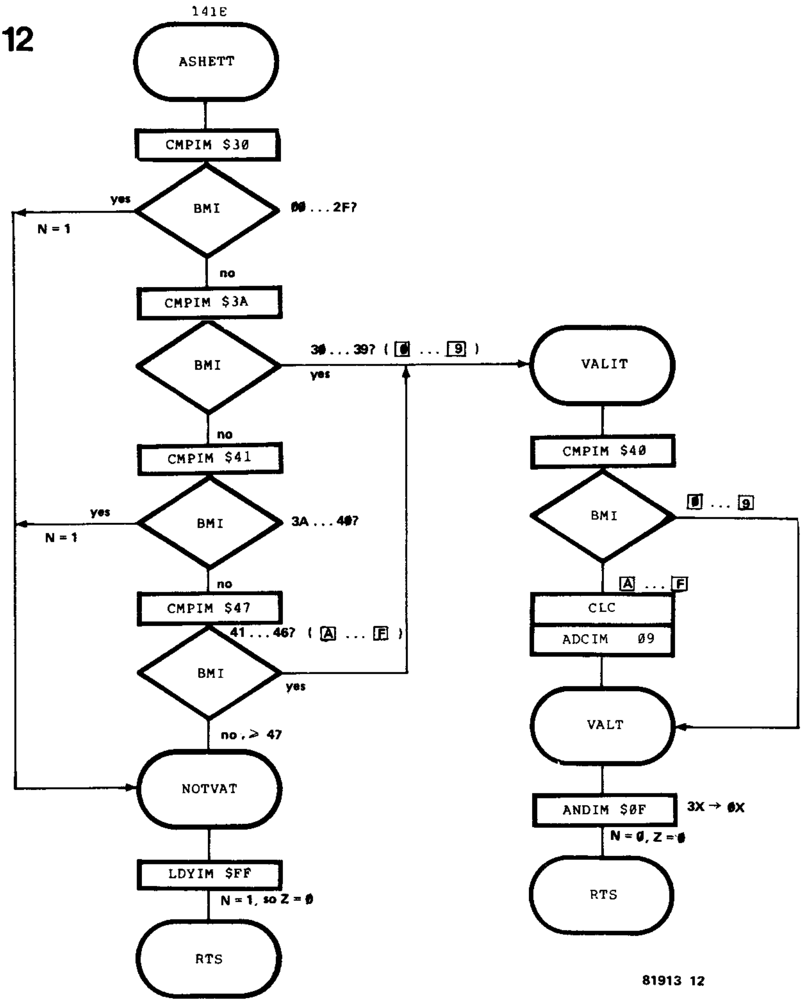


Figure 12. Subroutine ASHETT is called during the HEXNUM subroutine (figure 11) to check the validity of a supposedly hexadecimal key. If a non-valid key is depressed, the error message 'WHAT?' is displayed.

- The low order nibble in buffer INH is the same as the previous high order INL nibble.
- The high order nibble in buffer INH is the same as the previous low order INH nibble.
- The previous high order INH nibble has disappeared.

In other words, the contents of buffers INH and INL always correspond to the data belonging to the last four hexadecimal keys to be depressed,

unless the buffers were cleared in the meantime (subroutine RESIN, see point 14). If subroutine HEXNUM is called less than four times in succession, the unused nibbles will be zero.
Just before the end of the HEXNUM subroutine, the Y index register is loaded with the value ØØ. As a result, the Z flag will be set and the N flag reset, which is exactly the opposite of what happens when a non-valid ASCII code is received.

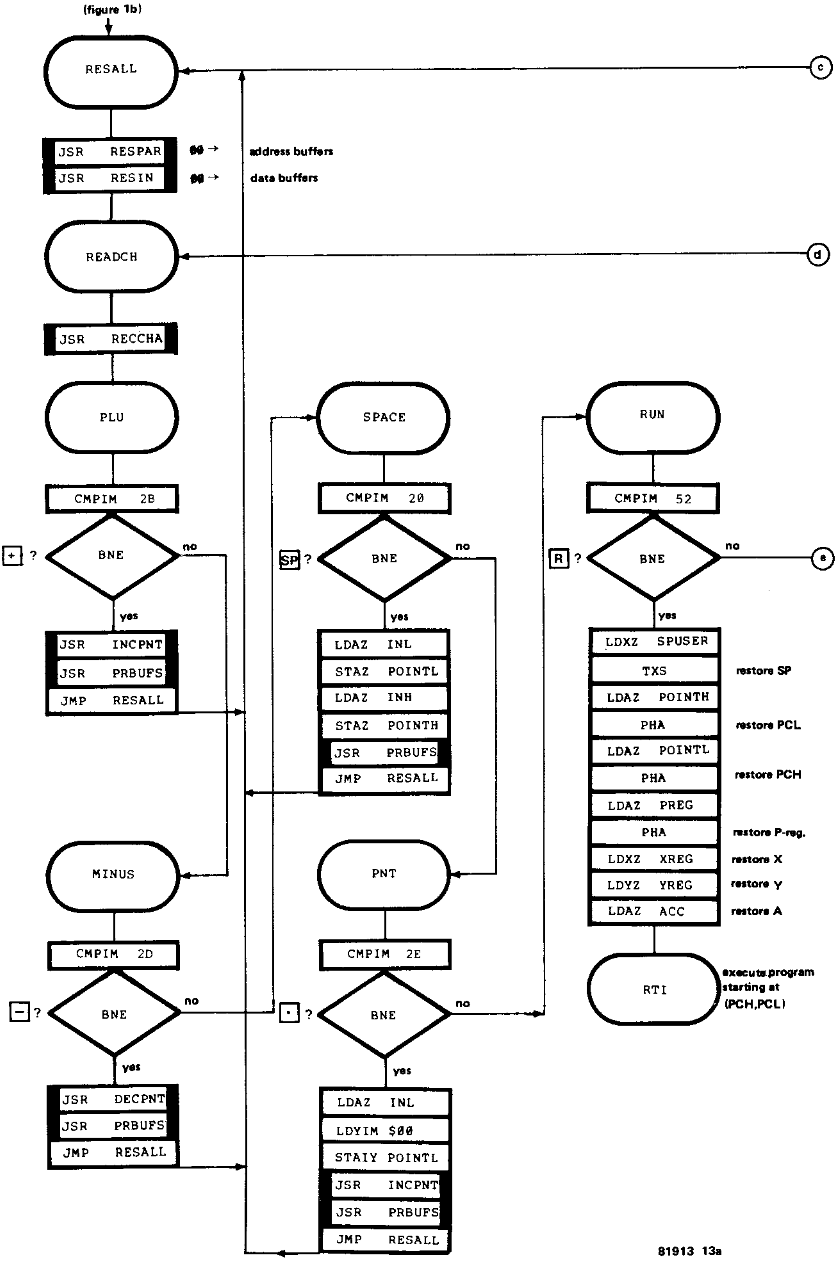
14 The **main PM routine** and the remaining PM subroutines will now be dealt with. The complete flowchart is shown in figures 13a . . . 13c. The central point of the main routine is constituted by the subroutine RESALL (top left-hand corner of figure 13a). Following this label the computer calls subroutines **RESPAR** and **RESIN** (figures 14a and 14b respectively). The subroutine RESPAR has the task of clearing the two address buffers PARA and PARB (both 16 bits), while the RESIN subroutine performs the same function for the data buffers INH and INL. This occurs as soon as the contents of the buffers have become superfluous after a particular key operation has been performed. The next label, READCH, constitutes another important part of the main PM routine, during data processing in particular (for instance, the entry of a work address or work data). Straight after the READCH label the PM routine jumps to subroutine RECCHA: the computer waits for a (new) key to be depressed. Then, the routine tests the various possibilities:
Is it the '+' key? If so carry out the '+' key routine. If not, is it the '–' key? If so, carry out the '–' key routine. If not, check to see whether it was the space key, etc. etc.

15 The **PLUS** key routine; the sequence of instructions in figure 13a following on from the PLU label. As mentioned in chapter 12, the work address has to be incremented and the new work address together with the data contained therein has to be printed on a new line. To start with, the subroutine **INCPNT** is called (see figure 15a). The current work address is indicated by the address pointer POINTH and POINTL. Incrementing this pointer by one (INCPNT, figure 15a) is carried out in the usual manner. After this, the new work address and the corresponding data is printed (subroutine PRBUFS, see point 10 and figure 8) and the program returns to the label RESALL.

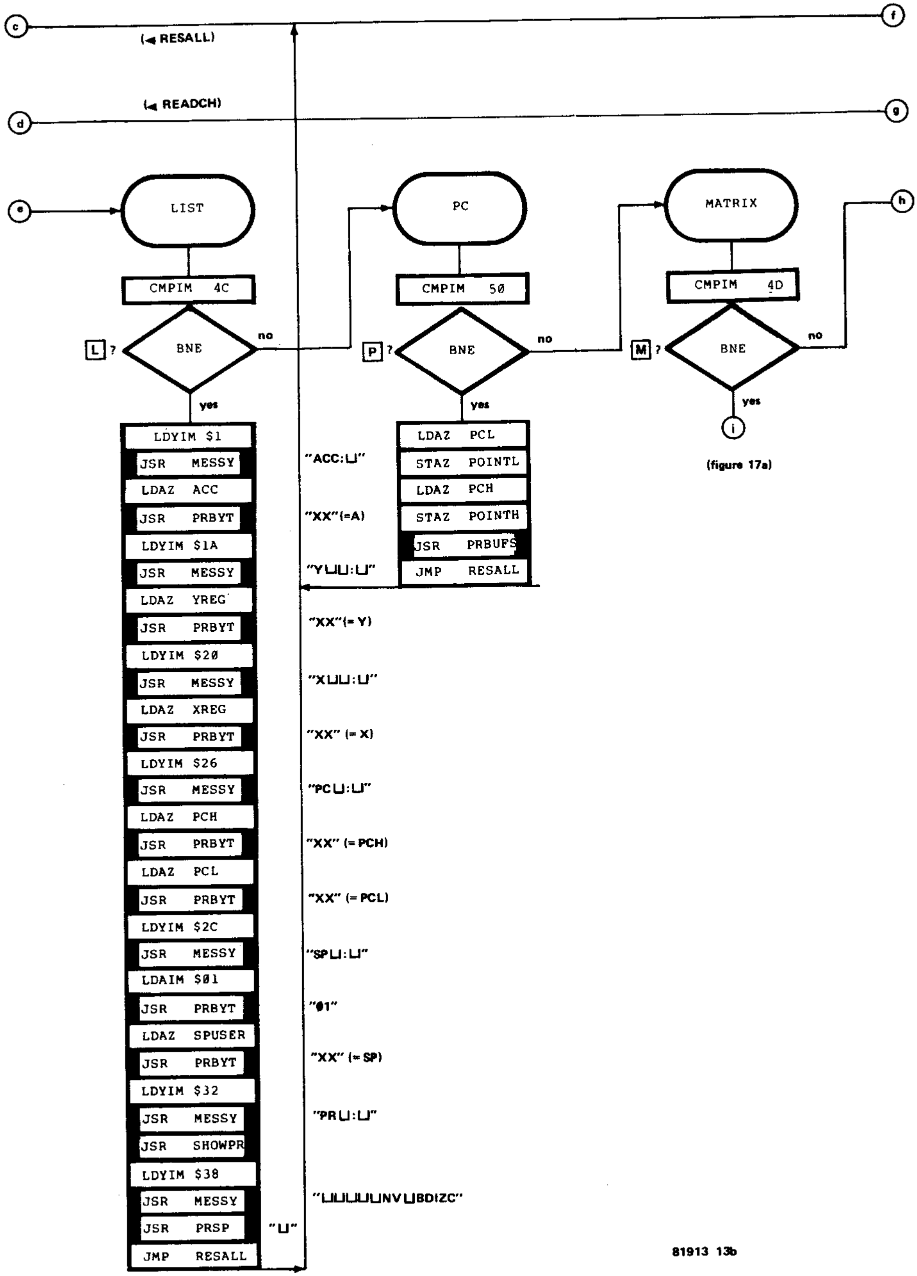
16 The **MINUS** key routine; the sequence of instructions following the MINUS label in figure 13a. Here, the current work address has to be decremented by one; the new work address and the inherent data has to be printed. This is very similar to the PLUS key routine described in point 15. The only difference being that the subroutine **DECPNT** is called rather than the subroutine INCPNT. In this instance the contents of POINTH and POINTL are decremented by one, otherwise everything else is the same as before.

17 The **SPACE** key routine; the sequence of instructions following the SPACE label in figure 13a. A work address is entered and displayed. The address data is entered by previously depressing up to four hexadeci-

13a


(figure 1b)
RESALL
JSR RESPAR
JSR RESIN
00 → address buffers
00 → data buffers
READCH
JSR RECCHA
c
d
PLU
CMPIM 2B
+ ?
BNE
no
yes
JSR INCPNT
JSR PRBUFS
JMP RESALL
SPACE
CMPIM 20
SP ?
BNE
no
yes
LDAZ INL
STAZ POINTL
LDAZ INH
STAZ POINTH
JSR PRBUFS
JMP RESALL
RUN
CMPIM 52
R ?
BNE
no
e
yes
LDXZ SPUSER
TXS
restore SP
LDAZ POINTH
PHA
restore PCL
LDAZ POINTL
PHA
restore PCH
LDAZ PREG
PHA
restore P-reg.
LDXZ XREG
restore X
LDYZ YREG
restore Y
LDAZ ACC
restore A
RTI
execute program starting at (PCH,PCL)
MINUS
CMPIM 2D
− ?
BNE
no
yes
JSR DECPNT
JSR PRBUFS
JMP RESALL
PNT
CMPIM 2E
· ?
BNE
no
yes
LDAZ INL
LDYIM $00
STAIY POINTL
JSR INCPNT
JSR PRBUFS
JMP RESALL
81913 13a

13b


c
(◄ RESALL)
f
(◄ READCH)
d
g
e
LIST
CMPIM 4C
L ?
BNE
no
yes
LDYIM $1
JSR MESSY
"ACC:⊔"
LDAZ ACC
JSR PRBYT
"XX"(=A)
LDYIM $1A
JSR MESSY
"Y⊔⊔:⊔"
LDAZ YREG
JSR PRBYT
"XX"(= Y)
LDYIM $2Ø
JSR MESSY
"X⊔⊔:⊔"
LDAZ XREG
JSR PRBYT
"XX" (= X)
LDYIM $26
JSR MESSY
"PC⊔:⊔"
LDAZ PCH
JSR PRBYT
"XX" (= PCH)
LDAZ PCL
JSR PRBYT
"XX" (= PCL)
LDYIM $2C
JSR MESSY
"SP⊔:⊔"
LDAIM $Ø1
JSR PRBYT
"Ø1"
LDAZ SPUSER
JSR PRBYT
"XX" (= SP)
LDYIM $32
JSR MESSY
"PR⊔:⊔"
JSR SHOWPR
LDYIM $38
JSR MESSY
"⊔⊔⊔⊔⊔NV⊔BDIZC"
JSR PRSP
"⊔"
JMP RESALL
PC
CMPIM 5Ø
P ?
BNE
no
yes
LDAZ PCL
STAZ POINTL
LDAZ PCH
STAZ POINTH
JSR PRBUFS
JMP RESALL
MATRIX
CMPIM 4D
M ?
BNE
no
yes
h
i
(figure 17a)
81913 13b

13c

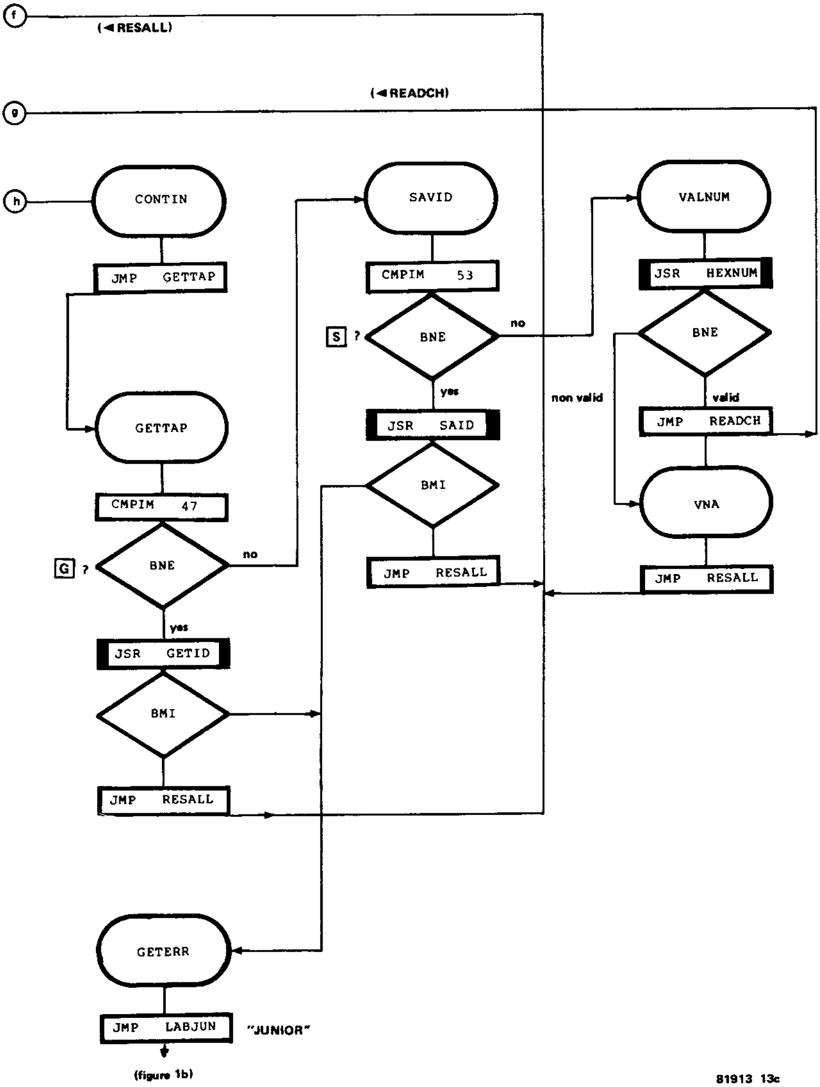
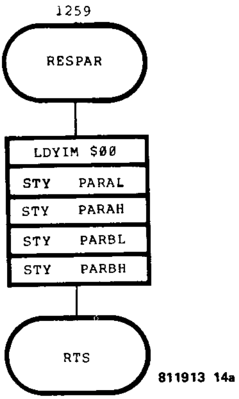


Figure 13. The main PM routine which processes the various key functions can be split up into three sections. These are shown in figures 13a, 13b and 13c. Since the M key routine contains so many instructions, it is shown separately in figures 17a and 17b.

14a



14b

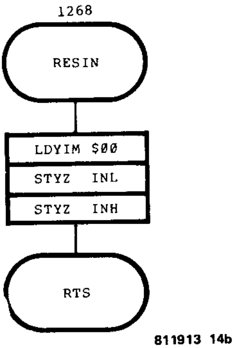
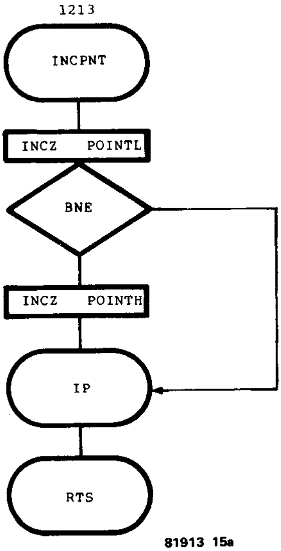


Figure 14. Subroutine RESPAR (figure 14a) makes sure that the address buffers PARA and PARB are cleared for the entry of the first and last address, respectively. Subroutine RESIN (figure 14b) clears the data buffers INH and INL.

15a



15b

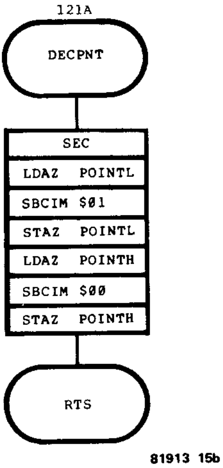


Figure 15. Subroutines INCPNT (figure 15a) and DECPNT (figure 15b) serve to increment or decrement the contents of the address pointer POINT, respectively.

mal keys and is stored in the buffers INH and INL. The contents of INH and INL are then copied into POINTH and POINTL. The work address is now established and can be printed. For this subroutine PRBUFS is called (see figure 8 and point 10). Following that the program returns to RESALL.

18 The **FULL STOP** key routine; the sequence of instructions in figure 13a following the PNT label. Data is loaded into a previously specified work address. One of the keys '+', '–' or 'space' is implemented. The contents of the data buffer INL are first copied into the accumulator, buffer INH is not required here as only one data byte is concerned. If more than two data keys are depressed, the two most recent ones will comprise the work data.

Once the work data is stored in the accumulator, it is transferred to the location reserved for the work address by means of the instruction STA - POINTL, Y. Afterwards, the work address is incremented by one (subroutine INCPNT) and displayed together with the data contained therein (subroutine PRBUFS). This makes preparation for the following work data entry.

19 The **R** key routine; all the instructions following the RUN label in figure 13a. This routine is identical to the one belonging to the GO key in the original monitor program (see figure 5a on page 108 of Book 2). During the STEP routine (see figure 1b) the contents of all the registers were stored away. During the RUN routine the original register contents are restored. The RTI (return from interrupt) instruction which ends the RUN routine ensures that a program is carried out from the start address specified by the contents of locations PCH and PCL, or, if the program is being executed in the STEP mode, that the following instruction is carried out. The opcode of this instruction is stored at the address specified by PCH and PCL (PC = program counter).

20 Before discussing the **L** key routine, let us consider a special subroutine, **SHOWPR** (see figure 16), which is used during a program listing to make sure that the contents of the status (P) register are printed bit by bit.

The SHOWPR subroutine starts by copying the contents of the status register (PREG) into location PRTEMP. In addition, the X index register, which acts as the bit counter, is loaded with an initial value of Ø8. The position of the various flags in the status register can be seen from figure 1b on page 61 of Book 1.

After the label SPRA in the SHOWPR subroutine the contents of location PRTEMP are shifted to the left eight times in succession by means of the instruction ASL - PRTEMP. This means that initially bit 7 is shifted into the carry flag. Depending on whether the carry bit is now logic one or logic zero, either a 1 or Ø will be printed by the subroutine PRNIBL (see point 9 and figure 7a). After this the next (and subsequent) bit is prepared. The process is repeated until the contents of the X index register are zero.

16

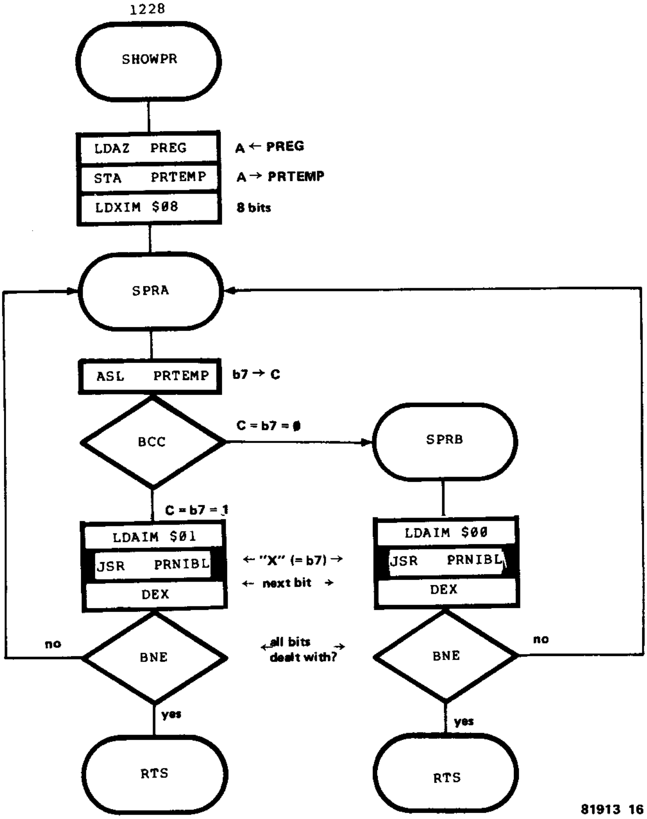


Figure 16. Subroutine SHOWPR is called during a certain phase of the L key routine and prints the contents of the status register bit by bit (the state of the various flags).

This is what happens:

when X = Ø8 :	the value of the N flag is printed;
when X = Ø7 :	the value of the V flag is printed;
when X = Ø6 :	a random value is printed;
when X = Ø5 :	the value of the B flag is printed;
when X = Ø4 :	the value of the D flag is printed;
when X = Ø3 :	the value of the I flag is printed;
when X = Ø2 :	the value of the Z flag is printed;
when X = Ø1 :	the value of the C flag is printed.

21 The L key routine; the section of program following the LIST label in figure 13b. The task of this subroutine is to print the memory contents in an orderly manner. The LIST routine is made up of a whole series of instructions and many jumps to other subroutines. This

subroutine can be dealt with rapidly by looking at a practical example (see chapter 12 in Book 3). The text that is printed during a certain phase is shown in inverted commas. The inverted commas themselves, however, are not printed! How a space is symbolised is also shown in figure 13b. All the subroutines that are called by the LIST routine have been discussed earlier: MESSY in point 11, PRBYT in point 9, SHOWPR in point 20 and PRSP in point 12.

22 The **P** key routine; the sequence of instructions following the label PC in figure 13b. Depressing the P key causes the contents of the program counter (PCH and PCL) to be copied into the current work address. The function of the P key is identical to that of the PC key of the original monitor program. This key routine can again be explained in very few words.
The contents of location PCH and PCL are copied into locations POINTH and POINTL. This gives a new work address, which is printed via the subroutine PRBUFS.

23 The **M** key routine: the large number of instructions following the label MATRIX in figure 13b. In fact, there are so many that they require extra space: figures 17a and 17b; Figures 13b and 17a are joined via 'junction' ①.
At the top left-hand corner of figure 17a, the routine starts by printing the message 'HEXDUMP:'. This is accomplished by loading the Y index register with the value $ 52 and calling the MESSY subroutine. This brings us to the subroutine **INPAR** which we will describe now (the subject of the M key routine is continued in point 25).

24 The **INPAR** subroutine (see figure 18) is either partially or completely run during the routines associated with the M, G and S keys. These keys are used to enter certain parameters, as you will remember from chapter 12. The INPAR subroutine controls the parameter setting:

enter first address data - comma key enter last address data or ID - CR key

The INPAR subroutine starts by waiting for a key to be depressed via the subroutine RECCHA. The question now is: is it the comma key? If not, either valid (hexadecimal) data or non-valid data (all other keys except ',') must have been entered. The HEXNUM subroutine (see point 13 and figure 11) processes valid hexadecimal data in the buffers INH and INL.
As soon as the comma key is depressed (label IPA in figure 18) the first data entry will be complete. This information is then transferred to the address buffers PARAH and PARAL for the first address. In the case of a hex dump this memory location will contain the first data to be printed (see point 25) or it will be the first address of a data block which is to be stored on cassette (subroutine SAID - see point 28). Next, the data buffers INH and INL are cleared (subroutine RESIN).
This brings us to the label IPB in figure 18. Again, the computer waits for a key to be depressed and then checks to see whether it was the carriage return key. If so, it means that all the data has been entered. If not, it

17a

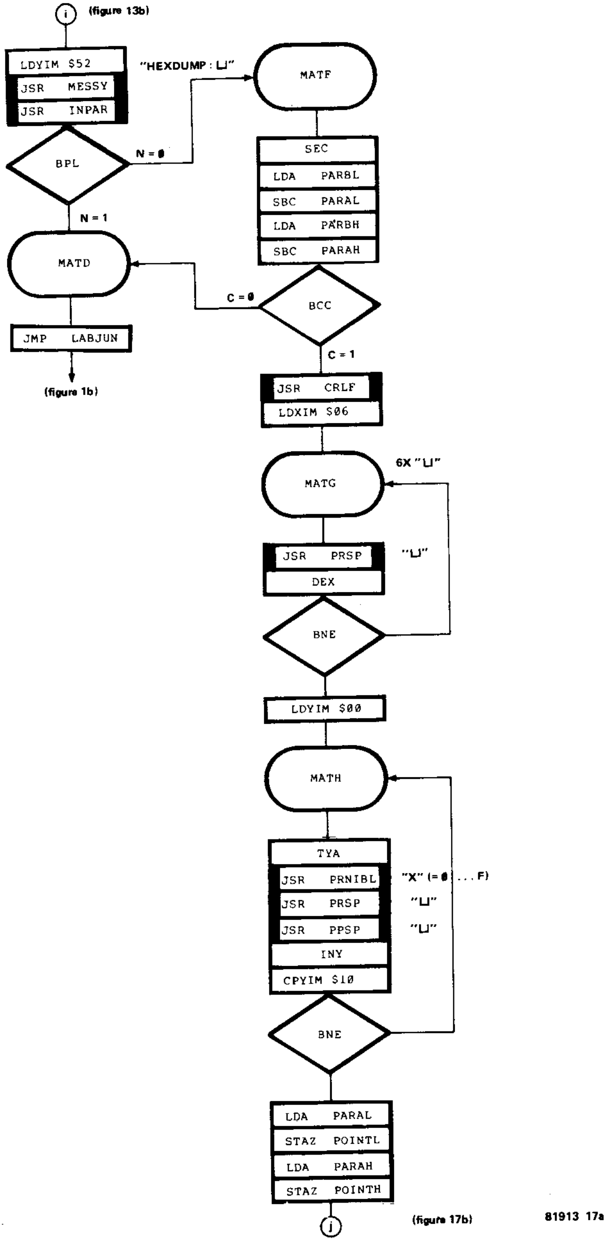
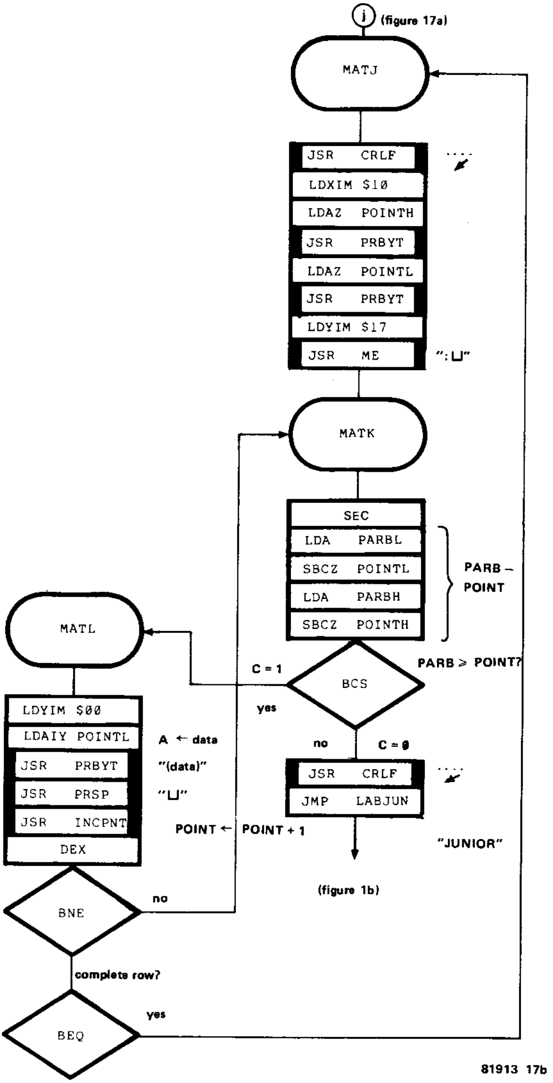


Figure 17. Apart from the first two instructions (see figure 13b) all the instructions pertaining to the M key routine are shown in figures 17a and 17b.

17b



must have been more data and the procedure just described above is repeated. If on the other hand, it was the CR key, the second data entry will have been completed and the program will have reached the label IPC. This is where the last data to be entered is stored in locations PARBH and PARBL. In this instance the buffers INH and INL are not cleared, unlike when the comma key was detected. Finally, the contents of the Y index register are cleared, which sets the Z flag and resets the N flag.

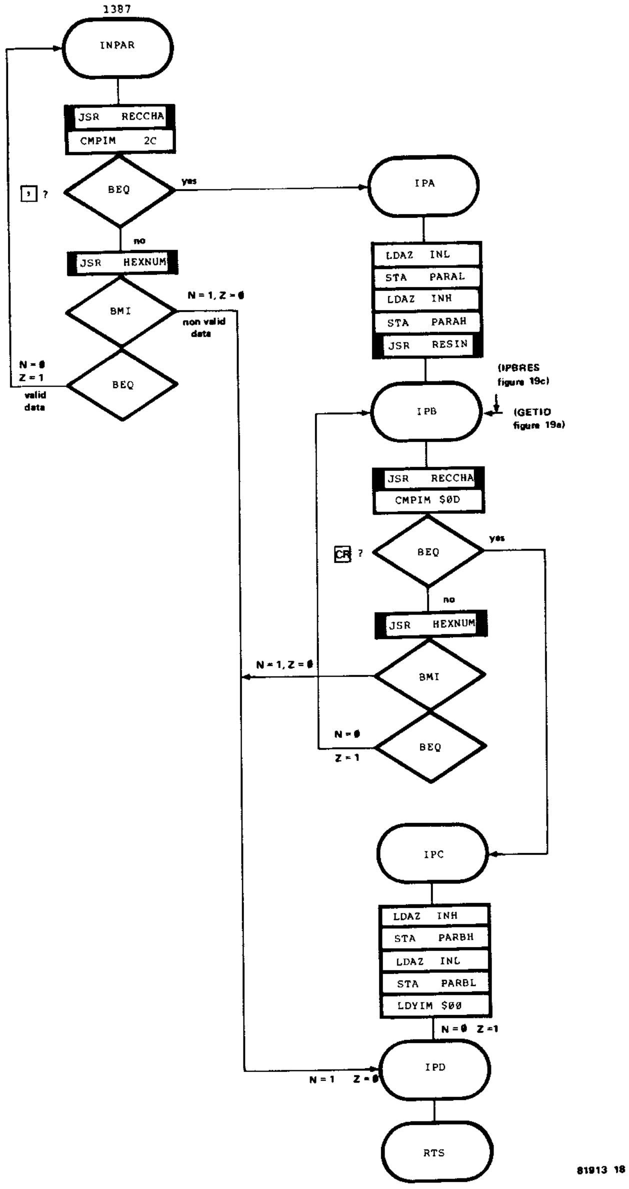


Figure 18. Subroutine INPAR is of vital importance to the M, G and S key functions, in other words, those requiring certain parameters to be input.

25 The M key routine continued – back to figure 17a. After the end of the INPAR subroutine (point 24) the state of the N flag determines what happens next. If non-valid data was entered during the INPAR subroutine, the program moves on to label MATD and from there to the LABJUN label (figure 1b) where the computer reports 'JUNIOR'. Following this, the central label in the PM main routine, RESALL, is reached. We can now try again by depressing the M key. If, however, correct data was entered, the program jumps to label MATF. The next sequence of instructions checks to see whether anything has gone wrong. The 16 bit number contained in locations PARBH and PARBL is subtracted from the 16 bit number contained in location PARAH and PARAL. If the result is negative, the carry flag will be reset and the program will jump to label LABJUN via label MATD to report 'JUNIOR'. The reason for this is that the last address (PARB) should never be lower than the first address (PARA). This is true of both a hex dump listing and of a data transmission to cassette (key S; see point 28).

If the addresses are all right (carry flag = 1) the listing of the hex dump can be started. The instructions between labels MATG and MATH in figure 17a make sure that six spaces are 'printed' one after the other from the start of a new line. This is for the sake of clarity. After the spaces have been output, the numbers Ø . . . F are printed above the hex dump columns. Each figure is separated from the next by two further spaces. At the end of figure 17a the contents of the work address buffers POINTH and POINTL are made the same as those of the first address – the contents of locations PARAH and PARAL. That completes the first half of the M key routine.

26 Now to continue with the second half, which brings us to figure 17b. After the label MATJ, the data to be dumped is printed from the start of a new line (CRLF). Initially, the start address (the contents of POINTH and POINTL) is printed at the beginning of a new line. Once the address has been printed, a colon and a space are produced via the subroutine ME. This is similar to the MESSY subroutine, except for the initial carriage return and line feed (see figure 9a and point 11). In addition, the X index register (initial value = 1Ø) is used to keep a track of the amount of data to be printed in a particular row.

The instructions after label MATK check whether all the data in the hex dump has been printed. This can be deduced from the value in the address pointer (POINT) when compared to the contents of the last address buffer PARB. The state of the carry flag determines which path the computer is to take. If the POINT address has not yet exceeded the contents of buffer PARB, the relevant data is fetched from the current POINT location and printed (label MATL). Then a space is produced (subroutine PRSP), the value in POINT is incremented (subroutine INCPNT; see point 15 and figure 15a) and the contents of the X index register are decremented to prepare for the next print operation. The state of the Z flag (the instructions BNE and BEQ) determines whether the next lot of data is to be printed on the same line or on a fresh line. If the contents of the X register are not yet zero then the following data has to be printed on the same line. If the contents of the X index register are zero then the computer jumps back to label MATJ to print out the next address and then the next line of the hex dump.

Once all the data in the hex dump has been printed (the contents of PARB are less than those of POINT), the computer jumps to the CRLF subroutine and then to label LABJUN, to output the text 'JUNIOR'. Since subroutine CRLF is called just before the jump to label LABJUN and the text 'JUNIOR' is also printed at the start of a new line (see subroutine MESSY and JUNIOR) a gap of one line is left between the last line of the hex dump and the 'JUNIOR' message.

N.B. If the amount of data to be printed is a factor of sixteen, in other words, if the hex dump is made up only of complete lines, the last address to be printed will belong to a 'fictional' row of data. This is because the computer does not 'know' that it has reached the end of the hex dump until after label MATK. This means that although the address will be printed, the data contained therein will not be printed.

27 As far as the **G** key routine is concerned, we can leave figure 13b and turn to the third section of the main PM routine. This section is shown in figure 13c. The G key routine consists of all the instructions following the label GETTAP. Most of the work in this routine is accomplished by the subroutine **GETID** which is shown in figure 19a. Following the label GETID (figure 19a) the program jumps to subroutine **IPB.** This is not a new subroutine, but the closing stages of subroutine INPAR which was dealt with in point 24 and figure 18. When a program is being entered from cassette, data must be input (the program number, ID) the G key must be depressed followed by the CR key. (The situation where ID = FF will be considered later.) At the end of the IPB (= INPAR) subroutine the state of the N flag will dictate whether the whole procedure went smoothly and correctly, or not. If something went wrong the N flag will be set and the BMI instruction after the IPB subroutine has ended causes the computer to branch to label GA and from there return to the main section of GETTAP.

If an acceptable ID was entered, the contents of the accumulator are stored in location ID (see chapter 16). Prior to the end of the IPB subroutine, the accumulator is loaded with the contents of the low order data buffer INL. By the way; subroutine IPB is not exited from (unless something went wrong) until the CR key has been operated. The detour by way of the subroutine IPBRES is required to clear the contents of buffers INH and INL. Alternatively, the computer could have jumped to the instruction JSR - RESIN immediately before the IPB label in figure 18.

If everything went fine after the second jump to the IPB subroutine, the contents of locations INH and INL are copied into locations SAH and SAL respectively (see chapter 16). The program then proceeds to label GB so that the actual read from cassette operation can take place. Subroutine **RDTAPE** will be discussed in detail in chapter 16. During the RDTAPE subroutine the input/output parameters are re-defined. This is why the subroutine RESTTY (figure 19b) is called immediately afterwards. Here, the I/O status is restored to that shown in figure 1a. The computer then reports with the text 'READY' and the Z flag is set and the N flag is reset before the end of the subroutine GETID.

Back to the G key routine in figure 13c. If the N flag was set during the GETID subroutine, the program continues to label LABJUN via the label GETERR to report 'JUNIOR'. If, on the other hand, the N flag is reset, the entire process will have been a success and the computer returns to the focal point RESALL in the main PM routine.

28 The S key routine consists of all the instructions following the label SAVID in figure 13c. Again, the lion's share of the work is carried out by the subroutine **SAID**, which is shown in figure 20. We shall describe this subroutine first. When the S key is depressed, the following parameters have to be met:
S key - first data - comma - second data - comma - third data - CR key
where:
the first data represents an ID (ØØ . . . FF)
the second data represents the start address, SA
the third data represents the end address, EA (= LA + 1; see chapter 11 in Book 3).
The SAID subroutine starts by checking to see whether a comma key was operated. If not, the first data is not yet completely entered. This involves the subroutine HEXNUM: valid hexadecimal data sets the N flag thereby making the program jump to label SIB. As soon as the comma key is depressed, the program jumps to label SIC. Here, the contents of buffer INL are stored in location ID and the computer checks to see whether the programmer has inadvertently entered the value ØØ or FF for the ID. In this case the program will make a quick error exit via the label SIA, where the N flag is set and the Z flag is reset.
The second and third data entries and the final CR entry are processed by calling the subroutine INPAR which is shown in figure 18 and described in point 20. Before this, however, the data buffers INH and INL are cleared via the subroutine RESIN.
As soon as the CR key is depressed, the computer can start with the actual storage of the data being received from the cassette. The start address buffers SAH and SAL are loaded with the contents of buffers PARAH and PARAL for the first address. Similarly, the end address buffers EAH and EAL are loaded with the contents of locations PARBH and PARBL for the last address. The computer then jumps to the cassette subroutine **DUMP** which will be discussed in detail in chapter 16. As with the RDTAPE subroutine (point 27) the input/output parameters are altered during the DUMP subroutine. Therefore, the subroutine RESTTY is called after the end of subroutine DUMP.
After the end of the program entry the computer reports 'READY' and the contents of the Y index register are cleared, thereby setting the Z flag and resetting the N flag.
Back to the S key routine in figure 13c. After the SAID subroutine has been executed, the state of the N flag indicates whether all the data arrived safely or not. If so, the computer returns to the main cross-roads in PM (label RESALL); if not, the computer reports the error 'JUNIOR' via labels GETERR and LABJUN.

19a

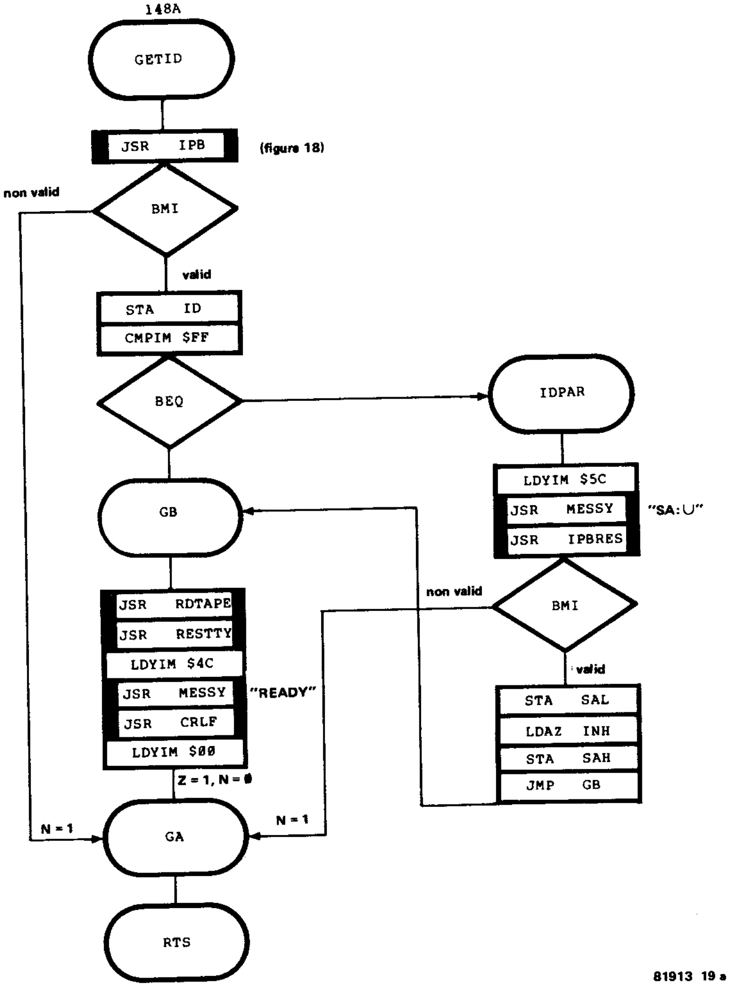
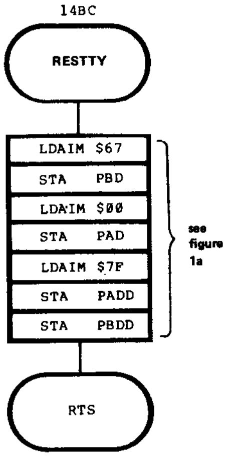


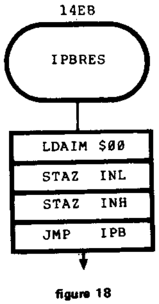
Figure 19. Subroutine GETID makes up most of the G key routine (figure 19a). Subroutine RESTTY (figure 19b) restores the input/output parameters to their original condition after one of the cassette subroutines RDTAPE or DUMP have been executed. Subroutine IPBRES (figure 19c) clears the contents of the data buffers INH and INL during part of the INPAR subroutine.

19b



81913 19b

19c



81913 19c

29 Finally, the **data** key routine. This involves all the instructions which follow the label VALNUM in figure 13c. The main routine will now have reached the stage where any keys that mean something to PM have been filtered out. These are:

+	(point 15)
–	(point 16)
SP	(point 17)
•	(point 18)
R	(point 19)
L	(point 21)
P	(point 22)
M	(points 23, 25 and 26)
G	(point 27)
S	(point 28)

Any keys not mentioned above will not have been filtered out. These are either non-hexadecimal keys – usually pressed by mistake – or hexadecimal keys. In the case of the latter, this concerns information pertaining to the entry of a work address or work data. Key information which is part of the routines relating to the key functions M, G and S is not dealt with by the main PM program loop.

The subroutine HEXNUM (figure 11) processes the hexadecimal data. This was discussed in point 13. Any non-valid keys produce the error message 'WHAT?'. In this case the program jumps back to the central label RESALL. When a valid key is depressed, the associated data nibble is processed inside the buffers INH and INL. Next, the program jumps to label READCH (see figure 13a). In this instance the data buffers INH and INL are not cleared as the data entry may well not be complete.

That covers all the software concerning the PM routine, but a word should be added about the BREAK key routine.

20

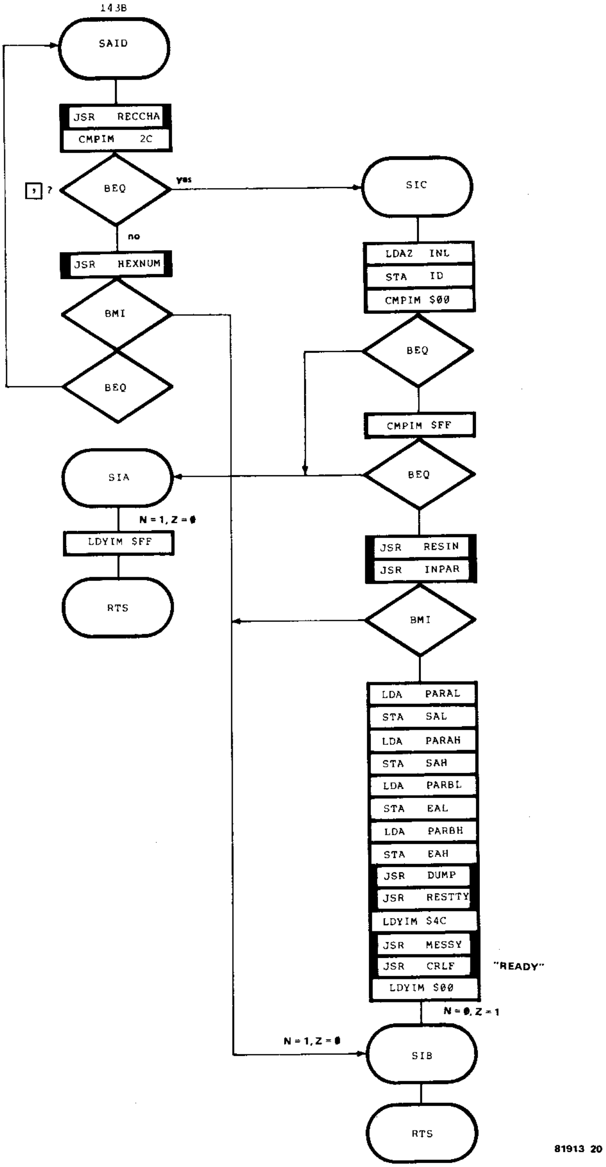


Figure 20. Subroutine SAID performs the lion's share of the S key routine. Among other things, it contains a jump to the cassette routine DUMP to allow a program to be stored on cassette.

30 As already mentioned (see chapter 12 in Book 3 and point 8) depressing the BREAK key will only be effective during a printing session involving the subroutine PRCHA. In practice, however, it will be seen that this key can lead to a 'JUNIOR' report, even while the computer is waiting for a key to be depressed (subroutine RECCHA). Thus, the behaviour of the computer will be identical to that during the LABJUN routine in figure 1b (see points 1 and 6).
How this is possible can be explained as follows:
If the BREAK key is depressed for at least nine bit periods in a row, the ASCII code ØØ is transmitted. Since this code is incomprehensible to the Junior Computer, it reports 'WHAT?'. At least that is what it is supposed to do in theory. In fact, as soon as the error message starts to be printed, the computer will find out that the BREAK key is still depressed and so will report 'JUNIOR' . . . but not before the BREAK key has been released!

15

The PME software

766 bytes of text editor

The new PM Editor was introduced in chapter 13. Now that we know how to 'drive' the new software, it may be interesting to take a good look 'under the bonnet' to find out how the 'engine' works – byte by byte. The program is based partly on existing subroutines and partly on totally new subroutines which, as always, can be incorporated into user programs.
Although the PME program contains relatively few bytes compared to the rest of the software in this book, each routine will be dealt with and fully explained in turn.

1 Let us start by looking at PME **in general**. As an aid to this, a block diagram has been drawn and is illustrated in figure 1.
PME is structured as follows:
The main routine is preceded by an initialisation routine which is only run now and again. As can be seen from the flowchart, PME originates from three sources, much in the same way as a river is formed by the convergence of a number of brooks and streams. The three sources are: the cold, warm CEND and lukewarm start entry routines. A fourth 'tributary', the ASSEND routine, should really be considered as an intermediate routine which is instigated by the depression of the 'ST' key on the main keyboard.
Normally, any use of PME will lead to its main routine, which is everything after the WARM label in figure 1. The procedure carried out by this main routine should be very familiar by now. Once a depressed key has been detected, the program carries out an interrogation procedure to find out which key was depressed: was it the 'A' key? If so, carry out the 'A' key routine. If not, was it the 'B' key? If so, execute the 'B' key routine. If not, was it the 'C' key? etc. After executing the corresponding key routine, the program returns to the central label WARM of the main PME routine, where the computer waits for a new key to be depressed and then the subsequent key operation is carried out, and so on.
Sometimes the computer will return to the WARM label via a 'half-way'

1

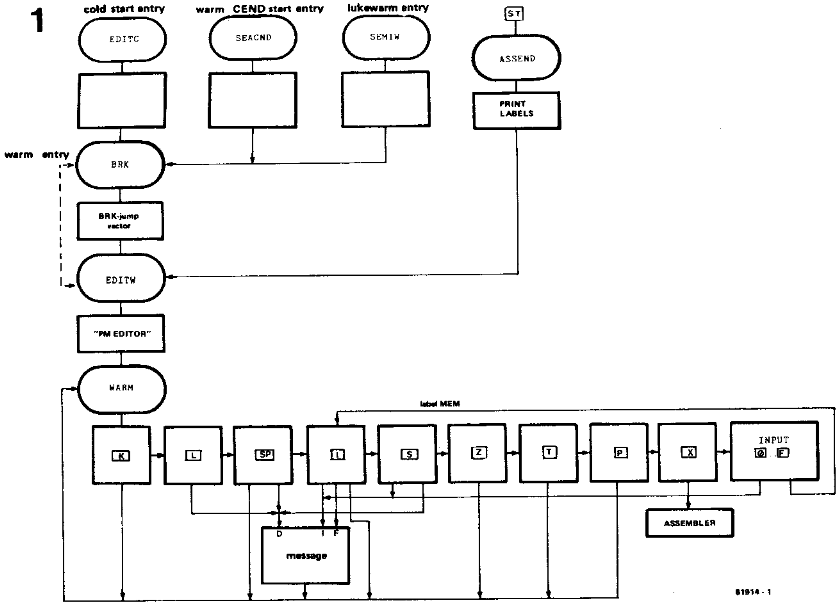


Figure 1. The block diagram of the PME system program.

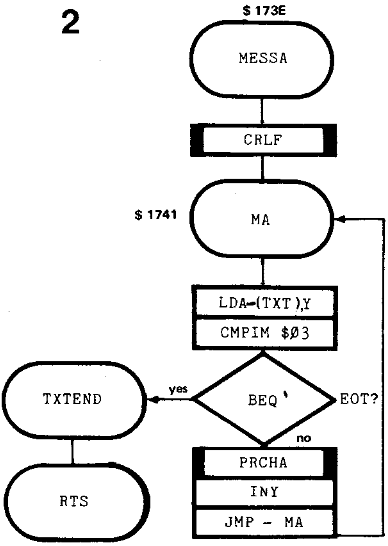
stage, after a 'DONE', 'ILLEGAL KEY' or 'FULL' report. In the case of the 'X' key routine, however, the computer will not return to the WARM label, but will jump to the assembler. The WARM label is reached by means of a detour (label ST, label ASSEND, label EDITW). The latter part of the '1' key routine (from label MEM) constitutes the final section of the INPUT routine.

Now let us take a much closer look at the subroutines involved in the PME program.

2 **The subroutine MESSA** is shown in figure 2. This is used by PME to print a text. The subroutine is virtually identical to MESSY, which was described in chapter 14. The only difference is that the first address of the look-up table is TXT, rather than MESS. The following texts are stored in the look-up table: 'BEGAD, ENDAD:', 'ILLEGAL KEY', 'FULL', 'DONE', 'PM EDITOR', 'LAB $' and ': $'. The particular text to be printed depends on the value contained in the Y index register just before the jump to subroutine MESSA.

3 **The subroutine PRINS** is shown in figure 3 and has the task of printing an address, the instruction contained in that address and a number of spaces. The number of spaces is determined by the length of the printed instruction – for a single byte instruction, 12 spaces are printed; for a double byte instruction, 9 spaces are printed and for a three

2



```
TXT = $ 175Ø
Y = ØØ ← 42  45  47  41  44  2C  45  4E  44  41  44  3A  2Ø  Ø3
          B   E   G   A   D   ,   E   N   D   A   D   :   ⊔  EOT
Y = ØE ← 49  4C  4C  45  47  41  4C  2Ø  4B  45  59  Ø3
          I   L   L   E   G   A   L   ⊔   K   E   Y  EOT
Y = 1A ← 46  55  4C  4C  Ø3
          F   U   L   L  EOT
Y = 1F ← 44  4F  4E  45  Ø3
          D   O   N   E  EOT
Y = 24 ← 5Ø  4D  2Ø  45  44  49  54  4F  52  Ø3
          P   M   ⊔   E   D   I   T   O   R  EOT
Y = 2E ← 4C  41  42  2Ø  24  Ø3
          L   A   B   ⊔   $  EOT
Y = 34 ← 3A  2Ø  24  Ø3
          :   ⊔   $  EOT
```

81914-2

Figure 2. The PME subroutine MESSA is virtually identical to the PM subroutine MESSY, except that a different look-up table is required for the various PME reports.

byte instruction, 6 spaces are printed.

Firstly, which instruction is printed? After all, there are quite a few! An instruction is printed when its opcode is contained in the address pointed to by the current address pointer CURAD. It is wise to remember that CURAD is always pointing to the first address of the last instruction to be printed in the left-hand column of the video screen (or the printer paper).

The PRINS routine starts by calling the CRLF subroutine. This was discussed in point 12 of chapter 14. This subroutine starts the printing operation at the beginning of a new line, which is of course the beginning of a line in the left-hand column. Following this another subroutine is called: OPLEN. This routine was mentioned frequently during the discussion of the original editor routines. It is described on page 165 in Book 2 (chapter 8). The opcode of an instruction is stored in the accumulator and the computer checks to see whether the instruction is one, two or three bytes in length. The length of the instruction is stored in location BYTES. Next, the X index register is loaded with the contents of location BYTES. Therefore, the computer knows how many bytes are to be printed. However, prior to the instruction itself being printed, the computer first outputs the address of the instruction opcode, namely the contents of CURADH and CURADL. This is accomplished with the aid of the subroutine PRBYT (see point 9 in chapter 12). Then the value ØF is loaded into address location LABELS. This value determines the number of spaces to be printed after the instruction has been dealt with.

3

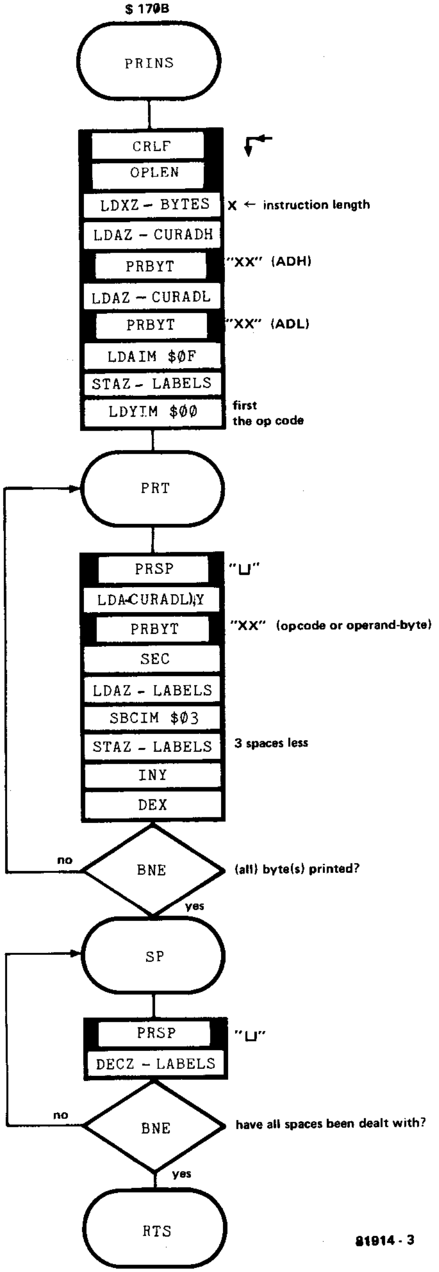


Figure 3. The PRINS subroutine prints the instruction indicated by the CURAD pointer along with the address of the opcode. A number of spaces are also printed depending on the length of the instruction.

This brings us to the whole section between the labels PRT and SP in figure 3. This section of the PRINS routine is run one, two or three times depending on the initial value contained in the X index register – the length of the instruction. When this section of program is reached, the initial value contained in the Y index register is zero. The routine following the PRT label starts by printing a space (PRSP, see point 12 in chapter 14). Then the contents of the address location pointed to by the CURAD pointer plus the value in the Y index register (ØØ) is printed. In other words, the opcode of the particular instruction. Again, the actual printing is accomplished by the PRBYT subroutine. The next few instructions cause the contents of location LABELS to be decremented by three. This effectively reduces the number of spaces to be printed later by a factor of three so that the total output, when the right-hand column is also printed, is nice and neat. The contents of the Y index register are then incremented by one and the value in the X index register is decremented by one. If the contents of the X index register are not yet zero, another byte has to be printed and the program loops back to the PRT label.

The final section of PRINS, everything after the label SP, is carried out once the contents of the X index register are zero – all the bytes in the instruction have been printed. This final section is concerned with printing a number of spaces after the instruction. The amount of spaces printed will depend on the final contents of location LABELS, that is:

15 (the original value in LABELS) minus three multiplied by the number of times the routine between the labels PRT and SP was run.

Therefore, 12 spaces are printed after a single byte instruction, 9 after a double byte instruction and 6 after a triple byte instruction. Once all the spaces have been printed, the program returns to the main routine.

4 **What happens after a cold start entry into PME?** The instructions contained in the initialisation routine given in figure 4a are executed. The instructions between the labels EDITC and BRK are only executed during a cold start entry into the PME, whereas those following the BRK label are executed in certain other instances as well. We shall return to these later.

Immediately following the EDITC label, the subroutine RESIN is called. We first encountered this subroutine in point 14 of chapter 14. It clears the data buffers INH and INL. Following this, the contents of the Y index register are made zero and subroutine MESSA is called, so that the computer reports 'BEGAD, ENDAD:'. Strictly speaking, this is not a statement, but a request for information. The operator is asked to indicate the BEGAD and ENDAD addresses. As you will remember, this involves entering the data for the start address, then a comma, then the data for the end address and finally depressing the carriage return (CR) key. The computer then disposes of another already present subroutine (INPAR) to deal with this information. The INPAR subroutine, which was previously used during the M key routine in the PM program (hex dump), was considered in detail in point 24 of chapter 14. If anything went wrong during the data entry, the N flag will be set and the Z flag will be reset at the end of INPAR. The following BMI instruction causes the program to branch back to the start of EDITC if wrong data was entered. This means that the

4a

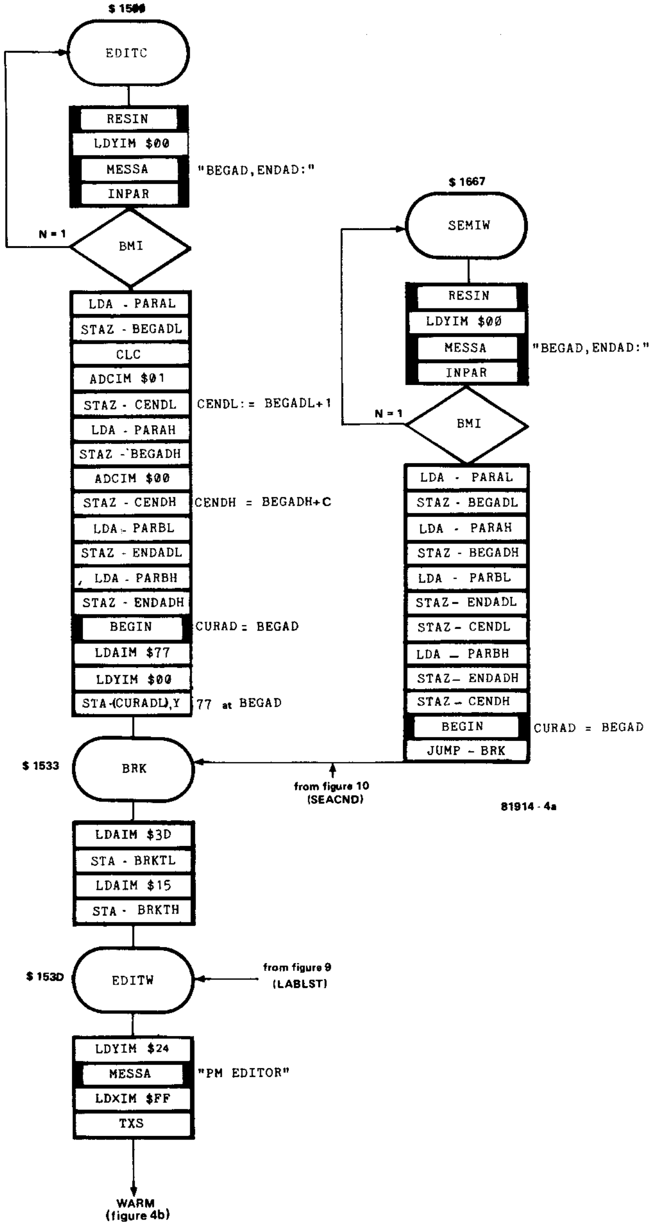


Figure 4a. The initialisation routines EDITC and SEMIW, including a common section following the BRK label.

whole procedure has to be repeated, so that 'BEGAD, ENDAD:' appears on the screen once again. In a number of cases the computer also reports 'WHAT?' This is because the subroutine HEXNUM is called at two places during the INPAR routine (see figure 18 of chapter 14). If a non-hexadecimal key is depressed, an error report will follow.

As far as the HEXNUM subroutine is concerned, you are referred to figure 11 and point 13 in chapter 14. Once the BEGAD and ENDAD addresses have been correctly defined, the contents of the first address buffers PARAH and PARAL are transferred to memory locations BEGADH and BEGADL, respectively. The contents of the second address buffers PARBH and PARBL are then transferred into locations ENDADH and ENDADL, respectively. At the same time, locations CENDH and CENDL are loaded with the contents of BEGADH and BEGADL + 1, so that the current end address pointer points to one location higher than the initial (BEGAD) address in the current memory range. This is necessary because the EOF character 77 is to be stored at BEGAD. As instructions and labels are added to the program, the EOF character and the CEND pointer will move up to a higher location and point to a higher address, respectively. If, on the other hand, instructions or labels are deleted, the EOF character and the CEND pointer move down the memory range.

Before the EOF character is stored at BEGAD, the current address pointer must also point to the start address. This is accomplished during the subroutine BEGIN, which was first introduced in chapter 8 of Book 2. After the PME initialisation routine in figure 4a has been run, the first instruction in the program will be printed. The opcode of the instruction will be held in location BEGAD, therefore, straight after a cold start entry, this will be the single byte 'instruction' with the pseudo-opcode 77.

5 **A warm start entry into PME.** The second part of the PME initialisation routine in figure 4a is sometimes run after a warm start entry into PME. A warm start entry begins at label BRK. If this method of starting PME is chosen, the contents of locations BEGAD and ENDAD will have already been established (during the initialisation routines SEMIW, SEACND or the intermediate routine LABLST – see later on for details). In any case, the BRK jump vector will have to be defined immediately after a warm start entry. For prior to the start, the computer was in the PM mode where the BRK vector points to the label LABJUN causing the text 'JUNIOR' to be printed (see chapter 14).

It is not surprising, therefore, that immediately after the BRK label in figure 4a, the BRK jump vector is loaded with the address $153D. This address pertains to the label EDITW, where we have just arrived. This label is followed by a few instructions which cause the text 'PM EDITOR' to be printed and the stack pointer to be reset to FF. The latter is useful after pressing the BREAK key, as it is possible that not only does the printing process have to be stopped, but the stack may be 'full'. (The so-called 'overflow' state is not possible where the stack is concerned. As soon as the highest location (Ø1ØØ) is reached, the process automatically continues from the next lowest location in the stack, which in this case is Ø1FF.)

N.B. A warm start entry could also take place at address $153D (label EDITW) instead of at address $1533 (label BRK). This means that the

BRK jump vector will be defined according to the PM program and will remain so unless altered by the user. Furthermore, the BRK jump vector will be determined by PM during the execution of the instructions up to the BRK label. This includes the various initialisation routines. Therefore, if the BREAK key is depressed while the computer is in the process of printing out 'BEGAD, ENDAD:' or while the INPAR subroutine is being run, the message 'JUNIOR' will appear on the screen as opposed to the expected 'PM EDITOR'. The latter will only be printed if the section of program following the BRK label has been run previously.

6 **The main PME routine.** It is now time to discuss the main PME routine, which is shown in figure 4b, starting with the **'K' key routine.** This involves the instructions immediately following the central WARM label, at the top left-hand corner of figure 4b. First an instruction is printed. Which instruction is determined by the contents of the current address pointer CURAD. The printing is accomplished with the aid of the PRINS subroutine which was described earlier (see figure 3). This is followed by the subroutine RECCHA, during which the computer waits for a key to be depressed. As soon as this occurs, the computer stores the ASCII code of the depressed key in the accumulator. The RECCHA subroutine was described in point 5 and figure 5 of chapter 14.

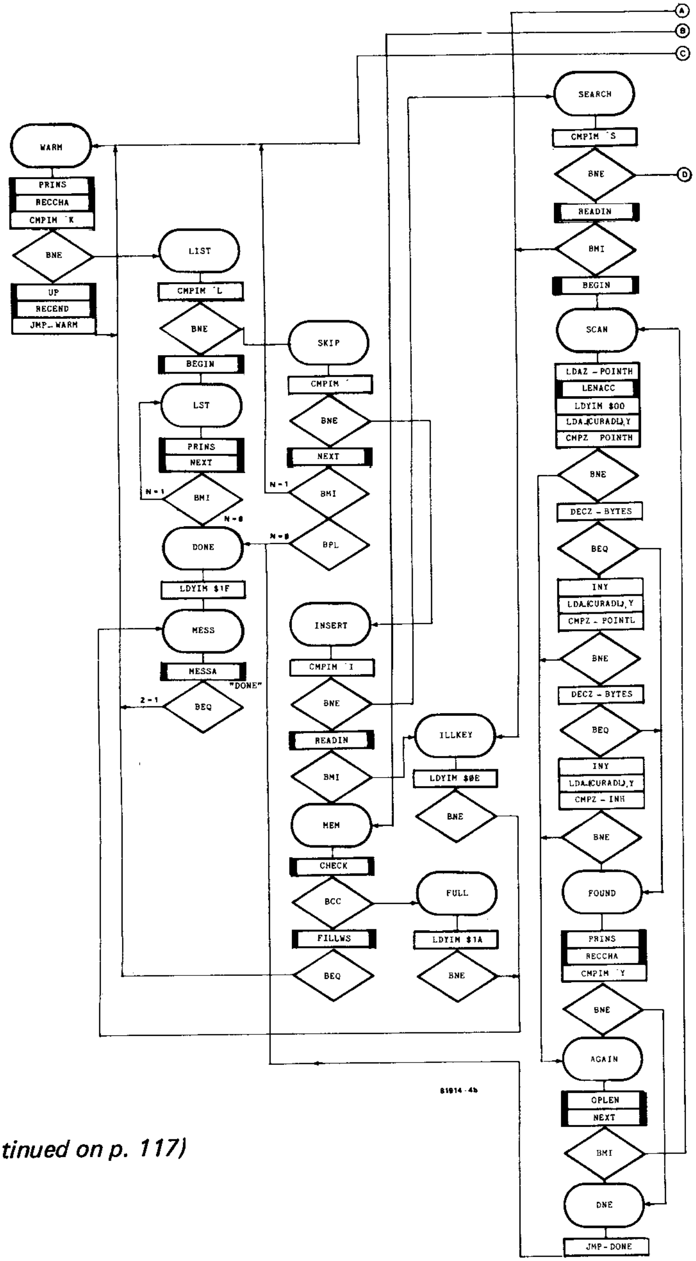
When the depressed key is a 'K', the UP routine is called. This is an original editor routine and causes the current instruction (the one pointed to by the current address pointer) is deleted from memory and that subsequent instructions and labels are shifted down the address range by the number of bytes corresponding to the length of the deleted instruction. The program or user file will now be one instruction shorter. After the UP routine, another original editor subroutine, RECEND is called. This decrements the contents of the current end address pointer by the value held in location BYTES. This is because location BYTES contains the length of the instruction just deleted.

Location BYTES is also used during the UP routine. Subroutines OPLEN and LENACC do not have to be called, as the deleted instruction is the last to have been printed and the value held in BYTES corresponds to its length. The UP subroutine was described in figures 17 and 18 of chapter 8 in Book 2, while RECEND can be found in figure 14 in chapter 8 of Book 2.

At the end of the 'K' routine, the computer jumps back to the central label WARM belonging to the PME main routine. Every requirement has now been met; the last instruction has been deleted from memory. Since the contents of the CURAD pointer remain unchanged, the instruction immediately following the one just erased is printed after the return to the WARM label.

7 **The 'L' key routine.** This routine consists of all the instructions starting at the LIST label in figure 4b and ending with the computer report 'DONE'. This key routine sees to it that all the instructions in the current program are printed out. The printing procedure must stop as soon as the current end address pointer CEND and the current address pointer CURAD contain the same value.

4b



(continued on p. 117)

Figure 4b. The first section of the main PME routine consists of the key routines for the 'K', 'L', 'SP' (space bar), 'I' and 'S' keys.

When the computer detects that the 'L' key has been depressed, the BEGIN subroutine is called. Firstly, the initial instruction, indicated by the contents of BEGAD, is printed. The remainder of the 'L' key routine is taken up by the program loop around the label LST. This involves subroutines PRINS and NEXT and the BMI instruction. The NEXT subroutine is yet another one which is 'borrowed' from the original editor and is also described in chapter 8 of Book 2. Readers are invited to refresh their memories as far as this subroutine is concerned, it is described on page 151 and in figure 10 of chapter 8. The contents of the current address pointer CURAD are incremented by the value related to the last figure held in location BYTES, that is the length of the last instruction to be printed during the PRINS routine. The final section of the NEXT subroutine checks to see that the contents of the current address pointer do not exceed those of the current end address pointer, or rather, that the contents of CURAD are smaller than or equal to the contents of CEND.

During this final section, the contents of CEND are subtracted from the contents of CURAD. Thus if:

CEND > CURAD N = 1 and C = Ø
CEND ⩽ CURAD N = Ø and C = 1

The status of the N flag is valid provided the result of the subtraction is not less than −127, in other words, provided the most significant bit in the result is.one.

Once the EOF character has been printed (this is stored at a location equal to one less than the address pointed to by CEND) and the NEXT subroutine has been run again, the contents of CURAD are made equal to those of CEND (the instruction length of the pseudo-opcode 77 is one). As a result, the N flag will be reset and so the program loop around the LST label will be interrupted.

The instruction concerning the printing of the 'DONE' message is next on the list. During the MESSA subroutine the Z flag is set (because the accumulator contains the EOF character), therefore the computer returns to the central WARM label in the main PME routine. During the subsequent PRINS routine, the opcode of the instruction stored at the location indicated by the CEND pointer is printed. This is unlikely to be an instruction entered by the operator, but more likely a 'random' instruction which was 'entered' when the computer was switched on. In any case, the last instruction to be printed is clearly not part of the current program as it is held in an address indicated by the CEND pointer.

8 **The 'SPACE' key routine.** The 'SPACE' key routine consists of all the instructions following the SKIP label in figure 4b. Depressing the SPACE bar causes a print-out of the instruction stored in memory immediately after the last one to be printed. In other words, the routine is similar to the 'L' key routine, but one instruction is printed at a time. The instruction to be printed is determined by the contents of the current address pointer CURAD. It is not surprising that this routine involves the NEXT subroutine. This is followed by the two instructions BMI and BPL, which both react to the status of the N flag. The computer will, therefore, either branch back to the central label WARM, or branch to the DONE

label. The latter will only occur when the contents of CURAD exceed those of CEND, as in the case of the LIST routine.

9 **The READIN/READ subroutine.** Before examining the 'I' key routine, let us take a BREAK and look at three other PME subroutines. First of all, the READIN/READ subroutine is shown in figure 5. This has a similar function to the RDINST subroutine contained in the original editor program (see page 153 and figure 11 in Book 2). The purpose of the READIN/READ subroutine is to process any instruction that the operator enters into the computer memory banks. It is used during the INSERT, SEARCH and INPUT key routines.

Just as the GETBYT subroutine was called once, twice or three times, depending on the length of the instruction, during RDINST, the BYTIN/BYT subroutine is called once, twice or three times during the READIN/READ subroutine.

The BYTIN/BYT subroutine is shown in figure 6. As the name of the latter suggests, two hexadecimal key operations are processed into a high order nibble and a low order nibble, which are then joined together to form a data byte. This is then stored in the accumulator. The BYTIN/BYT routine starts by calling the RECCHA subroutine, where the computer waits for a key to be depressed. During the subroutine ASHETT following the BYT label, the ASCII code of the depressed key is converted into a corresponding data nibble. If it was not a hexadecimal key the N flag is reset and the subroutine is exited from. The ASHETT subroutine was described in figure 12 and point 13 of chapter 14.

If the N flag is set after the ASHETT routine, the data nibble will be held in the low order position in the accumulator. Four successive shift operations in the accumulator cause the nibble to be moved to the high order position. This is then temporarily stored in location NIBBLE. Then the computer jumps to the subroutines RECCHA and ASHETT once again to wait for and subsequently process the next key to be depressed. Provided that the second depressed key is also hexadecimal, the second data nibble is ORed with the first to provide the complete data byte. The computer will then return to the READIN/READ subroutine.

If, for any reason, a non-hexadecimal key was depressed during BYTIN/BYT the subroutine in exited from with the N flag set and the Z flag reset.

N.B. The BYT subroutine is very similar to the BYTIN subroutine, except that the computer does not wait for the first hexadecimal key to be depressed. The BYT subroutine is called during the INPUT key routine.

Now back to the READIN/READ subroutine in figure 5, which starts by calling the BYTIN/BYT subroutine. The first two data nibbles, thus the opcode of the entered instruction, are held in the accumulator. At least, this is the case if hexadecimal keys were depressed. If not, the computer will proceed directly to the RTS label.

Following the label READ, the opcode of the instruction is stored in location POINTH. Subroutine LENACC then sets to work to find out the length of the instruction, and therefore how often, if at all, to keep calling the BYTIN/BYT subroutine (the contents of the Y index register after the end of LENACC). The length of the instruction is transferred to memory

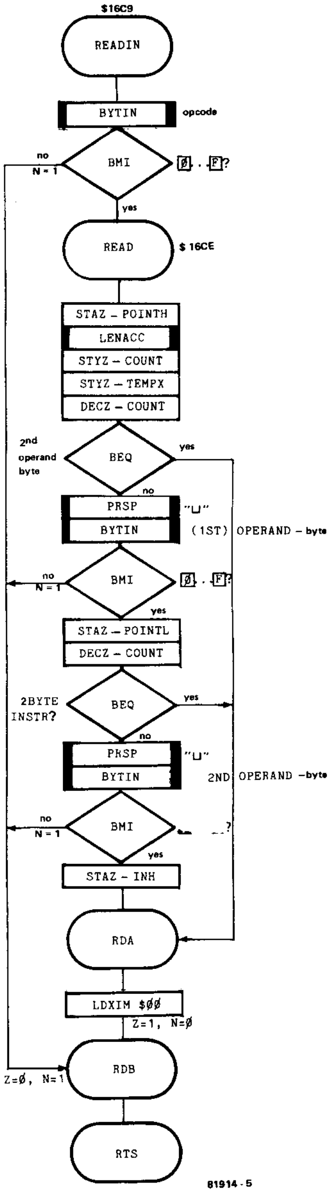


Figure 5. The subroutine READIN/READ examines an instruction and is used during the 'I', 'S' and INPUT key routines.

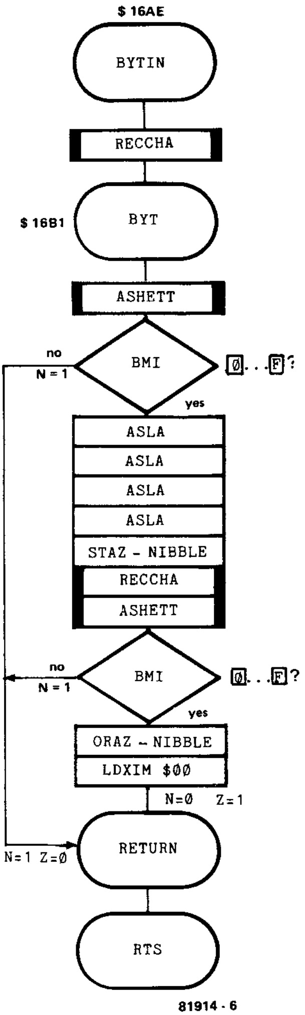


Figure 6. The BYTIN/BYT subroutine combines the data nibbles of two depressed keys. This subroutine is called at least once during the READIN/READ routine.

locations TEMPX and COUNT. If the instruction turns out to be two or three bytes long, a space (PRSP) is printed and the computer waits for the entry of the first (or only) operand byte (BYTIN). This byte is then transferred to the data buffer POINTL. If the instruction is three bytes long this procedure is repeated and the second operand byte is stored in the data buffer INH. Last but not least, the contents of the X index register are made zero: As a result, the Z flag will be set and the N flag will be reset. If a non-hexadecimal key was detected in any of the above procedures, the routine is exited from with the N flag set and the Z flag reset. Whatever happens, the subroutine always ends with the RTS instruction.

N.B. Apart from waiting for the first data byte (the opcode) to be entered (the first call to subroutine BYTIN), the READ subroutine is identical to READIN. The READ subroutine is called during the INPUT key routine.

10 **The CHECK subroutine.** The CHECK subroutine performs the task of ascertaining whether or not there is sufficient memory space available for a new instruction to be INPUT or INSERTED. As you will remember from chapter 13, four memory locations are required at the end of a user program. Three locations for the first label to be dealt with during assembly and one for the EOF character. This means that the lowest possible position for the CEND pointer to indicate is two locations below the address indicated by the ENDAD pointer. If the contents of CEND drop below this, due to the length of the new instruction, the instruction will not be accepted for storage.

At the end of the CHECK routine, the status of the carry flag will indicate whether or not there is sufficient memory space available. The CHECK subroutine is shown in figure 7 and starts by calling the subroutine ADCEND, which is contained in the original editor routine. It was described on page 155 of Book 2 and in figure 13 of chapter 8. The current end address pointer CEND is incremented by an amount corresponding to the length of the instruction. We may assume that, just prior to the jump to the CHECK routine, the contents of location BYTES corresponds to the length of the instruction that needs to be stored. Subsequently, the carry flag is assigned a value according to the result of the subtraction:

the contents of ENDAD minus $Ø2 minus the contents of CEND.

Depending on the result, the computer will either branch directly to the end of the subroutine or continue. If the carry flag is reset, a borrow operation must have been necessary during the subtraction. This means that the contents of ENDAD are less than the sum of the contents of CEND and $Ø2. This concludes that there is no room for the instruction.

If, on the other hand, the subtraction leads to a zero or positive result, the contents of CEND will still be above the danger level and the instruction can be stored. Therefore, before the return instruction, the subroutine RECEND is executed. This is the opposite of ADCEND and was described on page 159 and in figure 13 of chapter 8 in Book 2. The contents of the current end address pointer CEND are temporarily restored to their previous value, since the instruction is 'registered' straight after the CHECK routine (see the discussion on the 'I' key routine). Firstly, however, sufficient room has to be reserved in the memory banks, which is why the CEND pointer has to be restored to its previous value for a moment.

N.B. Note that if the CHECK routine is left when the carry flag is reset, the CEND pointer will not be adjusted by means of the RECEND routine. As a result, the EOF character 77 will stay where it is (no instruction is added as there is no room for it) and at the same time the contents of CEND will be incremented. As soon as there appears to be no more room

7

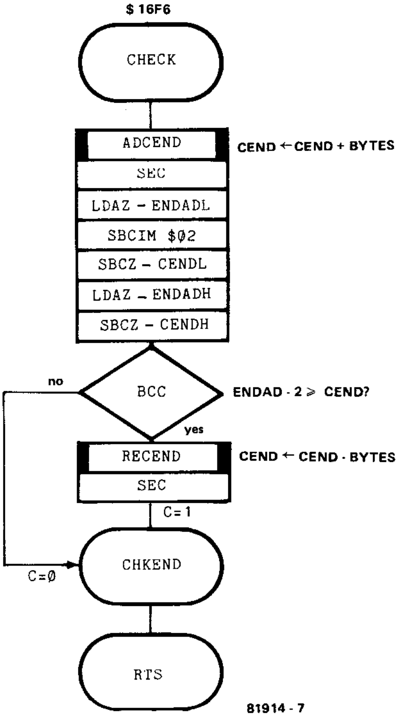


Figure 7. The CHECK subroutine is used to determine whether or not there is sufficient room in the memory area specified by the values contained in BEGAD and ENDAD for any extra instructions.

for further instructions, the CEND pointer will no longer indicate an address which is one location higher than that where the EOF character is situated. This situation can be remedied and will be explained later.

11 **The 'I' key routine.** Now that we are familiar with the subroutines involved, we can go on to describe the 'I' key routine. This consists of all the instructions following the INSERT label in figure 4b. As soon as the computer detects that the 'I' key has been depressed, the READIN subroutine is called. This subroutine is exited from in two ways:
when N = Ø and Z = 1 where:
the opcode of the instruction being inserted is held in the data buffer POINTH;
the first, or only, operand byte of the instruction to be inserted is held in data buffer POINTL;
the second operand byte, if any, is held in data buffer INH.
when N = 1 and Z = Ø where:
a non-hexadecimal key must have been depressed during the entry of the instruction.
In the latter case, the program will branch to the label ILLKEY and the

computer will report 'ILLEGAL KEY', followed by a return to the central label WARM.

If the instruction is entered in the correct manner, the sequence of instructions following the MEM label will be processed. The subroutine CHECK is called to see whether or not there is sufficient room for the instruction being inserted. If not, the computer branches to label FULL and the computer reports the corresponding message. If there is sufficient room, however, the FILLWS routine is called and the instruction is inserted into the sequence already held in memory.

The FILLWS subroutine is a well known original editor routine and is described in figure 12 of chapter 8 in Book 2. It ensures that sufficient space is reserved for the new instruction (the DOWN subroutine). After the current end address pointer CEND has been adjusted, one, two or three memory locations are loaded with the contents of POINTH (opcode), POINTL (first operand) and INH (second operand), depending on the length of the instruction. Since the Z flag will be set at the end of the FILLWS routine, the computer will once again return to the central WARM label. The contents of the CURAD pointer remain unchanged, so the instruction that has just been inserted is printed.

N.B. The section of the 'I' key routine following the MEM label also constitutes the second part of the INPUT key routine.

12 **The 'S' key routine.** There are quite a number of instructions concerned with the SEARCH routine, as can be seen from the right-hand column in figure 4b. When the computer detects that the 'S' key has been depressed, the subroutine READIN is called. In other words, the instruction to be searched for is specified in location POINTH (opcode) and, depending on the length of the instruction, in locations POINTL and INH as well. If a non-valid key is depressed during the instruction entry, the processor branches to the ILLKEY label and the message 'ILLEGAL KEY' is printed.

Provided the instruction was entered correctly, the search procedure will take place, starting at the label SCAN. First, the opcode of the instruction concerned is loaded into the accumulator and the length of the instruction is deduced via the subroutine LENACC. The length of the instruction is stored in location BYTES. By the way, just before the SCAN label the contents of the current address pointer are made the same as those of BEGAD by means of the BEGIN subroutine, so that the search operation starts at the first instruction in the program. The opcode of that instruction is loaded into the accumulator and is then compared with the opcode of the instruction being searched for, in other words, the contents of POINTH. If the two opcodes are different, the instruction can be ignored whereupon the computer will branch to the AGAIN label to examine the next instruction in the program.

By decrementing the contents of BYTES by one and testing the result with the aid of a BEQ instruction, the computer can determine whether or not the instruction being searched for (not the one being examined!) is a single byte instruction. Note that this only takes place if the two opcodes are the same. If the searched for instruction is in fact one byte long, the program will branch to the label FOUND, where the instruction is printed along with the address where it was found. If not, the next byte is fetched from memory. This does not necessarily have to be the first (or only) operand of the instruction being examined, because the previously found opcode could well have been data of a different kind.

What does happen is that the next memory byte is compared to the first operand of the instruction being searched for. Again, one of two things may occur. If the two bytes are different the program will branch to the label AGAIN. If they are the same, the computer checks to see whether the instruction being searched for is two bytes long. If so, the program branches to the FOUND label, if not, the next byte is retrieved from memory and compared to the second operand byte of the instruction being searched for. If these two bytes are also identical, the program branches to the FOUND label.

The section of the 'S' key routine following the SCAN label could well have been structured differently. After ascertaining that the two opcodes were the same, the two instruction lengths could have been compared and the program would branch to AGAIN if they were different. The problem here is that although this method would save a certain amount of time, it would require quite a lot of memory space.

Following the label FOUND, the PRINS subroutine is called: the instruction being searched for has now been found and is printed along with its corresponding address. The flowchart shows quite clearly that the search operation starts at the beginning of the program: see the BEGIN subroutine just prior to the SCAN label.

In chapter 13 we described how the same instruction can be searched for further on in memory. For this the 'Y' key had to be depressed. As can be seen, the subroutine RECCHA follows the PRINS routine. Therefore, the computer waits for a key to be depressed. If the 'Y' key is depressed at this stage, the computer will branch to the label AGAIN to continue the search. If any other key is depressed, the program will branch to the DONE label, where the message 'DONE' is printed to complete the 'S' key routine.

Following the label AGAIN, the subroutine OPLEN is called, which determines the length of the last instruction to be examined. Then the NEXT subroutine is called to adjust the contents of the CURAD pointer so that it points to the opcode of the next instruction in the program being examined. The status of the N flag at the end of the NEXT subroutine informs the computer whether or not there is another instruction to follow. This matter was discussed in detail during the explanation of the 'L' key routine. If there are no more instructions in the program, the processor proceeds to the DNE label to print the message 'DONE' and from there it will move back to the central WARM label. As can be seen from figure 4b (page 108), the only exit from the 'S' key routine is via the DNE label, as illustrated in chapter 13. Therefore, every search routine has to end with the message 'DONE'.

13 **The 'Z' key routine.** The 'Z' key routine consists of the relatively small number of instructions following the BACK label in figure 4c. When the computer establishes that the 'Z' key has been depressed, the contents of the address pointer TABLE are made the same as those of

4c *(continued from p. 108)*

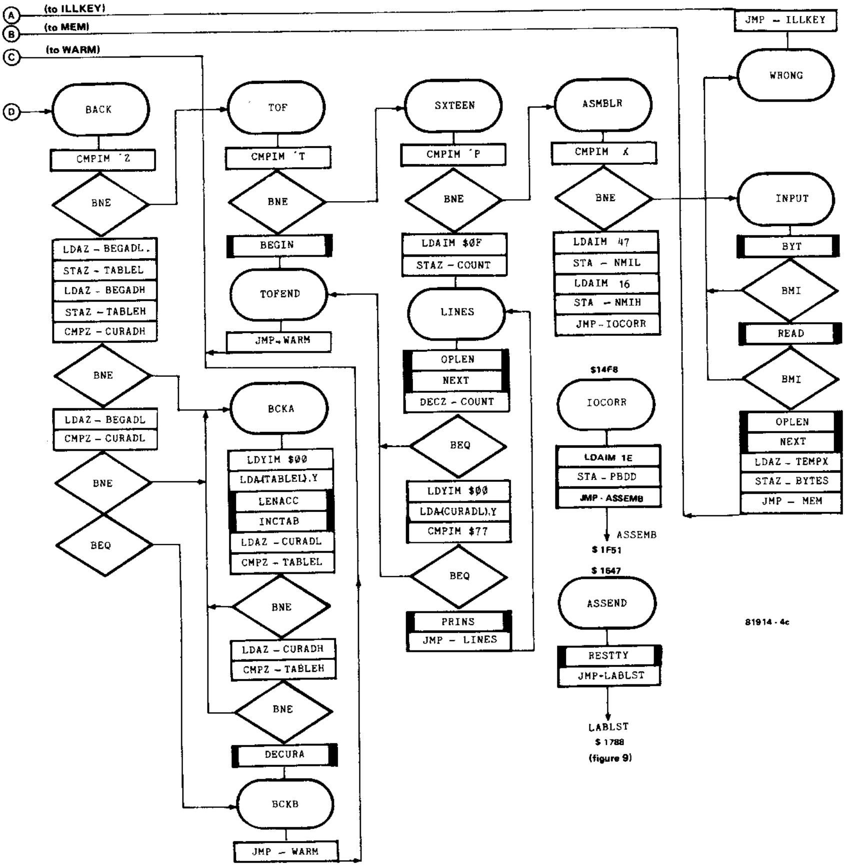


Figure 4c. The second section of the main PME routine consists of the key routines for the 'Z', 'T', 'P' and 'X' keys and also the INPUT routine (hexadecimal keys).

BEGAD. The TABLE pointer has only been used in the assembler mode until now (see chapter 9 in Book 2), but its services are required here as an auxiliary address pointer. When the length of an instruction is being determined, this location points to the address directly in front of that belonging to the last instruction to be printed. The length of the instruction must be known so that it can be subtracted from the contents of the CURAD pointer, and so enable the previous instruction to be printed after the jump to the WARM label. This corresponds to the 'decrement instruction' function which is executed by means of the 'Z' key.

As soon as the contents of TABLE and BEGAD are the same, the computer determines whether or not the contents of TABLE are also the same as those of CURAD. In other words, whether the first instruction in the program was the last to be printed. If this is the case, then there can be no 'previous' instruction and the program returns to the WARM label via the label BCKB. As a result, the first instruction in the program is printed again (since the contents of CURAD remain unaltered).

Things are totally different when the CURAD pointer does not point to the first address in the program (BEGAD). The program loop following the BCKA label is run a number of times depending on the length of the instruction concerned. This effectively increments the contents of TABLE by an amount equal to the instruction length until the contents of TABLE are the same as those of CURAD. Then the CURAD pointer has to be decremented by the value equal to the last contents of location BYTES. This is why the opcode of the instruction being examined is entered into the accumulator immediately after the label BCKA. The length of the instruction is determined by the subroutine LENACC. The contents of TABLE are then incremented by an amount equal to the instruction length. This is accomplished during the subroutine INCTAB, which is shown in figure 8b.

Once the contents of TABLE have been incremented, the computer checks to see whether they are already the same as those of CURAD. If not, the program returns to the label BCKA where the next round in the search commences. If, on the other hand, the contents of TABLE and CURAD are the same, the program jumps to the DECURA subroutine, which is illustrated in figure 8a. Here, the last contents of location BYTES are subtracted from the contents of the CURAD pointer. This will be the same value that was added to the contents of TABLE to make them equal to those of CURAD. All that remains is to jump back to the central label WARM. On the way, the computer passes through the section of program

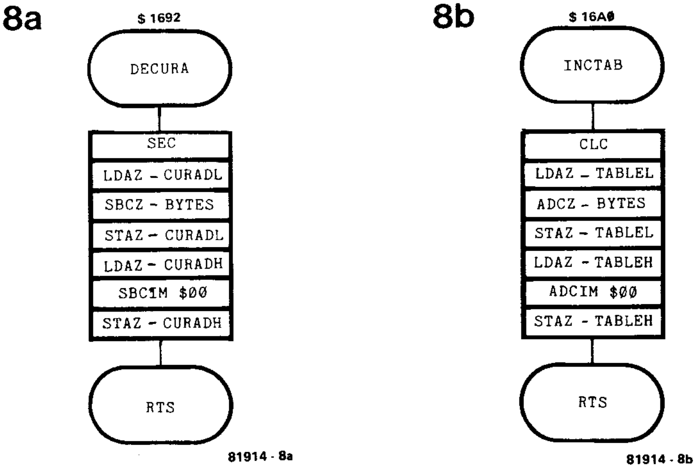


Figure 8. The subroutines DECURA (figure 8a) and INCTAB (figure 8b) are used during the 'Z' key routine (the instruction 'backspace' function).

following the BCKB label, so that the previous instruction is printed after the label WARM.

14 **The 'T' key routine.** The description of the 'T' key routine can be very brief. It consists of the instructions following the label TOF in figure 4c. When the computer detects that the 'T' key has been depressed, the BEGIN subroutine is called. This simply makes the contents of the CURAD pointer the same as those of BEGAD. The computer then returns to the WARM label via the label TOFEND, whereby the first instruction (or label) in the program is printed.

15 **The 'P' key routine.** The 'P' key routine consists of all the instructions following the SXTEEN label in figure 4c. When the computer detects that the 'P' key has been depressed, the value $ØF (decimal 15) is stored in memory location COUNT. This location acts as an instruction counter. This means that in addition to the instruction already printed on the display another 15 instructions will be listed. This gives a total output of sixteen instructions. In fact, the first instruction has already been printed before the 'P' key is depressed. Fourteen instructions are printed during the program loop starting at label LINES and the sixteenth instruction is printed when the computer returns to the central WARM label.

Following the label LINES, the subroutine OPLEN is called to establish the length of each instruction to be printed. The contents of the current address pointer CURAD are incremented accordingly by means of the NEXT subroutine. Then the value contained in location COUNT is decremented by one. If this brings the value in COUNT to zero, the computer returns to the WARM label via the intermediate TOFEND label. As a result, the sixteenth instruction is printed. If the contents of COUNT are not yet zero, the computer checks to see whether the EOF character 77 has been reached. This would mean that the final instruction in the program had been reached, in which case this is the last 'instruction' to be printed after the return to the central label WARM via the label TOFEND.

If the EOF character has not yet been reached, the computer calls the PRINS routine to print the instruction and then returns to the label LINES. The first four instructions following this label are executed 15 times, unless the EOF character is encountered, because the contents of location COUNT have to be decremented 15 times before they will be zero. The remainder of the program following the label LINES is only executed 14 times (again, unless the EOF character is encountered). On the 15th run the computer will branch back to the WARM label at the first BEQ instruction.

16 **The 'X' key routine.** The 'X' key routine consists of the instructions following the ASMBLR label in figure 4c and also those instructions after the IOCORR label. This key routine is directly concerned with assembling the program. When the computer detects that the 'X' key has been depressed, it specifies the NMI jump vector as being the address where the ASSEND label is held ($1647). After the program has

been assembled and the ST/NMI key on the main keyboard has been depressed, the computer will return to PME by way of ASSEND.
When the NMI vector is defined the program jumps to the label IOCORR. Port B data direction register is then loaded with the same value as at the end of the RESET initialisation routine in the original monitor program, namely $1E (see chapter 7 in Book 2). The rest of the input/output parameters do not have to be modified. There is now no reason why the processor cannot jump to the assembler. The assembly procedure was fully described in chapter 9 of book 2.

17 **After assembly**. The program will now be almost ready to run. As soon as the program is completely assembled at the instigation of the 'X' key, the six displays will light. The operator must now depress the ST key. This enables a non-maskable interrupt which informs the computer to execute the instructions following the ASSEND label.
Firstly, the RESTTY subroutine is called. This subroutine is used by the PM program to restore the input/output status to its original condition once the cassette subroutines DUMP or RDTAPE have been dealt with. Again, this is all old hat and readers are referred to figure 19b and points 27 and 28 in chapter 14. Really, only port B data direction register needs to be restored, but that would take up more memory bytes than simply calling the RESTTY subroutine.
As we already know, all the labels appearing in the program that has just been assembled are printed out. This is accomplished during the label print routine LABLST, which is shown in figure 9. During this routine, the address pointer TABLE, which was 'borrowed' earlier for an auxiliary address counter, is used for its intended purpose, which is to act as a table end address pointer.
Right at the start of the first assembly phase, location TABLE is loaded with a value equal to the current value in the ENDAD pointer minus $FF. The initial value stored in location LABELS is $FF. The table pointer TABLE + LABELS establishes at which location a label number is stored – including its high and low order address bytes. Initially, the table pointer indicates the same address as ENDAD. When a label has been processed (found and stored), the contents of LABELS are decremented by $Ø3. That is quite enough about chapter 9. Readers who feel they should brush up on the assembly procedure should take another look at chapter 9 in Book 2.
The LABLST routine in figure 9 copies the intermediate values held in location LABELS into the Y index register. Thus, the routine starts by making the contents of the Y index register equal to $FF. The contents of TABLE are, of course, equal to those of ENDAD minus $FF. The instructions after the label LBLSTB serve to print the various components of the labels in the correct order, so that the contents of the Y index register are decremented by one whenever a label parameter is encountered.
However, before the program reaches the label LBLSTB, two instructions under the label LBLSTA are executed. The CRLF subroutine is called so that the printing process starts at the beginning of a new line and the X index register, which is used as a label counter is loaded with an initial

9

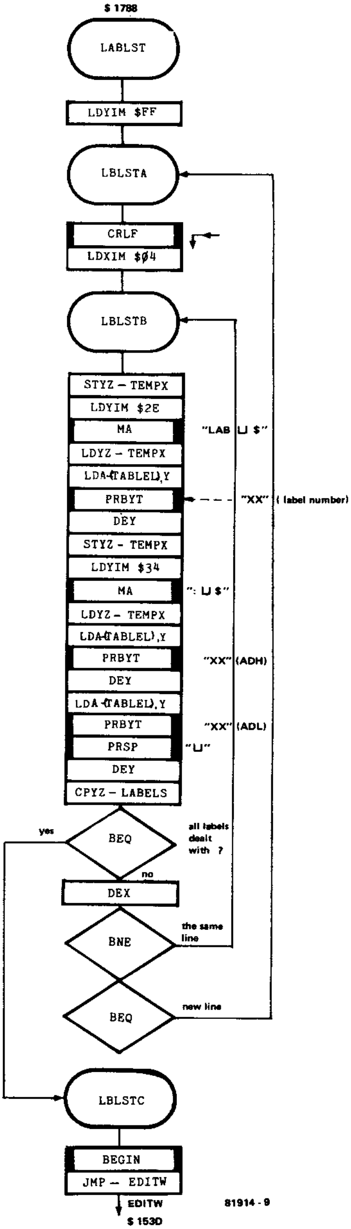


Figure 9. The 'print label' routine LBLST is executed after the program has been assembled and after the ST/NMI key on the main keyboard has been depressed. Up to four addresses and label numbers are printed on one line.

value of $Ø4: a maximum of four labels can be printed on any one line. Following the label LBLSTB the contents of the Y index register are stored in location TEMPX. This is necessary as the routine uses the Y index register for two separate tasks. For the computer to report 'LAB $', the Y index register is loaded with the value $2E. Then the MA subroutine (which is part of the subroutine MESSA) is called to print the message. After this, the number of the (first/next) label to be found and stored during the first phase of assembly is loaded into the accumulator. Thereupon, the label number is printed with the aid of the subroutine PRBYT. The computer then reports ': $' and the high and low order address values of the label are printed. Subsequently, a space is printed (PRSP) to separate the labels on one line from each other. The computer then checks to see whether or not there are any more labels to be printed. If there are, the contents of the X index register are decremented by one. The value in the X index register indicates the number of labels which can be printed on the same (current) line. Next, the processor returns to the label LBLSTA to continue the printing on a new line, or returns to LBLSTB to continue on the same line.

Once all the labels have been printed, the computer branches to the label LBLSTC. After the current address pointer has been loaded with the address of the first instruction in the assembled program (BEGIN), the processor jumps back to label EDITW (see figure 4a). Then the text 'PM EDITOR' is printed, followed by the first instruction in the program.

18 **The INPUT key routine.** The INPUT key routine consists of the instructions following the label INPUT in figure 4c. As mentioned in chapter 13, no non-hexadecimal key (function key) has to be depressed before this routine is carried out. This means that the computer will accept the instruction entry if at least two hexadecimal keys are depressed one after the other. The INPUT routine starts by calling the BYT subroutine, which is similar to BYTIN except for the first jump to the RECCHA subroutine. The latter is omitted as it comes into the picture after the central WARM label (see figure 6).

After the BYT subroutine, the opcode of the instruction being input is held in the accumulator. The status of the N flag informs the computer whether a non-hexadecimal key has been depressed, in which case the computer branches to the label ILLKEY via the label WRONG and will then report 'ILLEGAL KEY'.

If a hexadecimal key was depressed, the READ subroutine will be executed. This is almost identical to the READIN subroutine except for the fact that the first two instructions of the latter are missing (see figure 5). The READ subroutine starts by storing the opcode of the instruction just entered into address buffer POINTH. Next, the rest of the instruction is read. Again, the program will branch to ILLKEY if a wrong key is depressed.

Then it is the turn of the familiar OPLEN routine followed by the equally familiar NEXT routine. The contents of the CURAD pointer are incremented by a factor corresponding to the length of the input instruction. Location BYTES (as ever) contains the length of the new instruction once the value contained in TEMPX has been copied therein. As soon as this has happened, the program jumps to the MEM label. The remainder of the INPUT routine is exactly the same as the latter section of the 'I' key routine.

One further remark. The contents of the CURAD pointer will be incremented regardless of whether or not there is sufficient room for the extra instruction.

19 **The lukewarm start entry initialisation routine.** The lukewarm initialisation routine consists of all the instructions following the SEMIW label in figure 4a (page 105). This routine starts off in exactly the same way as the EDITC routine. In other words, the data buffers are cleared, the message 'BEGAD, ENDAD :' is printed and the operator defines the start and end addresses of the program (INPAR subroutine). If anything went wrong during the entry of the start and end addresses, the computer will return to the SEMIW label and start again. If everything was carried out correctly, the correct information is transferred to the address locations BEGADH, BEGADL, ENDADH, and ENDADL. Furthermore, the contents of the current end address pointer CEND are made the same as those of ENDAD and the contents of CURAD are made the same as those of BEGAD. The latter is accomplished with the aid of the BEGIN subroutine. Following the jump to the BRK label in figure 4a, the BRK jump vector is specified and the computer goes on to report 'PM EDITOR'. The program then proceeds to the central WARM label in the main PME routine.

The lukewarm start entry procedure is used to examine a 'finished' program, whether it be stored in RAM or in EPROM. The 'SP', 'S', 'L' and 'P' keys then have a very important function. When we described these key functions earlier in this chapter, we saw that the processes stop as soon as the address pointed to by the CEND pointer is reached. Well, one main advantage of the lukewarm start entry into PME is that the contents of CEND are made the same as those of ENDAD – chosen by the operator! Therefore, the computer user can specify exactly how much of the finished program he/she wants to examine. By the way, this type of entry was discussed in chapter 5 of Book 2 (original editor program). Only there, a whole series of data buffers had to be loaded 'by hand' to start with.

20 **The warm CEND start entry initialisation routine.** This is the final point to be discussed in this chapter. The warm CEND start entry into PME consists of the initialisation routine following the label SEACND in figure 10. Again, the program starts off in exactly the same manner as does EDITW and SEMIW, so there is no need to discuss this procedure further and we can move straight on to the SCNDA label. Once the contents of the current address pointer are made to indicate the first address in the program (BEGAD; by means of the BEGIN subroutine), the opcode of the instruction being examined is loaded into the accumulator and the computer tests to see whether it is the pseudo-opcode 77 (the EOF character). If so, the processor executes the sequence of instructions following the SCNDB label; if not, the contents of CURAD are incremented by the length of the last instruction and the next instruction indicated by the CURAD pointer is examined via the subroutines LENACC and NEXT.

As soon at the EOF character is detected the computer branches to the

10

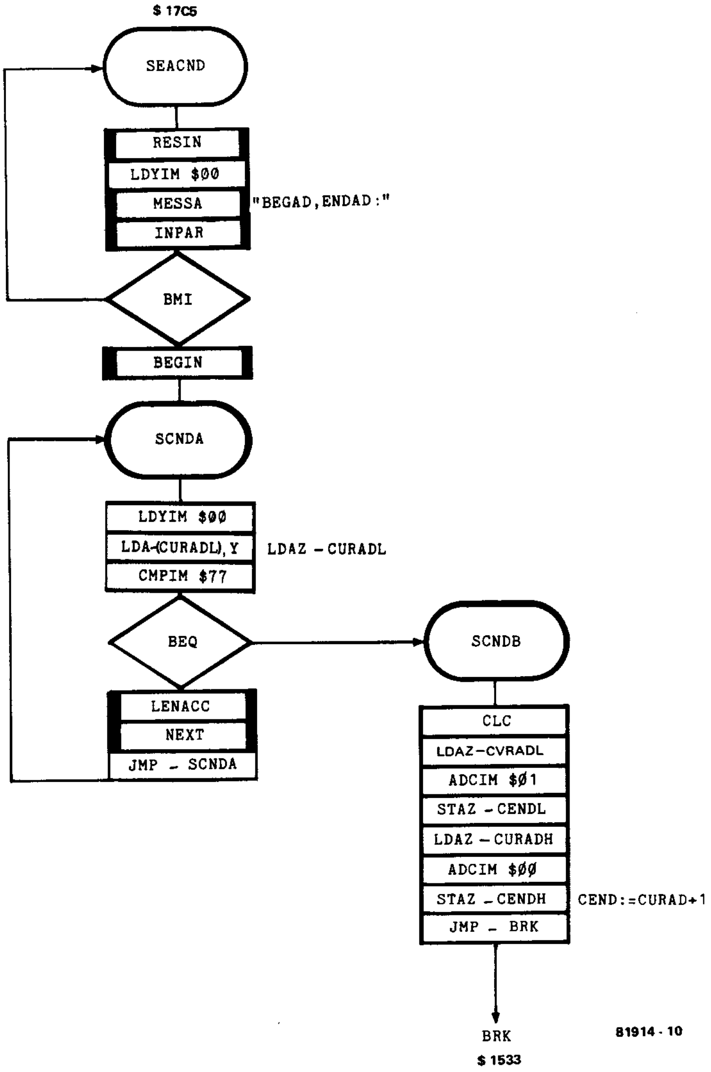


Figure 10. The initialisation routine SEACND is run when the PME program has been started via a warm CEND start entry. The contents of the current end address pointer CEND are restored to their original value, provided the program contains an EOF character 77.

label SCNDB where the contents of the CURAD pointer are incremented by one. This new value is then allocated to the current end address pointer CEND. The processor then moves on to the BRK label in figure 4a, the

BRK jump vector is defined and the message 'PM EDITOR' is printed. We are now back at the central WARM label in the main PME routine. The original contents of BEGAD and ENDAD are restored to the program read from cassette. The value contained in the ENDAD pointer does not necessarily have to be the same as the original, because at the beginning of the warm CEND start entry into PME the user can specify a higher end address if he/she wishes to provide plenty of space for an extra section of program. If this is not done, the computer will most likely report 'FULL' when extra instructions are entered.

After dealing with the instructions shown in figure 10, the final value contained in the current address pointer CURAD will be that location containing the EOF character 77. Not, in other words, the current end address. Figure 10 also shows what happens if the program does not contain an EOF character: the program loop starting at the label SCNDA is executed ad infinitum.

The warm CEND start entry is also applied when there is sufficient room for extra instructions in the memory area, but the computer has reported 'FULL'. Using the warm CEND start entry the contents of the ENDAD pointer can be altered while those of the BEGAD pointer remain the same. This means that the contents of the CEND pointer and the location of the EOF character will be correct for further instruction entries.

16

The cassette software

Data storage on magnetic tape

What is required in the way of software to enable the Junior Computer to store data on and retrieve data from ordinary magnetic tape? How is the Tape Monitor routine structured? These are just two of the many questions which will be answered during the course of this chapter. The two main sections of the cassette software are the DUMP routine which transfers information from the microcomputer to the cassette tape and the RDTAPE routine which does exactly the opposite. These subroutines can be called by either the Printer Monitor or the Tape Monitor routines and can also be included in user programs.

Firstly, this chapter will examine the DUMP routine and its associated subroutines (or rather, sub-subroutines!) in points 1 to 9. This will be followed by a detailed description of the RDTAPE routine and its sub-subroutines in points 10 to 19. This will bring us to the discussion of the main section of the Tape Monitor program and its subroutines in points 20 to 28. The chapter concludes with a brief discourse of the PLL test software in points 29 and 30. However, before you can retrieve data from cassette tape you must be able to store it . . .

1 **The DUMP routine.** The complete flowchart for the DUMP routine is shown in figures 1a (first half) and 1b (second half). By far the best idea seems to start at the very top of figure 1a! In other words, with the section of program between the labels DUMP and DUMPT. As can be seen, a total of four memory locations, HIGHER, LOWER, FIRST and SECOND, are first loaded with specific data. These locations play an important part in determining the bit time and the relative durations of the 2400 Hz and 3600 Hz tones – the time taken for a data byte or an

ASCII character code to be transferred to tape. This aspect will be considered in greater detail in points 3 and 4.

Following the DUMPT label in figure 1a the processor programs the input/output ports of the PIA. Readers may be forgiven for thinking that this should have occurred sooner. However, the four memory locations have to be defined first, because the subroutine is not only called during the PM and TM programs, but can also be included in user programs. Therefore, if the operator wishes to alter the bit time (the baud rate) he/she only has to modify the first group of instructions and then call the routine starting at the DUMPT label (not DUMP!). In point 9, for instance, the baud rate is modified to suit the KIM system.

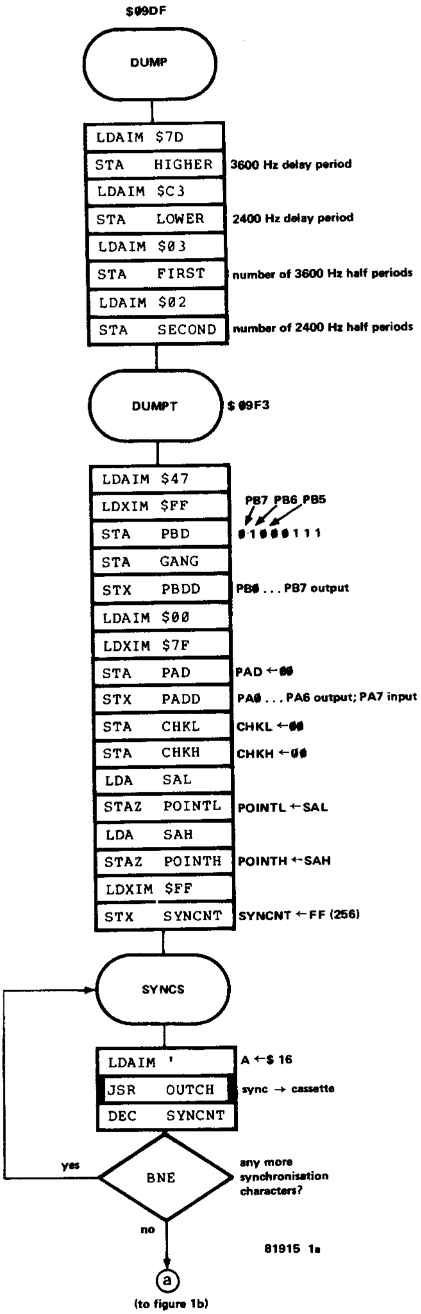
The first nine instructions after the label DUMPT are used to define the input and output parameters, so that data transfer from computer to cassette can take place. How is port B affected? All eight input/output lines belonging to port B are programmed as outputs. Furthermore, the value $47 is loaded into port B data register. As a result:

- port line PB7, which is used as the cassette data input and output, will be logic zero.
- port line PB6, which acts as a relay control output, will be logic one. This means that transistor T2 of the cassette interface circuitry will be turned off, the INPUT relay will not be activated and the green INPUT LED (D4) will be unlit (see figure 2 on page 13 of Book 3).
- port line PB5, which also acts as a relay control output, will be logic zero. This means that transistor T3 of the cassette interface circuitry will conduct, the OUTPUT relay will be activated and the red OUTPUT LED (D5) will be lit.
- the data presented to port lines PB4, PB3, PB2 and PB1, the DCBA inputs of the keyboard and display decoder (see IC7 in the circuit diagram on pages 14 and 15 of Book 1 and figure 9 on page 119 in Book 2), will be ØØ11, respectively. This means that the unused output (3) of the decoder is selected and the displays, regardless of the information presented to the segment drivers via port A, will all be turned off and the depression of any key will have no effect. This is why the display is unlit (only the red OUTPUT LED is lit) and the main keyboard is disabled while data is being transferred to cassette tape.
- port line PBØ, which is the usual serial data output line, is held at logic one. Since the only peripheral device being used at this particular moment is the cassette recorder, the RS232 line remains in the quiescent state. Any data transmitted by the computer takes place by way of port line PB7, not PBØ.

Now for port A. Port lines PAØ . . . PA6, which correspond to the segment data, are programmed as outputs and port line PA7 is programmed as an input, as usual. Since the value $ØØ has been loaded into port A data register, all the segments should be lit (as far as port A is concerned). However, this is prevented by the information being held in port B data register. Provided no characters are transmitted by any peripheral equipment, port line PA7 will be logic one. This corresponds to the quiescent state of the serial data line. In any case, any characters sent by peripheral devices are totally unintelligible as far as the DUMP routine is concerned.

To continue with the description of the DUMP routine, it should be noted

1a



```
LDAIM  $7D
STA    HIGHER   3600 Hz delay period
LDAIM  $C3
STA    LOWER    2400 Hz delay period
LDAIM  $03
STA    FIRST    number of 3600 Hz half periods
LDAIM  $02
STA    SECOND   number of 2400 Hz half periods

LDAIM  $47
LDXIM  $FF      PB7 PB6 PB5
STA    PBD      01000111
STA    GANG
STX    PBDD     PB0 . . . PB7 output
LDAIM  $00
LDXIM  $7F
STA    PAD      PAD ←00
STX    PADD     PA0 . . . PA6 output; PA7 input
STA    CHKL     CHKL ←00
STA    CHKH     CHKH ←00
LDA    SAL
STAZ   POINTL   POINTL ←SAL
LDA    SAH
STAZ   POINTH   POINTH ←SAH
LDXIM  $FF
STX    SYNCNT   SYNCNT ←FF (256)

LDAIM  '        A ←$ 16
JSR    OUTCH    sync → cassette
DEC    SYNCNT
BNE             any more
                synchronisation
                characters?
```

Figure 1a. The first section of the DUMP/DUMPT routine, which transfers data from the computer to ordinary cassette tape.

1b

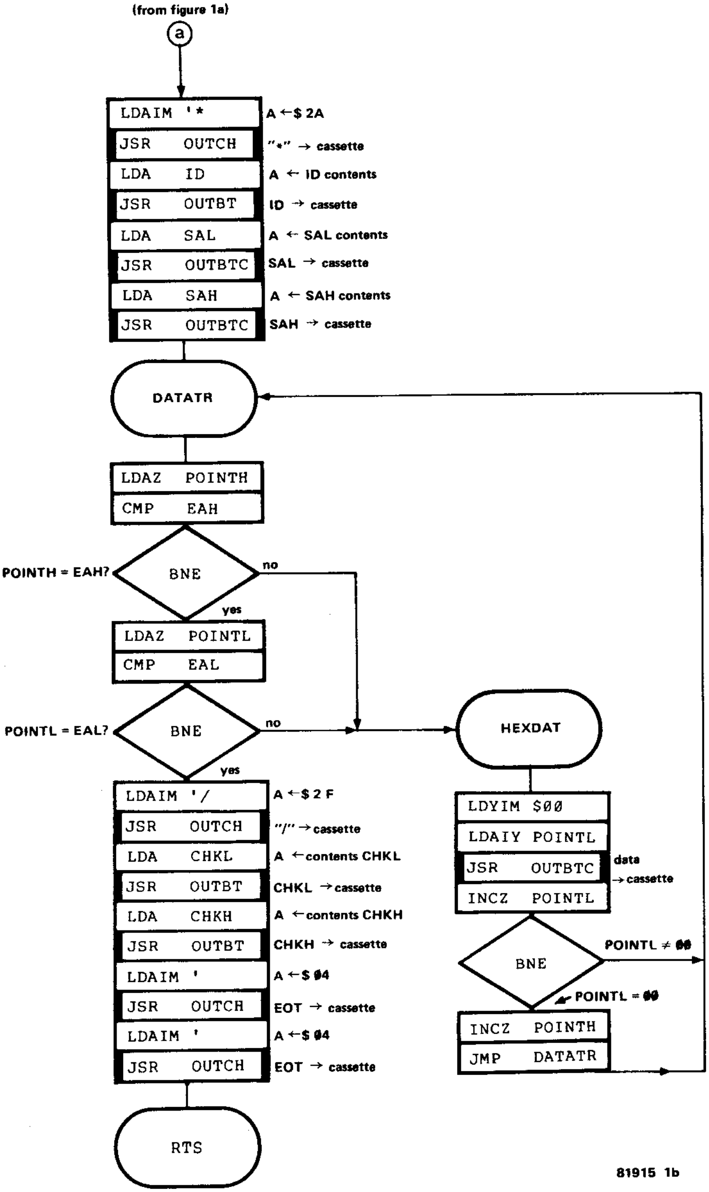


Figure 1b. The second section of the DUMP/DUMPT routine.

that address location GANG is loaded with the same information as port B data register. The purpose of this particular memory location will be explained in detail in point 2. Memory locations CHKH and CHKL, used for calculating the status of the control bytes, are initially assigned a value of zero. This brings us to the actual preparations involved in transferring data from the machine to the tape recorder. The contents of memory locations SAH and SAL, to provide the first address of the data block to be transferred, are copied into the buffers POINTH and POINTL, respectively. The last two instructions prior to the label SYNCS cause the value $FF (hexadecimal = 255 decimal) to be loaded into memory location SYNCNT. This location is used as a counter for the synchronisation characters which precede a tape recording. The remainder of the DUMP routine is described in point 7. We now move on to discuss some of the subroutines involved in transmitting bits, bytes and characters which are called during the DUMP routine.

2 Various methods of data transmission were described in Book 3. As far as the software is concerned, the relative procedures to generate the bit period and the high and low signal frequencies are as follows:

- Port line PB7 acts as the 'doorway', through which the data signal is passed from the Junior Computer to the cassette recorder.
- A data bit is encoded into a squarewave signal which always starts with a frequency of 3600 Hz and which always ends with a frequency of 2400 Hz.
- The signal at the output (port line PB7) consists of a mixture of the two frequencies — the ratio is determined by whether the data bit being transmitted is logic zero or logic one.
- A **high** bit (logic one) will produce a signal consisting of **three** half periods of 3600 Hz followed by **four** half periods of 2400 Hz.
- A **low** bit (logic zero) will produce a signal consisting of **six** half periods of 3600 Hz followed by **two** half periods of 2400 Hz.

Now for the subroutines!

a. The HIGH subroutine is shown in figure 2a. This provides an output signal which is made up of three half periods of 3600 Hz.

b. The LOW subroutine is shown in figure 2b. This provides an output signal made up of two half periods of 2400 Hz.

It can be seen that for a logic one bit to be transmitted, the HIGH subroutine has to be called once, after which the LOW subroutine is called twice. In order to transmit a logic zero bit, on the other hand, the HIGH subroutine has to be called twice, after which the LOW subroutine is called once. Furthermore, it should be noted that the HIGH and LOW subroutines always alternate during the transmission of an ASCII character, regardless of the logic level of the bit in question.

What exactly happens in the HIGH subroutine in figure 2a? The computer has to wait for three half periods of 3600 Hz during which the logic level on port line PB7 is inverted three times. The number of wait periods is stored in location FIRST the contents of which are copied into the X index register at the beginning of the HIGH subroutine.

The interval timer belonging to the PIA (peripheral interface adapter) is put to good use during the HIGH and LOW subroutines. During the HIGH

2a

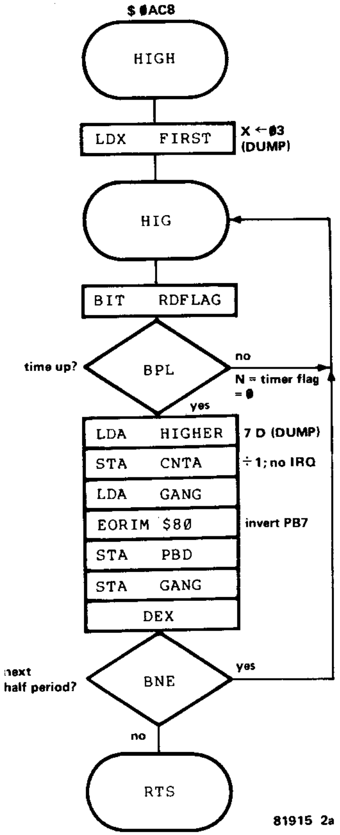


Figure 2a. The HIGH subroutine provides the 3600 Hz period of a data bit transmission.

subroutine, the timer is started by loading the value $7D into location CNTA. This provides a division factor of one and ensures that a time out does not enable an interrupt request. As soon as a time out occurs, the wait loop consisting of the instructions BIT-RDFLAG and BPL in figure 2a will be terminated. The timer is restarted almost immediately after a time out. Following this, the logic level present on port line PB7 is inverted. This brings us back to location GANG.

Being a true Exclusive-OR (see page 65 in Book 1), the instruction EOR # 8Ø causes bit 7 of the accumulator contents to be inverted, whereas the other seven bits remain unchanged.

Then why not
LDA - PBD
EOR # 8Ø
STA - PBD
instead of

```
LDA - GANG
EOR #8Ø
STA - PBD
STA - GANG ??
```

Well, the reason is quite simple. When a port line which has been programmed as an output is read, the computer will always 'see' a logic one, which is not what we want! This is why a 'shadow' memory location (GANG) has to be included and why the STA - PBD instruction is always accompanied by a STA - GANG instruction.

The instructions following the HIG label in figure 2a are executed a total of three times, since the initial value in the X index register is three — the contents of location FIRST. The contents of the X index register are decremented by one just before the BNE instruction. The latter causes the processor to branch back to the HIG label if the contents of the X index register are not yet equal to zero.

What about the delay times? At the start of the countdown the interval timer is loaded with the value $7D. In other words, 7 x 16 + 13 + 1 = 126 µs go by until a time out occurs. There is of course a slight delay between starting the timer and inverting port line PB7. This delay is 10 µs, the time taken to execute the instructions LDA - GANG, EOR #8Ø and STA - PBD. However, the time duration between successive PB7 inversions is just as long as that between successive timer starts.

The actual delay between successive inversions is 126 µs plus the time needed to execute the instructions LDA - HIGHER and STA - CNTA (8 µs) plus the duration of the loop around the BIT - RDFLAG/BPL instructions (7 µs). Therefore, the total time delay is somewhere between 134 and 141 µs. Since a full period of 3600 Hz corresponds to 277.8 µs, a half period corresponds to 138.9 µs. As can be seen, the difference between the theoretical value and the value obtained in practice is quite marginal.

3 Not much needs to be said about the LOW subroutine (figure 2b) as it is virtually identical to the HIGH subroutine. Only now two half periods of 2400 Hz are involved, which is why the X index register is loaded with the value $Ø2 (the contents of location SECOND) at the start of the subroutine. The program loop BIT - RDFLAG/BPL is terminated as soon as a time out occurs. It is important to note that this could be caused by either a previous execution of the HIGH subroutine or the LOW subroutine, as both routines are exited from **before** the final time out.

Following this, about 8 to 15 µs later (allowing the computer up to 7 µs to detect a time out), the interval timer is started yet again. Another 10 µs later the logic level on port line PB7 is inverted. The contents of the X index register are then decremented and the process is repeated once more. However, the computer does not wait for the time out resulting from the last timer start, but proceeds to the RTS instruction. Thus, the time out is not detected until the program loop BIT - RDFLAG/BPL of the **next** HIGH or LOW subroutine.

The interval timer is initially loaded with the value $C3, this being the original contents of memory location LOWER. In other words, 12 x 16 + 3 + 1 µs pass by before a time out occurs. In addition, the computer spends

2b

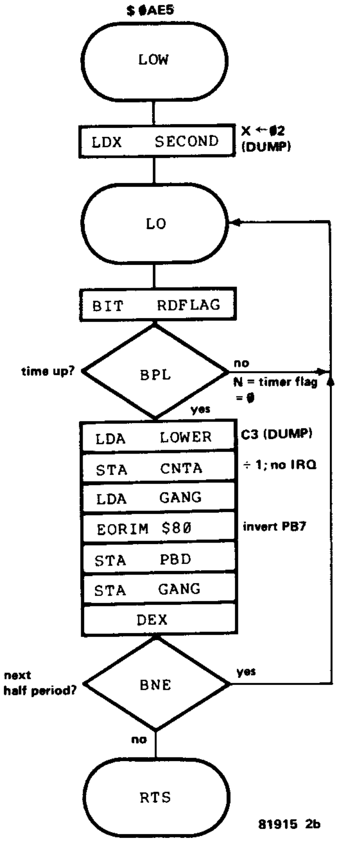


Figure 2b. The LOW subroutine provides the 2400 Hz period of a data bit transmission.

about 8 μs executing the instructions LDA - LOWER and STA - CNTA, which brings the current total up to 204 μs. Again, assuming the computer to be about 7 μs late in detecting a time out, the total will rise to around 211 μs. A full period of 2400 Hz is 416.7 μs, therefore a half period would be 208.3 μs. Again, we can see that the practical value is pretty close to the theoretical one.

4 A few general remarks about the HIGH and LOW subroutines.

First of all, the important thing to note about the HIGH and LOW subroutines is that the logic level present on port line PB7 is inverted every half period. The precise logic level is irrelevant, as what is critical is the frequency information, in other words, the time duration between successive inversions.

The HIGH subroutine is always followed by either another HIGH subroutine or a LOW subroutine. This is because the process of transmitting a data bit to cassette always starts with at least one HIGH subroutine and ends with at least one LOW subroutine. This means that after the final 3600 Hz half period, the computer waits for a time out in the BIT - RDFLAG/BPL loop of the following HIGH or LOW subroutine.

Things are slightly different in the case of the LOW subroutine. As soon as the last LOW subroutine has been dealt with, the last 2400 Hz half period has to be processed. The corresponding time out will be detected during the next BIT - RDFLAG/BPL loop, that is, the subsequent HIGH subroutine. The computer will spend less time in this loop, as in between loops the computer deals with the instructions pertaining to the next bit (label ONE in figure 3, see point 5), data nibble (label NIBOUT) or ASCII character (label OUTCH) to be transmitted. Clearly, the execution time of these instructions must not be longer than the interval between time outs. Thus, the computer has about 200 µs to deal with these instructions, which should be long enough to process about 50 instructions each taking around 4 µs. Plenty of time in fact, so there is nothing to worry about.

Now let us see how the bits, bytes and ASCII characters are transmitted to the cassette tape sequentially.

5 **The NIBOUT/OUTCH subroutine** is shown in figure 3 and is devoted to transmitting all eight bits of an ASCII character, the code of which is stored in the accumulator, to the cassette deck. The task is carried out by the instructions following the label OUTCH. In some instances the section of routine following the NIBOUT label is also executed. It may be an idea therefore, to examine this section in detail before we go any further.

A data byte is in fact split up into two nibbles before it can be transmitted to the tape recorder. These data nibbles have to be translated into their equivalent ASCII characters. Prior to the jump to the NIBOUT label, the contents of the accumulator are a hexadecimal number between Ø . . . F. The reason for this is explained in point 6 and is illustrated in figure 4 (the OUTBTC/OUTBT subroutine). The ASCII code for numbers Ø . . . 9 are 3Ø . . . 39 and the ASCII code for numbers A . . . F are 41 . . . 46. In the first instance, therefore, the hexadecimal value $3Ø has to be added to the contents of the accumulator and in the second instance the value $37 has to be added. In other words, the contents of the accumulator are always incremented by at least $3Ø and sometimes as much as $37. The instruction CMP #ØA is used to separate the 'goats' (Ø . . . 9) from the 'sheep' (A . . . F).

The OUTCH section of the subroutine starts by loading the value $Ø8 into memory location BITS, which is used as a bit counter. This means that the program from label ONE onwards is executed a total of eight times in succession. Each time a single bit of the ASCII character is transmitted to the cassette tape. All the bits in the accumulator are shifted one position to the right with the aid of the instruction LSRA. The extreme right-hand bit is shifted into the carry position. After this, the shifted contents of the accumulator are temporarily stored on the stack (the instruction PHA). The computer is now ready to transmit the data bit. The state of the carry

3

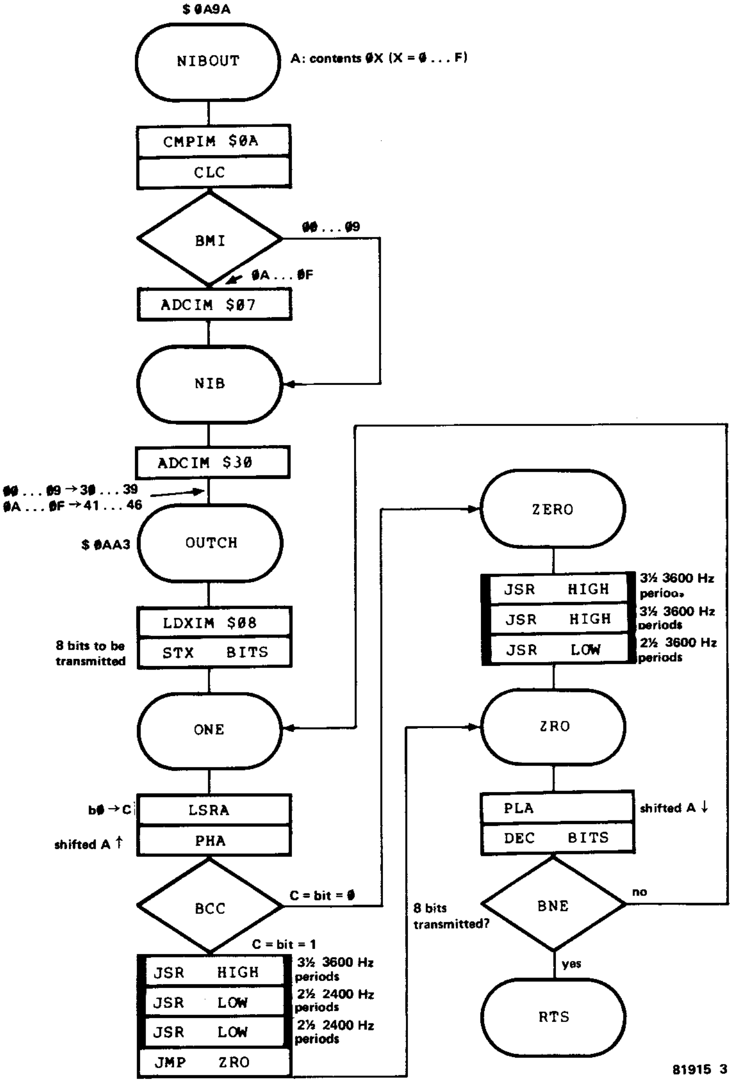


Figure 3. The NIBOUT/OUTCH subroutine converts a data nibble to its corresponding ASCII character and takes care of the transmission of the ASCII character.

flag and the instruction BCC determine the direction in which the processor is to move next. If the bit to be transmitted is logic one, the processor runs the HIGH subroutine once and the LOW subroutine twice. If, on the other hand, the bit to be transmitted is logic zero, the HIGH subroutine is executed twice and the LOW subroutine is executed once (following label ZERO). In either case, the computer ends up at the label ZRO where the previous accumulator contents are restored by the instruction PLA. The contents of location BITS are then decremented and the result determines whether or not more bits are to be transmitted. If the result is not zero, more bits are to be transmitted and the processor branches back to the label ONE so that the data in the accumulator can be shifted along one place again. The subroutine is exited from when the contents of location BITS turn out to be zero (all eight bits have been transmitted).

6 **The OUTBTC/OUTBT subroutine** is shown in figure 4 and performs the operation of splitting the data byte to be transmitted into two data nibbles so that they can in turn be converted into ASCII. This task is in fact carried out by the sequence of instructions following the label OUTBT. In some instances, however, something else has to be taken care of first. The 8 bit number constituted by the data byte has to be added to the 16 bit number contained in locations CHKH and CHKL so that a check-sum operation can be performed during retrieval from tape to see whether the data obtained is correct or not. The addition is carried out between the labels OUTBTC and OUTBT, as shown in figure 4.

Now let us assume that the data byte to be transmitted is stored in the accumulator just before the OUTBTC subroutine is called. The data byte is immediately copied into the Y index register (the instruction TAY). After the carry flag has been cleared, the data byte is added to the number contained in memory locations CHKH and CHKL using ordinary 16 bit addition. After restoring the original accumulator contents (the instruction TYA) the OUTBTC section of the program is complete.

The OUTBT section of the subroutine also starts by temporarily storing the contents of the accumulator in the Y index register. Then four LSRA instructions follow, as a result of which the original high order nibble moves into the low order nibble position and the original low order nibble is lost. The high order nibble now becomes zero. The contents of the accumulator (the original high order nibble) is then transferred to tape during the NIBOUT subroutine (see point 5 and figure 3). After this, the original contents of the accumulator are restored (the instruction TYA) and are masked by the value $ØF. This means that the original low order nibble of the data byte can now be transferred to tape (again using the subroutine NIBOUT). This, of course, brings us to the end of the OUTBTC/OUTBT subroutine.

7 **The DUMP routine continued** (see point 1). Our previous discussion of the DUMP/DUMPT routine led us as far as the SYNCS label in figure 1a. This is where 255 synchronisation characters were transmitted to the cassette tape. In other words, the accumulator is loaded with the

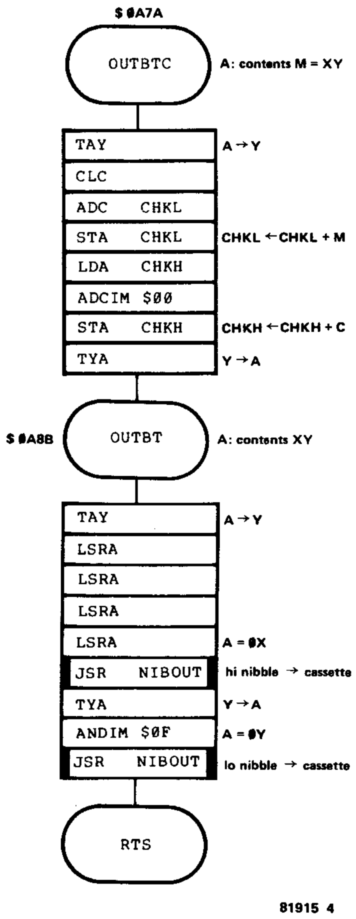


Figure 4. The OUTBTC/OUTBT subroutine transmits a data byte to cassette in the form of two data nibbles and, if necessary, updates the contents of locations CHKH and CHKL.

code for the ASCII character SYN ($16) 255 times and this character is transferred to the tape recorder via the subroutine OUTCH a total of 255 times (see point 5 and figure 3). The contents of memory location SYNCNT are decremented by one a total of 256 times. The last time this happens the following BNE instruction causes the computer to leave the program at the bottom of figure 1a and continue from the top of figure 1b. As you will remember from chapter 11 in Book 3, the synchronisation characters are followed by the 'start-of-data transmission' character '*'. Not surprisingly, therefore, the accumulator is loaded with the value

$2A, the ASCII code for '*', at the top of figure 1b. After this the program calls the subroutine OUTCH. Next, the program number (ID) and the memory locations corresponding to the beginning of the data block to be transmitted, SAH and SAL, are transferred to cassette. In all three cases a data byte is transmitted, which calls for the subroutine OUTBT to be executed. However, the start address (SAH and SAL) has to be included in the check-sum process, which is why the OUTBTC subroutine is called (see point 6 and figure 4).

8 The final section of the DUMP routine consists of the instructions following the label DATATR in figure 1b. As its name suggests, this stands for 'data transfer'. For the moment has come for the computer to transfer the data block. In other words, all the data between the start address (SA) and the end address (EA), or rather, up to and including the last address (LA) where LA = EA – 1. The familiar address buffer POINTH and POINTL is used to keep track of the data being transmitted. In figure 1a (see point 1) the contents of POINT were made the same as the start address SA.

The instructions following the label DATATR check to see whether the contents of POINT are the same as those of EA (the end address). If so, the data transmission proceeds with the 'tail end' of the program. More about this later. If not, the processor branches to the label HEXDAT. Here the contents of the location indicated by POINT are loaded into the accumulator and, once the contents of locations CHKH and CHKL have been modified, they are transferred to the cassette tape, via the subroutine OUTBTC. After this, the contents of POINT are incremented so that they indicate the next memory location from which data transfer is to take place and the program returns to the label DATATR for the next data byte to be transmitted.

As soon as the entire data block has been transferred to tape, the contents of POINT and those of EA are the same, the program moves on to the tail end. Transmission of the data block is terminated with the 'end-of-data' character '/' (ASCII code $2F), the final contents of locations CHKH and CHKL and two 'end-of-transmission' (EOT) characters (ASCII code $Ø4). This brings us to the end of both the data transfer and the DUMP routine, except for one small item . . .

9 As we mentioned previously, the DUMP subroutine may also be utilised in user software. If the transfer speed (Baud rate) required is the same as that for the Junior Computer, all that needs to be added is the instruction JSR - DUMP.

If, on the other hand, a different baud rate is required, the contents of at least two of the four locations HIGHER, LOWER, FIRST and SECOND need to be modified. Basically, what happens is that all four locations have to be re-defined and the program has to jump to the DUMPT (not the DUMP) routine.

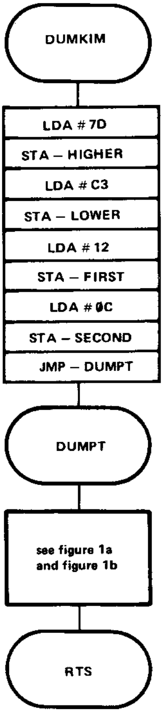
Readers wishing to work with speeds similar to those of the KIM computer should use the values given in the DUMKIM routine which is shown in figure 5. Firstly, the four memory locations are defined and then the computer jumps to the DUMPT routine, which has already been discussed

at length.
It can be seen that the contents of locations HIGHER and LOWER remain unchanged. This is because the signal frequencies associated with the data transfer are the same in both the KIM and the Junior Computer (see figure 6 on page 86 in Book 3). However, the contents of locations FIRST and SECOND are six times greater, as the transmission speed of the KIM computer is six times slower than that of the Junior Computer.

10 **The RDTAPE subroutine** is the opposite of the DUMP routine in that it allows the user to load a program into the computer from a cassette tape. The flowchart of the RDTAPE routine is shown in figures 6a and 6b. The first section of the program (the instructions between the labels RDTAPE and SYNC in figure 6a) serve to establish the input/output parameters of ports A and B. The value $32 is loaded into port B data register while the value $7E is loaded into port B data direction register. This means that:

- port line PB7, which is used as the serial data input/output line, is pro-

5



81915 5

Figure 5. The DUMKIM routine is similar to the DUMP routine, but the baud rate has been adapted to suit the KIM computer.

6a

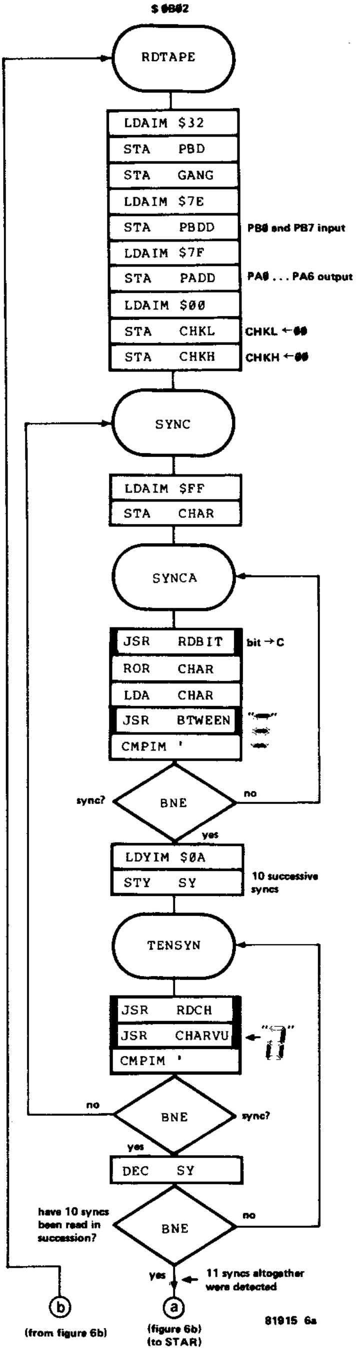


Figure 6a. The first section of the RDTAPE routine, which reads data previously stored on cassette into memory.

6b

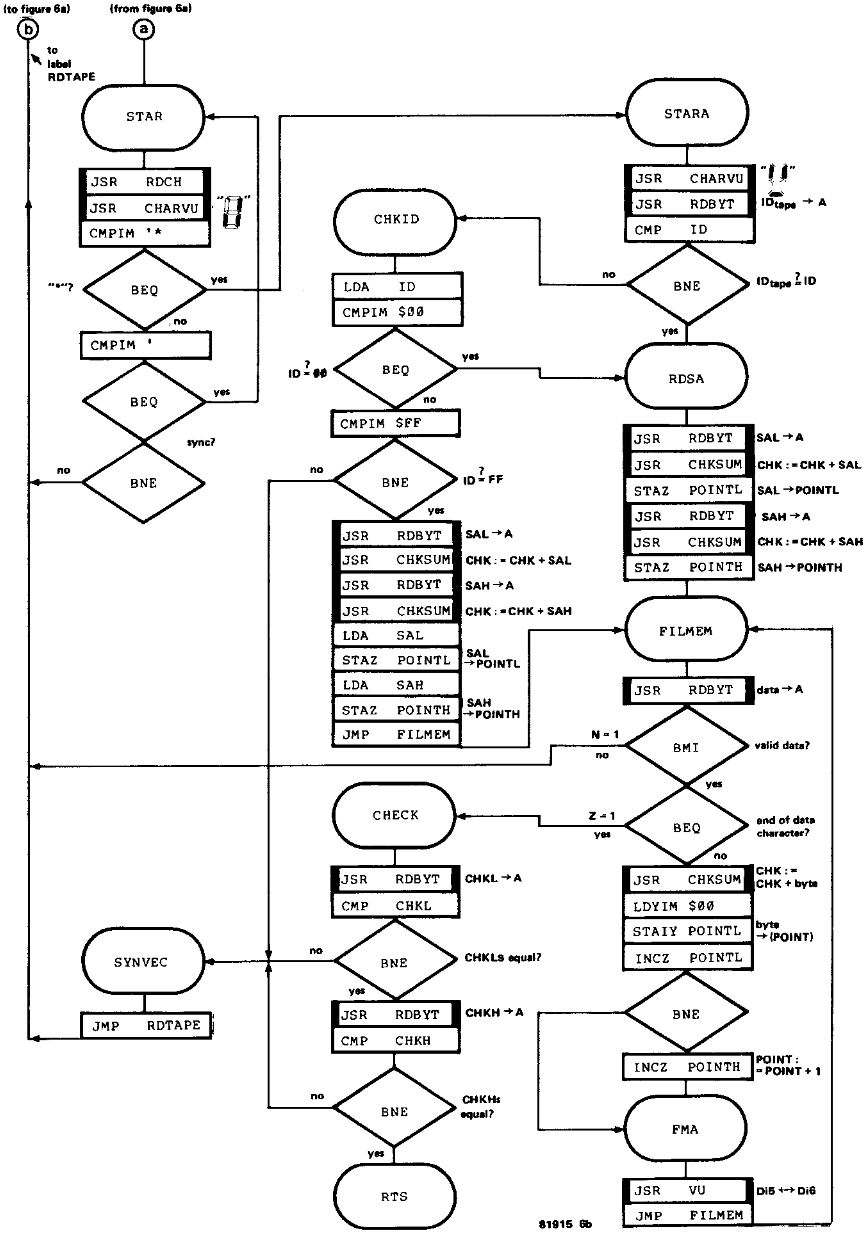


Figure 6b. The second section of the RDTAPE routine.

grammed as an input and is held at logic zero.

- port line PB6, which is used as a relay control output, is held at logic zero. This means that transistor T2 of the cassette interface circuitry is turned on, the INPUT relay is activated and the green INPUT LED is lit.
- port line PB5, which is used as a relay control output, is held at logic one. This means that the transistor T3 of the cassette interface circuitry is turned off, the OUTPUT relay is de-activated and the red OUTPUT LED is unlit.
- the data presented to port lines PB4, PB3, PB2 and PB1, the DCBA inputs of the keyboard and display decoder (see point 1), will be 1ØØ1, respectively. This means that output 9 is active and that display 6, at the far right, is enabled. As you will remember from chapter 11 in Book 3, display 5 also lights up. More about this in point 15, where we will take a look at the CHARVU/VU routine.
- port line PBØ, which is used a serial data input, is programmed as an output here and so it can be considered as being switched off.

Now for port A. The value $7F is loaded into port A data direction register, which means that port lines PAØ . . . PA6 are programmed as outputs. These lines form the segment 'data bus' as usual. The function of port line PA7 has not been altered either, it still acts as a serial data input. However, the RDTAPE routine is not interested in any characters transmitted from peripheral devices, only in the information being relayed from the cassette recorder.

The start of the RDTAPE routine may look a little odd, as it does not contain an instruction to load port A data register. The reason for this is quite straightforward: the data register is not assigned any particular value until the computer executes the BTWEEN subroutine (see point 16) or the CHARVU/VU subroutine (see point 15). As we will see later, the information held in port A data register does not remain constant.

Before we delve too deeply into the RDTAPE routine, let us take a look at all the subroutines used by the program.

Note: the rest of the RDTAPE routine will be dealt with in point 17.

11 **The RDBIT subroutine** is one of the most elementary subroutines called by RDTAPE and is shown in figure 7. The RDBIT subroutine reads a data bit from the cassette and sets or resets the carry flag depending on whether the input bit was logic one or logic zero. Now for the details. Unfortunately, there are quite a few details to consider, as the procedure is rather a complicated one.

To understand the RDBIT subroutine a little better, let us take a look at the pulse diagrams given in figures 8a . . . 8f. Figures 8a (logic zero bit) and 8d (logic one bit) should look familiar. In fact, they were first introduced in chapters 10 and 11 of Book 3. In both cases, part of the phase-locked loop (PLL) output signal is involved. Unfortunately, the PLL output suffers from a little 'jitter' and therefore the more likely signals are represented in figures 8b and 8e. The PLL jitter is indicated as slight fluctuations between the high and low logic levels, very similar to contact bounce where keys or switches are concerned. In reality, things are slightly more complicated, but the main thing to note here is that the signal level fluctuates and that the fluctuations have nothing to do with the end of a

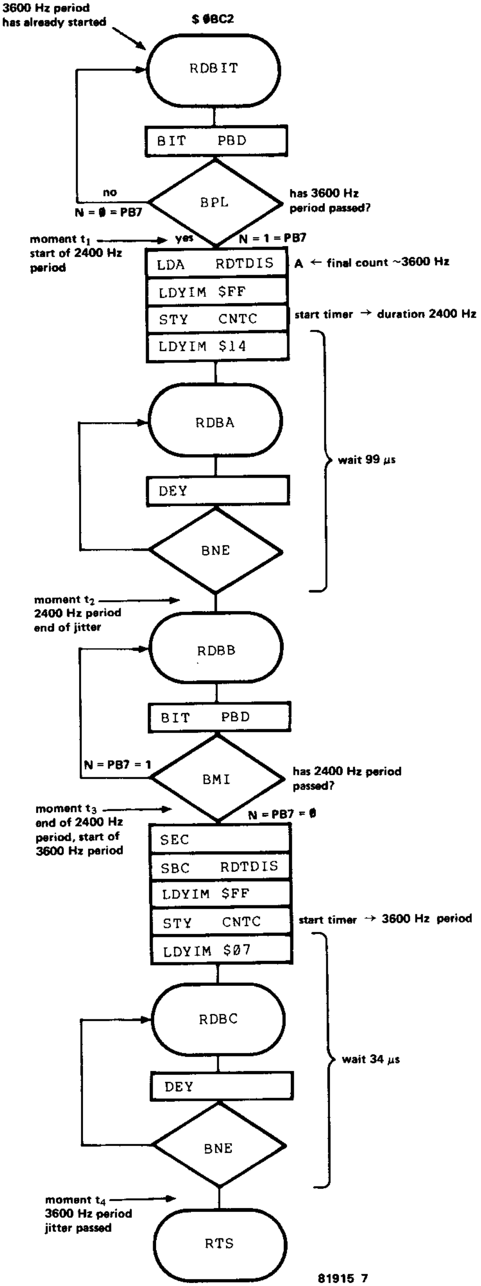


Figure 7. The RDBIT subroutine reads in a data bit from cassette. The logic level of the bit is reflected by the status of the carry flag.

8

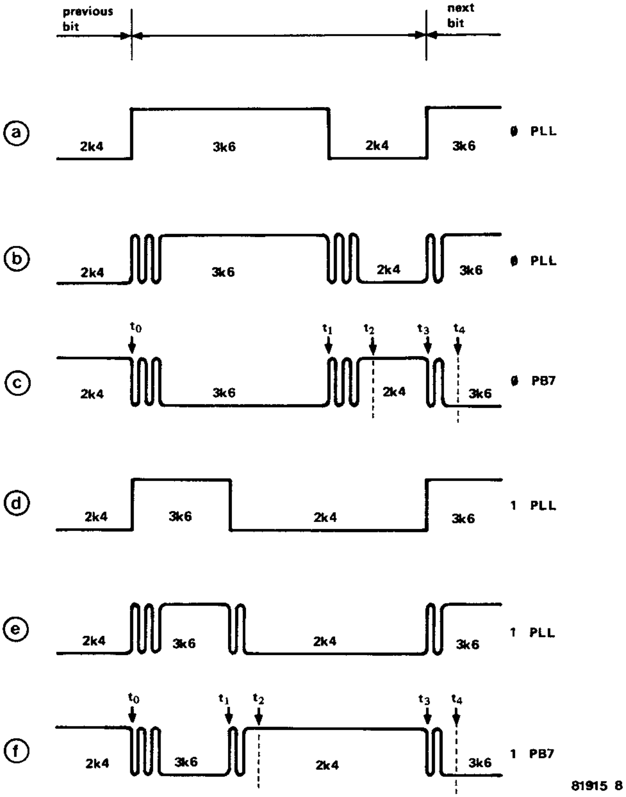


Figure 8. The output signals from the PLL during a low bit transmission a . . . c, and a high bit transmission d . . . f.

2400 Hz or 3600 Hz period.

Before the PLL signal reaches port line PB7, which acts as the serial input, it is inverted, which explains the requirement for figures 8c and 8f. The resultant input signal presented to port line PB7 combines parts of figure 8c with parts of figure 8f. The 'bit train' is related to the logic levels of the consecutive bits being read from the tape. Whatever the bit train looks like, the 3600 Hz (first phase) and 2400 Hz (second phase) signals always alternate. This is the basic principle behind the RDBIT routine.

Now let us take a closer look at the RDBIT routine by examining figure 7. The routine starts with a wait loop consisting of the instructions BIT - PBD and BPL. The computer sits in this loop until such time as the N flag, bit 7 in port B data register (PB7), becomes logic one. From figures 8c and 8f it is quite apparent that port line PB7 becomes logic one at the end of 3600 Hz period preceding a 2400 Hz period. At that point the computer will be terminating the first phase of the bit and starting the second.

This occurs at moment t_1. Immediately afterwards, the instruction LDA - RDTDIS is executed. This instruction loads the accumulator with the current value contained in the interval timer resistor.

Obviously, there is no point in reading the value of the interval timer unless the timer has been started at some time or another. Well, in fact it was. At moment t_1 the contents of the timer were read immediately after the end of a 3600 Hz period. Before that, the timer was started at the beginning of the 3600 Hz period, at moment t_0. This happened prior to the jump to the RDBIT subroutine, during the last phase of the previous RDBIT routine. Whenever this routine is executed, a data bit is read and stored in the carry flag. During the last phase of RDBIT the next RDBIT is prepared, the first RDBIT phase being prepared during the last phase of the previous RDBIT. This is because the bit train has regular 3600 Hz - 2400 Hz - 3600 Hz intervals etc.

Back to figure 7. After the last 3600 Hz signal has been read, the interval timer is started again. The instructions LDY # FF and STY - CNTC ensure that the contents of the timer register are decremented from their initial value of $FF (255) every 64 μs. No interrupt request occurs after a time out. This procedure was described in detail in chapter 6 of Book 2.

We have now reached the label RDBA. The computer waits in the loop DEY/BNE until the period of PLL jitter (as a result of the transition from 3600 Hz to 2400 Hz) has definitely passed. In other words, the computer waits until moment t_2 in either figure 8c or figure 8f has arrived. The computer has to do this, as otherwise the next wait loop (the loop starting at label RDBB, which determines when the 2400 Hz period has passed) will be terminated prematurely, with disastrous results.

As soon as the 2400 Hz period is over, moment t_3, the BMI instruction causes the computer to continue with the final section of the RDBIT routine. This brings us to the interesting SEC and SBC - RDTDIS instructions, which we will examine separately in point 12. At this stage all you need to know is that following these instructions the status of the carry flag is the same as the logic level of the data bit just read from tape. After this has been done, the interval timer is started once again to prepare the way for a new jump to the RDBIT subroutine.

The final instructions in the RDBIT subroutine (DEY and BNE) provide a short delay loop to ensure that any PLL jitter has ceased before the computer moves on to the next stage of the program. In other words, the computer does not enter the first delay loop of the next RDBIT subroutine until moment t_4.

12 Now for the instructions SEC and SBC - RDTDIS which we encountered in the RDBIT subroutine. The SEC instruction simply ensures that the carry flag is set before the subtraction, so the borrow is reset (zero). As a result of the instruction SBC - RDTDIS, the current value in the timer register is subtracted from the contents of the accumulator when the 2400 Hz period has ended. Earlier on, the accumulator was loaded with a value relative to the timer register contents at the end of the 3600 Hz period (the instruction LDA - RDTDIS).

The timer is loaded with the value $FF both at the beginning of a 2400 Hz period and a 3600 Hz period. The count is decremented every 64 μs.

Obviously, the longer a 2400 Hz or 3600 Hz period lasts, the lower the timer count will be.

Going back to the subtraction instruction; if the value in the interval timer register is less than or equal to the contents of the accumulator prior to the subtraction being performed, the result will leave the carry flag set (C = 1). This is because nothing had to be borrowed, so that the borrow flag is still zero (carry = $\overline{\text{borrow}}$). If, on the other hand, the value in the interval timer register is greater than the contents of the accumulator prior to the subtraction being performed, the result will reset the carry flag (C = Ø).

If the contents of the accumulator are greater than those of the timer register (C = 1), the 3600 Hz period must have been longer than the 2400 Hz period. This means that a logic one bit was involved (see figure 8). It follows, therefore, that when the carry flag is reset (C = Ø), a logic zero bit was involved, Thus,
when C = 1 a logic one bit was received and
when C = Ø a logic zero bit was received.

As an analogy, take a look at the two towers, A and B, in figure 9. Both towers are made up of 255 ($FF) building blocks. During the final phase of a 3600 Hz period, one brick is removed from tower A every 64 µs. The same happens to tower B during the final phase of a 2400 Hz period. When both periods have passed, tower A will be higher than tower B if a logic one bit was received (A – B = positive) and tower A will be smaller than tower B if a logic zero bit was received (A – B = negative). Note that the PLL jitter wait loops have no effect on the two timer counts whatsoever.

The nominal ratios of 3600 Hz signal to 2400 Hz signal are either 2 : 1 or 1 : 2, so the result of the subtraction is always either clearly positive or clearly negative, even if the PLL is not spot on tune (for instance, because of a slightly different tape speed). For more details, read chapter 11 of Book 3.

The total duration of a complete bit can be calculated to be about 1250 µs:
a logic zero bit = 833 µs of 3600 Hz + 417 µs of 2400 Hz
a logic one bit = 417 µs of 3600 Hz + 833 µs of 2400 Hz.

A bit period of 1250 µs corresponds to a baud rate of 800 bits per second. Since the timer is decremented every 64 µs, about 13 decrements will occur in the space of 833 µs and 6 decrements in the space of 417 µs. This means that there is never a risk of the timer contents reaching zero, as the initial value was 255 (= $FF). In other words, the towers in figure 9 will never be totally demolished. It also means that the system can be used to read in tapes recorded on a KIM computer, which is a factor of 6 times slower. It makes no difference whether 13 or 6 x 13 is subtracted from 255, the result of the subtraction will always be positive.

One aspect of the RDBIT subroutine still has to be clarified. The first bit to be read from cassette tape is not preceded by a RDBIT subroutine to detect the start of a 3600 Hz period. However, this does not matter in the least, since the first bit corresponds to the first synchronisation character to be read from cassette. Even if things go terribly wrong with the first bit or the entire first synchronisation character, the computer still has another 254 opportunities to detect one!

9

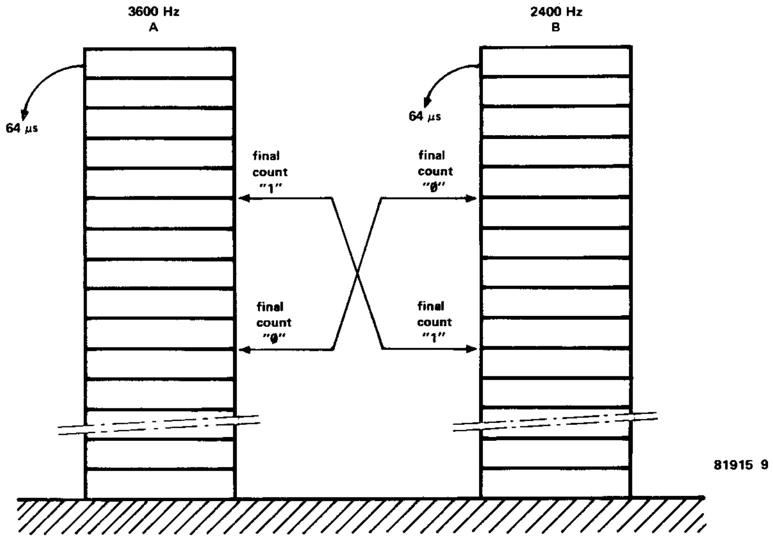


Figure 9. The tower model illustrates how the RDBIT subroutine works. Tower A is higher than tower B in the case of a logic high bit, but lower in the case of a logic zero bit.

Finally, a remark along the lines of point 4. Clearly, the time it takes to carry out the instructions before the following RDBIT subroutine must not exceed the 3600 Hz period. There are roughly 400 µs available for the last four instructions (including the RTS) of the RDBIT subroutine, plus the instructions preceding the next run of the RDBIT subroutine.

13 **The RDCH subroutine** is shown in figure 10. This routine performs the operation of reading in an ASCII character, in other words, a series of eight bits in succession.

During the discussion on the DUMP routine, we discovered that the least significant bit of the data byte (bØ) is transferred to the cassette first, the most significant bit (b7) being transmitted last. This is illustrated in figure 3 and was also explained in the paragraph on the OUTCH subroutine in point 5. Therefore, if eight data bits are read from the cassette in succession, all of course belonging to the same ASCII character, bit Ø will be the first data bit to arrive and bit 7 the last.

The RDCH subroutine starts by loading the value $Ø8 into the X index register. This register is used as bit counter during the RDCH subroutine. The processor then performs the instructions JSR - RDBIT (read data bit from cassette), ROR - CHAR (shift all the data bits in location CHAR one position to the right and copy the status of the carry flag into the b7 position), and DEX (prepare for the next bit).

When all the data bits in the byte have been read in, the instructions ROL - CHAR and LSR - CHAR are executed. As a result, bit 7 in location CHAR is made logic zero. Bit 7 could well be used as a parity bit, but we

10

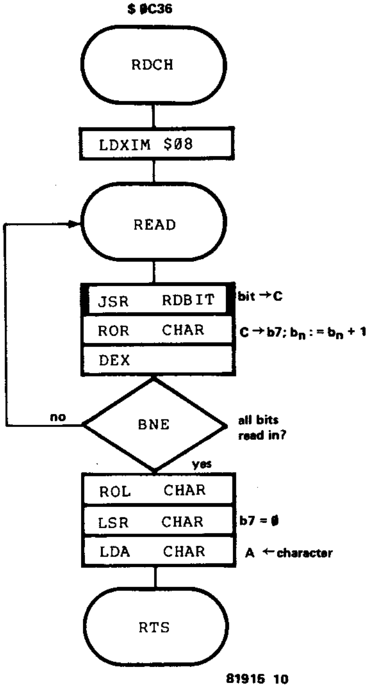


Figure 10. The RDCH subroutine reads an ASCII character from cassette.

have already discovered that the Junior Computer does not use b7 for parity. (During transmission of the ASCII character to cassette, bit 7 was made logic zero.) The final two instructions in the RDCH subroutine simply copy the contents of memory location CHAR into the accumulator (LDA - CHAR) and end the routine (RTS).

14 **The RDBYT subroutine.** Now that we know how a data bit and an ASCII character are read into memory from cassette (points 11 . . . 13), it seems quite logical to find out how a data byte (two ASCII characters) is read in. This requires the RDBYT subroutine which is shown in figure 11.

During the DUMP routine the high order nibble of the data byte was transmitted to cassette first, followed by the low order nibble, see point 6. Therefore, when the information is read back from tape, the high order nibble will obviously reach the computer first, followed by the low order nibble.

The RDBYT subroutine starts by calling the RDCH routine, which we have just dealt with. We should all know by now what the latter routine does – an ASCII character is read from cassette tape and stored in the accumulator. The ASCII character could either be a coded data nibble, or

11

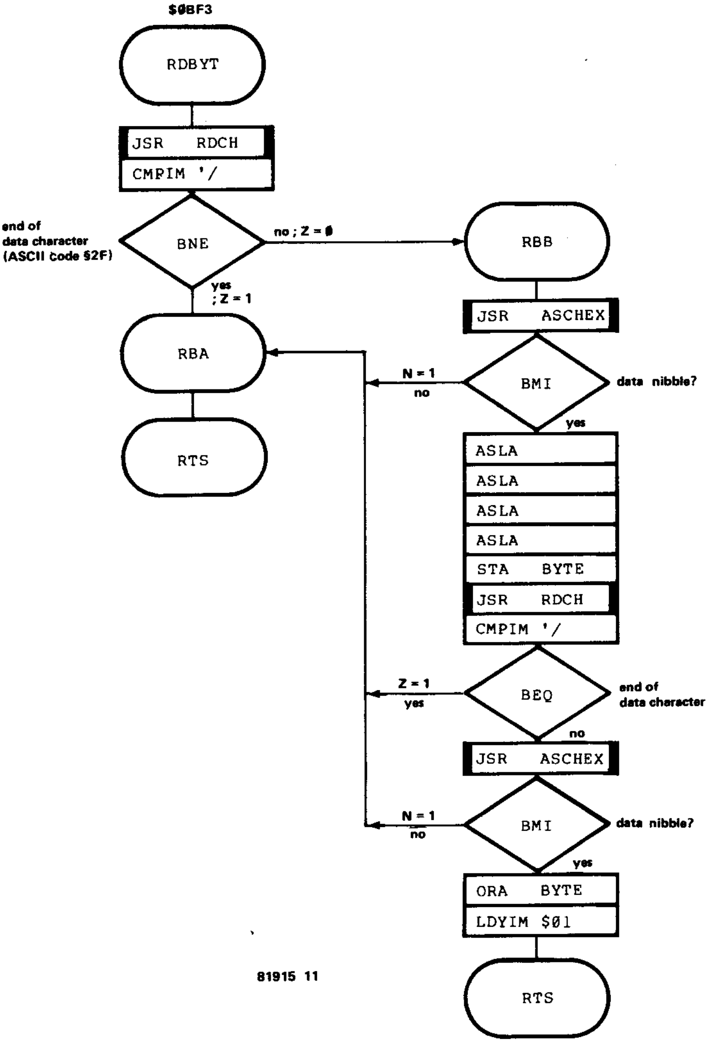


Figure 11. The RDBYT subroutine reads a data byte from cassette by reading a data nibble (ASCII character) twice in succession.

it could be the 'end-of-data' character '/'. In the second instance, the RDBYT subroutine will be exited from immediately as the Z flag will be set. In the first instance, the computer will proceed to label RBB and call the subroutine ASCHEX, which is illustrated in figure 12.

If the ASCII code of a data nibble is involved, this will have to be converted back to the corresponding hexadecimal number. This does not

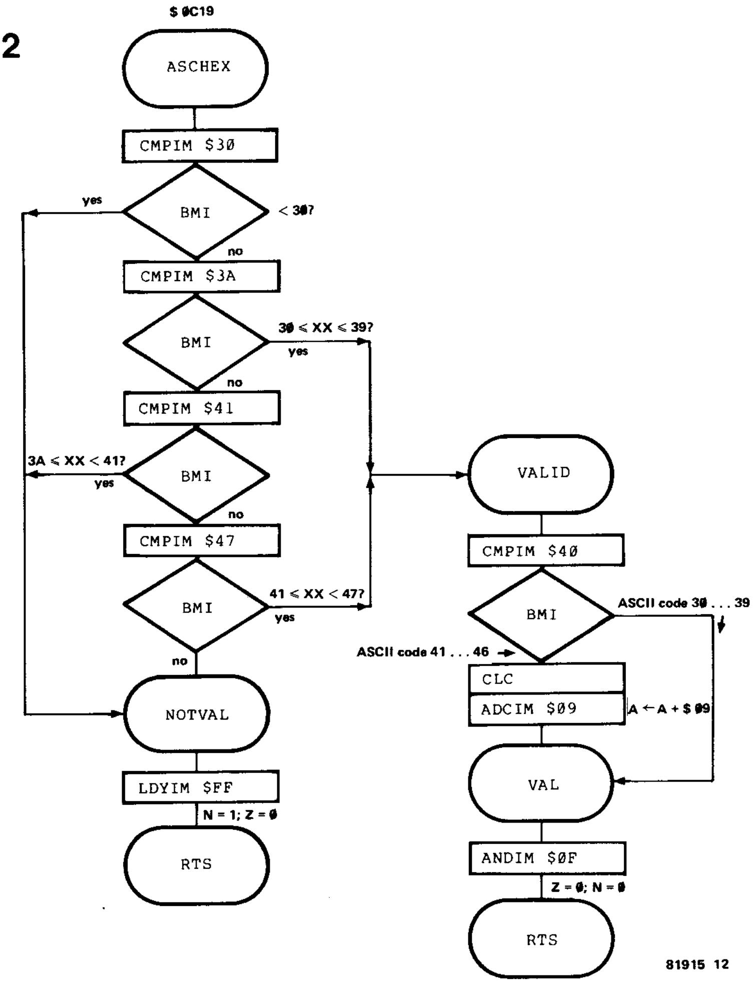


Figure 12. The ASCHEX subroutine converts an ASCII character code back to its hexadecimal value.

happen however, until the computer is absolutely sure that the ASCII code of a data nibble is involved.

Hexadecimal numbers Ø . . . 9 are coded 3Ø . . . 39 in ASCII and the numbers A . . . F are coded 41 . . . 46. If the ASCII character is not one of the values mentioned here, this will be 'reported' by the status of the N flag, whereupon the processor will exit from the ASCHEX and RDBYT subroutines.

The ASCHEX subroutine starts with a 'question-and-answer' game consisting of four compare (= CMP) and four branch (= BMI) instructions.

ASCII codes not belonging to a hexadecimal character lead the processor to the label NOTVAL where the N flag is set and the routine exited from. ASCII codes which do represent a hexadecimal character, however, lead the processor to the VALID label where the ASCII character is 're-structured' to provide the correct data.

As far as the numbers Ø . . . 9 are concerned, all that is required is for the high order nibble of the ASCII character to be made zero. This is accomplished with the instruction AND #ØF. Where the numbers A . . . F are concerned, however, the value Ø9 needs to be added to the contents of the accumulator first. This 'masking' process also ensures that the N flag is reset before the processor exits from the ASCHEX subroutine.

Back to the RDBYT subroutine and figure 11. The BMI instruction following the first jump to subroutine ASCHEX instruction (label RBB) will cause the processor to exit from the RDBYT subroutine if the ASCII character just read was not a hexadecimal number (the N flag is set). If it was a hexadecimal number, four shift left (= ASLA) instructions are executed – the hexadecimal value is moved up to occupy the high order nibble of the accumulator contents. This value is then copied into location BYTE. The computer then jumps to subroutine RDCH to read in another ASCII character. Again, if this turns out to be the end-of-data character '/', the computer will leave the RDBYT subroutine (label RBA) after setting the Z flag. If it is not the end-of-data character, the ASCHEX subroutine is called again to ascertain whether or not a hexadecimal value is concerned. If so, the contents of location BYTE are ORed with the second data nibble to form the complete data byte. Just before the end of the subroutine, the Y index register is loaded with the value $Ø1. This causes both the N flag and the Z flag to be reset.

15 It is now time for a few RDTAPE subroutines which help to display the state of affairs during a read from cassette. These subroutines provide the three indications given in figure 7 of chapter 11 on page 89 of Book 3. Let us examine the **CHARVU/VU subroutine** in figure 13 first of all. In most instances all of the instructions between the labels CHARVU and RTS are executed, but in one instance (see the BTWEEN subroutine in figure 14) only the instructions following the VU label are carried out. This happens when a data block is being searched for, is found and is entered into memory.

The CHARVU subroutine starts by saving the contents of the accumulator on the stack. Next, bits Ø . . . 6 in the accumulator are inverted using the instruction EOR #7F. This is then stored in port A data register to become the new segment code for the displays. Now that we have mentioned segment codes, figure 15 contains the 128 different segment code possibilities. Codes belonging to a particular row all share the same high order nibble value, whereas those belonging to a particular column all share an identical low order nibble value. The reason for the inversion of the accumulator contents will be given in point 17, during the further discussion of the RDTAPE subroutine. After port A data register has been loaded the contents of the accumulator are restored to their previous value by means of the instruction PLA. The contents of the accumulator must be saved as the ASCII character needs to be further processed. However,

13

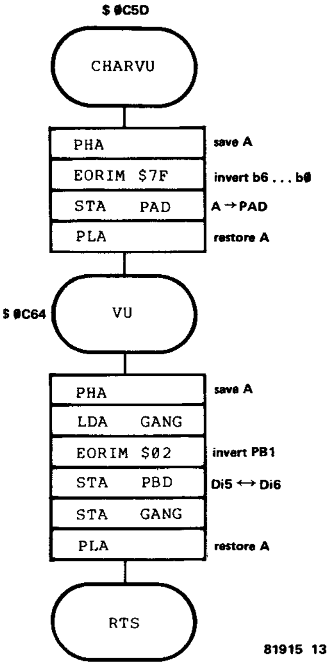


Figure 13. The CHARVU/VU subroutine inverts the bit data and passes it on to the display segments. The VU section ensures that displays 5 and 6 are 'multiplexed'.

14

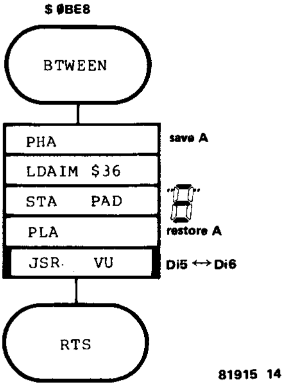


Figure 14. The BTWEEN subroutine is used to display the 'search' pattern of three horizontal bars when a data block is being looked for on a cassette tape.

15

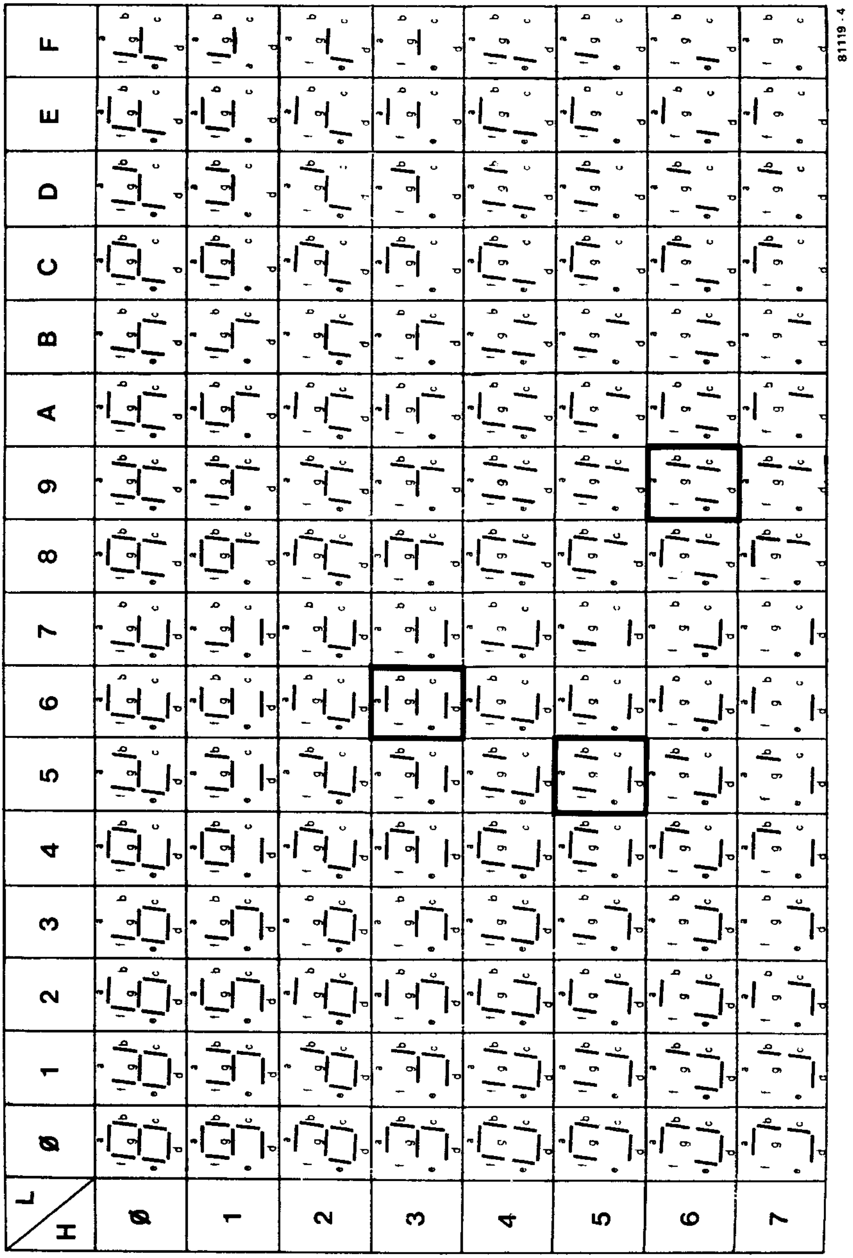


Figure 15. The 128 different segment patterns which can be displayed. The patterns seen during the cassette routines are shown boxed.

before that can be done the ASCII character must report to the display (in inverted form).

For the same reason, the first instruction after the VU label also pushes the accumulator contents onto the stack. The accumulator is then loaded with the current contents of location GANG, which are the same as those of port B data register. The logic level at port line PB1 is then inverted by means of the instructions EOR #Ø2 and STA - PBD (the result is also stored in location GANG). Let us examine this a little closer:

	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PBØ
A = PBD:	Ø	Ø	1	1	Ø	Ø	1	Ø
EOR #Ø2:	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø
result:	Ø	Ø	1	1	Ø	Ø	Ø	Ø

(zero if bits are the same, one if not)

Port B data register contained an initial value of $32, which was established at the beginning of the RDTAPE routine. As a result of this, display 6 was enabled – see point 10. The EXOR instruction causes the logic level at PB1 to be inverted. This means that display 5 is enabled and display 6 is disabled. In other words, the information presented to the display jumps from Di6 to Di5. If, on the other hand, display 5 was enabled before the VU subroutine was called, it will be disabled and Di6 enabled after the EXOR instruction. Thus, every time that the VU subroutine is run, one display is enabled and the other disabled. Provided subroutine CHARVU or subroutine VU is called a number of times (as will happen when data is being read from tape) then it will appear that both displays are lit at the same time. This is due to the multiplexing principle which we first encountered in chapter 7 of Book 2.

16 **The BTWEEN subroutine** is shown in figure 14 and is called whenever the 'between' character (three horizontal bars) needs to be displayed. This occurs when the tape passing the playback head contains no data. In other words, between data blocks (see page 89 of Book 3).

The BTWEEN subroutine starts by saving the contents of the accumulator on the stack. The code for the three horizontal bars ($36) is then loaded into the accumulator and from there passed on to port A data register. The accumulator contents are then restored and the subroutine VU is called to switch between displays 5 and 6. This situation continues until an ASCII character is read from tape and a different segment code is passed on to port A data register.

17 **The RDTAPE subroutine continued.** Now let us continue with the discussion of the RDTAPE routine which we left in point 10. We had got as far as the SYNC label in figure 6a. The routine continues by loading the value $FF into memory location CHAR. This location was discovered in point 13 when the RDCH subroutine was discussed. The initial contents of location CHAR were not established at the beginning of the RDCH subroutine, nor did they need to be. The reasons why they need to be established now will be explained a little further on.

After the label SYNCA, the computer reads in the very first data bit from the tape. It accomplishes this with the aid of the RDBIT subroutine (see points 11 and 12). As soon as a bit is detected it is shifted into bit 7 of the accumulator by means of the instructions ROR - CHAR and LDA - CHAR. The subroutine BTWEEN is then called to display the three horizontal bars (see point 16). Then the accumulator contents are compared with the value $16 to see whether a synchronisation character has been received yet ($16 is the ASCII code for the synchronisation character).

At this particular moment the computer cannot possibly have detected a synchronisation character as the RDBIT subroutine has only been run once. After the instructions following the SYNCA label have been executed once the value contained in location CHAR will be either
Ø1111111 or
11111111 depending on the value of the detected bit.
Compare this with the hexadecimal code for the synchronisation character:
ØØØ1Ø11Ø.
The initial value contained in location CHAR (=$FF) prevents a synchronisation character from being detected until at least eight bits have been read in succession.
When a synchronisation character is at last detected, the value $ØA is loaded into memory location SY. The processor has now reached the label TENSYN. At least ten synchronisation pulses have to be read in from tape without being interrupted by a different character. As soon as this happens the computer interprets it as the beginning of a data block to be entered into the memory banks. If the detected character is not a synchronisation character, the computer will branch back to the SYNC label to repeat the procedure.
During the program loop around the TENSYN label, the jump to RDCH subroutine instruction is followed by a jump to CHARVU subroutine instruction. This means that something is going to happen to the display. Unless a different character is read in during the RDCH subroutine the value $16 will be held in the accumulator. By inverting bits Ø . . . 6 in the accumulator (see point 15) port A data register is loaded with the value $69. As can be seen from figure 15, the second situation in figure 7 of chapter 11 will be displayed to indicate the fact that a synchronisation character is being read. As the subroutine CHARVU is called repeatedly, during both the loop around the TENSYN label and the loop around the STAR label (see top of figure 6b), displays 5 and 6 will be alternately enabled continuously and will therefore both appear to be 'on' at the same time.

18 Once the computer has detected ten synchronisation characters in a row (eleven in all, as one was detected before the label TENSYN was reached), the processor arrives at label STAR at the top of figure 6b. The program loop around this label is not terminated until all the other synchronisation characters (as many as 255 – 11 = 244) have been received. There are two ways in which the loop can be terminated. One way is if the start-of-data character '*' is detected. The processor will then continue from the label STARA. The second method is if a character other than a synchronisation character or a start-of-data character is detected. If this second instance should occur then something has gone wrong and the whole procedure has to start all over again . . . back to the label RDTAPE.

Therefore, the label STARA is reached as soon as the start-of-data character has been read in from tape. The ASCII code for this character is $2A. During the CHARVU subroutine this value is inverted and stored in port A data register. This provides the segment code $55, which corresponds to the third situation in figure 7 of chapter 11. This means that the computer is reading the specified data block.

During the next section of the RDTAPE routine the computer checks to see whether the next data byte to come along (the label identifier) corresponds to the contents of location ID. In other words, whether the ID entered by the operator is the same as the ID written on the tape. If so, the processor continues with the instructions following the RDSA label. If, on the other hand, the two numbers are not the same, the computer checks to see whether the operator entered a 'special' ID (ØØ or FF).

If the operator entered an ID other than ØØ, FF or that stored on the tape, the processor jumps back to the start of the RDTAPE routine via the label SYNVEC. We have now come to the conclusion that the data block encountered on the tape was the wrong one and we shall have to wait until the correct data block comes along. Obviously, it all depends on what is stored on cassette. In between the jump to subroutine CHARVU (label STARA) and jump to subroutine BTWEEN (label SYNCA) instructions, the segment code for the third situation in figure 7 of chapter 11 will appear on the displays very briefly. This indicates that the processor has returned to the start of the RDTAPE routine (see figure 6a).

Apart from the ID being the wrong one, there are three other possibilities:

16

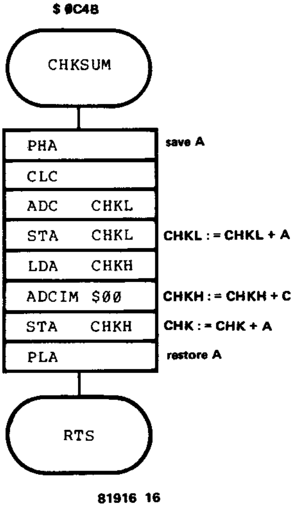


Figure 16. The CHKSUM subroutine adds the value of the current data byte to the 16 bit contents of locations CHKH and CHKL.

A. An identifier between Ø1 . . . FE has been found. The processor moves on to execute the instructions between the labels RDSA and FILMEM. As a result, the low order start address (SAL) is read in to the computer via the RDBYT subroutine. The CHKSUM subroutine is then called, see figure 16. This routine simply saves the contents of the accumulator, adds this current value to the 16 bit number held in locations CHKH and CHKL and restores the previous accumulator contents. Upon the return from the CHKSUM routine the accumulator contents are stored in the address buffer POINTL. The high order byte of the start address is dealt with in the same manner and the result stored in address buffer POINTH. Effectively, the address buffer POINT now contains the start address of the data block being received and the value contained in location CHK has been updated.

B. The operator entered an ID of ØØ. The processor once again finds itself at label RDSA and continues as above. This is correct, as when an ID of ØØ is entered, the identifier on the tape is ignored and the first data block to come along is stored in memory.

C. The operator entered an ID of FF. In this instance the values of the high and low order bytes of the start address stored on tape are added to the contents of location CHK. Then the start address previously entered by the operator is copied into the address buffer POINT. This is once again correct as both the identifier and the start address on tape are ignored when an ID of FF is entered. However, the high and low order bytes of the start address have to be added to the contents of CHK so that the final checksum operation produces the correct result. In all cases we have now reached the label FILMEM.

19 The instructions following the FILMEM label in figure 6b read in the entire data block previously recorded on tape and enter it into the computer memory. The process once again starts with the RDBYT subroutine, which was described in point 14 and is illustrated in figures 11 and 12. If, at the end of the RDBYT subroutine, the data concerned is invalid (N = 1), the computer will return to the start of the RDTAPE subroutine. If, on the other hand, the processor encounters the end-of-data character '/', it will proceed to the label CHECK. However, if the data concerned is still valid, the contents of location CHK are processed once more (JSR - CHKSUM). After this, the data that has just been read in is entered into a memory location determined by the current contents of the address buffer POINT. As mentioned previously, the initial contents of POINT correspond to the start address of the data block being entered into memory from cassette.

Once the data has been stored in memory, the contents of POINT are incremented, which brings us to the label FMA. Here, the subroutine VU is called. This routine was described in point 15 and illustrated in figure 13. It causes the two displays, Di5 and Di6, to be enabled and disabled alternately. By the way, the segment code which was loaded into port A data register during the CHARVU subroutine following the label STARA will still be displayed. The last instruction in this particular program section simply causes the processor to jump back to the label FILMEM so that the next data byte can be processed.

To end the discussion of the RDTAPE routine we will take a quick look at the final instructions after the CHECK label in figure 6b. When the processor encounters the end-of-data character, '/', the contents of memory location CHKL which were stored on the cassette are loaded into the accumulator via the RDBYT subroutine. This value is then compared to the contents of location CHKL which has just been calculated from the data being received by the computer. If the two values are not the same, an error has occurred somewhere and the process will have to be repeated, so the computer branches back to the start of the RDTAPE routine, via the SYNVEC label. If, on the other hand, the two final values are the same, the processor will move on to test whether the two values for CHKH are also the same. If they are, the processor exits from the subroutine via the RTS instruction. If they are not the same, the processor branches back to the start of RDTAPE once again.

If either the two values for CHKL or the two for CHKH are different, the data block (albeit somewhat corrupted) is stored in computer memory, but the RDTAPE subroutine is not exited from. This can be seen on the displays, as the third situation in figure 7 of chapter 11 will be followed by the first situation. Normally speaking, the display will be unlit when the computer leaves the RDTAPE routine after a successful data storage procedure. In addition, the Printer Monitor program informs us of the fact by reporting the text 'READY'. When called by the Tape Monitor program, the computer again informs us that it has finished reading in a data block from cassette by reporting 'id xx' on the displays. This particular aspect will be dealt with later on in this chapter.

20 **The main routine of the Tape Monitor program.** The flow chart of the main section of the Tape Monitor (TM) program is illustrated in figures 17a . . . 17c. The structure of the routine is very similar to that of the other system programs for the Junior Computer. To start with, there is an initialisation routine consisting of the instructions between the labels TPINIT and TPI, and the following instructions between the labels TPI and TPTXT. The latter label also acts as a focal point: the central label of the main TM routine. The computer returns to this label at the end of the PAR key routine (see point 24), once the various parameters (ID, SAH, EAH, EAL, BEGADH, BEGADL, ENDADH and ENDADL) have been set up. After the GET key routine (see point 22) and the SAVE routine (see point 23), the computer runs the second section of the initialisation routine, starting at label TMI, before returning to the TPTXT label. After the SEF and EDIT key routines (see point 25) the TM program is exited from completely and the computer enters the editor mode.

Straight after the central TPTXT label, the name of the particular memory location is displayed, along with its current contents. The computer then waits for a key to be depressed. Depending on which key is depressed, the computer will execute either a key routine or, if a hexadecimal key was depressed, the data key routine.

Upon close examination of the TM program software it will become apparent that the memory location DISCNT has a very important part to play in the proceedings. The value contained in this location determines which of the nine parameters have to be displayed and/or altered. There-

17a

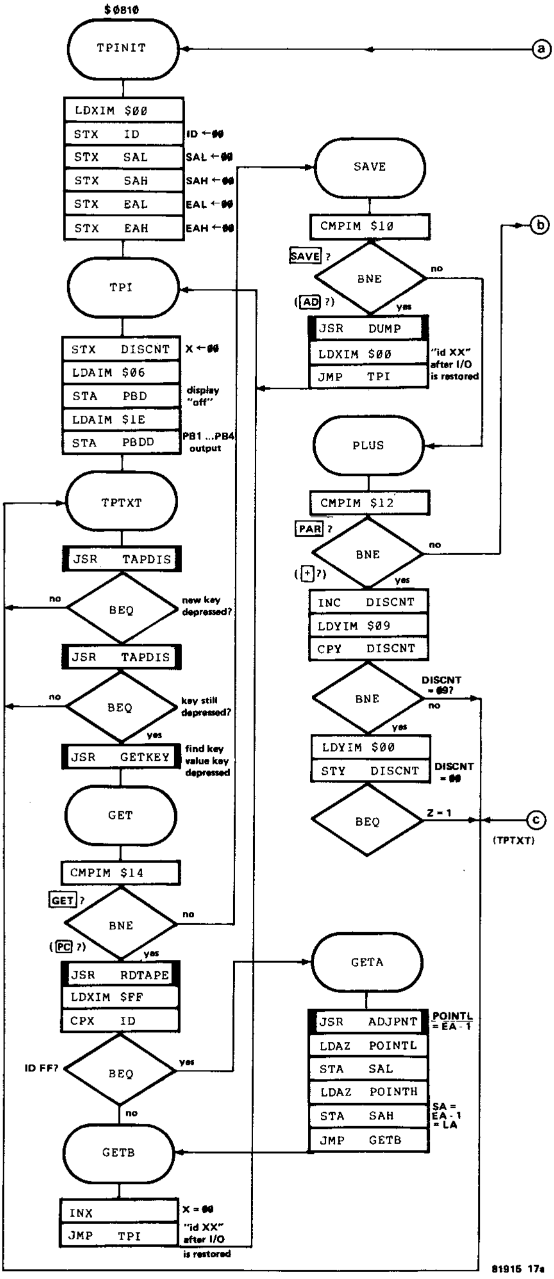


Figure 17a. The first section of the main Tape Monitor program.

17b

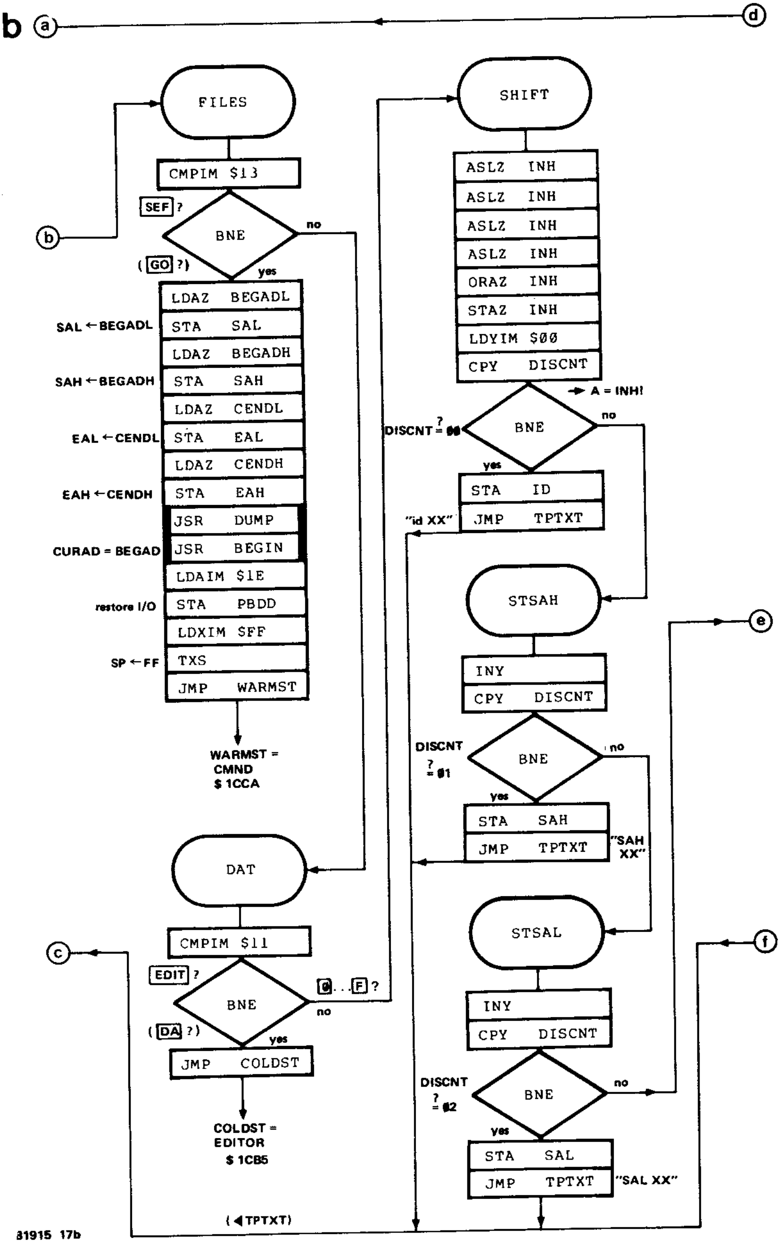


Figure 17b. The second section of the main Tape Monitor program.

17c

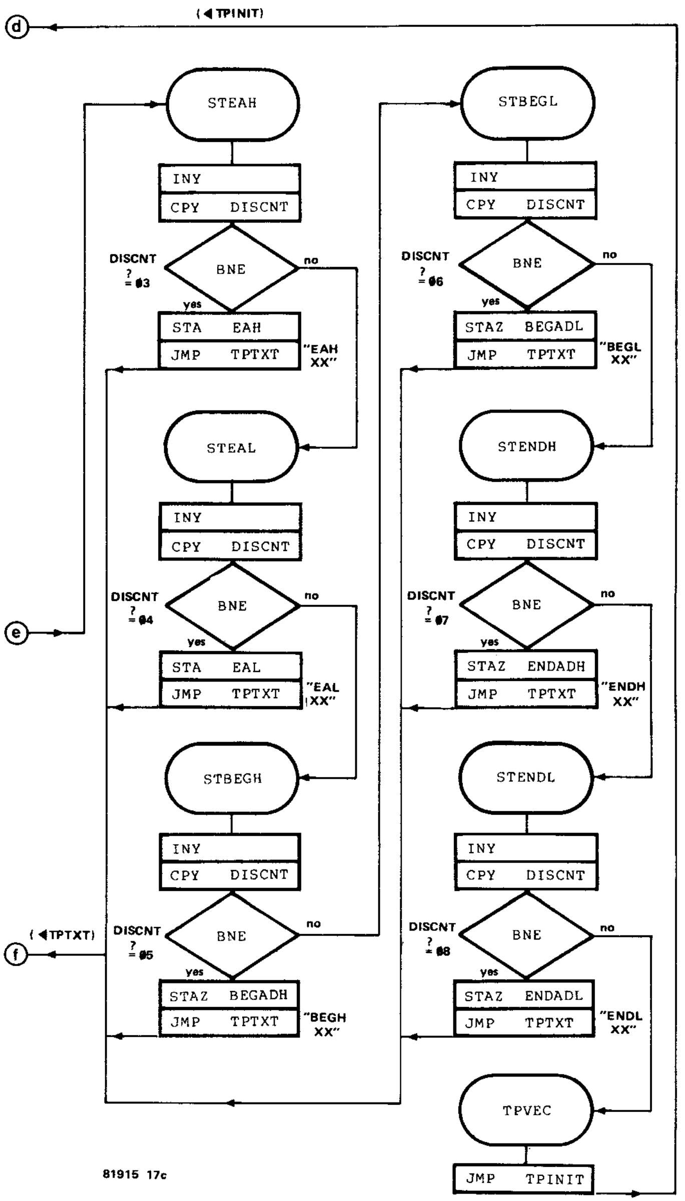


Figure 17c. The third and final section of the main Tape Monitor program.

fore, depending on the actual contents of location DISCNT (between ØØ and Ø8), one of the nine parameters will be displayed. Thus:
DISCNT = ØØ : display/alter ID
DISCNT = Ø1 : display/alter SAH
DISCNT = Ø2 : display/alter SAL
DISCNT = Ø3 . display/alter EAH
DISCNT = Ø4 : display/alter EAL
DISCNT = Ø5 : display/alter BEGADH
DISCNT = Ø6 : display/alter BEGADL
DISCNT = Ø7 : display/alter ENDADH
DISCNT = Ø8 : display/alter ENDADL
This gives us a general impression of what the Tape Monitor program actually does – now for the finer details.

21 **The TM program initialisation routine.** The main TM routine starts by loading the value $ØØ into the five locations associated with the cassette procedure; ID, SAL, SAH, EAL and EAH. Following the label TPI the parameter counter location DISCNT is also loaded with the value $ØØ. This point can be reached from three different routes and in each case the value $ØØ is held in the X index register. Therefore, whenever the program is executed from the TPI label onwards, the message 'id xx' will appear on the display.
The next part of the procedure is to load port B data register with the value $Ø6 and port B data direction register with the value $1E. This means that port lines PB1 . . . PB4 are programmed as outputs and that the display is switched off. Following the TPTXT label the processor calls the TAPDIS subroutine twice in succession. This routine determines which of the nine messages are to be displayed. This routine is described in point 27 and is illustrated in figure 19. The GETKEY subroutine is then executed to obtain the value of a depressed key. Therefore, by the time the processor reaches the label GET, the value of the depressed key will be held in the accumulator. Depending on which key was depressed, either a specific key routine will be executed or the data key routine will be run.

22 **The GET key routine** consists of all the instructions following the GET label at the bottom left-hand side of figure 17a. When the computer detects the fact that the GET key has been depressed, the RDTAPE subroutine is called. This routine was described in detail in points 10 . . . 19. All the data that is received from cassette is stored in memory. Following the RDTAPE subroutine, the computer checks to see whether the ID of the data block was FF. If not, the processor proceeds to label GETB where the contents of the X index register are made zero and the processor returns to the central label TPTXT, via the label TPI. The message 'id xx' then appears on the display. The slight detour by way of the TPI label was necessary in order to correct the input/output situation. The input/output status was changed during the RDTAPE routine.
Where the entered ID was FF, the processor carries out the instructions after the GETA label before reaching the GETB label. The first of these instructions call the subroutine ADJPNT, illustrated in figure 18, which

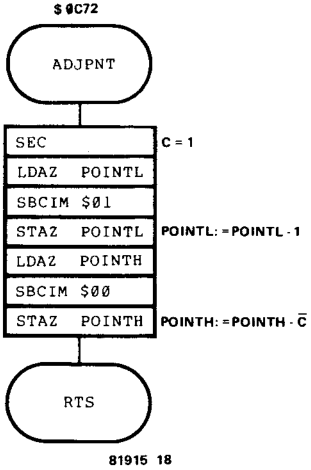


Figure 18. The ADJPNT subroutine decrements the contents of the address buffer POINT and is called during the execution of the GET key routine.

performs the simple operation of decrementing the contents of address buffer POINT. The reason for this is that location POINT indicates the address of the most recent data byte to be read from tape during the RDTAPE routine. At the end of the RDTAPE routine the contents of POINT are incremented by one. Therefore, when the complete data block has been read in to memory, location POINT indicates an address which is one higher than the address of the last byte in the data block. In other words: POINT = EA = LA + 1.

The end address is not directly defined on cassette, but is deduced from the start address (either on tape or entered manually) plus the number of bytes in the data block plus one.

Upon the return from the ADJPNT subroutine the contents of location POINT indicate the final address (LA) of the data block that has just been read in. And for a very good reason. The four instructions following the jump to the ADJPNT subroutine instruction cause a new start address to be defined. This applies when a number of data blocks belonging to unassembled sections of the same program have to be 'glued' together, so that the end-of-file characters have to be removed. The trick of entering an ID of FF was discussed in some length in chapter 11 of Book 3. It may be an idea to take another look at figure 11 of that chapter as well.

From the software of the GET key routine it can be seen that no data needs to be entered before the next data block is read in to memory. After all, the start address has been defined automatically and the contents of location ID will still be FF. All that has to be done before depressing the GET key is to make sure that the next data block on the cassette tape is the correct one!

23 **The SAVE key routine** consists of all (!) the instructions following the label SAVE in figure 17a. When the computer detects that the SAVE key has been depressed, the DUMP subroutine is called (see points 1 . . . 9). A previously defined data block is then stored on cassette. Beforehand memory locations ID, SAH, SAL, EAH and EAL must have been loaded with the correct information.

At the end of the DUMP subroutine the contents of the X index register are cleared. This means that the message 'id xx' will be displayed at the end of the procedure, when the processor reaches the central label TPTXT via the TPI label. As in the GET key routine, the section of program after the TPI label has to be executed to restore the input/output parameters.

24 **The PAR key routine** consists of all the instructions following the PLUS label in figure 17a. When the PAR key is depressed, the contents of the parameter counter DISCNT are incremented by one. If the result turns out to be $Ø9, the value $ØØ is loaded into location DISCNT. Therefore, depressing the PAR key simply causes the contents of the next of the nine parameter locations to be displayed. The order in which the nine parameters are presented was given in point 20. After the message 'ENDLxx' and a subsequent PAR key operation, the message 'id xx' is displayed.

25 **The SEF key routine** consists of all the instructions following the FILES label in figure 17b. When the SEF key is depressed, the contents of location BEGAD are stored in location SA (start address) and the contents of CEND are copied into location EA (end address). CEND is, of course, the current end address of the (as yet) unassembled program which is to be stored on cassette. Obviously, the ID must be defined before the SEF key is operated.

After the DUMP routine has been executed the Tape Monitor program is exited from and a warm start entry is made into the editor mode. However, before this can happen, the BEGIN subroutine is executed: the current address pointer CURAD is loaded with the contents of location BEGAD which also acts as the start address. Furthermore, port B data direction register is updated. Finally, the contents of the stack pointer are set to $FF, after which the processor makes a warm start entry into the editor.

The EDIT key routine consists of the instructions after the DAT label in figure 17b. All that happens when the EDIT key is depressed is that the processor makes a cold start entry into the editor.

26 **The data key routine.** By this time all the non-hexadecimal keys will have been filtered out, which leaves us with the data keys Ø . . . F. This involves the remainder of the main Tape Monitor program – the data key routine. This consists of all the instructions following the SHIFT label in figure 17b and overflows into figure 17c.

A hexadecimal key is processed into the new low order nibble of the data buffer INH. The previous low order nibble becomes the new high order nibble. There should be no need to go into too much detail here as all this has been described before.

The modified contents of the buffer INH must be stored in one of the nine parameter locations. Which one depends on the contents of the parameter counter DISCNT. The data is stored in location

ID	(label SHIFT)	if DISCNT = ØØ
SAH	(label STSAH)	if DISCNT = Ø1
SAL	(label STSAL)	if DISCNT = Ø2
EAH	(label STEAH)	if DISCNT = Ø3
EAL	(label STEAL)	if DISCNT = Ø4
BEGADH	(label STBEGH)	if DISCNT = Ø5
BEGADL	(label STBEGL)	if DISCNT = Ø6
ENDADH	(label STENDH)	if DISCNT = Ø7
ENDADL	(label STENDL)	if DISCNT = Ø8

Once the particular parameter has been entered, the processor returns to the central label TPTXT to display the result.

This just about ends the discussion of the main Tape Monitor program. Only one subroutine and one sub-subroutine still need to be mentioned.

27 **The TAPDIS subroutine** is illustrated in figure 19. This subroutine has the job of making sure that one of the nine parameter messages appears on the display. Displays 1 . . . 4 indicate the parameter location while displays 5 and 6 show its contents. We will briefly recap on part of chapter 7 (Book 2) to see that the next display to be lit depends on the value contained in the X index register:

when X = Ø8 display 1 will be enabled
when X = ØA display 2 will be enabled
when X = ØC display 3 will be enabled
when X = ØE display 4 will be enabled
when X = 1Ø display 5 will be enabled and
when X = 12 display 6 will be enabled

Immediately after the TAPDIS label in figure 19, port lines PAØ . . . PA6 are programmed as outputs. These port lines will contain the segment data to be displayed. Following the label SID the value $Ø8 is loaded into the X index register. This means that display 1 is enabled (during the TDISP routine which follows shortly). However, before the TDISP routine is called, the computer must know which location to display. This in turn depends on the contents of memory location DISCNT. A form of 'interrogation', similar to that for the key routines, is then carried out to determine the value held in location DISCNT. The Y index register is used as the 'interrogator' initially, but when the computer has discovered which memory location is to be displayed, the Y index register assumes a different role. It then serves as an index for the look-up table which is used during the TDISP routine.

When the label COMPNT in figure 19 is reached (this is when the TDISP subroutine is called), the following situation arises:

DISCNT = ØØ; contents of ID	stored in INH; index = ØØ
DISCNT = Ø1, contents of SAH	stored in INH; index = Ø4
DISCNT = Ø2; contents of SAL	stored in INH; index = Ø8
DISCNT = Ø3; contents of EAH	stored in INH; index = ØC
DISCNT = Ø4; contents of EAL	stored in INH; index = 1Ø
DISCNT = Ø5, contents of BEGADH	stored in INH; index = 14
DISCNT = Ø6; contents of BEGADL	stored in INH: index = 18
DISCNT = Ø7, contents of ENDADH	stored in INH; index = 1C
DISCNT = Ø8; contents of ENDADL	stored in INH; index = 2Ø

19

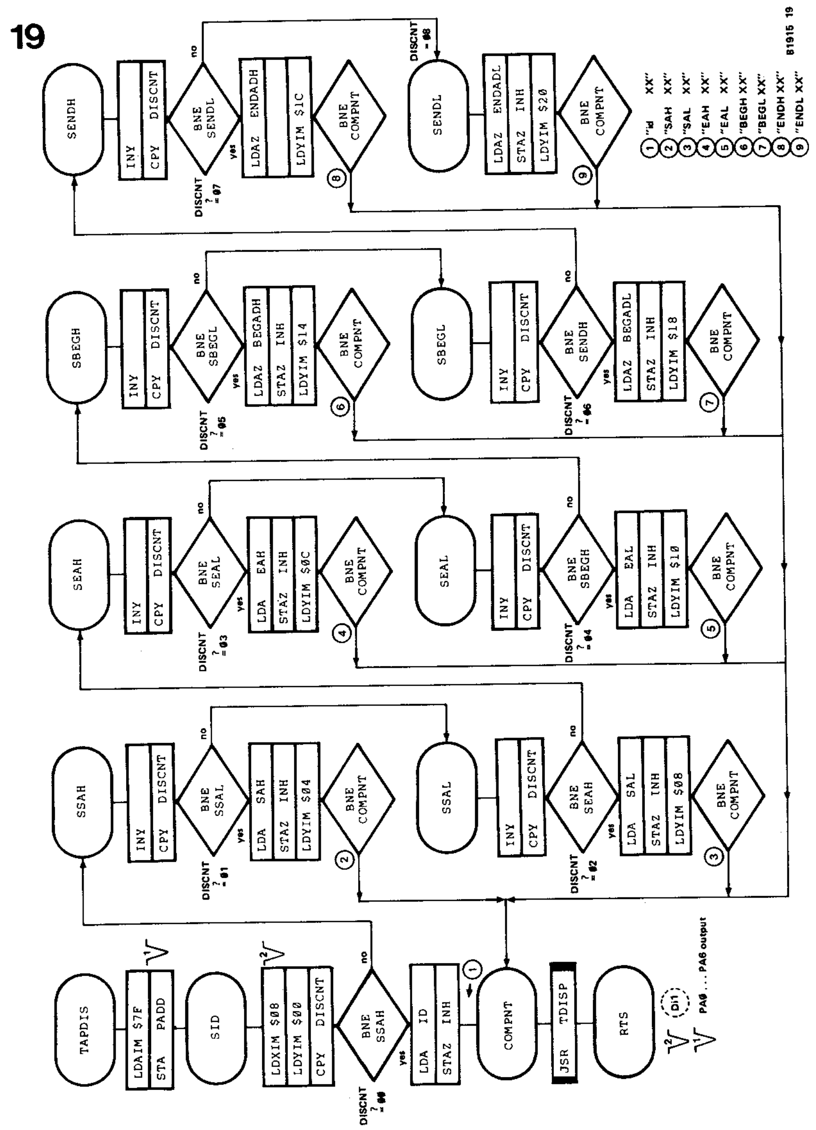


Figure 19. The TAPDIS subroutine establishes which of the nine parameter locations are to be displayed and/or modified.

28 **The TDISP subroutine** is illustrated in figure 20 and has the task of displaying the name of one of the nine parameter locations along with the data held therein. In addition, the routine checks whether another key was depressed or not.

20

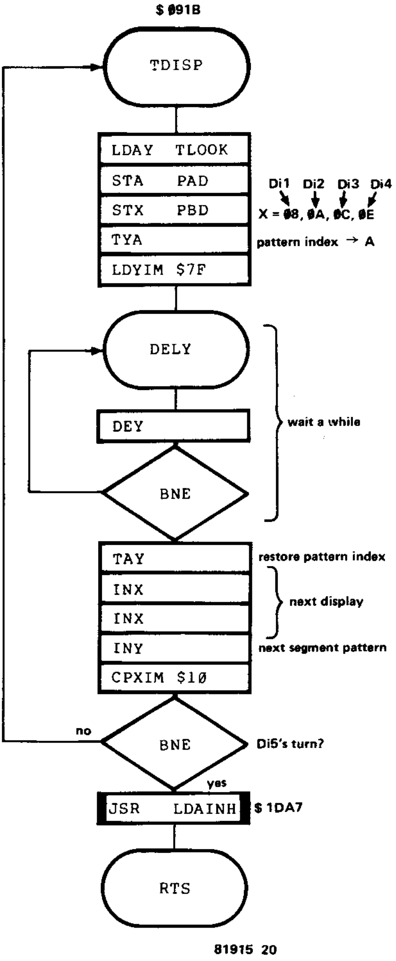


Figure 20. The TDISP subroutine is used to enable each of the displays in turn and also checks to see whether a new key has been depressed.

One execution of the program loop starting at the TDISP label can be compared with the CONVD subroutine in the original monitor program (see figure 14 in chapter 7 in Book 2). First of all, the accumulator is loaded with data derived from the look-up table TLOOK. The data selected depends on the current contents of the Y index register. The data from the look-up table acts as the segment code and is therefore stored in port A data register. The X index register is used to determine which particular display is to be enabled and its contents, therefore, are transferred to port B data register.

21

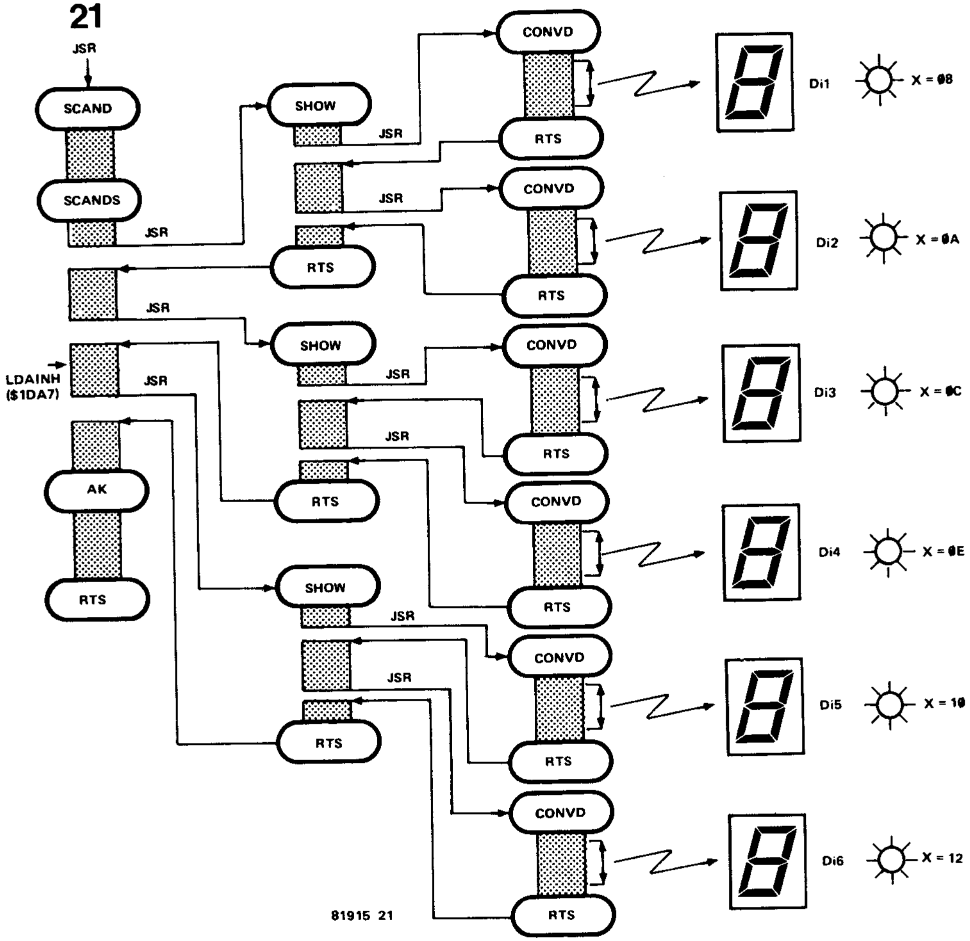


Figure 21. The LDAINH subroutine is part of the subroutine SCAND/SCANDS contained in the original monitor program.

After saving the contents of the Y index register in the accumulator (the instruction TYA), the Y index register is used for a short delay loop. The length of the delay determines for how long each display is actually lit. After the delay the previous contents of the Y index register are restored whereupon the value in the X index register is incremented twice to enable the next display and the value in the Y index register is incremented by one to index the next segment code to be loaded from the look-up table. When the value held in the X index register is $1Ø (it is time to enable display 5), the processor terminates the loop and calls the LDAINH subroutine.

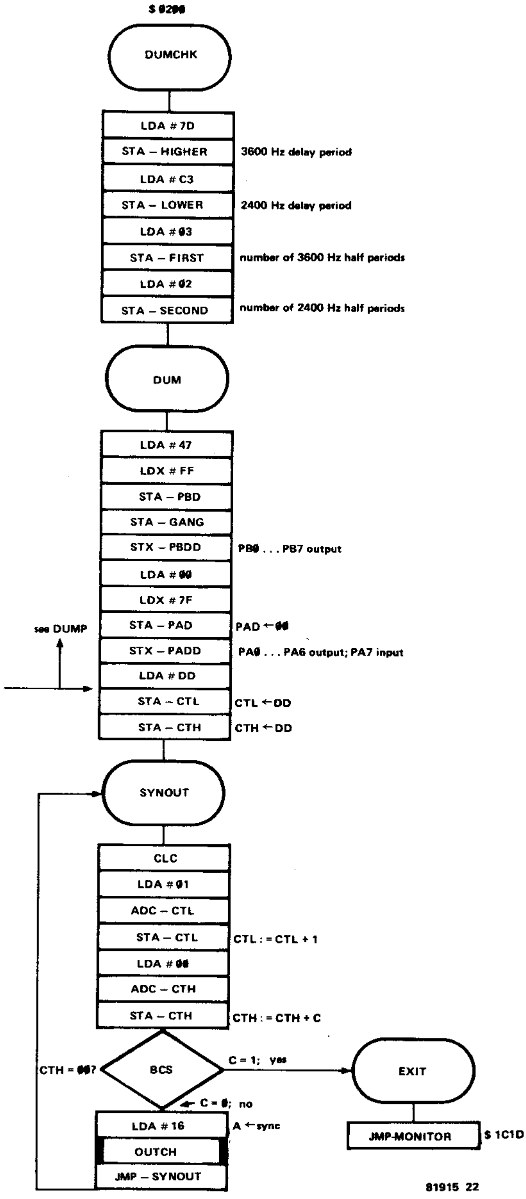


Figure 22. The DUMCHK routine can be used to record a series of synchronisation characters on to tape so that the PLL can be calibrated correctly.

The LDAINH subroutine is new, but not totally unfamiliar (see figure 12 on page 122 of Book 2 and address $1DA7 in the listing at the back of this book). As can be seen, the processor jumps to the middle of the SCAND/SCANDS subroutine in the original monitor program. From figure 21 (which is identical to figure 11 in chapter 7), it is apparent that the SCAND/SCANDS subroutine is called when the contents of the data buffer INH have to be displayed.

Once the subroutine LDAINH (alias SCAND/SCANDS) has been executed, the contents of the accumulator will reveal whether or not another key has been depressed. This comprises the final section of the routine in figure 21 following the label AK.

This brings us to the end of the discussion on the Tape Monitor program and almost to the end of this chapter.

29 In chapter 11 of Book 3 we described two methods for calibrating the phase-locked loop (PLL) of the cassette interface. In both instances a test program is required. If the PLL is calibrated with the aid of the display, both a read and a write operation are required. If, on the other hand, an oscilloscope is used, data need only be stored on cassette beforehand. To recap on the details, see pages 101 . . . 108 in chapter 11 of Book 3.

When using the display to calibrate the PLL, a series of un-interrupted synchronisation characters have to be written onto the cassette tape. This is accomplished with the aid of the DUMCHK program given in figure 22. Readers have already seen the hex dump for this in the first half of table 2 on page 102 of Book 3. The start address is Ø2ØØ.

The first few instructions of the DUMCHK routine are identical to those of the DUMP routine (see figure 1a). It is not until three instructions before the SYNOUT label that the routine differs. Here, the locations CTL and CTH are both loaded with the value $DD. The 16 bit figure contained in locations CTH (high order byte) and CTL (low order byte) is incremented each time the processor has executed the program loop around the label SYNOUT. As long as the result of the addition does not generate a carry, the computer continues to produce synchronisation characters (the instructions LDA # 16, JSR - OUTCH and JMP - SYNOUT). However, should the addition generate a carry, the computer will branch to the EXIT label from whence it will return to the original monitor program. The start address (Ø2ØØ) will then appear on the display.

The carry flag is set when the contents of memory location CTH become zero; in other words, after the addition when both CTH and CTL contain the value $FF. Just before the processor reaches the label SYNOUT for the first time, locations CTH and CTL were both loaded with the value $DD, giving a 16 bit figure of $DDDD. This means that a total of

$1ØØØØ - $DDDD = $2223 = 8739 (decimal)

synchronisation characters are generated. As each character contains 8 bits and each bit lasts 1250 µs (see point 3) the total duration of the recording is 1250 x 8 x 8738 µs = 87 seconds, or 1½ minutes.

As would be expected, the opposite of the DUMCHK routine has the task of reading the synchronisation characters from the cassette and is called RDCHK. This subroutine is shown in figure 23. Again, the hex dump can

23

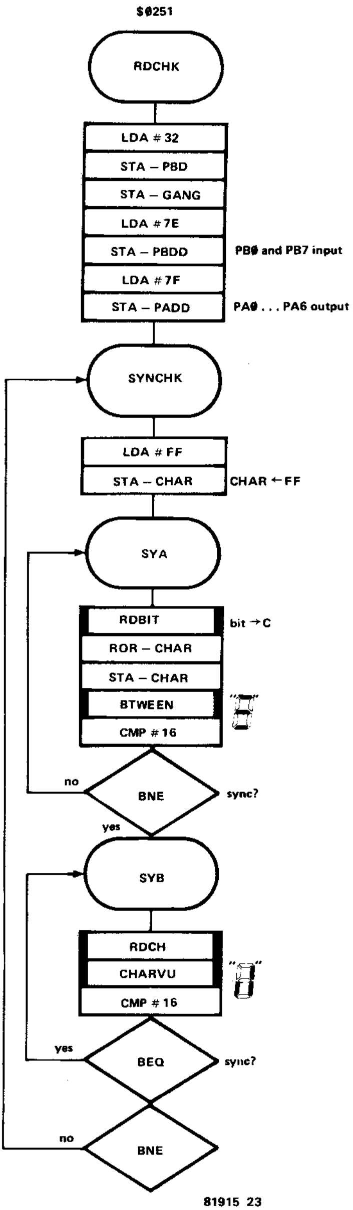
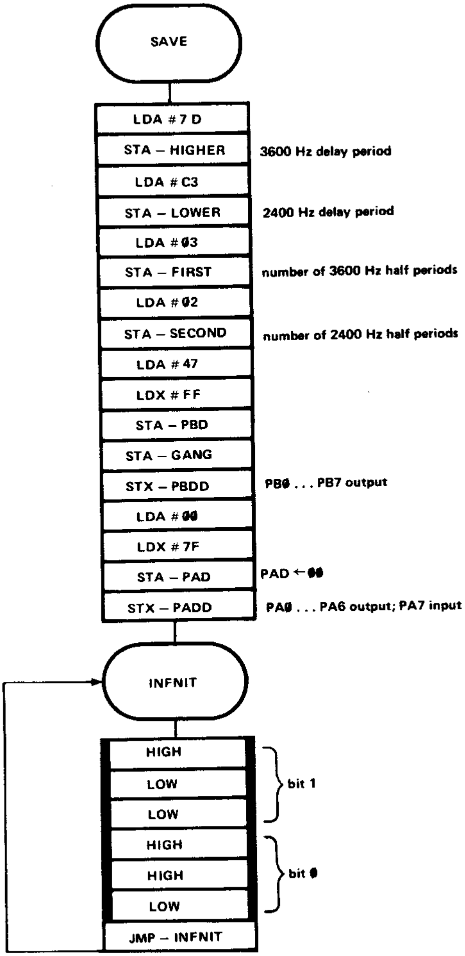


Figure 23. The RDCHK routine can be used to read a series of synchronisation characters from tape so that the PLL can be set up accurately.

24



81915 24

Figure 24. The SAVE routine transfers a continuous stream of alternating ones and zeros to cassette so that the PLL can be set up using an oscilloscope.

be found on page 102 of Book 3. The first few instructions, between the labels RDCHK and SYB are virtually identical to those of the RDTAPE subroutine between the labels RDTAPE and TENSYN. Therefore, let us move straight on to the instructions following the SYB label. A character is read in by means of the RDCH subroutine and the ASCII code for that character is held in the accumulator. Next, the computer checks to see whether or not it was a synchronisation character. If so, the processor returns to the SYB label. If not, the processor returns to the label SYNCHK.

The programs given in figures 22 and 23 are intended purely for calibrating the PLL by way of the display. The question is: what can be seen and when? This depends on two instructions in figure 23. Immediately after the return from the BTWEEN subroutine the 'search' pattern, three horizontal bars, appears on the display. After the CHARVU subroutine has been executed, the ASCII code for the synchronisation character is inverted and passed on to the segments of the display. The indication that the computer is reading synchronisation characters will remain on the display for as long as synchronisation characters continue to be read, provided of course that the PLL has been correctly calibrated. If any other pattern appears on the display, then there could be a gap on the tape where no synchronisation characters have been recorded, or the PLL is incorrectly adjusted. However, this will only happen for a very brief period indeed before the display reverts to the search pattern or the synchronisation pattern.

30 The SAVE routine in figure 24 is part of the auxiliary software required when the PLL is calibrated with the aid of an oscilloscope. Again, the hex dump has already been given on page 103 of Book 3.

The SAVE routine transmits a series of alternating ones and zeros. Not surprisingly, the instructions between the labels SAVE and INFNIT are identical to those of the DUMP/DUMPT routine. Also, there is an infinite program loop around the label INFNIT, as a result of which a bit train consisting of alternate ones and zeros is generated. The latter is simply a question of calling the HIGH and LOW subroutines the right number of times and in the right order. The infinite loop can, of course, be exited from if the RST key is depressed.

Appendix 1

The complete listing of the PME system program

The following pages contain the complete listing of the PME system program. Each book page corresponds to a single assembler page consisting of 56 lines.
Assembler page 12 contains the instructions belonging to the BINAR and PMBINA routines which are required to switch back to binary arithmetic. See appendix 4 for further details.

A few addresses are well worth noting:

$15ØØ :	EDITC
$1533 :	BRK
$153D :	EDITW
$1667 :	SEMIW
$17C5 :	SEACND
$17F6 :	BINAR
$17FA:	PMBINA

```
14F8                   ORG   $14F8


                SOURCE LISTING OF THE PM EDITOR


                WRITTEN BY G. NACHBAR

                DATE : 25 JUNE 1981


                THE PM EDITOR USES HEXADECIMAL LABELS

                ******************************
                POINTERS AND TEMPS IN PAGE ZERO
                ******************************

14F8            BEGADL *      $00E2
14F8            BEGADH *      $00E3
14F8            ENDADL *      $00E4
14F8            ENDADH *      $00E5
14F8            CURADL *      $00E6
14F8            CURADH *      $00E7
14F8            CENDL  *      $00E8
14F8            CENDH  *      $00E9
14F8            TABLEL *      $00EC
14F8            TABLEH *      $00ED
14F8            LABELS *      $00EE
14F8            BYTES  *      $00F6
14F8            COUNT  *      $00F7
14F8            INH    *      $00F9
14F8            POINTL *      $00FA
14F8            POINTH *      $00FB
14F8            TEMPX  *      $00FD
14F8            NIBBLE *      $00FE

                ***********************************
                PME'S POINTERS AND TEMPS IN PAGE 1A
                ***********************************

14F8            PARAL  *      $1A63
14F8            PARAH  *      $1A64
14F8            PARBL  *      $1A65
14F8            PARBH  *      $1A66
14F8            NMIL   *      $1A7A
14F8            NMIH   *      $1A7B
14F8            BRKTL  *      $1A7C
14F8            BRKTH  *      $1A7D
14F8            PBDD   *      $1A83


                ******************************
                ADDRESSES IN THE STANDARD EPROM
                ******************************


14F8            SAVE   *      $1C00
14F8            BEGIN  *      $1ED3
14F8            OPLEN  *      $1E5C
14F8            LENACC *      $1E60
14F8            ADCEND *      $1EDC
14F8            RECEND *      $1EEA
14F8            UP     *      $1E83
14F8            FILLWS *      $1E47
14F8            NEXT   *      $1EF8
14F8            ASSEMB *      $1F51


```

```
                *****************
                SUBROUTINES IN PM
                *****************


14F8            INPAR  *      $1387
14F8            RECCHA *      $12AE
14F8            PRCHA  *      $1334
14F8            PRBYT  *      $128F
14F8            PRSP   *      $11F3
14F8            ASHETT *      $141E
14F8            CRLF   *      $11E8
14F8            RESIN  *      $1268
14F8            RESTTY *      $14BC

14F8            STEP   *      $14CF



                IOCORR: ADAPTS THE PBDD TO THE VERSION D SITUATION,
                PRIOR TO ASSEMBLING

14F8 A9 1E      IOCORR LDAIM $1E
14FA 8D 83 1A          STA   PBDD   PB0 IS INPUT
14FD 4C 51 1F          JMP   ASSEMB ASSEMBLE THE PRCGRAM

                EDITC: COLD START ENTRY OF PME

1500 20 68 12   EDITC  JSR   RESIN  RESET INPUT BUFFERS
1503 A0 00             LDYIM $00
1505 20 3E 17          JSR   MESSA  "BEGAD,ENDAD:"
1508 20 87 13          JSR   INPAR  GET PARAMETERS
150B 30 F3             BMI   EDITC  TRY IT AGAIN, IF NOT DONE PROPERLY
150D AD 63 1A          LDA   PARAL  LOAD BEGADL
1510 85 E2             STAZ  BEGADL
1512 18                CLC
1513 69 01             ADCIM $01
1515 85 E8             STAZ  CENDL  CENDL = BEGADL+1
1517 AD 64 1A          LDA   PARAH
151A 85 E3             STAZ  BEGADH LOAD BEGADH
151C 69 00             ADCIM $00
151E 85 E9             STAZ  CENDH  CENDH = BEGADH+AARRY
1520 AD 65 1A          LDA   PARBL
1523 85 E4             STAZ  ENDADL LOAD ENDAD
1525 AD 66 1A          LDA   PARBH
1528 85 E5             STAZ  ENDADH LOAD ENDADH
152A 20 D3 1E          JSR   BEGIN  PRINT POINTER CURAD
152D A9 77             LDAIM $77    EQUALS BEGAD
152F A0 00             LDYIM $00
1531 91 E6             STAIY CURADL EOF 77 ON FIRST ADDRESS BEGAD

                BRK: WARM START ENTRY OF PME, INCLUDING THE SPECIFI-
                CATION OF THE BRK JUMP VECTOR

1533 A9 3D      BRK    LDAIM $3D
1535 8D 7C 1A          STA   BRKTL
1538 A9 15             LDAIM $15
153A 8D 7D 1A          STA   BRKTH

                EDITW: MAY BE USED AS A WARM START ENTRY OF PME
                PROVIDED THE BRK JUMP VECTOR HAS BEEN SPECIFIED
                BEFOREHAND.
153D A0 24      EDITW  LDYIM $24
153F 20 3E 17          JSR   MESSA  "PM EDITOR"
1542 A2 FF             LDXIM $FF
1544 9A                TXS          RESET STACK POINTER

                THE CENTRAL LABEL "WARM" OF PME STARTS STARTS
                WITH THE K-KEY ROUTINE
```

```
               ("DELETE")

1545 20 0B 17  WARM   JSR    PRINS  PRINT CURRENT INSTRUCTION
1548 20 AE 12         JSR    RECCHA WAIT FOR A DEPRESSED KEY
154B C9 4B            CMPIM 'K      IS IT THE K-KEY?
154D D0 09            BNE    LIST   CONTINUE IF NOT
154F 20 83 1E         JSR    UP     ADAPT WORKSPACE
1552 20 EA 1E         JSR    RECEND ADAPT CURRENT END ADDRESS POINTER
1555 4C 45 15         JMP    WARM   READY
               LABEL LIST: L-KEY ROUTINE

1558 C9 4C     LIST   CMPIM 'L      L-KEY DEPRESSED?
155A D0 12            BNE    SKIP   IF NOT CONTINUE
155C 20 D3 1E         JSR    BEGIN  START AT BEGAD
155F 20 0B 17  LST    JSR    PRINS  PRINT CURRENT INSTRUCTION
1562 20 F8 1E         JSR    NEXT   ADAPT INSTRUCTION POINTER
1565 30 F8            BMI    LST
1567 A0 1F     DONE   LDYIM $1F
1569 20 3E 17  MESS   JSR    MESSA  PRINT "DONE"
156C F0 D7            BEQ    WARM   READY

               LABEL SKIP: SPACE BAR KEY ROUTINE
               LABEL INSERT: I-KEY ROUTINE

156E C9 20     SKIP   CMPIM '        SPACE BAR DEPRESSED?
1570 D0 07            BNE    INSERT CONTINUE IF NOT
1572 20 F8 1E         JSR    NEXT   ADAPT INSTRUCTION POINTER
1575 30 CE            BMI    WARM   READY
1577 10 EE            BPL    DONE   PRINT "DONE" IF NO FURTHER INSTR. AVAILABLE
1579 C9 49     INSERT CMPIM 'I      I-KEY DEPRESSED?
157B D0 17            BNE    SEARCH IF NOT CONTINUE
157D 20 C9 16         JSR    READIN READ NEW INSTRUCTION
1580 30 0A            BMI    ILLKEY PRINT "ILLEGAL KEY" IF NEW ISTR. ISN'T VALID
1582 20 F6 16  MEM    JSR    CHECK
1585 90 09            BCC    FULL   PRINT "FULL" IF MEMORY IS FULL
1587 20 47 1E         JSR    FILLWS LOAD NEW INSTR. IN MEMORY
158A F0 B9            BEQ    WARM   READY
158C A0 0E     ILLKEY LDYIM $0E
158E D0 D9            BNE    MESS   PRINT "ILLEGAL KEY"
1590 A0 1A     FULL   LDYIM $1A
1592 D0 D5            BNE    MESS   PRINT "FULL"

               LABEL SEARCH: S-KEY ROUTINE
               ANY BYTE PATTERN (INSTRUCTION) MAY BE SEARCHED FOR.
               IF THE INSTRUCTION SEARCHED FOR IS FOUND,
               ONE MAY SEARCH FOR THE SAME INSTRUCTION ON A HIGHER
               ADDRESS, BY PRESSING THE KEY "Y". THE SEARCHING PRO-
               CESS ALWAYS HAS TO BE CONCLUDED BY THE MESSAGE "DONE".
               THIS OCCURS EITHER BECAUSE ALL OR NO INSTRUCTIONS ARE
               FOUND, OR BY PRESSING AN ARBITRARY SOFTWARE-KEY (I.E.
               A KEY GENERATING AN ASCII CODE WHEN DEPRESSED),
               WITH THE EXEPTION OF THE 'Y'-KEY.
1594 C9 53     SEARCH CMPIM 'S      S-KEY DEPRESSED?
1596 D0 40            BNE    BACK   IF NOT CONTINUE
1598 20 C9 16         JSR    READIN READ INSTR. TO BE SEARCHED FOR
159B 30 EF            BMI    ILLKEY "ILLEGAL KEY" IF NOT PROPERLY DONE
159D 20 D3 1E         JSR    BEGIN  START AT BEGAD
15A0 A5 FB     SCAN   LDAZ   POINTH GET OPCODE OF SEARCH INSTR.
15A2 20 60 1E         JSR    LENACC GET LENGTH OF SEARCH INSTR.
15A5 A0 00            LDYIM $00
15A7 B1 E6            LDAIY CURADL GET OPCODE OF THE CURRENT INSTR.
15A9 C5 FB            CMPZ   POINTH COMPARE IT AGAINST OPCODE OF THE SEARCH INST.
15AB D0 20            BNE    AGAIN  IF NO MATCH THEN NEXT INSTR.
15AD C6 F6            DECZ   BYTES
15AF F0 12            BEQ    FOUND  CONTINUE, IF INSTR. LENGTH IS 2 OR 3
15B1 C8               INY
15B2 B1 E6            LDAIY CURADL GET NEXT BYTE IN MEMORY
15B4 C5 FA            CMPZ   POINTL COMP. IT AGAINST 1ST OPERAND BYTE
15B6 D0 15            BNE    AGAIN  IF NO MATCH NEXT INSTR.
```

```
0207: 15B8 C6 F6              DECZ  BYTES
0208: 15BA F0 07              BEQ   FOUND  CONTINUE IF LENGTH OF CURR. INSTR. IS 3
0209: 15BC C8                 INY
0210: 15BD B1 E6              LDAIY CURADL GET NEXT BYTE IN MEMORY
0211: 15BF C5 F9              CMPZ  INH    COMP. IT AGAINST 2ND OPERAND BYTE
0212: 15C1 D0 0A              BNE   AGAIN  IF NO MATCH NEXT INSTR.
0213: 15C3 20 0B 17  FOUND    JSR   PRINS  PRINT OUT THE SPECIFIED INSTRUCTION
0214: 15C6 20 AE 12           JSR   RECCHA WAIT FOR A DEPRESSED KEY
0215: 15C9 C9 59              CMPIM 'Y     Y-KEY DEPRESSED?
0216: 15CB D0 08              BNE   DNE    IF NOT PRINT "DONE"
0217: 15CD 20 5C 1E  AGAIN    JSR   OPLEN  GET LENGTH OF THE CURR. INSTR.
0218: 15D0 20 F8 1E           JSR   NEXT   ADAPT INSTR. POINTER
0219: 15D3 30 CB              BMI   SCAN   CONTINUE SEARCH IF STILL INSTR. AVAILABLE
0220: 15D5 4C 67 15  DNE      JMP   DONE   ELSE PRINT "DONE"
0221:
0222:                LABEL BACK: Z-KEY ROUTINE
0223:
0224: 15D8 C9 5A     BACK     CMPIM 'Z     Z-KEY DEPRESSED?
0225: 15DA D0 30              BNE   TOF    IF NOT CONTINUE
0226: 15DC A5 E2              LDAZ  BEGADL
0227: 15DE 85 EC              STAZ  TABLEL
0228: 15E0 A5 E3              LDAZ  BEGADH
0229: 15E2 85 ED              STAZ  TABLEH TABLE:=BEGAD
0230: 15E4 C5 E7              CMPZ  CURADH  TABLEH:=CURADH?
0231: 15E6 D0 08              BNE   BCKA   IF NOT INCREASE TABLE
0232: 15E8 A5 E2              LDAZ  BEGADL
0233: 15EA C5 E6              CMPZ  CURADL TABLEL:=CURADL?
0234: 15EC D0 02              BNE   BCKA   IF NOT INCREASE TABLE
0235: 15EE F0 19              BEQ   BCKB   ELSE PRINT FIRST INSTRUCTION
0236: 15F0 A0 00     BCKA     LDYIM $00
0237: 15F2 B1 EC              LDAIY TABLEL GET LENGTH OF INSTR. AT
0238: 15F4 20 60 1E           JSR   LENACC WHICH POINT IS POINTED
0239: 15F7 20 A0 16           JSR   INCTAB INCREASE POINT BY BYTES
0240: 15FA A5 E6              LDAZ  CURADL
0241: 15FC C5 EC              CMPZ  TABLEL TABLEL:=CURADL?
0242: 15FE D0 F0              BNE   BCKA   IF NOT INCREASE TABLE
0243: 1600 A5 E7              LDAZ  CURADH
0244: 1602 C5 ED              CMPZ  TABLEH TABLEH:=CURADH
0245: 1604 D0 EA              BNE   BCKA   IF NOT INCREASE TABLE
0246: 1606 20 92 16           JSR   DECURA DECREASE INSTRUCTION POINTER BY BYTES
0247: 1609 4C 45 15  BCKB     JMP   WARM
0248:
0249:                LABEL TOF: T-KEY ROUTINE
0250:                LABEL SXTEEN: P-KEY ROUTINE
0251:
0252: 160C C9 54     TOF      CMPIM 'T     T-KEY DEPRESSED?
0253: 160E D0 06              BNE   SXTEEN IF NOT CONTINUE
0254: 1610 20 D3 1E           JSR   BEGIN  INSTR. POINTER = BEGAD
0255: 1613 4C 45 15  TOFEND   JMP   WARM   READY
0256: 1616 C9 50     SXTEEN   CMPIM 'P     P-KEY DEPRESSED?
0257: 1618 D0 1C              BNE   ASMBLR IF NOT CONTINUE
0258: 161A A9 0F              LDAIM $0F    15 INSTR. TO BE PRINTED
0259: 161C 85 F7              STAZ  COUNT
0260: 161E 20 5C 1E  LINES    JSR   OPLEN  GET LENGTH OF CURRENT INSTR.
0261: 1621 20 F8 1E           JSR   NEXT   ADAPT INSTR. POINTER
0262: 1624 C6 F7              DECZ  COUNT
0263: 1626 F0 EB              BEQ   TOFEND READY IF 15 INSTR. HAVE BEEN PRINTED
0264: 1628 A0 00              LDYIM $00
0265: 162A B1 E6              LDAIY CURADL
0266: 162C C9 77              CMPIM $77
0267: 162E F0 E3              BEQ   TOFEND READY IF OPCODE IS 77
0268: 1630 20 0B 17           JSR   PRINS  PRINT INSTRUCTION
0269: 1633 4C 1E 16           JMP   LINES  NEXT INSTRUCTION
0270:                LABEL ASMBLR: X-KEY ROUTINE
0271:                LABEL ASSEND: RESTORE I/O AFTER ASSEMBLING
0272:
0273: 1636 C9 58     ASMBLR   CMPIM 'X     X-KEY DEPRESSED?
0274: 1638 D0 13              BNE   INPUT  IF NOT CONTINUE
0275: 163A A9 47              LDAIM ASSEND SPECIFY NMI JUMP VECTOR
```

```
0276: 163C 8D 7A 1A          STA   NMIL
0277: 163F A9 16             LDAIM ASSEND /256
0278: 1641 8D 7B 1A          STA   NMIH
0279: 1644 4C F8 14          JMP   IOCORR PREPARE I/O PRIOR TO ASSEMBLING
0280: 1647 20 BC 14   ASSEND JSR   RESTTY RESTORE I/O: PM SITUATION
0281: 164A 4C 88 17          JMP   LABLST LIST THE HEXADECIMAL LABELS
0282:
0283:                 LABEL INPUT: I-KEY ROUTINE
0284:                 NOTE: NO NON-NUMERICAL KEY HAS TO BE DEPRESSED
0285:                 PME AUTOMATICALLY ASSUMES THE INPUT KEY FUNCTION
0286:                 HAS BEEN OPTED FOR AS SOON AS THE DATA BELONGING
0287:                 TO THE NEW INSTR. HAS BEEN SPECIFIED
0288:
0289: 164D 20 B1 16   INPUT  JSR   BYT    GET THE 2ND NIBBLE OF THE 1ST BYTE
0290: 1650 30 12            BMI   WRONG
0291: 1652 20 CE 16          JSR   READ   GET THE OTHER BYTE(S)
0292: 1655 30 0D             BMI   WRONG
0293: 1657 20 5C 1E          JSR   OPLEN  GET LENGTH OF THE CURRRENT INSTR.
0294: 165A 20 F8 1E          JSR   NEXT   ADAPT INSTR. POINTER
0295: 165D A5 FD             LDAZ  TEMPX  GET LENGTH OF NEW INSTR.
0296: 165F 85 F6             STAZ  BYTES  AND STORE IT IN BYTES
0297: 1661 4C 82 15          JMP   MEM    LOAD MEMORY WITH NEW INSTRUCTION
0298: 1664 4C 8C 15   WRONG  JMP   ILLKEY PRINT "ILLEGAL KEY"
0299:
0300:                 END OF THE PME MAIN ROUTINE
0301:
0302:
0303:
0304:                 SEMI WARM START ("LUKE WARM") ENTRY OF PME
0305:                 FOLLOWING THE SPECIFICATION BY THE USER
0306:                 OF BEGAD AND ENDAD THE INSTRUCTION POINTER
0307:                 CURAD IS POINTED AT BEGAD AND THE CURRENT
0308:                 END ADDRESS POINTER CEND IS POINTED AT ENDAD.
0309:                 AFTER THIS PME IS ENTERED BY THE WARM START ENTRY
0310:
0311:
0312: 1667 20 68 12   SEMIW  JSR   RESIN  RESET INPUT BUFFERS
0313: 166A A0 00            LDYIM $00
0314: 166C 20 3E 17          JSR   MESSA  "BEGAD,ENDAD:"
0315: 166F 20 87 13          JSR   INPAR  GET BEGAD AND ENDAD
0316: 1672 30 F3             BMI   SEMIW  TRY IT AGAIN IF NOT DONE PROPERLY
0317: 1674 AD 63 1A          LDA   PARAL
0318: 1677 85 E2             STAZ  BEGADL
0319: 1679 AD 64 1A          LDA   PARAH
0320: 167C 85 E3             STAZ  BEGADH
0321: 167E AD 65 1A          LDA   PARBL
0322: 1681 85 E4             STAZ  ENDADL
0323: 1683 85 E8             STAZ  CENDL
0324: 1685 AD 66 1A          LDA   PARBH
0325: 1688 85 E5             STAZ  ENDADH
0326: 168A 85 E9             STAZ  CENDH  CEND = ENDAD
0327: 168C 20 D3 1E          JSR   BEGIN  CURAD = BEGAD
0328: 168F 4C 33 15          JMP   BRK    JUMP TO WARM START
0329:
0330:
0331:
0332:                 *****************
0333:                 SUBROUTINES OF PME
0334:                 *****************
0335:
0336:
0337:                 DECURA
0338:                 THE INSTRUCTION POINTER IS DECREASED BY THE
0339:                 NUMBER OF BYTES - BEING THE LENGTH OF THE
0340:                 INSTRUCTION - WHICH PRECEEDS THE CURRENT
0341:                 INSTRUCTION IN MEMORY.
0342:
0343: 1692 38         DECURA SEC
0344: 1693 A5 E6            LDAZ  CURADL
```

```
1695 E5 F6            SBCZ  BYTES
1697 85 E6            STAZ  CURADL CURADL:=CURADL-BYTES
1699 A5 E7            LDAZ  CURADH
169B E9 00            SBCIM $00
169D 85 E7            STAZ  CURADH CURADH:=CURADH-BORROW
169F 60               RTS


               SUBROUTINR INCTAB
               INCRESES THE POINTER TABLE BY THE AMOUNT DETERMINED
               BY THE CONTENTS OF BYTES (INSTRUCTION LENGTH)

16A0 18        INCTAB CLC
16A1 A5 EC            LDAZ  TABLEL
16A3 65 F6            ADCZ  BYTES
16A5 85 EC            STAZ  TABLEL TABLE:=TABLE + BYTES
16A7 A5 ED            LDAZ  TABLEH
16A9 69 00            ADCIM $00
16AB 85 ED            STAZ  TABLEH
16AD 60               RTS

               THE SUBROUTINE BYTIN  LOADS THE ACCU WITH
               WITH DATA WHICH BELONGS TO TWO NUMERICAL KEYS
               DEPRESSED. RETURNING FROM SUBROUTINE N=1 AND
               Z=0 IF A NON-NUMERICAL KEY WAS DEPRESSED. TWO
               SUCCESIVELY DEPRESSED NUMERICAL KEYS WILL
               RESULT INTO N=0 AND Z=1

16AE 20 AE 12  BYTIN  JSR   RECCHA WAIT FOR A DEPRESSED KEY
16B1 20 1E 14  BYT    JSR   ASHETT CONVERT IT TO A DATA NIBBLE
16B4 30 12            BMI   RETURN ERROR EXIT IF KEY <> 0...F
16B6 0A               ASLA         ENTERED DATA IS NEW HIGHER DATA NIBBLE
16B7 0A               ASLA
16B8 0A               ASLA
16B9 0A               ASLA
16BA 85 FE            STAZ  NIBBLE SAVE HIGHER DATA NIBBLE
16BC 20 AE 12         JSR   RECCHA WAIT FOR A DEPRESSED KEY
16BF 20 1E 14         JSR   ASHETT CONVERT IT TO A DATA NIBBLE
16C2 30 04            BMI   RETURN ERROR EXIT IF KEY <> 0...F
16C4 05 FE            ORAZ  NIBBLE ISERT NEW LOWER DATA NIBBLE
16C6 A2 00            LDXIM $00    RESET N-FLAG, SET Z-FLAG
16C8 60        RETURN RTS


               THE SUBROUTINE READIN LOADS EITHER ONE,
               TWO OR THREE DATA BUFFERS DEPENDING
               OF THE INSTRUCTION LENGTH SPECIFIED BY
               THE OPCODE.
               NORMAL EXIT: Z=1, N=0
               ERROR EXIT: Z=0, N=1

16C9 20 AE 16  READIN JSR   BYTIN  WAIT FOR OPCODE
16CC 30 27            BMI   RDB    ERROR IF KEY <> 0...F
16CE 85 FB     READ   STAZ  POINTH STORE OPCODE IN POINTH
16D0 20 60 1E         JSR   LENACC GET INSTRUCTION LENGTH
16D3 84 F7            STYZ  COUNT  AND COPY IT INTO COUNT
16D5 84 FD            STYZ  TEMPX  AS WELL AS IN TEMPX
16D7 C6 F7            DECZ  COUNT
16D9 F0 18            BEQ   RDA    RETURN IF ONE BYTE INSTR.
16DB 20 F3 11         JSR   PRSP   PRINT A SPACE
16DE 20 AE 16         JSR   BYTIN  WAIT FOR (1ST) OPERAND
16E1 30 12            BMI   RDB    ERROR IF KEY <> 0...F
16E3 85 FA            STAZ  POINTL STORE (1ST) OPERND IN POINTL
16E5 C6 F7            DECZ  COUNT
16E7 F0 0A            BEQ   RDA    RETURN IF TWO BYTE INSTR.
16E9 20 F3 11         JSR   PRSP   PRINT A SPACE
16EC 20 AE 16         JSR   BYTIN  WAIT FOR 2ND OPERAND
16EF 30 04            BMI   RDB    ERROR IF KEY <> 0...F
16F1 85 F9            STAZ  INH    STORE 2ND OPERAND IN INH
```

```
16F3 A2 00     RDA    LDXIM $00    Z=1, N=0
16F5 60        RDB    RTS



                    THE SUBROUTINE CHECK VERIFIES IF THERE IS STILL
                    ROOM IN THE WORKSPACE AREA FOR A NEW INSTRUCTION.
                    THE WORKSPACE IS LIMITED BY BEGAD AND ENDAD.
                    CHECK RETURNS WITH:
                    C=0 IF THERE IS NO ROOM FOR THE INSTR.
                    C=1 IF THERE IS ROOM FOR THE NEW INSTRUCTION

16F6 20 DC 1E  CHECK  JSR   ADCEND ADJUST CURRENT ENDADDRESS POINTER
16F9 38               SEC
16FA A5 E4            LDAZ  ENDADL
16FC E9 02            SBCIM $02
16FE E5 E8            SBCZ  CENDL
1700 A5 E5            LDAZ  ENDADH CARRY DETERMINED BY ENDAD-2-CEND
1702 E5 E9            SBCZ  CENDH
1704 90 04            BCC   CHKEND EXIT IF NO ROOM ANYMORE
1706 20 EA 1E         JSR   RECEND READJUST CURRENT END ADDRESS POINTER
1709 38               SEC
170A 60        CHKEND RTS



                    THE SUBROUTINE PRINS PRINTS AN INSTRUCTION
                    SPECIFIED BY THE INSTRUCTION POINTER CURAD.
                    THE CONTENTS OF THE INSTRUCTION POINTER IS ALSO
                    PRINTED. BY PRINTING A CERTAIN NUMBER OF SPACES
                    THE POSITION OF THE CARRIAGE IS IDENTICAL TO THE
                    FIRST POSITION OF THE SYSTEM COMMAND COLUMN.


170B 20 E8 11  PRINS  JSR   CRLF   PRINT A NEW LINE
170E 20 5C 1E         JSR   OPLEN  GET LENGTH OF INSTRUCTION
1711 A6 F6            LDXZ  BYTES  TO BE PRINTED
1713 A5 E7            LDAZ  CURADH
1715 20 8F 12         JSR   PRBYT  PRINT HIGHER ADDRESS BYTE
1718 A5 E6            LDAZ  CURADL
171A 20 8F 12         JSR   PRBYT  PRINT LOWER ADDRESS BYTE
171D A9 0F            LDAIM $0F
171F 85 EE            STAZ  LABELS MAXIMUM NUMBER OF SPACES
1721 A0 00            LDYIM $00
1723 20 F3 11  PRT    JSR   PRSP   PRINT A SPACE
1726 B1 E6            LDAIY CURADL GET BYTE TO BE PRINTED
1728 20 8F 12         JSR   PRBYT  AND PRINT IT
172B 38               SEC
172C A5 EE            LDAZ  LABELS DECREASE NUMBER OF SPACES
172E E9 03            SBCIM $03    TO BE PRINTED
1730 85 EE            STAZ  LABELS BY 3
1732 C8               INY          SET UP FOR NEXT BYTE
1733 CA               DEX          TO BE PRINTED
1734 D0 ED            BNE   PRT    IF THERE IS ANYONE
1736 20 F3 11  SP     JSR   PRSP   PRINT A NUMBER OF SPACES
1739 C6 EE            DECZ  LABELS
173B D0 F9            BNE   SP
173D 60               RTS          RETURN


                    THE SUBROUTINE MESSA IS IDENTICAL WITH THE
                    PM SUBROUTINE MESSY, EXEPT FOR A DIFFERENT
                    LOOKUP TABLE


173E 20 E8 11  MESSA  JSR   CRLF   PRINT A NEW LINE
1741 B9 50 17  MA     LDAY  TXT    GET CHARACTER TO BE PRINTED
1744 C9 03            CMPIM $03    EOT CHARACTER?
1746 F0 07            BEQ   TXTEND IF YES RETURN
1748 20 34 13         JSR   PRCHA  PRINT A CHARACTER
174B C8               INY          SET UP FOR NEXT CHARACTER
```

```
0483: 174C 4C 41 17          JMP    MA
0484: 174F 60         TXTEND RTS           READY
0485:
0486:
0487:                 LOOKUP TABLE TXT
0488:
0489: 1750 42         TXT    =      'B     Y=00
0490: 1751 45                =      'E
0491: 1752 47                =      'G
0492: 1753 41                =      'A
0493: 1754 44                =      'D
0494: 1755 2C                =      ',
0495: 1756 45                =      'E
0496: 1757 4E                =      'N
0497: 1758 44                =      'D
0498: 1759 41                =      'A
0499: 175A 44                =      'D
0500: 175B 3A                =      ':
0501: 175C 20                =      '
0502: 175D 03                =      $03
0503: 175E 49                =      'I     Y=0E
0504: 175F 4C                =      'L
0505: 1760 4C                =      'L
0506: 1761 45                =      'E
0507: 1762 47                =      'G
0508: 1763 41                =      'A
0509: 1764 4C                =      'L
0510: 1765 20                =      '
0511: 1766 4B                =      'K
0512: 1767 45                =      'E
0513: 1768 59                =      'Y
0514: 1769 03                =      $03
0515: 176A 46                =      'F     Y=1A
0516: 176B 55                =      'U
0517: 176C 4C                =      'L
0518: 176D 4C                =      'L
0519: 176E 03                =      $03
0520: 176F 44                =      'D     Y=1F
0521: 1770 4F                =      'O
0522: 1771 4E                =      'N
0523: 1772 45                =      'E
0524: 1773 03                =      $03
0525: 1774 50                =      'P     Y=24
0526: 1775 4D                =      'M
0527: 1776 20                =      '
0528: 1777 45                =      'E
0529: 1778 44                =      'D
0530: 1779 49                =      'I
0531: 177A 54                =      'T
0532: 177B 4F                =      'O
0533: 177C 52                =      'R
0534: 177D 03                =      $03
0535: 177E 4C                =      'L     Y=2E
0536: 177F 41                =      'A
0537: 1780 42                =      'B
0538: 1781 20                =      '
0539: 1782 24                =      '$
0540: 1783 03                =      $03
0541: 1784 3A                =      ':     Y=34
0542: 1785 20                =      '
0543: 1786 24                =      '$
0544: 1787 03                =      $03
0545:
0546:                 THE INSTRUCTIONS FOLLOWING THE LABEL LABLST
0547:                 DEAL WITH THE PRINTING OF ALL HEXADECIMAL LABELS
0548:                 AFTER THE PROGRAM HAS BEEN ASSEMBLED
0549:
0550:
0551: 1788 A0 FF      LABLST LDYIM $FF
```

```
178A 20 E8 11  LBLSTA JSR   CRLF   PRINT ON A NEW LINE
178D A2 04            LDXIM $04    4 LABELS ON EACH LINE
178F 84 FD     LBLSTB STYZ  TEMPX  SAVE Y
1791 A0 2E            LDYIM $2E    "LAB $"
1793 20 41 17         JSR   MA
1796 A4 FD            LDYZ  TEMPX  RESTORE Y
1798 B1 EC            LDAIY TABLEL GET LABEL NUMBER
179A 20 8F 12         JSR   PRBYT  PRINT LABEL NUMBER
179D 88               DEY
179E 84 FD            STYZ  TEMPX  SAVE Y
17A0 A0 34            LDYIM $34    ": $"
17A2 20 41 17         JSR   MA
17A5 A4 FD            LDYZ  TEMPX  RESTORE Y
17A7 B1 EC            LDAIY TABLEL GET HIGHER ADDRESS BYTE
17A9 20 8F 12         JSR   PRBYT  AND PRINT IT
17AC 88               DEY
17AD B1 EC            LDAIY TABLEL GET LOWER ADDRESS BYTE
17AF 20 8F 12         JSR   PRBYT  AND PRINT IT
17B2 20 F3 11         JSR   PRSP   PRINT A SPACE
17B5 88               DEY          SETUP FOR NEXT LABEL
17B6 C4 EE            CPYZ  LABELS ALL LABELS PRINTED
17B8 F0 05            BEQ   LBLSTC IF NOT SET UP FOR NEXT LABEL
17BA CA               DEX
17BB D0 D2            BNE   LBLSTB ON THE SAME LINE
17BD F0 CB            BEQ   LBLSTA OR ON A NEW LINE
17BF 20 D3 1E  LBLSTC JSR   BEGIN  INSTRUCTION POINTER = BEGAD
17C2 4C 3D 15         JMP   EDITW  WARM START ENTRY OF PME WITHOUT
                            BRK JUMP VECTOR SPECIFICATION



               THE WARM CEND ENTRY (LABEL SEACND) OF PME
               RESTORES THE POSITION OF THE CURRENT END
               ADDRESS POINTER CEND, AFTER SPECIFICATION,
               BY THE USER, OF BEGAD AND ENDAD, AND AFTER FINDING A
               PSEUDO OPCODE 77, I.E. AN EOF CHARACTER. THIS IS
               FOLLOWED BY A JUMP TO THE WARM START ENTRY OF PME.


17C5 20 68 12  SEACND JSR   RESIN  RESET INPUT BUFFERS
17C8 A0 00            LDYIM $00
17CA 20 3E 17         JSR   MESSA  "BEGAD,ENDAD"
17CD 20 87 13         JSR   INPAR  GET BEGAD AND ENDAD
17D0 30 F3            BMI   SEACND TRY IT AGAIN IF NOT PROPERLY DONE
17D2 20 D3 1E         JSR   BEGIN  INSTR. POINTER=BEGAD
17D5 A0 00     SCNDA  LDYIM $00
17D7 B1 E6            LDAIY CURADL GET CURRENT OPCODE
17D9 C9 77            CMPIM $77    IS IT OPCODE 77?
17DB F0 09            BEQ   SCNDB  IF NOT CONTINUE
17DD 20 60 1E         JSR   LENACC GET CURRENT INSTR. LENGTH
17E0 20 F8 1E         JSR   NEXT   ADJUST INSTR. POINTER
17E3 4C D5 17         JMP   SCNDA  AND CHECK AGAIN FOR OPCODE 77
17E6 18        SCNDB  CLC
17E7 A5 E6            LDAZ  CURADL
17E9 69 01            ADCIM $01
17EB 85 E8            STAZ  CENDL  CENDL:=CURADL+1
17ED A5 E7            LDAZ  CURADH
17EF 69 00            ADCIM $00
17F1 85 E9            STAZ  CENDH  CENDH:=CURADH+CARRY
17F3 4C 33 15         JMP   BRK    WARM START ENTRY OF PME


               THE INSTRUCTIONS FOLLOWING THE LABEL BINAR
               AND THE INSTRUCTIONS FOLLOWING THE LABEL PMBINA ARE
               AN EXTENSION OF THE STANDARD MONITOR AND OF PM
               RESPECTIVELY. IN CASE OF SINGLE STEPPING THROUGH
               AN USER PROGRAM WITH DECIMAL ARITHMETIC, THERE
               WILL BE A TEMPORARY SWTCH BACK ON BINARY ARITHMETIC
               PROVIDED THE NMI JUMP VECTOR HAS BEEN SPECIFIED
```

```
0621:                   ON 17F6 OR 17FA RESPECTIVELY.
0622:
0623:
0624: 17F6 D8           BINAR  CLD              BINARY ARITHMETIC
0625: 17F7 4C 00 1C            JMP    SAVE      SAVE INPUT PROPER
0626: 17FA D8           PMBINA CLD              BINARY ARITHMETIC
0627: 17FB 4C CF 14            JMP    STEP      SAVE INPUT PROPER
0628:
0629:
0630:                   ******* END OF THE PME PROGRAM *******
0631:
0632:
0633:                   VICINUS ELABORABAT PROGRAMMAM XXV VI MCMLXXXI
0634:                   VIR NOCTIS IMPOSIT PROGRMMAM IN MACHINAM
0635:
ID=

-T

    SYMBOL TABLE 3000 3276
    ADCENL 1EDC     AGAINH 15CD     ASHET\ 141E     ASMBLZ 1636
    ASSEMJ 1F51     ASSEND 1647     BACK   15D8     BCKA   15F0
    BCKB   1609     BEGADH 00E3     BEGADL 00E2     BEGINH 1ED3
    BINAR  17F6     BRKTHH 1A7D     BRKTLH 1A7C     BRK    1533
    BYTESH 00F6     BYTIN  16AE     BYT    16B1     CENDHH 00E9
    CENDLH 00E8     CHECK  16F6     CHKEND 170A     COUNT  00F7
    CRLF   11E8     CURADH 00E7     CURADL 00E6     DECURA 1692
    DNE    15D5     DONE   1567     EDITC  1500     EDITW  153D
    ENDADH 00E5     ENDADL 00E4     FILLWS 1E47     FOUND  15C3
    FULL   1590     ILLKEY 158C     INCTAB 16A0     INH    00F9
    INPAR  1387     INPUT  164D     INSERT 1579     IOCORR 14F8
    LABELS 00EE     LABLST 1788     LBLSTA 178A     LBLSTB 178F
    LBLSTC 17BF     LENACC 1E60     LINES  161E     LIST   1558
    LST    155F     MA     1741     MEM    1582     MESS   1569
    MESSA  173E     NEXT   1EF8     NIBBLE 00FE     NMIH   1A7B
    NMIL   1A7A     OPLEN  1E5C     PARAH  1A64     PARAL  1A63
    PARBH  1A66     PARBL  1A65     PBDD   1A83     PMBINA 17FA
    POINTH 00FB     POINTL 00FA     PRBYT  128F     PRCHA  1334
    PRINS  170B     PRSP   11F3     PRT    1723     RDA    16F3
    RDB    16F5     READ   16CE     READIN 16C9     RECCHA 12AE
    RECEND 1EEA     RESIN  1268     RESTTY 14BC     RETURN 16C8
    SAVE   1C00     SCAN   15A0     SCNDA  17D5     SCNDB  17E6
    SEACND 17C5     SEARCH 1594     SEMIW  1667     SKIP   156E
    SP     1736     STEP   14CF     SXTEEN 1616     TABLEH 00ED
    TABLEL 00EC     TEMPX  00FD     TOFEND 1613     TOF    160C
    TXTEND 174F     TXT    1750     UP     1E83     WARM   1545
    WRONG  1664
-E
7EE4 0636
-
```

The hex dump of the PME system program

The first section of the PM/PME system program has already been printed in hex format in appendix 5 on page 190 of Book 3. The extended version from address $14F8 . . . $17F5 belongs to PME. The remainder, from $17F6 . . . $17FD, is used by the two routines given in appendix 4.

```
        Ø  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
14FØ:  F9 4C A2 13 FF FF FF FF A9 1E 8D 83 1A 4C 51 1F
15ØØ:  2Ø 68 12 AØ ØØ 2Ø 3E 17 2Ø 87 13 3Ø F3 AD 63 1A
151Ø:  85 E2 18 69 Ø1 85 E8 AD 64 1A 85 E3 69 ØØ 85 E9
152Ø:  AD 65 1A 85 E4 AD 66 1A 85 E5 2Ø D3 1E A9 77 AØ
153Ø:  ØØ 91 E6 A9 3D 8D 7C 1A A9 15 8D 7D 1A AØ 24 2Ø
154Ø:  3E 17 A2 FF 9A 2Ø ØB 17 2Ø AE 12 C9 4B DØ Ø9 2Ø
155Ø:  83 1E 2Ø EA 1E 4C 45 15 C9 4C DØ 12 2Ø D3 1E 2Ø
156Ø:  ØB 17 2Ø F8 1E 3Ø F8 AØ 1F 2Ø 3E 17 FØ D7 C9 2Ø
157Ø:  DØ Ø7 2Ø F8 1E 3Ø CE 1Ø EE C9 49 DØ 17 2Ø C9 16
158Ø:  3Ø ØA 2Ø F6 16 9Ø Ø9 2Ø 47 1E FØ B9 AØ ØE DØ D9
159Ø:  AØ 1A DØ D5 C9 53 DØ 4Ø 2Ø C9 16 3Ø EF 2Ø D3 1E
15AØ:  A5 FB 2Ø 6Ø 1E AØ ØØ B1 E6 C5 FB DØ 2Ø C6 F6 FØ
15BØ:  12 C8 B1 E6 C5 FA DØ 15 C6 F6 FØ Ø7 C8 B1 E6 C5
15CØ:  F9 DØ ØA 2Ø ØB 17 2Ø AE 12 C9 59 DØ Ø8 2Ø 5C 1E
15DØ:  2Ø F8 1E 3Ø CB 4C 67 15 C9 5A DØ 3Ø A5 E2 85 EC
15EØ:  A5 E3 85 ED C5 E7 DØ Ø8 A5 E2 C5 E6 DØ Ø2 FØ 19
15FØ:  AØ ØØ B1 EC 2Ø 6Ø 1E 2Ø AØ 16 A5 E6 C5 EC DØ FØ
16ØØ:  A5 E7 C5 ED DØ EA 2Ø 92 16 4C 45 15 C9 54 DØ Ø6
161Ø:  2Ø D3 1E 4C 45 15 C9 5Ø DØ 1C A9 ØF 85 F7 2Ø 5C
162Ø:  1E 2Ø F8 1E C6 F7 FØ EB AØ ØØ B1 E6 C9 77 FØ E3
163Ø:  2Ø ØB 17 4C 1E 16 C9 58 DØ 13 A9 47 8D 7A 1A A9
164Ø:  16 8D 7B 1A 4C F8 14 2Ø BC 14 4C 88 17 2Ø B1 16
165Ø:  3Ø 12 2Ø CE 16 3Ø ØD 2Ø 5C 1E 2Ø F8 1E A5 FD 85
166Ø:  F6 4C 82 15 4C 8C 15 2Ø 68 12 AØ ØØ 2Ø 3E 17 2Ø
167Ø:  87 13 3Ø F3 AD 63 1A 85 E2 AD 64 1A 85 E3 AD 65
168Ø:  1A 85 E4 85 E8 AD 66 1A 85 E5 85 E9 2Ø D3 1E 4C
169Ø:  33 15 38 A5 E6 E5 F6 85 E6 A5 E7 E9 ØØ 85 E7 6Ø
16AØ:  18 A5 EC 65 F6 85 EC A5 ED 69 ØØ 85 ED 6Ø 2Ø AE
```

```
        Ø  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
16BØ:  12 2Ø 1E 14 3Ø 12 ØA ØA ØA ØA 85 FE 2Ø AE 12 2Ø
16CØ:  1E 14 3Ø Ø4 Ø5 FE A2 ØØ 6Ø 2Ø AE 16 3Ø 27 85 FB
16DØ:  2Ø 6Ø 1E 84 F7 84 FD C6 F7 FØ 18 2Ø F3 11 2Ø AE
16EØ:  16 3Ø 12 85 FA C6 F7 FØ ØA 2Ø F3 11 2Ø AE 16 3Ø
16FØ:  Ø4 85 F9 A2 ØØ 6Ø 2Ø DC 1E 38 A5 E4 E9 Ø2 E5 E8
17ØØ:  A5 E5 E5 E9 9Ø Ø4 2Ø EA 1E 38 6Ø 2Ø E8 11 2Ø 5C
171Ø:  1E A6 F6 A5 E7 2Ø 8F 12 A5 E6 2Ø 8F 12 A9 ØF 85
172Ø:  EE AØ ØØ 2Ø F3 11 B1 E6 2Ø 8F 12 38 A5 EE E9 Ø3
173Ø:  85 EE C8 CA DØ ED 2Ø F3 11 C6 EE DØ F9 6Ø 2Ø E8
174Ø:  11 B9 5Ø 17 C9 Ø3 FØ Ø7 2Ø 34 13 C8 4C 41 17 6Ø
175Ø:  42 45 47 41 44 2C 45 4E 44 41 44 3A 2Ø Ø3 49 4C
176Ø:  4C 45 47 41 4C 2Ø 4B 45 59 Ø3 46 55 4C 4C Ø3 44
177Ø:  4F 4E 45 Ø3 5Ø 4D 2Ø 45 44 49 54 4F 52 Ø3 4C 41
178Ø:  42 2Ø 24 Ø3 3A 2Ø 24 Ø3 AØ FF 2Ø E8 11 A2 Ø4 84
179Ø:  FD AØ 2E 2Ø 41 17 A4 FD B1 EC 2Ø 8F 12 88 84 FD
17AØ:  AØ 34 2Ø 41 17 A4 FD B1 EC 2Ø 8F 12 88 B1 EC 2Ø
17BØ:  8F 12 2Ø F3 11 88 C4 EE FØ Ø5 CA DØ D2 FØ CB 2Ø
17CØ:  D3 1E 4C 3D 15 2Ø 68 12 AØ ØØ 2Ø 3E 17 2Ø 87 13
17DØ:  3Ø F3 2Ø D3 1E AØ ØØ B1 E6 C9 77 FØ Ø9 2Ø 6Ø 1E
17EØ:  2Ø F8 1E 4C D5 17 18 A5 E6 69 Ø1 85 E8 A5 E7 69
17FØ:  ØØ 85 E9 4C 33 15 D8 4C ØØ 1C D8 4C CF 14 77 77
```

Appendix 3

The complete listings of the Tape Monitor and Printer Monitor system programs.

The following pages contain a complete listing of the TM and PM system programs. Each book page corresponds to a single assembler page consisting of 56 lines.
The listing itself consists of the following:

- The main Tape Monitor routine
- The TM subroutines
- The DUMP/DUMPT subroutine
- Subroutines used by DUMP/DUMPT
- The RDTAPE subroutines
- Subroutines used by RDTAPE
- The main Printer Monitor routine
- The PM subroutines
- The look-up table MESS
- The initialisation routine STEP
- The IBRES subroutine
- A survey of all the labels and their addresses

```
0800              ORG   $0800  SOFTWARE OF JUNIOR COMPUTER 3,4


               SOURCE LISTING OF THE TAPE/PRINTER MONITOR

               WRITTEN BY A. NACHTMANN


               DATE: 17 DEC. 1980



               ***************************************
               *** DEFINITIONS OF DUMPT & RDTAPE ***
               ***************************************


               DUMPT STORES A PROGRAM ON TAPE
               1) STORE A PROGRAM ON TAPE
               2) ID IS THE IDENTITY NUMBER
               3) SA IS THE START ADDRESS
               4) EA IS THE END ADDRESS + 1
               5) ID = 00 & ID = FF SHOULD NOT BE USED

               RDTAPE READS A PROGRAM FROM TAPE
               1) READ A PROGRAM FROM TAPE WITH A CERTAIN ID
               2) DISCARD DATA IN MEMORY BEGINNING AT SA
               3) ID = 00 & ID = FF ARE SPECIAL IDS



               THE DATA FORMAT ON TAPE IS ASCII:

               255SYN<>*<>ID<>SAL<>SAH<>DATA<>/<>CHKL<>CHKH<>EOT<>EOT

               I/O DEFINITION

0800           PAD    *     $1A80  PORT A DATA REGISTER
0800           PADD   *     $1A81  PORT A DATA DIRECTION
0800           PBD    *     $1A82  PORT B DATA REGISTER
0800           PBDD   *     $1A83  PORT B DATA DIRECTION


               TIMER REGISTERS

0800           CNTA   *     $1AF4  CLK1T, DISABLE TIMER IRQ
0800           CNTC   *     $1AF6  CLK64T, DISABLE TIMER IRQ
0800           RDFLAG *     $1AD5  READ FLAG REGISTER, B7 IS TIMER FLAG
0800           RDTDIS *     $1AD4  READ CONTENTS OF TIMER CELL, IRQ DISABLEI


               TEMPORARY DATA BUFFERS

0800           SY     *     $1A69  SYN COUNTER
0800           BYTE   *     $1A6A  BYTE FROM TAPE
0800           CHAR   *     $1A6B  CHARACTER FROM TAPE
0800           HIGHER *     $1A6C  3600 HZ HALF PERIODE DELAY
0800           LOWER  *     $1A6D  2400 HZ HALF PERIODE DELAY
0800           CHKL   *     $1A6E  CHECK SUM LOW
0800           CHKH   *     $1A6F  CHECK SUM HIGH
0800           SAL    *     $1A70  START ADDRESS LOW
0800           SAH    *     $1A71  START ADDRESS HIGH
0800           EAL    *     $1A72  END ADDRESS LOW + 1
0800           EAH    *     $1A73  END ADDRESS HIGH
0800           SYNCNT *     $1A74
0800           BITS   *     $1A75  AMOUNT OF BITS
0800           FIRST  *     $1A76  HALF PERIODE AMOUNT OF 3600 HZ
0800           SECOND *     $1A77  HALF PERIODE AMOUNT OF 2400 HZ
0800           GANG   *     $1A78  TEMP OF PBD-BITS
```

```
0800      ID      *     $1A79  ID OF THE DUMPT PROGRAM
0800      NMI     *     $1A7A  NMI VECTOR

          ****************************************
          *** DEFINITIONS OF THE TTY MONITOR ***
          ****************************************


          I/O DEFINITION: SEE DUMPT & RDTAPE


          CPU REGISTERS

0800      PCL     *     $00EF  PROGRAM COUNTER
0800      PCH     *     $00F0
0800      PREG    *     $00F1  STATUS REGISTER
0800      SPUSER  *     $00F2  USER STACK POINTER
0800      ACC     *     $00F3  ACCUMULATOR
0800      YREG    *     $00F4  INDEX Y
0800      XREG    *     $00F5  INDEX X


          INPUT BUFFERS

0800      INL     *     $00F8  LOWER BYTE
0800      INH     *     $00F9  UPPER BYTE


          ADDRESS POINTER

0800      POINTL  *     $00FA
0800      POINTH  *     $00FB


          TEMPORARAY DATA BUFFERS

0800      TEMP    *     $00FC
0800      TEMPX   *     $00FD
0800      STPBIT  *     $1A59  NUMBER OF STOP BITS + 1
0800      CNTL    *     $1A5A  BIT TIME BUFFER
0800      CNTH    *     $1A5B
0800      CNTHL   *     $1A5C  HALF BIT TIME BUFFER
0800      CNTHH   *     $1A5D
0800      TIML    *     $1A5E  COUNT DOWN BUFFER
0800      TIMH    *     $1A5F
0800      TEMPA   *     $1A60  TEMPS
0800      TEMPB   *     $1A61
0800      CHA     *     $1A62  CHARACTER BUFFER
0800      PARAL   *     $1A63  PARAMETER BUFFERS
0800      PARAH   *     $1A64
0800      PARBL   *     $1A65
0800      PARBH   *     $1A66
0800      PRTEMP  *     $1A67  TTY BUFFER
0800      BRKT    *     $1A7C  BREAK TEST VECTOR




          ************************************
          *** BUFFERS & EXTERNAL ADDRESSES ***
          ************************************

0800      DISCNT  *     $1A68  DISPLAY COUNTER
0800      COLDST  *     $1CB5  EDITOR COLD START
0800      WARMST  *     $1CCA  EDITOR WARM START
0800      BEGIN   *     $1ED3  EDITOR SUBROUTINE
0800      RESET   *     $1C1D  RESET OF VERSION D
0800      GETKEY  *     $1DF9  COMPUTE THE KEY VALUE
0800      LDAINH  *     $1DA7  PART OF SCAND/SCANDS
```

```
0800                 BEGADL *      $00E2  BEGIN ADDRESS POINTER
0800                 BEGADH *      $00E3
0800                 ENDADL *      $00E4  END ADDRESS POINTER
0800                 ENDADH *      $00E5
0800                 CENDL  *      $00E8  CURRENT END ADDRESS POINTER
0800                 CENDH  *      $00E9

                     ***********************
                     *** TAPE MANAGEMENT ***
                     ***********************


                     >PC KEY: READ A DATA BLOCK WITH ID=01...FE FROM TAPE
                              1) IF ID = 00, NO SA MAY BE ENTERED
                              2) IF ID = FF, SAH & SAL SHOULD BE ENTERED
                     >AD KEY: SAVE A DATA BLOCK FROM SA TO EA-1 ON TAPE
                              1) ENTER ID = 01...FE
                              2) ENTER SAH & SAL
                              3) ENTER EAH & EAL + 1
                     >DA KEY: 1) EDITOR COLD START ENTRY
                              2) FILE IS DEFINED BY BEGH,BEGL & ENDH,ENDL
                     >+  KEY: DISPLAY ID-SAH-SAL-EAH-EAL-BEGH-BEGL-ENDH-
                                        ENDL-ID-SAH...
                     >GO KEY: SAVE AN EDITOR FILE BETWEEN BEGAD AND CEND
                                 ON TAPE; THEN JUMP TO WARM START ENTRY OF
                                 THE EDITOR AND DISPLAY THE FIRST INSTRUCTION



0800 20 72 0C GETA   JSR    ADJPNT DECREMENT FILE POINTER
0803 A5 FA           LDAZ   POINTL POINT := SA
0805 8D 70 1A        STA    SAL
0808 A5 FB           LDAZ   POINTH
080A 8D 71 1A        STA    SAH
080D 4C 4E 08        JMP    GETB

0810 A2 00    TPINIT LDXIM  $00
0812 8E 79 1A        STX    ID     RESET ALL TAPE PARAMETERS
0815 8E 70 1A        STX    SAL
0818 8E 71 1A        STX    SAH
081B 8E 72 1A        STX    EAL
081E 8E 73 1A        STX    EAH

0821 8E 68 1A TPI    STX    DISCNT RESET DISPLAY COUNTER
0824 A9 06           LDAIM  $06
0826 8D 82 1A        STA    PBD    DISPLAY OFF
0829 A9 1E           LDAIM  $1E
082B 8D 83 1A        STA    PBDD

082E 20 36 09 TPTXT  JSR    TAPDIS DISPLAY TAPE PARAMETERS
0831 D0 FB           BNE    TPTXT  KEY RELEASED?

0833 20 36 09 TTXT   JSR    TAPDIS
0836 F0 FB           BEQ    TTXT   ANY KEY DEPRESSED?
0838 20 36 09        JSR    TAPDIS DEBOUNCE KEY
083B F0 F6           BEQ    TTXT   KEY STILL DEPRESSED?
083D 20 F9 1D        JSR    GETKEY RETURN WITH KEY IN ACCU

0840 C9 14    GET    CMPIM  $14    PC KEY?
0842 D0 0E           BNE    SAVE
0844 20 02 0B        JSR    RDTAPE READ DATA FROM TAPE
0847 A2 FF           LDXIM  $FF
0849 EC 79 1A        CPX    ID     CHECK, IF ID = FF
084C F0 B2           BEQ    GETA   IF YES, ADJUST FILE POINTER

084E E8       GETB   INX           RESET DISCNT
084F 4C 21 08        JMP    TPI    SHOW 'ID  XX'

0852 C9 10    SAVE   CMPIM  $10    AD KEY?
```

```
0208: 0854 D0 08              BNE   PLUS
0209: 0856 20 DF 09           JSR   DUMP   WRITE DATA ON TAPE
0210: 0859 A2 00              LDXIM $00
0211: 085B 4C 21 08           JMP   TPI    SHOW ´ID  XX´
0212:
0213: 085E C9 12      PLUS    CMPIM $12    + KEY?
0214: 0860 D0 11              BNE   FILES
0215: 0862 EE 68 1A           INC   DISCNT SET UP PARAMETER COUNTER
0216: 0865 A0 09              LDYIM $09    LIMIT COUNT
0217: 0867 CC 68 1A           CPY   DISCNT
0218: 086A D0 C2              BNE   TPTXT
0219: 086C A0 00              LDYIM $00    RESET PARAMETER COUNTER
0220: 086E 8C 68 1A           STY   DISCNT
0221: 0871 F0 BB              BEQ   TPTXT
0222:
0223: 0873 C9 13      FILES   CMPIM $13    GO KEY?
0224: 0875 D0 25              BNE   DAT
0225: 0877 A5 E2              LDAZ  BEGADL
0226: 0879 8D 70 1A           STA   SAL    BEGAD := SA
0227: 087C A5 E3              LDAZ  BEGADH
0228: 087E 8D 71 1A           STA   SAH
0229: 0881 A5 E8              LDAZ  CENDL  CEND := EA
0230: 0883 8D 72 1A           STA   EAL
0231: 0886 A5 E9              LDAZ  CENDH
0232: 0888 8D 73 1A           STA   EAH
0233:
0234: 088B 20 DF 09           JSR   DUMP   WRITE DATA = FILE ON TAPE
0235: 088E 20 D3 1E           JSR   BEGIN  SHOW THE FIRST INSTRUKTION
0236: 0891 A9 1E              LDAIM $1E    DIFINE I/O
0237: 0893 8D 83 1A           STA   PBDD
0238: 0896 A2 FF              LDXIM $FF    RESET STACK
0239: 0898 9A                 TXS
0240: 0899 4C CA 1C           JMP   WARMST RETURN TO EDITOR
0241:
0242: 089C C9 11      DAT     CMPIM $11    DA KEY?
0243: 089E D0 03              BNE   SHIFT  IT WAS A DATA KEY
0244: 08A0 4C B5 1C           JMP   COLDST EDITOR COLD START ENTRY
0245:
0246: 08A3 06 F9      SHIFT   ASLZ  INH    SHIFT KEY INTO DISPLAY BUFFER
0247: 08A5 06 F9              ASLZ  INH
0248: 08A7 06 F9              ASLZ  INH
0249: 08A9 06 F9              ASLZ  INH
0250: 08AB 05 F9              ORAZ  INH
0251: 08AD 85 F9              STAZ  INH
0252: 08AF A0 00              LDYIM $00    RESET PARAMETER COUNTER
0253: 08B1 CC 68 1A           CPY   DISCNT ID TO DISPLAY?
0254: 08B4 D0 06              BNE   STSAH
0255: 08B6 8D 79 1A           STA   ID     SAVE ID
0256: 08B9 4C 2E 08           JMP   TPTXT
0257:
0258: 08BC C8         STSAH   INY
0259: 08BD CC 68 1A           CPY   DISCNT SAH TO DISPLAY?
0260: 08C0 D0 06              BNE   STSAL
0261: 08C2 8D 71 1A           STA   SAH    SAVE SAH
0262: 08C5 4C 2E 08           JMP   TPTXT
0263:
0264: 08C8 C8         STSAL   INY
0265: 08C9 CC 68 1A           CPY   DISCNT SAL TO DISPLAY?
0266: 08CC D0 06              BNE   STEAH
0267: 08CE 8D 70 1A           STA   SAL    SAVE SAL
0268: 08D1 4C 2E 08           JMP   TPTXT
0269:
0270: 08D4 C8         STEAH   INY
0271: 08D5 CC 68 1A           CPY   DISCNT EAH TO DISPLAY?
0272: 08D8 D0 06              BNE   STEAL
0273: 08DA 8D 73 1A           STA   EAH    SAVE EAH
0274: 08DD 4C 2E 08           JMP   TPTXT
0275:
0276: 08E0 C8         STEAL   INY
```

```
08E1 CC 68 1A           CPY   DISCNT EAL TO DISPLAY?
08E4 D0 06              BNE   STBEGH
08E6 8D 72 1A           STA   EAL    SAVE EAL
08E9 4C 2E 08           JMP   TPTXT

08EC C8          STBEGH INY
08ED CC 68 1A           CPY   DISCNT BEGH TO DISPLAY?
08F0 D0 05              BNE   STBEGL
08F2 85 E3              STAZ  BEGADH SAVE BEGADH
08F4 4C 2E 08           JMP   TPTXT

08F7 C8          STBEGL INY
08F8 CC 68 1A           CPY   DISCNT BEGADL TO DISPLAY?
08FB D0 05              BNE   STENDH
08FD 85 E2              STAZ  BEGADL SAVE BEGADL
08FF 4C 2E 08           JMP   TPTXT

0902 C8          STENDH INY
0903 CC 68 1A           CPY   DISCNT ENDADH TO DISPLAY?
0906 D0 05              BNE   STENDL
0908 85 E5              STAZ  ENDADH SAVE ENDADH
090A 4C 2E 08           JMP   TPTXT

090D C8          STENDL INY
090E CC 68 1A           CPY   DISCNT ENDADL TO DISPLAY?
0911 D0 05              BNE   TPVEC  ERROR EXIT
0913 85 E4              STAZ  ENDADL SAVE ENDADL
0915 4C 2E 08           JMP   TPTXT


0918 4C 10 08    TPVEC  JMP   TPINIT

                 SUBROUTINES OF 'TAPE MANAGEMENT'

091B B9 BB 09    TDISP  LDAY  TLOOK  TEXT LOOKUP
091E 8D 80 1A           STA   PAD    OUTPUT TEXT PATTERN
0921 8E 82 1A           STX   PBD    DISPLAY ENABLE
0924 98                 TYA          SAVE INDEX Y
0925 A0 7F              LDYIM $7F

0927 88          DELY   DEY          DELAY
0928 D0 FD              BNE   DELY
092A A8                 TAY          RESTORE INDEX Y
092B E8                 INX          SET UP FOR NEXT DISPLAY
092C E8                 INX
092D C8                 INY          ADJUST TEXT POINTER
092E E0 10              CPXIM $10    4 DISPLAYS SCANNED?
0930 D0 E9              BNE   TDISP
0932 20 A7 1D           JSR   LDAINH DISPLAY DATA BUFFER INH
0935 60                 RTS

0936 A9 7F       TAPDIS LDAIM $7F
0938 8D 81 1A           STA   PADD   I/O DEFINITION

093B A2 08       SID    LDXIM $08    INIT DISPLAY SWITCH
093D A0 00              LDYIM $00
093F CC 68 1A           CPY   DISCNT ID TO DISPLAY?
0942 D0 09              BNE   SSAH
0944 AD 79 1A           LDA   ID     ID TO DISPLAY
0947 85 F9              STAZ  INH

0949 20 1B 09    COMPNT JSR   TDISP  SHOW ANY PARAMETER
094C 60                 RTS

094D C8          SSAH   INY
094E CC 68 1A           CPY   DISCNT SAH TO DISPLAY?
0951 D0 09              BNE   SSAL
0953 AD 71 1A           LDA   SAH    SAH TO DISPLAY
0956 85 F9              STAZ  INH
```

```
0346: 0958 A0 04            LDYIM $04    LOOKUP POINTER
0347: 095A D0 ED            BNE   COMPNT SHOW SAH
0348:
0349: 095C C8         SSAL  INY
0350: 095D CC 68 1A         CPY   DISCNT SAL TO DISPLAY?
0351: 0960 D0 09            BNE   SEAH
0352: 0962 AD 70 1A         LDA   SAL    SAL TO DISPLAY
0353: 0965 85 F9            STAZ  INH
0354: 0967 A0 08            LDYIM $08    LOOKUP POINTER
0355: 0969 D0 DE            BNE   COMPNT SHOW SAL
0356:
0357: 096B C8         SEAH  INY
0358: 096C CC 68 1A         CPY   DISCNT EAH TO DISPLAY?
0359: 096F D0 09            BNE   SEAL
0360: 0971 AD 73 1A         LDA   EAH    EAH TO DISPLAY
0361: 0974 85 F9            STAZ  INH
0362: 0976 A0 0C            LDYIM $0C    LOOKUP POINTER
0363: 0978 D0 CF            BNE   COMPNT SHOW EAH
0364:
0365: 097A C8         SEAL  INY
0366: 097B CC 68 1A         CPY   DISCNT
0367: 097E D0 09            BNE   SBEGH
0368: 0980 AD 72 1A         LDA   EAL
0369: 0983 85 F9            STAZ  INH
0370: 0985 A0 10            LDYIM $10    LOOKUP POINTER
0371: 0987 D0 C0            BNE   COMPNT SHOW EAL
0372:
0373: 0989 C8         SBEGH INY
0374: 098A CC 68 1A         CPY   DISCNT
0375: 098D D0 08            BNE   SBEGL
0376: 098F A5 E3            LDAZ  BEGADH
0377: 0991 85 F9            STAZ  INH
0378: 0993 A0 14            LDYIM $14
0379: 0995 D0 B2            BNE   COMPNT
0380:
0381: 0997 C8         SBEGL INY
0382: 0998 CC 68 1A         CPY   DISCNT
0383: 099B D0 08            BNE   SENDH
0384: 099D A5 E2            LDAZ  BEGADL
0385: 099F 85 F9            STAZ  INH
0386: 09A1 A0 18            LDYIM $18
0387: 09A3 D0 A4            BNE   COMPNT
0388:
0389: 09A5 C8         SENDH INY
0390: 09A6 CC 68 1A         CPY   DISCNT
0391: 09A9 D0 08            BNE   SENDL
0392: 09AB A5 E5            LDAZ  ENDADH
0393: 09AD 85 F9            STAZ  INH
0394: 09AF A0 1C            LDYIM $1C
0395: 09B1 D0 96            BNE   COMPNT
0396:
0397: 09B3 A5 E4      SENDL LDAZ  ENDADL
0398: 09B5 85 F9            STAZ  INH
0399: 09B7 A0 20            LDYIM $20
0400: 09B9 D0 8E            BNE   COMPNT
0401:
0402:
0403:
0404:                 7 SEGMENT TEXT LOOKUP TABLE
0405:
0406: 09BB 66         TLOOK =     $66    ID PATTERN
0407: 09BC 21               =     $21
0408: 09BD 7F               =     $7F
0409: 09BE 7F               =     $7F
0410:
0411: 09BF 52               =     $52    SAH PATTERN
0412: 09C0 08               =     $08
0413: 09C1 09               =     $09
0414: 09C2 7F               =     $7F
```

```
0415:
0416: 09C3 52                =     $52     SAL PATTERN
0417: 09C4 08                =     $08
0418: 09C5 47                =     $47
0419: 09C6 7F                =     $7F
0420:
0421: 09C7 06                =     $06     EAH PATTERN
0422: 09C8 08                =     $08
0423: 09C9 09                =     $09
0424: 09CA 7F                =     $7F
0425:
0426: 09CB 06                =     $06     EAL PATTERN
0427: 09CC 08                =     $08
0428: 09CD 47                =     $47
0429: 09CE 7F                =     $7F
0430:
0431: 09CF 03                =     $03     BEGH PATTERN
0432: 09D0 06                =     $06
0433: 09D1 42                =     $42
0434: 09D2 09                =     $09
0435:
0436: 09D3 03                =     $03     BEGL PATTERN
0437: 09D4 06                =     $06
0438: 09D5 42                =     $42
0439: 09D6 47                =     $47
0440:
0441: 09D7 06                =     $06     ENDH PATTERN
0442: 09D8 2B                =     $2B
0443: 09D9 21                =     $21
0444: 09DA 09                =     $09
0445:
0446: 09DB 06                =     $06     ENDL PATTERN
0447: 09DC 2B                =     $2B
0448: 09DD 21                =     $21
0449: 09DE 47                =     $47
0450:                ******************************
0451:                * DUMP/DUMPT IS A SUBROUTINE *
0452:                ******************************
0453:
0454: 09DF A9 7D     DUMP   LDAIM $7D     3600HZ HALF PERIODE DELAY
0455: 09E1 8D 6C 1A         STA   HIGHER
0456: 09E4 A9 C3           LDAIM $C3     2400 HZ HALF PERIODE DELAY
0457: 09E6 8D 6D 1A         STA   LOWER
0458: 09E9 A9 03           LDAIM $03     TOGGLE 3 TIMES AT 3600 HZ
0459: 09EB 8D 76 1A         STA   FIRST
0460: 09EE A9 02           LDAIM $02     TOGGLE 2 TIMES AT 2400 HZ
0461: 09F0 8D 77 1A         STA   SECOND
0462:
0463: 09F3 A9 47     DUMPT  LDAIM $47     PBD PATTERN
0464: 09F5 A2 FF           LDXIM $FF     PBDD PATTERN
0465: 09F7 8D 82 1A         STA   PBD     INPUT OFF,OUTPUT ON,STROBE DISABLED
0466: 09FA 8D 78 1A         STA   GANG    SAVE BITS OF PBD
0467: 09FD 8E 83 1A         STX   PBDD    PB7...PB0 IS OUTPUT
0468: 0A00 A9 00           LDAIM $00     PAD PATTERN
0469: 0A02 A2 7F           LDXIM $7F     PADD PATTERN
0470: 0A04 8D 80 1A         STA   PAD     SEGMENTS ON BUT DISABLED
0471: 0A07 8E 81 1A         STX   PADD    ALL LINES OUTPUT EXEPT PA7
0472: 0A0A 8D 6E 1A         STA   CHKL    RESET CHECK SUM
0473: 0A0D 8D 6F 1A         STA   CHKH
0474: 0A10 AD 70 1A         LDA   SAL     INITIALIZE DUMPT POINTER
0475: 0A13 85 FA           STAZ  POINTL
0476: 0A15 AD 71 1A         LDA   SAH
0477: 0A18 85 FB           STAZ  POINTH
0478: 0A1A A2 FF           LDXIM $FF     SET SYNC COUNTER
0479: 0A1C 8E 74 1A         STX   SYNCNT
0480:
0481: 0A1F A9 16     SYNCS  LDAIM         SYN CHARACTER
0482: 0A21 20 A3 0A         JSR   OUTCH   OUTPUT 255 SYN CHARACTERS
0483: 0A24 CE 74 1A         DEC   SYNCNT
```

```
0484: 0A27 D0 F6                BNE   SYNCS
0485:
0486: 0A29 A9 2A                LDAIM ´*     OUTPUT START CHARACTER
0487: 0A2B 20 A3 0A             JSR   OUTCH
0488: 0A2E AD 79 1A             LDA   ID     OUTPUT ID
0489: 0A31 20 8B 0A             JSR   OUTBT
0490: 0A34 AD 70 1A             LDA   SAL    OUTPUT START ADDRESS
0491: 0A37 20 7A 0A             JSR   OUTBTC AND START CHECK SUM COMPUTATION
0492: 0A3A AD 71 1A             LDA   SAH
0493: 0A3D 20 7A 0A             JSR   OUTBTC
0494:
0495: 0A40 A5 FB         DATATR LDAZ  POINTH
0496: 0A42 CD 73 1A             CMP   EAH    ENTIRE FILE TRANSMITTED?
0497: 0A45 D0 23                BNE   HEXDAT
0498: 0A47 A5 FA                LDAZ  POINTL
0499: 0A49 CD 72 1A             CMP   EAL
0500: 0A4C D0 1C                BNE   HEXDAT
0501:
0502: 0A4E A9 2F                LDAIM ´/     OUTPUT END OF DATA CHARACTER
0503: 0A50 20 A3 0A             JSR   OUTCH  STOP WITH CHECK SUM COMPUTATION
0504: 0A53 AD 6E 1A             LDA   CHKL   OUTPUT CHECK SUM
0505: 0A56 20 8B 0A             JSR   OUTBT
0506: 0A59 AD 6F 1A             LDA   CHKH
0507: 0A5C 20 8B 0A             JSR   OUTBT
0508: 0A5F A9 04                LDAIM ´      EOT CHARACTER
0509: 0A61 20 A3 0A             JSR   OUTCH  OUTPUT EOT CHARACTER
0510: 0A64 A9 04                LDAIM ´      EOT CHARACTER
0511: 0A66 20 A3 0A             JSR   OUTCH
0512:
0513: 0A69 60                   RTS          RETURN TO CALLER
0514:
0515: 0A6A A0 00         HEXDAT LDYIM $00
0516: 0A6C B1 FA                LDAIY POINTL FETCH CURRENT DATA BYTE
0517: 0A6E 20 7A 0A             JSR   OUTBTC TRANSMIT CURRENT DATA BYTE
0518: 0A71 E6 FA                INCZ  POINTL AND COMPUT CHECK SUM
0519: 0A73 D0 CB                BNE   DATATR SET UP FOR NEXT DATA BYTE
0520: 0A75 E6 FB                INCZ  POINTH
0521: 0A77 4C 40 0A             JMP   DATATR
0522:
0523:                    *** END OF DUMP/DUMPT ***
0524:                    DUMP´S SUBROUTINES
0525:
0526:
0527:                    OUTBTC OUTPUTS A HEX BYTE AS TWO ASCII CHARACTERS
0528:                    TO THE TAPE RECORDER. ALSO THE CHECK SUM IS COM-
0529:                    PUTED.
0530:
0531:                    OUTBT OUTPUTS A HEX BYTE AS TWO ASCCI CHARACTERS
0532:                    TO THE TAPE RECORDER WITHOUT CHECK SUM COMPUTATION.
0533:
0534:                    NIBOUT CONVERTS A HEX NIBBLE TO AN ASCII CHARACTER
0535:                    AND TRANSMITS THE 8 BIT ASCII WORD TO THE TAPE.
0536:
0537:                    HIGH AND LOW PRODUCE THE DELAYS OF THE 3600 HZ AND
0538:                    THE 2400 HZ FREQUENCY.
0539:
0540:
0541:                    *** OUTBTC/OUTBT ***
0542:
0543: 0A7A A8            OUTBTC TAY          SAVE ACCU
0544: 0A7B 18                   CLC
0545: 0A7C 6D 6E 1A             ADC   CHKL   CHECK SUM COMPUTATION
0546: 0A7F 8D 6E 1A             STA   CHKL
0547: 0A82 AD 6F 1A             LDA   CHKH
0548: 0A85 69 00                ADCIM $00    CHK:= CHK + BYTE
0549: 0A87 8D 6F 1A             STA   CHKH
0550: 0A8A 98                   TYA          GET ACCU AGAIN
0551:
0552: 0A8B A8            OUTBT  TAY          SAVE ACCU TEMP
```

```
0553: 0A8C 4A               LSRA         GET UPPER NIBBLE
0554: 0A8D 4A               LSRA
0555: 0A8E 4A               LSRA
0556: 0A8F 4A               LSRA
0557: 0A90 20 9A 0A         JSR   NIBOUT OUTPUT UPPER NIBBLE AS ASCII CHAR.
0558: 0A93 98               TYA          GET BYTE AGAIN
0559: 0A94 29 0F            ANDIM $0F    GET LOWER NIBBLE
0560: 0A96 20 9A 0A         JSR   NIBOUT OUTPUT LOWER NIBBLE AS ASCII CHAR.
0561: 0A99 60               RTS
0562:
0563:                *** END OF OUTBTC/OUTBT ***
0564:
0565:
0566:                *** NIBOUT/OUTCH ***
0567:
0568: 0A9A C9 0A     NIBOUT CMPIM $0A    CONVERT A NIBBLE TO AN ASCII CHAR.
0569: 0A9C 18               CLC
0570: 0A9D 30 02            BMI   NIB
0571: 0A9F 69 07            ADCIM $07
0572:
0573: 0AA1 69 30     NIB    ADCIM $30
0574:
0575: 0AA3 A2 08     OUTCH  LDXIM $08    SET UP FOR 8 BITS
0576: 0AA5 8E 75 1A         STX   BITS
0577:
0578: 0AA8 4A        ONE    LSRA         SHIFT OUT BIT BY BIT
0579: 0AA9 48               PHA          SAVE CHARACTER
0580: 0AAA 90 0C            BCC   ZERO
0581: 0AAC 20 C8 0A         JSR   HIGH   START AT 3600 HZ
0582: 0AAF 20 E5 0A         JSR   LOW
0583: 0AB2 20 E5 0A         JSR   LOW    END AT 2400 HZ
0584: 0AB5 4C C1 0A         JMP   ZRO
0585:
0586: 0AB8 20 C8 0A  ZERO   JSR   HIGH   START AT 3600 HZ
0587: 0ABB 20 C8 0A         JSR   HIGH
0588: 0ABE 20 E5 0A         JSR   LOW    END AT 2400 HZ
0589:
0590: 0AC1 68        ZRO    PLA          GET CHARACTER AGAIN
0591: 0AC2 CE 75 1A         DEC   BITS   ALL BITS SHIFTED OUT
0592: 0AC5 D0 E1            BNE   ONE
0593: 0AC7 60               RTS
0594:
0595:                *** END OF NIBOUT/OUTCH ***
0596:
0597:
0598:                *** HIGH ***
0599:
0600: 0AC8 AE 76 1A  HIGH   LDX   FIRST  AMOUNT OF HALF PERIODES
0601:
0602: 0ACB 2C D5 1A  HIG    BIT   RDFLAG TIME OUT?
0603: 0ACE 10 FB            BPL   HIG
0604: 0AD0 AD 6C 1A         LDA   HIGHER SET UP TIMER FOR SHORT PERIODE
0605: 0AD3 8D F4 1A         STA   CNTA   DISABLE TIMER IRQ, CLK1T
0606: 0AD6 AD 78 1A         LDA   GANG
0607: 0AD9 49 80            EORIM $80    TOGGLE OUTPUT
0608: 0ADB 8D 82 1A         STA   PBD
0609: 0ADE 8D 78 1A         STA   GANG   SAVE STATUS QUO
0610: 0AE1 CA               DEX
0611: 0AE2 D0 E7            BNE   HIG
0612: 0AE4 60               RTS
0613:
0614:                *** END OF HIGH ***
0615:
0616:
0617:                *** LOW ***
0618:
0619: 0AE5 AE 77 1A  LOW    LDX   SECOND AMOUNT OF HALF PERIODES
0620:
0621: 0AE8 2C D5 1A  LO     BIT   RDFLAG TIME OUT?
```

```
0AEB 10 FB             BPL   LO
0AED AD 6D 1A          LDA   LOWER  SET UP TIMER FOR LONG PERIODE
0AF0 8D F4 1A          STA   CNTA   DISABLE TIMER IRQ, CLK1T
0AF3 AD 78 1A          LDA   GANG
0AF6 49 80             EORIM $80    TOGGLE OUTPUT
0AF8 8D 82 1A          STA   PBD
0AFB 8D 78 1A          STA   GANG   SAVE STATUS QUO
0AFE CA                DEX
0AFF D0 E7             BNE   LO
0B01 60                RTS

               *** END OF LOW ***

               ***************************
               * RDTAPE IS A SUBROUTINE *
               ***************************

               ID = 00: IGNORE ID ON TAPE; DISCARD DATA BLOCK
               AT SAL, SAH COMING FROM TAPE

               ID = FF: IGNORE ID ON TAPE; DISCARD DATA BLOCK
               AT SAL, SAH STORED IN JUNIORS MEMORY

0B02 A9 32     RDTAPE  LDAIM $32    INPUT RECORDER IS ON
0B04 8D 82 1A          STA   PBD    OUTPUT RECORDER IS OFF, PB7 IS ENABLED
0B07 8D 78 1A          STA   GANG   STROBE ´9´ IS ENABLED
0B0A A9 7E             LDAIM $7E    PB0 IS INPUT
0B0C 8D 83 1A          STA   PBDD   PB7 IS INPUT
0B0F A9 7F             LDAIM $7F    PA6...PA0 IS OUTPUT
0B11 8D 81 1A          STA   PADD
0B14 A9 00             LDAIM $00    SEGMENTS ON
0B16 8D 6E 1A          STA   CHKL   RESET CHECK SUM
0B19 8D 6F 1A          STA   CHKH

0B1C A9 FF     SYNC    LDAIM $FF    RESET FOR SYN CHARACTER
0B1E 8D 6B 1A          STA   CHAR

0B21 20 C2 0B  SYNCA   JSR   RDBIT  READ A BIT FROM TAPE
0B24 6E 6B 1A          ROR   CHAR   RIGHT SHIFT
0B27 AD 6B 1A          LDA   CHAR   GET CURRENT CHARACTER
0B2A 20 E8 0B          JSR   BTWEEN DISPLAY NOT SYN CHARACTER
0B2D C9 16             CMPIM ´      SYN CHARACTER?
0B2F D0 F0             BNE   SYNCA  IF NOT, RESYNC
0B31 A0 0A             LDYIM $0A    TRY IT FOR 10 SYNS AT LEAST
0B33 8C 69 1A          STY   SY     SYN COUNTER

0B36 20 36 0C  TENSYN  JSR   RDCH
0B39 20 5D 0C          JSR   CHARVU DISPLAY SYN CHARACTER
0B3C C9 16             CMPIM ´      STILL SYN CHARACTER?
0B3E D0 DC             BNE   SYNC   IF NOT, RETURN
0B40 CE 69 1A          DEC   SY     10 SYNS RECEIVED?
0B43 D0 F1             BNE   TENSYN RETURN IF LESS THAN 10 SYNS

0B45 20 36 0C  STAR    JSR   RDCH   WAIT FOR ´*´ CHARACTER
0B48 20 5D 0C          JSR   CHARVU
0B4B C9 2A             CMPIM ´*
0B4D F0 06             BEQ   STARA
0B4F C9 16             CMPIM ´     STILL SYN CHARACTER?
0B51 F0 F2             BEQ   STAR  IF YES, THEN WAIT
0B53 D0 AD             BNE   RDTAPE IF NOT, THEN RESYNC

0B55 20 5D 0C  STARA   JSR   CHARVU DISPLAY ´*´
0B58 20 F3 0B          JSR   RDBYT  READ ID FROM TAPE
0B5B CD 79 1A          CMP   ID     REQUESTED ID?
0B5E D0 3E             BNE   CHKID

0B60 20 F3 0B  RDSA    JSR   RDBYT  READ SAL FROM TAPE
0B63 20 4B 0C          JSR   CHKSUM CHECK SUM COMPUTATION
0B66 85 FA             STAZ  POINTL SET UP STORE POINTER
```

```
 0B68 20 F3 0B          JSR   RDBYT  READ SAH FROM TAPE
 0B6B 20 4B 0C          JSR   CHKSUM
 0B6E 85 FB             STAZ  POINTH

 0B70 20 F3 0B   FILMEM JSR   RDBYT  READ DATA BYTE FROM TAPE
 0B73 30 8D             BMI   RDTAPE NOT VALID HEX CHARACTER
 0B75 F0 13             BEQ   CHECK  END OF DATA CHARACTER?
 0B77 20 4B 0C          JSR   CHKSUM
 0B7A A0 00             LDYIM $00
 0B7C 91 FA             STAIY POINTL STORE BYTE IN MEMORY
 0B7E E6 FA             INCZ  POINTL SET POINTER FOR NEXT BYTE
 0B80 D0 02             BNE   FMA
 0B82 E6 FB             INCZ  POINTH

 0B84 20 64 0C   FMA    JSR   VU     DISPLAY '*'
 0B87 4C 70 0B          JMP   FILMEM READ NEXT DATA BYTE FROM TAPE

 0B8A 20 F3 0B   CHECK  JSR   RDBYT  READ CHECK SUM FROM TAPE
 0B8D CD 6E 1A          CMP   CHKL   AND COMPARE IT
 0B90 D0 09             BNE   SYNVEC
 0B92 20 F3 0B          JSR   RDBYT
 0B95 CD 6F 1A          CMP   CHKH
 0B98 D0 01             BNE   SYNVEC
 0B9A 60                RTS          RETURN TO CALLER

 0B9B 4C 02 0B   SYNVEC JMP   RDTAPE

 0B9E AD 79 1A   CHKID  LDA   ID
 0BA1 C9 00             CMPIM $00    ID = 00?
 0BA3 F0 BB             BEQ   RDSA
 0BA5 C9 FF             CMPIM $FF    ID = FF?
 0BA7 D0 F2             BNE   SYNVEC
 0BA9 20 F3 0B          JSR   RDBYT  READ SA FROM TAPE, BUT IGNORE IT
 0BAC 20 4B 0C          JSR   CHKSUM
 0BAF 20 F3 0B          JSR   RDBYT
 0BB2 20 4B 0C          JSR   CHKSUM
 0BB5 AD 70 1A          LDA   SAL    USE SA STORED IN BUFFER
 0BB8 85 FA             STAZ  POINTL
 0BBA AD 71 1A          LDA   SAH
 0BBD 85 FB             STAZ  POINTH
 0BBF 4C 70 0B          JMP   FILMEM

                 *** END OF RDTAPE ***

                 SUBROUTINES OF RDTAPE

                 *** RDBIT ***

                 RDBIT READS 1 BIT FROM TAPE.
                 LOG 1: IT RETURNS WITH C = 1
                 LOG 0: IT RETUNS WITH C = 0

 0BC2 2C 82 1A   RDBIT  BIT   PBD    3600 HZ?
 0BC5 10 FB             BPL   RDBIT
 0BC7 AD D4 1A          LDA   RDTDIS GET COUNT DOWN
 0BCA A0 FF             LDYIM $FF
 0BCC 8C F6 1A          STY   CNTC   START TIMER FOR 2400 HZ COUNT DOWN
 0BCF A0 14             LDYIM $14

 0BD1 88         RDBA   DEY          DELAY JITTER TIME
 0BD2 D0 FD             BNE   RDBA

 0BD4 2C 82 1A   RDBB   BIT   PBD    2400 HZ?
 0BD7 30 FB             BMI   RDBB
 0BD9 38                SEC
 0BDA ED D4 1A          SBC   RDTDIS SET OR RESET C-FLAG
 0BDD A0 FF             LDYIM $FF
 0BDF 8C F6 1A          STY   CNTC   START TIMER FOR 3600 HZ COUNT DOWN
 0BE2 A0 07             LDYIM $07    DELAY FOR JITTER
```

```
0760:
0761: 0BE4 88          RDBC   DEY
0762: 0BE5 D0 FD             BNE   RDBC
0763: 0BE7 60                RTS
0764:
0765:                 *** END OF RDBIT ***
0766:
0767:
0768:
0769:                 *** BTWEEN ***
0770:
0771:                 DISPLAY THE BETWEEN CHARACTER
0772:
0773: 0BE8 48         BTWEEN PHA          SAVE ACCU
0774: 0BE9 A9 36             LDAIM $36    OUTPUT BETWEEN CHARACTER
0775: 0BEB 8D 80 1A          STA   PAD
0776: 0BEE 68                PLA          GET ACCU AGAIN
0777: 0BEF 20 64 0C          JSR   VU     DON´T CHANGE BETWEEN
0778: 0BF2 60                RTS
0779:
0780:                 *** END OF BTWEEN ***
0781:                 *** RDBYT ***
0782:
0783:
0784:                 READ 1 HEX BYTE = 2 ASCII CHARACTERS FROM TAPE.
0785:                 COMPOSE THE HEX VALUES OF THE CHARACTERS IN ACCU
0786:                 AND RETURN.
0787:
0788:                 1) N = 1: A NOT VALID HEX CHARACTER WAS TRANSMITTED
0789:                 2) Z = 1: END OF DATA TRANSMISSION
0790:                 3) Z = 0: VALID HEX BYTE IN ACCU
0791:                 4) THE N-FLAG HAS ALWAYS PRIORITY
0792:
0793: 0BF3 20 36 0C   RDBYT  JSR   RDCH   READ ANY ASCII CHARACTER FROM TAPE
0794: 0BF6 C9 2F            CMPIM ´/     END OF DATA CHARACTER?
0795: 0BF8 D0 01            BNE   RBB
0796:
0797: 0BFA 60         RBA    RTS          ERROR EXIT
0798:
0799: 0BFB 20 19 0C   RBB    JSR   ASCHEX ASCII HEX CONVERSION
0800: 0BFE 30 FA             BMI   RBA    NOT VALID CHARACTER
0801: 0C00 0A                ASLA         SHIFT NIBBLE TO LEFT
0802: 0C01 0A                ASLA
0803: 0C02 0A                ASLA
0804: 0C03 0A                ASLA
0805: 0C04 8D 6A 1A          STA   BYTE   SAVE HIGH ORDER NIBBLE
0806: 0C07 20 36 0C          JSR   RDCH   READ NEXT CHARACTER
0807: 0C0A C9 2F             CMPIM ´/     END OF DATA CHARACTER
0808: 0C0C F0 EC             BEQ   RBA
0809: 0C0E 20 19 0C          JSR   ASCHEX ASCII HEX CONVERSION
0810: 0C11 30 E7             BMI   RBA    NOT VALID CHARACTER
0811: 0C13 0D 6A 1A          ORA   BYTE   BYTE = HIGH ORDER AND LOW ORDER NIBBLE
0812: 0C16 A0 01             LDYIM $01    BE SHURE THAT CHARACTER
0813: 0C18 60                RTS          NORMAL EXIT
0814:
0815:                 *** END OF RDBYT ***
0816:                 *** ASCHEX ***
0817:
0818:
0819:                 CONVERT AN ASCII CHARACTER TO A HEX DATA NIBBLE.
0820:
0821:                 1) RETURN WITH CONVERTED HEX NUMBER IN ACCU
0822:                 2) N = 1, IF NOT VALID HEX NUMBER
0823:                 3) Z = 1, IF VALID HEX NUMBER
0824:                 4) *** ASCHEX IS ALSO USED IN THE PRINTER SOFTWARE ***
0825:
0826: 0C19 C9 30      ASCHEX CMPIM $30    IGNORE 00...2F
0827: 0C1B 30 0C             BMI   NOTVAL
0828: 0C1D C9 3A             CMPIM $3A
```

```
0C1F 30 0B               BMI   VALID
0C21 C9 41               CMPIM $41    IGNORE 3A...40
0C23 30 04               BMI   NOTVAL
0C25 C9 47               CMPIM $47    IGNORE 47...7F
0C27 30 03               BMI   VALID

0C29 A0 FF        NOTVAL LDYIM $FF    SET N-FLAG
0C2B 60                  RTS          ERROR EXIT

0C2C C9 40        VALID  CMPIM $40    ASCII HEX CONVERSION
0C2E 30 03               BMI   VAL
0C30 18                  CLC
0C31 69 09               ADCIM $09

0C33 29 0F        VAL    ANDIM $0F    HEX DATA IS LOW ORDER NIBBLE IN ACCU
0C35 60                  RTS

                  *** END OF ASCHEX ***
                  *** RDCH ***


                  READ AN ASCII CHARACTER FROM TAPE AND
                  STORE IT IN ACCU

0C36 A2 08        RDCH   LDXIM $08    SET UP FOR 8 BITS

0C38 20 C2 0B     READ   JSR   RDBIT  READ A BIT FROM TAPE
0C3B 6E 6B 1A            ROR   CHAR   SHIFT BIT INTO CHARACTER
0C3E CA                  DEX          ALL BITS READ?
0C3F D0 F7               BNE   READ
0C41 2E 6B 1A            ROL   CHAR   B7 MUST BE ZERO
0C44 4E 6B 1A            LSR   CHAR
0C47 AD 6B 1A            LDA   CHAR   RECEIVED CHARACTER TO ACCU
0C4A 60                  RTS

                  *** END OF RDCH ***

                  *** CHKSUM ***


                  COMPUTE CHECK SUM OF RECEIVED DATA

0C4B 48           CHKSUM PHA          SAVE ACCU
0C4C 18                  CLC
0C4D 6D 6E 1A            ADC   CHKL   CHK := CHK + BYTE
0C50 8D 6E 1A            STA   CHKL
0C53 AD 6F 1A            LDA   CHKH
0C56 69 00               ADCIM $00
0C58 8D 6F 1A            STA   CHKH
0C5B 68                  PLA          GET ACCU AGAIN
0C5C 60                  RTS

                  *** END OF CHKSUM ***


                  *** CHARVU ***


                  OUTPUT ANY CHARACTER TO 7 SEGMENT DISPLAY
                  CHARACTER CHANGE IS ENABLED/DISABLED

0C5D 48           CHARVU PHA          SAVE ACCU
0C5E 49 7F               EORIM $7F    OUTPUT INVERTED CHAR. TO SEGMENTS
0C60 8D 80 1A            STA   PAD
0C63 68                  PLA          RESTORE ACCU

0C64 48           VU     PHA
0C65 AD 78 1A            LDA   GANG
0C68 49 02               EORIM $02    CHANGE DISPLAYS
```

```
0898: 0C6A 8D 82 1A         STA   PBD
0899: 0C6D 8D 78 1A         STA   GANG   SAVE STATUS QUO
0900: 0C70 68               PLA
0901: 0C71 60               RTS
0902:
0903:                *** END OF CHARVU ***
0904:
0905:
0906:                ADJPNT ADDRESS POINTER ADJUSTMENT
0907: 0C72 38        ADJPNT SEC          16 BIT SUBTRACTION
0908: 0C73 A5 FA            LDAZ  POINTL
0909: 0C75 E9 01            SBCIM $01
0910: 0C77 85 FA            STAZ  POINTL
0911: 0C79 A5 FB            LDAZ  POINTH
0912: 0C7B E9 00            SBCIM $00
0913: 0C7D 85 FB            STAZ  POINTH
0914: 0C7F 60               RTS
0915:
0916:                *** END OF ADJPNT ***
ID=B4
R

X

0001: 1000                  ORG   $1000
0002:
0003:
0004:                *******************************************
0005:                *** MAIN PROGRAM OF THE PRINTER MONITOR ***
0006:                *******************************************
0007:
0008:
0009:                COMMANDS OF THE PRINTER MONITOR:
0010:
0011:                > ´-´  DECREMENT THE CURRENT ADDRESS BY ONE
0012:                > ´+´  INCREMENT THE CURRENT ADDRESS BY ONE
0013:                > ´SPACE´  1) PRINT THE ADDRESS IN THE INPUT BUFFER
0014:                           2) SHOW THE DATA OF THIS ADDRESS
0015:                > ´.´  STORE INPUT DATA AT THE CURRENT ADDRESS
0016:                > ´R´   START PROGRAM EXECUTION AT THE CURRENT ADDRESS
0017:                > ´L´  LIST THE CONTENTS OF ALL CPU-REGISTERS
0018:                > ´P´  PRINT OUT THE LAST PROGRAM CONTER
0019:                > ´M´  PRINT A HEXDUMP SPECIFIED BY THE INPUT PARAMETER
0020:                > ´G´ READ A DATA BLOCK WITH A CERTAIN ID FROM TAPE
0021:                > ´S´ STORE A DATA BLOCK BETWEEN SA AND EA-1 ON TAPE
0022:
0023:
0024:
0025: 1000 D8        INITPR CLD          RESET SEQUENCE OF THE PRINTER PROGRAM
0026: 1001 78               SEI          IRQ LINE IS DISABLED
0027: 1002 A9 67            LDAIM $67
0028: 1004 8D 82 1A         STA   PBD    PBD: 01100111
0029: 1007 A9 00            LDAIM $00
0030: 1009 8D 80 1A         STA   PAD    PAD: 00000000
0031: 100C A2 FE            LDXIM $FE    RESET BIT TIME COUNTER
0032: 100E 8E 5A 1A         STX   CNTL
0033: 1011 E8               INX
0034: 1012 8E 5B 1A         STX   CNTH
0035: 1015 9A               TXS          RESET COMPUTER STACK POINTER
0036: 1016 86 F2            STXZ  SPUSER RESET USER STACK POINTER
0037: 1018 A9 7F            LDAIM $7F
0038: 101A 8D 81 1A         STA   PADD   PADD: 01111111
0039: 101D 8D 83 1A         STA   PBDD   PBDD: 01111111
0040: 1020 A2 02            LDXIM $02
0041: 1022 8E 59 1A         STX   STPBIT TRANSMIT NONE PARITY & ONE STOP BIT
0042: 1025 A9 5F            LDAIM LABJUN
0043: 1027 8D 7C 1A         STA   BRKT   SETUP BREAK VECTOR
0044: 102A A9 10            LDAIM LABJUN /256
0045: 102C 8D 7D 1A         STA   BRKT   +01
0046: 102F A9 CF            LDAIM STEP   SETUP STEP BY STEP VECTOR
```

```
0047: 1031 8D 7A 1A          STA    NMI
0048: 1034 A9 14             LDAIM  STEP    /256
0049: 1036 8D 7B 1A          STA    NMI     +01
0050:
0051: 1039 2C 80 1A   STRTBT BIT    PAD     WAIT FOR A START BIT
0052: 103C 30 FB             BMI    STRTBT
0053: 103E 20 E0 12          JSR    COMTIM  COMPUTE THE START BIT TIME
0054: 1041 4E 5F 1A          LSR    TIMH    DIVIDE BY 2
0055: 1044 6E 5E 1A          ROR    TIML
0056: 1047 AD 5E 1A          LDA    TIML
0057: 104A 8D 5C 1A          STA    CNTHL   SAVE HALF START BIT TIME
0058: 104D AD 5F 1A          LDA    TIMH
0059: 1050 8D 5D 1A          STA    CNTHH
0060: 1053 A2 08             LDXIM  $08
0061: 1055 20 03 13          JSR    DELHBT
0062: 1058 20 BE 12          JSR    RECD    GET THE REST OF THE CHARACTER
0063: 105B C9 7F             CMPIM  $7F     WAS IT THE RUBOUT CHARACTER?
0064: 105D D0 A1             BNE    INITPR
0065:
0066: 105F 20 46 12   LABJUN JSR    JUNIOR  PRINT ´JUNIOR´
0067: 1062 20 E8 11          JSR    CRLF
0068: 1065 A2 FF             LDXIM  $FF
0069: 1067 9A                TXS            RESET STACK POINTER WHEN A BREAK OCCURS
0070: 1068 86 F2             STXZ   SPUSER
0071:
0072: 106A 20 59 12   RESALL JSR    RESPAR  RESET PARAMETER BUFFER
0073: 106D 20 68 12          JSR    RESIN   RESET INPUT BUFFER
0074:
0075: 1070 20 AE 12   READCH JSR    RECCHA  WAIT FOR A CHARACTER
0076:
0077: 1073 C9 2B      PLU    CMPIM  ´+
0078: 1075 D0 09             BNE    MINUS
0079: 1077 20 13 12          JSR    INCPNT  INCREMENT CURRENT ADDRESS
0080: 107A 20 F8 11          JSR    PRBUFS  OPEN NEXT CELL
0081: 107D 4C 6A 10          JMP    RESALL
0082:
0083: 1080 C9 2D      MINUS  CMPIM  ´-
0084: 1082 D0 09             BNE    SPACE
0085: 1084 20 1A 12          JSR    DECPNT  DECREMENT CURRENT ADDRESS
0086: 1087 20 F8 11          JSR    PRBUFS  OPEN PREVIOUS CELL
0087: 108A 4C 6A 10          JMP    RESALL
0088:
0089: 108D C9 20      SPACE  CMPIM  ´
0090: 108F D0 0E             BNE    PNT
0091: 1091 A5 F8             LDAZ   INL     OUTPUT THE ADDRESS
0092: 1093 85 FA             STAZ   POINTL  IN THE INPUT BUFFER
0093: 1095 A5 F9             LDAZ   INH
0094: 1097 85 FB             STAZ   POINTH
0095: 1099 20 F8 11          JSR    PRBUFS  OPEN CURRENT CELL
0096: 109C 4C 6A 10          JMP    RESALL
0097:
0098: 109F C9 2E      PNT    CMPIM  ´.
0099: 10A1 D0 0F             BNE    RUN
0100: 10A3 A5 F8             LDAZ   INL
0101: 10A5 A0 00             LDYIM  $00
0102: 10A7 91 FA             STAIY  POINTL  STORE CURRENT DATA BYTE IN MEMORY
0103: 10A9 20 13 12          JSR    INCPNT
0104: 10AC 20 F8 11          JSR    PRBUFS  OPEN NEXT CELL
0105: 10AF 4C 6A 10          JMP    RESALL
0106:
0107: 10B2 C9 52      RUN    CMPIM  ´R
0108: 10B4 D0 13             BNE    LIST
0109: 10B6 A6 F2             LDXZ   SPUSER  START PROGRAM EXECUTION
0110: 10B8 9A                TXS            AT THE CURRENT DISPLAYED ADDRESS
0111: 10B9 A5 FB             LDAZ   POINTH
0112: 10BB 48                PHA
0113: 10BC A5 FA             LDAZ   POINTL
0114: 10BE 48                PHA
0115: 10BF A5 F1             LDAZ   PREG
```

```
0116: 10C1 48               PHA
0117: 10C2 A6 F5            LDXZ  XREG
0118: 10C4 A4 F4            LDYZ  YREG
0119: 10C6 A5 F3            LDAZ  ACC
0120: 10C8 40               RTI
0121:
0122: 10C9 C9 4C      LIST  CMPIM ´L
0123: 10CB D0 52            BNE   PC
0124: 10CD A0 14            LDYIM $14
0125: 10CF 20 D6 11         JSR   MESSY  ACC:
0126: 10D2 A5 F3            LDAZ  ACC
0127: 10D4 20 8F 12         JSR   PRBYT
0128: 10D7 A0 1A            LDYIM $1A
0129: 10D9 20 D6 11         JSR   MESSY  Y  :
0130: 10DC A5 F4            LDAZ  YREG
0131: 10DE 20 8F 12         JSR   PRBYT
0132: 10E1 A0 20            LDYIM $20
0133: 10E3 20 D6 11         JSR   MESSY  X  :
0134: 10E6 A5 F5            LDAZ  XREG
0135: 10E8 20 8F 12         JSR   PRBYT
0136: 10EB A0 26            LDYIM $26
0137: 10ED 20 D6 11         JSR   MESSY  PC :
0138: 10F0 A5 F0            LDAZ  PCH
0139: 10F2 20 8F 12         JSR   PRBYT
0140: 10F5 A5 EF            LDAZ  PCL
0141: 10F7 20 8F 12         JSR   PRBYT
0142: 10FA A0 2C            LDYIM $2C
0143: 10FC 20 D6 11         JSR   MESSY  SP :
0144: 10FF A9 01            LDAIM $01
0145: 1101 20 8F 12         JSR   PRBYT
0146: 1104 A5 F2            LDAZ  SPUSER
0147: 1106 20 8F 12         JSR   PRBYT
0148: 1109 A0 32            LDYIM $32
0149: 110B 20 D6 11         JSR   MESSY  PR :
0150: 110E 20 28 12         JSR   SHOWPR PRINT OUT FLAGS
0151: 1111 A0 38            LDYIM $38
0152: 1113 20 D6 11         JSR   MESSY  NV BDIZC
0153: 1116 20 F3 11         JSR   PRSP
0154: 1119 4C 6A 10         JMP   RESALL
0155:
0156: 111C 4C B0 11   CONTIN JMP  GETTAP
0157:
0158:
0159: 111F C9 50      PC    CMPIM ´P
0160: 1121 D0 0E            BNE   MATRIX
0161: 1123 A5 EF            LDAZ  PCL    RESTORE LAST PROGRAM COUNTER
0162: 1125 85 FA            STAZ  POINTL
0163: 1127 A5 F0            LDAZ  PCH
0164: 1129 85 FB            STAZ  POINTH
0165: 112B 20 F8 11         JSR   PRBUFS OPEN CURRENT CELL
0166: 112E 4C 6A 10         JMP   RESALL
0167:
0168: 1131 C9 4D      MATRIX CMPIM ´M
0169: 1133 D0 E7            BNE   CONTIN
0170: 1135 A0 52            LDYIM $52
0171: 1137 20 D6 11         JSR   MESSY  HEXDUMP:
0172: 113A 20 87 13         JSR   INPAR  READ PARAMETERS
0173: 113D 10 03            BPL   MATF
0174:
0175: 113F 4C 5F 10   MATD  JMP   LABJUN RETURN, IF INVALID CHARACTER
0176:
0177: 1142 38         MATF  SEC          VALID INPUT PARAMETERS?
0178: 1143 AD 65 1A         LDA   PARBL
0179: 1146 ED 63 1A         SBC   PARAL  PARA < PARB?
0180: 1149 AD 66 1A         LDA   PARBH
0181: 114C ED 64 1A         SBC   PARAH
0182: 114F 90 EE            BCC   MATD
0183: 1151 20 E8 11         JSR   CRLF
0184: 1154 A2 06            LDXIM $06
```

```

1156 20 F3 11  MATG   JSR   PRSP
1159 CA               DEX
115A D0 FA            BNE   MATG
115C A0 00            LDYIM $00    RESET COLUMN COUNTER

115E 98        MATH   TYA
115F 20 9B 12         JSR   PRNIBL OUTPUT COLUMNS
1162 20 F3 11         JSR   PRSP
1165 20 F3 11         JSR   PRSP
1168 C8               INY
1169 C0 10            CPYIM $10    PRINT COLUMS 0...F
116B D0 F1            BNE   MATH
116D AD 63 1A         LDA   PARAL
1170 85 FA            STAZ  POINTL SETUP MATRIX POINTER
1172 AD 64 1A         LDA   PARAH
1175 85 FB            STAZ  POINTH

1177 20 E8 11  MATJ   JSR   CRLF
117A A2 10            LDXIM $10
117C A5 FB            LDAZ  POINTH
117E 20 8F 12         JSR   PRBYT  OUTPUT CURRENT MATRIX ADDRESS
1181 A5 FA            LDAZ  POINTL
1183 20 8F 12         JSR   PRBYT
1186 A0 17            LDYIM $17
1188 20 D9 11         JSR   ME

118B 38        MATK   SEC
118C AD 65 1A         LDA   PARBL  HEXDUMP FINISHED?
118F E5 FA            SBCZ  POINTL
1191 AD 66 1A         LDA   PARBH
1194 E5 FB            SBCZ  POINTH
1196 B0 06            BCS   MATL
1198 20 E8 11         JSR   CRLF
119B 4C 5F 10         JMP   LABJUN HEXDUMP IS FINISHED

119E A0 00     MATL   LDYIM $00
11A0 B1 FA            LDAIY POINTL FETCH CURRENT DATA BYTE
11A2 20 8F 12         JSR   PRBYT  OUTPUT CURRENT DATA BYTE
11A5 20 F3 11         JSR   PRSP
11A8 20 13 12         JSR   INCPNT
11AB CA               DEX
11AC D0 DD            BNE   MATK
11AE F0 C7            BEQ   MATJ

11B0 C9 47     GETTAP CMPIM ´G
11B2 D0 0B            BNE   SAVID
11B4 20 8A 14         JSR   GETID  READ DATA FROM TAPE SPEC. BY ID
11B7 30 03            BMI   GETERR ILLEGAL CHARACTER?
11B9 4C 6A 10         JMP   RESALL NORMAL EXIT

11BC 4C 5F 10  GETERR JMP   LABJUN

11BF C9 53     SAVID  CMPIM ´S
11C1 D0 08            BNE   VALNUM NO COMMAND KEY WAS DEPRESSED
11C3 20 3B 14         JSR   SAID   STORE DATA ON TAPE SPECIFIED BY THE PARAMETERS
11C6 30 F4            BMI   GETERR ILLEGAL PARAMETER WAS ENTERED
11C8 4C 6A 10         JMP   RESALL

11CB 20 6F 12  VALNUM JSR   HEXNUM INPUT DATA TO BUFFER
11CE D0 03            BNE   VNA
11D0 4C 70 10         JMP   READCH

11D3 4C 6A 10  VNA    JMP   RESALL

               ******************************************
               *** SUBROUTINES OF THE PRINTER MONITOR ***
               ******************************************

```

```

                      MESSY PRINTS A MESSAGE, POINTED BY Y REGISTER

11D6 20 E8 11         MESSY  JSR   CRLF   PRINT A CR&LF

11D9 B9 BD 13         ME     LDAY  MESS   LOAD CHARACTERS
11DC C9 03                   CMPIM $03    ETX CHARACTER ?
11DE F0 07                   BEQ   MESEND
11E0 20 34 13                JSR   PRCHA  CHARACTER TO TTY
11E3 C8                      INY
11E4 4C D9 11                JMP   ME

11E7 60               MESEND RTS


                      CRLF PRINT CARRIAGE RETURN & LINE FEED
                      PRSP PRINT A SPACE

11E8 A9 0D            CRLF   LDAIM $0D
11EA 20 34 13                JSR   PRCHA  OUTPUT CR
11ED A9 0A                   LDAIM $0A

11EF 20 34 13         CLEND  JSR   PRCHA  OUTPUT LF
11F2 60                      RTS

11F3 A9 20            PRSP   LDAIM $20
11F5 4C EF 11                JMP   CLEND  OUTPUT A SPACE


                      PRBUFS OUTPUT ADDRESS AND DATA

11F8 20 E8 11         PRBUFS JSR   CRLF
11FB A5 FB                   LDAZ  POINTH
11FD 20 8F 12                JSR   PRBYT  OUTPUT HIGH ORDER ADDR
1200 A5 FA                   LDAZ  POINTL
1202 20 8F 12                JSR   PRBYT  OUTPUT LOW ORDER ADDR
1205 20 F3 11                JSR   PRSP
1208 A0 00                   LDYIM $00
120A B1 FA                   LDAIY POINTL FETCH DATA FROM MEMORY
120C 20 8F 12                JSR   PRBYT
120F 20 F3 11                JSR   PRSP
1212 60                      RTS


                      INCPNT INCRMENT ADDR POINTER BY ONE

1213 E6 FA            INCPNT INCZ  POINTL
1215 D0 02                   BNE   IP
1217 E6 FB                   INCZ  POINTH

1219 60               IP     RTS


                      DECPNT DECREMENT ADDR POINTER BY ONE

121A 38               DECPNT SEC
121B A5 FA                   LDAZ  POINTL 16 BIT SUBTRACTION
121D E9 01                   SBCIM $01
121F 85 FA                   STAZ  POINTL
1221 A5 FB                   LDAZ  POINTH
1223 E9 00                   SBCIM $00
1225 85 FB                   STAZ  POINTH
1227 60                      RTS


                      SHOWPR SHOW THE CONTENTS OF THE P REGISTER

1228 A5 F1            SHOWPR LDAZ  PREG
122A 8D 67 1A                STA   PRTEMP
```

```
122D A2 08            LDXIM $08    BIT COUNTER

122F 0E 67 1A  SPRA   ASL   PRTEMP SHIFT OUT THE BITS
1232 90 09            BCC   SPRB   IS IT A 0 OR 1 ?
1234 A9 01            LDAIM $01
1236 20 9B 12         JSR   PRNIBL PRINT A ´1´
1239 CA               DEX
123A D0 F3            BNE   SPRA   ALL BITS PRINTED ?
123C 60               RTS

123D A9 00     SPRB   LDAIM $00
123F 20 9B 12         JSR   PRNIBL PRINT A ´0´
1242 CA               DEX
1243 D0 EA            BNE   SPRA   ALL BITS PRINTED ?
1245 60               RTS


               JUNIOR PRINT ´JUNIOR´
               EDITOR PRINT ´EDITOR´
               ASSEM  PRINT ´ASSEMBLER´

1246 A0 00     JUNIOR LDYIM $00

1248 20 D6 11  JUN    JSR   MESSY
124B 20 E8 11         JSR   CRLF
124E 60               RTS

124F A0 07     EDITOR LDYIM $07
1251 4C 48 12         JMP   JUN

1254 A0 0E     ASSEM  LDYIM $0E
1256 4C 48 12         JMP   JUN


               RESET SUBROUTINES

1259 A0 00     RESPAR LDYIM $00
125B 8C 63 1A         STY   PARAL
125E 8C 64 1A         STY   PARAH
1261 8C 65 1A         STY   PARBL
1264 8C 66 1A         STY   PARBH
1267 60               RTS

1268 A0 00     RESIN  LDYIM $00
126A 84 F8            STYZ  INL
126C 84 F9            STYZ  INH
126E 60               RTS

               HEXNUM >CONVERT AN ASCII CHAR. TO A HEX NIBBLE
                      >SHIFT HEX DATA NIBBLE INTO BUFFER
                      >PRINT ´WHAT?´ IF CHARACTER WAS NOT VALID
                      >RETURN WITH Z=1, IF VALID CHARACTER
                      >RETURN WITH N=1, IF NOT VALID CHARACTER

126F 20 1E 14  HEXNUM JSR   ASHETT ASCII HEX CONVERSION
1272 30 10            BMI   HNUB   NOT VALID ?
1274 A2 04            LDXIM $04    SET NIBBLE COUNTER

1276 06 F8     HNUA   ASLZ  INL
1278 26 F9            ROLZ  INH
127A CA               DEX
127B D0 F9            BNE   HNUA
127D 05 F8            ORAZ  INL    NIBBLE TO INPUT BUFFER
127F 85 F8            STAZ  INL
1281 A0 00            LDYIM $00    SET Z FLAG
1283 60               RTS

1284 A0 46     HNUB   LDYIM $46
1286 20 D6 11         JSR   MESSY  ´WHAT?´
```

```
1289 20 E8 11          JSR    CRLF
128C A0 FF             LDYIM  $FF     SET N FLAG
128E 60                RTS


               PRBYT  >CONVERT AN BYTE STORED IN ACCU TO
                       TWO ASCII CHARACTERS AND PRINT THEM

128F 48        PRBYT   PHA            SAVE BYTE
1290 4A                LSRA           GET UPPER NIBBLE
1291 4A                LSRA
1292 4A                LSRA
1293 4A                LSRA
1294 20 A4 12          JSR    NIBASC  NIBBLE TO ASCII CONVERSION
1297 20 34 13          JSR    PRCHA   PRINT UPPER NIBBLE
129A 68                PLA            GET BYTE AGAIN

129B 29 0F     PRNIBL  ANDIM  $0F     GET LOWER NIBBLE
129D 20 A4 12          JSR    NIBASC
12A0 20 34 13          JSR    PRCHA   PRINT LOWER NIBBLE
12A3 60                RTS


               NIBASC >CONVERT A HEX DATA NIBBLE TO AN ASCII CHARACTER

12A4 C9 0A     NIBASC  CMPIM  $0A     NUMBER OR LETTER ?
12A6 18                CLC
12A7 30 02             BMI    NA
12A9 69 07             ADCIM  $07

12AB 69 30     NA      ADCIM  $30
12AD 60                RTS


               RECCHA >RECEIVE 1 ASCII CHARACTER FROM PRINTER
                      >RETURN WITH ASCII CHARACTER IN ACCU
                      >SAVE X REGISTER

12AE 2C 80 1A  RECCHA  BIT    PAD     WAIT FOR START BIT
12B1 30 FB             BMI    RECCHA
12B3 8E 61 1A          STX    TEMPB   SAVE INDEX X
12B6 A2 08             LDXIM  $08     BIT COUNTER IS 8
12B8 20 03 13          JSR    DELHBT  DELAY HALF BIT TIME

12BB 20 12 13  RECA    JSR    DELBIT  DELAY ONE BIT TIME

12BE 2C 80 1A  RECD    BIT    PAD     ONE/ZERO CHECK
12C1 10 0A             BPL    RECB    BRANCH IF ZERO
12C3 38                SEC            BIT IS ´1´
12C4 6E 62 1A          ROR    CHA     ROTATE CARRY INTO CHARACTER
12C7 CA                DEX            SET UP FOR NEXT BIT
12C8 D0 F1             BNE    RECA    ALL BITS READ?
12CA 4C D4 12          JMP    RECC

12CD 18        RECB    CLC            BIT IS ´0´
12CE 6E 62 1A          ROR    CHA
12D1 CA                DEX
12D2 D0 E7             BNE    RECA

12D4 20 12 13  RECC    JSR    DELBIT  WAIT FOR STOP BIT TIME
12D7 AD 62 1A          LDA    CHA
12DA 29 7F             ANDIM  $7F     BIT 7 MUST BE ZERO
12DC AE 61 1A          LDX    TEMPB   RESTORE INDEX X
12DF 60                RTS


               COMTIM >COMPUTE BIT TIME

12E0 18        COMTIM  CLC
```

```
0461: 12E1 AD 5A 1A        LDA   CNTL   16 BIT ADD
0462: 12E4 69 01           ADCIM $01
0463: 12E6 8D 5A 1A        STA   CNTL
0464: 12E9 AD 5B 1A        LDA   CNTH
0465: 12EC 69 00           ADCIM $00
0466: 12EE 8D 5B 1A        STA   CNTH
0467: 12F1 2C 80 1A        BIT   PAD    START BIT FINISHED ?
0468: 12F4 10 EA           BPL   COMTIM
0469: 12F6 AD 5A 1A        LDA   CNTL   SET UP FOR
0470: 12F9 8D 5E 1A        STA   TIML   HALF BIT TIME COMPUTATION
0471: 12FC AD 5B 1A        LDA   CNTH
0472: 12FF 8D 5F 1A        STA   TIMH
0473: 1302 60              RTS
0474:
0475:
0476:               DELBIT >DELAY 1 BIT TIME
0477:               DELHBT >DELAY 1/2 BIT TIME
0478:
0479: 1303 AD 5C 1A DELHBT LDA   CNTHL  FETCH 1/2 BIT TIME
0480: 1306 8D 5E 1A        STA   TIML
0481: 1309 AD 5D 1A        LDA   CNTHH
0482: 130C 8D 5F 1A        STA   TIMH
0483: 130F 4C 1E 13        JMP   CNTDN  START WITH BIT COUNT DOWN
0484:
0485: 1312 AD 5A 1A DELBIT LDA   CNTL   FETCH 1 BIT TIME
0486: 1315 8D 5E 1A        STA   TIML
0487: 1318 AD 5B 1A        LDA   CNTH
0488: 131B 8D 5F 1A        STA   TIMH
0489:
0490: 131E 38       CNTDN  SEC
0491: 131F AD 5E 1A        LDA   TIML   16 BIT SUBTRACTION
0492: 1322 E9 01           SBCIM $01    COUNT DOWN
0493: 1324 8D 5E 1A        STA   TIML
0494: 1327 AD 5F 1A        LDA   TIMH
0495: 132A E9 00           SBCIM $00
0496: 132C 8D 5F 1A        STA   TIMH
0497: 132F EA              NOP          EQUALIZE 4 MICRO SEC
0498: 1330 EA              NOP
0499: 1331 B0 EB           BCS   CNTDN  COUNT DOWN FINISHED ?
0500: 1333 60              RTS
0501:
0502:               PRCHA >TRANSMIT AN ASCII CHARACTER STORED IN
0503:                      ACCU TO PRINTER
0504:                     >SAVE INDEX X
0505:
0506: 1334 8E 60 1A PRCHA  STX   TEMPA  SAVE INDEX X
0507: 1337 8D 62 1A        STA   CHA
0508: 133A AD 82 1A        LDA   PBD
0509: 133D 29 FE           ANDIM $FE    TRNSMIT START BIT
0510: 133F 8D 82 1A        STA   PBD
0511: 1342 20 12 13        JSR   DELBIT DELAY OF START BIT
0512: 1345 A2 07           LDXIM $07    SET UP FOR 7 DATA BITS
0513:
0514: 1347 4E 62 1A PRA    LSR   CHA    SHIFT OUT CHARACTER
0515: 134A 90 30           BCC   PRC    BRANCH IF ´0´
0516: 134C AD 82 1A        LDA   PBD
0517: 134F 09 01           ORAIM $01    OUTPUT A LOG ´1´
0518: 1351 8D 82 1A        STA   PBD
0519:
0520: 1354 20 12 13 PRB    JSR   DELBIT DELAY 1 BIT TIME
0521: 1357 CA              DEX          SET UP FOR NEXT BIT
0522: 1358 D0 ED           BNE   PRA    ALL BITS TRANSMITTED ?
0523: 135A AE 59 1A        LDX   STPBIT X := AMOUNT OF STOP BITS + 1
0524:
0525: 135D AD 82 1A PRD    LDA   PBD
0526: 1360 09 01           ORAIM $01    FIRST NONE PARITY
0527: 1362 8D 82 1A        STA   PBD    AND THEN 1 STOP BIT
0528: 1365 20 12 13        JSR   DELBIT
0529: 1368 CA              DEX
```

```
0530: 1369 D0 F2           BNE   PRD
0531: 136B 2C 80 1A        BIT   PAD    TEST FOR BREAK
0532: 136E 10 04           BPL   BRKTST
0533: 1370 AE 60 1A        LDX   TEMPA  RESTORE INDEX X
0534: 1373 60              RTS
0535:
0536: 1374 2C 80 1A BRKTST BIT   PAD    KEY RELEASED ?
0537: 1377 10 FB           BPL   BRKTST
0538: 1379 6C 7C 1A        JMI   BRKT   JUMP TO AN USER SELECTABLE VECTOR
0539:
0540: 137C AD 82 1A PRC    LDA   PBD
0541: 137F 29 FE           ANDIM $FE    OUTPUT A LOG ´O´
0542: 1381 8D 82 1A        STA   PBD
0543: 1384 4C 54 13        JMP   PRB
0544:
0545:                INPAR >PARAMETER INPUT OF MATRIX
0546:
0547: 1387 20 AE 12 INPAR  JSR   RECCHA WAIT FOR A CHARACTER
0548: 138A C9 2C           CMPIM  ,     IS IT A COLON ?
0549: 138C F0 07           BEQ   IPA
0550: 138E 20 6F 12        JSR   HEXNUM FILLUP INPUT BUFFER
0551: 1391 30 29           BMI   IPD    RETURN, IF NOT VALID
0552: 1393 F0 F2           BEQ   INPAR  ELSE CONTINUE
0553:
0554: 1395 A5 F8    IPA    LDAZ  INL    INPUT TO PARAMETER BUFFER
0555: 1397 8D 63 1A        STA   PARAL
0556: 139A A5 F9           LDAZ  INH
0557: 139C 8D 64 1A        STA   PARAH
0558: 139F 20 68 12        JSR   RESIN
0559:
0560: 13A2 20 AE 12 IPB    JSR   RECCHA WAIT FOR A CHARACTER
0561: 13A5 C9 0D           CMPIM $0D    WAS IT A CARRIAGE RETURN ?
0562: 13A7 F0 07           BEQ   IPC
0563: 13A9 20 6F 12        JSR   HEXNUM
0564: 13AC 30 0E           BMI   IPD    VALID CHARACTER ?
0565: 13AE F0 F2           BEQ   IPB
0566:
0567: 13B0 A5 F9    IPC    LDAZ  INH    INPUT TO PARAMETER BUFFER
0568: 13B2 8D 66 1A        STA   PARBH
0569: 13B5 A5 F8           LDAZ  INL
0570: 13B7 8D 65 1A        STA   PARBL
0571: 13BA A0 00           LDYIM $00
0572:
0573: 13BC 60       IPD    RTS
0574:
0575:
0576:               STRING LOOKUP TABLE
0577:
0578: 13BD 4A       MESS   =     ´J     Y = 00
0579: 13BE 55              =     ´U
0580: 13BF 4E              =     ´N
0581: 13C0 49              =     ´I
0582: 13C1 4F              =     ´O
0583: 13C2 52              =     ´R
0584: 13C3 03              =     $03
0585: 13C4 45              =     ´E     Y = 07
0586: 13C5 44              =     ´D
0587: 13C6 49              =     ´I
0588: 13C7 54              =     ´T
0589: 13C8 4F              =     ´O
0590: 13C9 52              =     ´R
0591: 13CA 03              =     $03
0592: 13CB 41              =     ´A     Y = 0E
0593: 13CC 53              =     ´S
0594: 13CD 53              =     ´S
0595: 13CE 45              =     ´E
0596: 13CF 4D              =     ´M
0597: 13D0 03              =     $03
0598: 13D1 41              =     ´A     Y = 14
```

```
0599: 13D2 43            =      ´C
0600: 13D3 43            =      ´C
0601: 13D4 3A            =      ´:        Y = 17
0602: 13D5 20            =      ´
0603: 13D6 03            =      $03
0604: 13D7 59            =      ´Y        Y = 1A
0605: 13D8 20            =      ´
0606: 13D9 20            =      ´
0607: 13DA 3A            =      ´:
0608: 13DB 20            =      ´
0609: 13DC 03            =      $03
0610: 13DD 58            =      ´X        Y = 20
0611: 13DE 20            =      ´
0612: 13DF 20            =      ´
0613: 13E0 3A            =      ´:
0614: 13E1 20            =      ´
0615: 13E2 03            =      $03
0616: 13E3 50            =      ´P        Y = 26
0617: 13E4 43            =      ´C
0618: 13E5 20            =      ´
0619: 13E6 3A            =      ´:
0620: 13E7 20            =      ´
0621: 13E8 03            =      $03
0622: 13E9 53            =      ´S        Y = 2C
0623: 13EA 50            =      ´P
0624: 13EB 20            =      ´
0625: 13EC 3A            =      ´:
0626: 13ED 20            =      ´
0627: 13EE 03            =      $03
0628: 13EF 50            =      ´P        Y = 32
0629: 13F0 52            =      ´R
0630: 13F1 20            =      ´
0631: 13F2 3A            =      ´:
0632: 13F3 20            =      ´
0633: 13F4 03            =      $03
0634: 13F5 20            =      ´          Y = 38
0635: 13F6 20            =      ´
0636: 13F7 20            =      ´
0637: 13F8 20            =      ´
0638: 13F9 20            =      ´
0639: 13FA 4E            =      ´N
0640: 13FB 56            =      ´V
0641: 13FC 20            =      ´
0642: 13FD 42            =      ´B
0643: 13FE 44            =      ´D
0644: 13FF 49            =      ´I
0645: 1400 5A            =      ´Z
0646: 1401 43            =      ´C
0647: 1402 03            =      $03
0648: 1403 57            =      ´W        Y = 46
0649: 1404 48            =      ´H
0650: 1405 41            =      ´A
0651: 1406 54            =      ´T
0652: 1407 3F            =      ´?
0653: 1408 03            =      $03
0654: 1409 52            =      ´R        Y = 4C
0655: 140A 45            =      ´E
0656: 140B 41            =      ´A
0657: 140C 44            =      ´D
0658: 140D 59            =      ´Y
0659: 140E 03            =      $03
0660: 140F 48            =      ´H        Y = 52
0661: 1410 45            =      ´E
0662: 1411 58            =      ´X
0663: 1412 44            =      ´D
0664: 1413 55            =      ´U
0665: 1414 4D            =      ´M
0666: 1415 50            =      ´P
0667: 1416 3A            =      ´:
```

```
1417 20               =
1418 03               =     $03
1419 53               =     ´S     Y = 5C
141A 41               =     ´A
141B 3A               =     ´:
141C 20               =
141D 03               =     $03

               ASHETT = ASCHEX


               CONVERT AN ASCII CHARACTER TO A HEX DATA NIBBLE.

               1) RETURN WITH CONVERTED HEX NUMBER IN ACCU
               2) N = 1, IF NOT VALID HEX NUMBER
               3) Z = 1, IF VALID HEX NUMBER

141E C9 30     ASHETT CMPIM $30    IGNORE 00...2F
1420 30 0C            BMI   NOTVAT
1422 C9 3A            CMPIM $3A
1424 30 0B            BMI   VALIT
1426 C9 41            CMPIM $41    IGNORE 3A...40
1428 30 04            BMI   NOTVAT
142A C9 47            CMPIM $47    IGNORE 47...7F
142C 30 03            BMI   VALIT

142E A0 FF     NOTVAT LDYIM $FF    SET N-FLAG
1430 60               RTS          ERROR EXIT

1431 C9 40     VALIT  CMPIM $40    ASCII HEX CONVERSION
1433 30 03            BMI   VALT
1435 18               CLC
1436 69 09            ADCIM $09

1438 29 0F     VALT   ANDIM $0F
143A 60               RTS

143B 20 AE 12  SAID   JSR   RECCHA WAIT FOR A CHARACTER
143E C9 2C            CMPIM ´,
1440 F0 07            BEQ   SIC    IT WAS A DELIMITER
1442 20 6F 12         JSR   HEXNUM READ PARAMETER = ID
1445 30 3F            BMI   SIB
1447 F0 F2            BEQ   SAID

1449 A5 F8     SIC    LDAZ  INL    SAVE ID
144B 8D 79 1A         STA   ID
144E C9 00            CMPIM $00    ID = 00 & FF ARE NOT VALID
1450 F0 35            BEQ   SIA
1452 C9 FF            CMPIM $FF
1454 F0 31            BEQ   SIA
1456 20 68 12         JSR   RESIN  RESET INPUT BUFFER
1459 20 87 13         JSR   INPAR  READ SA AND EA
145C 30 28            BMI   SIB    NOT VALID CHARACTER
145E AD 63 1A         LDA   PARAL  SAVE ALL PARAMETERS
1461 8D 70 1A         STA   SAL
1464 AD 64 1A         LDA   PARAH
1467 8D 71 1A         STA   SAH
146A AD 65 1A         LDA   PARBL
146D 8D 72 1A         STA   EAL
1470 AD 66 1A         LDA   PARBH
1473 8D 73 1A         STA   EAH
1476 20 DF 09         JSR   DUMP   STORE DATA ON TAPE
1479 20 BC 14         JSR   RESTTY I/O RESET
147C A0 4C            LDYIM $4C    ´READY´
147E 20 D6 11         JSR   MESSY
1481 20 E8 11         JSR   CRLF
1484 A0 00            LDYIM $00    NORMAL EXIT

1486 60        SIB    RTS
```

```

1487 A0 FF      SIA    LDYIM $FF    ERROR EXIT
1489 60                RTS


148A 20 A2 13   GETID  JSR   IPB    READ ID
148D 30 17             BMI   GA     NOT VALID PARAMETER
148F 8D 79 1A          STA   ID     SAVE ID
1492 C9 FF             CMPIM $FF    SPECIAL ID ?
1494 F0 11             BEQ   IDPAR

1496 20 02 0B   GB     JSR   RDTAPE READ DATA FROM TAPE
1499 20 BC 14          JSR   RESTTY I/O RESET
149C A0 4C             LDYIM $4C    ´READY´
149E 20 D6 11          JSR   MESSY
14A1 20 E8 11          JSR   CRLF
14A4 A0 00             LDYIM $00    NORMAL EXIT

14A6 60         GA     RTS

14A7 A0 5C      IDPAR  LDYIM $5C    ´SA:
14A9 20 D6 11          JSR   MESSY
14AC 20 EB 14          JSR   IPBRES READ A SPECIAL ID
14AF 30 F5             BMI   GA     NOT VALID PARAMETER
14B1 8D 70 1A          STA   SAL    SAVE START ADDRESS
14B4 A5 F9             LDAZ  INH
14B6 8D 71 1A          STA   SAH
14B9 4C 96 14          JMP   GB



14BC A9 67      RESTTY LDAIM $67
14BE 8D 82 1A          STA   PBD
14C1 A9 00             LDAIM $00
14C3 8D 80 1A          STA   PAD
14C6 A9 7F             LDAIM $7F
14C8 8D 81 1A          STA   PADD
14CB 8D 83 1A          STA   PBDD
14CE 60                RTS
                ***************************************************
                *** STEP BY STEP PROGRAM OF THE PRINTER MONITOR ***
                ***************************************************


                > THE NMI VECTOR FOR STEP BY STEP IS SET AUTOMATICALLY
                > STEP SWITCH: ON POSITION
                > K4 AND K5 DISABLE THE SYNC SIGNAL OF THE PROCESSOR
                > A HARDWARE MODIFICATION IS REQIRED (SEE BOOK 3)


14CF 85 F3      STEP   STAZ  ACC    SAVE ACCU
14D1 68                PLA          GET PREG
14D2 85 F1             STAZ  PREG
14D4 68                PLA          GET PCL
14D5 85 EF             STAZ  PCL
14D7 85 FA             STAZ  POINTL PC OF THE NEXT INSTRUCTION
14D9 68                PLA          GET PCH
14DA 85 F0             STAZ  PCH
14DC 85 FB             STAZ  POINTH
14DE 84 F4             STYZ  YREG
14E0 86 F5             STXZ  XREG
14E2 BA                TSX          GET OLD STACK POINTER
14E3 86 F2             STXZ  SPUSER AND SAVE IT
14E5 20 F8 11          JSR   PRBUFS OPEN NEXT CELL
14E8 4C 6A 10          JMP   RESALL BACK TO MONITOR



```

```
0805:               *** SPECIAL ID ***
0806:
0807: 14EB A9 00     IPBRES LDAIM $00
0808: 14ED 85 F8            STAZ  INL     RESET INPUT BUFFER
0809: 14EF 85 F9            STAZ  INH
0810: 14F1 4C A2 13         JMP   IPB     CONTINUE
```

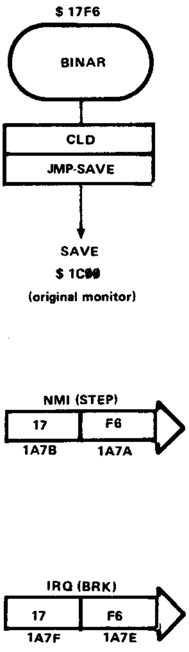
```
SYMBOL TABLE 3000 3594
ACC    00F3   ADJPNT 0C72   ASCHEX 0C19   ASHETT 141E
ASSEM  1254   BEGADH 00E3   BEGADL 00E2   BEGIN  1ED3
BITS   1A75   BRKT   1A7C   BRKTST 1374   BTWEEN 0BE8
BYTE   1A6A   CENDH  00E9   CENDL  00E8   CHAR   1A6B
CHARVU 0C5D   CHA    1A62   CHECK  0B8A   CHKH   1A6F
CHKID  0B9E   CHKL   1A6E   CHKSUM 0C4B   CLEND  11EF
CNTA   1AF4   CNTC   1AF6   CNTDN  131E   CNTH   1A5B
CNTHH  1A5D   CNTHL  1A5C   CNTL   1A5A   COLDST 1CB5
COMPNT 0949   COMTIM 12E0   CONTIN 111C   CRLF   11E8
DATATR 0A40   DAT    089C   DECPNT 121A   DELBIT 1312
DELHBT 1303   DELY   0927   DISCNT 1A68   DUMP   09DF
DUMPT  09F3   EAH    1A73   EAL    1A72   EDITOR 124F
ENDADH 00E5   ENDADL 00E4   FILES  0873   FILMEM 0B70
FIRST  1A76   FMA    0B84   GA     14A6   GANG   1A78
GB     1496   GETA   0800   GETB   084E   GETERR 11BC
GETID  148A   GETKEY 1DF9   GETTAP 11B0   GET    0840
HEXDAT 0A6A   HEXNUM 126F   HIGH   0AC8   HIGHER 1A6C
HIG    0ACB   HNUA   1276   HNUB   1284   ID     1A79
IDPAR  14A7   INCPNT 1213   INH    00F9   INITPR 1000
INL    00F8   INPAR  1387   IP     1219   IPA    1395
IPBRES 14EB   IPB    13A2   IPC    13B0   IPD    13BC
JUNIOR 1246   JUN    1248   LABJUN 105F   LDAINH 1DA7
LIST   10C9   LO     0AE8   LOWER  1A6D   LOW    0AE5
MATD   113F   MATF   1142   MATG   1156   MATH   115E
MATJ   1177   MATK   118B   MATL   119E   MATRIX 1131
ME     11D9   MESEND 11E7   MESS   13BD   MESSY  11D6
MINUS  1080   NA     12AB   NIBASC 12A4   NIBOUT 0A9A
NIB    0AA1   NMI    1A7A   NOTVAL 0C29   NOTVAT 142E
ONE    0AA8   OUTBT  0A8B   OUTBTC 0A7A   OUTCH  0AA3
PADD   1A81   PAD    1A80   PARAH  1A64   PARAL  1A63
PARBH  1A66   PARBL  1A65   PBDD   1A83   PBD    1A82
PC     111F   PCH    00F0   PCL    00EF   PLUS   085E
PLU    1073   PNT    109F   POINTH 00FB   POINTL 00FA
PRA    1347   PRBUFS 11F8   PRBYT  128F   PRB    1354
PRCHA  1334   PRC    137C   PRD    135D   PREG   00F1
PRNIBL 129B   PRSP   11F3   PRTEMP 1A67   RBA    0BFA
RBB    0BFB   RDBA   0BD1   RDBB   0BD4   RDBC   0BE4
RDBIT  0BC2   RDBYT  0BF3   RDCH   0C36   RDFLAG 1AD5
RDSA   0B60   RDTAPE 0B02   RDTDIS 1AD4   READ   0C38
READCH 1070   RECA   12BB   RECB   12CD   RECC   12D4
RECCHA 12AE   RECD   12BE   RESALL 106A   RESET  1C1D
RESIN  1268   RESPAR 1259   RESTTY 14BC   RUN    10B2
SAH    1A71   SAID   143B   SAL    1A70   SAVE   0852
SAVID  11BF   SBEGH  0989   SBEGL  0997   SEAH   096B
SEAL   097A   SECOND 1A77   SENDH  09A5   SENDL  09B3
SHIFT  08A3   SHOWPR 1228   SIA    1487   SIB    1486
SIC    1449   SID    093B   SPACE  108D   SPRA   122F
SPRB   123D   SPUSER 00F2   SSAH   094D   SSAL   095C
STAR   0B45   STARA  0B55   STBEGH 08EC   STBEGL 08F7
STEAH  08D4   STEAL  08E0   STENDH 0902   STENDL 090D
STEP   14CF   STPBIT 1A59   STRTBT 1039   STSAH  08BC
STSAL  08C8   SY     1A69   SYNC   0B1C   SYNCA  0B21
SYNCNT 1A74   SYNCS  0A1F   SYNVEC 0B9B   TAPDIS 0936
TDISP  091B   TEMP   00FC   TEMPA  1A60   TEMPB  1A61
TEMPX  00FD   TENSYN 0B36   TIMH   1A5F   TIML   1A5E
TLOOK  09BB   TPINIT 0810   TPI    0821   TPTXT  082E
TPVEC  0918   TTXT   0833   VALID  0C2C   VALIT  1431
VALNUM 11CB   VALT   1438   VAL    0C33   VNA    11D3
VU     0C64   WARMST 1CCA   XREG   00F5   YREG   00F4
ZERO   0AB8   ZRO    0AC1
```

Appendix 4

Working in decimal

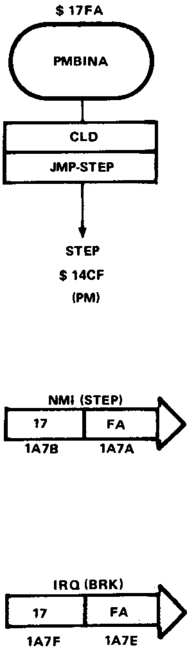
The detours involved to reach either the original monitor routine or the Printer Monitor during or after execution of a decimal program was mentioned at least twice in Book 3 (see chapter 12, page 151 and appendix 2). The 'detours' consisted of a few instructions stored in PIA RAM. Although the detours are still necessary, the corresponding instructions no longer need to be entered into the PIA RAM by the operator, as they are now located in the very last section of the PM/PME EPROM. The details are shown in figures 1a and 1b – the BINAR and PMBINA routines.

1a



81916 1a

1b



81916 1b

Appendix 5

BASIC on the Junior Computer

Although the Junior Computer is quite fluent in machine language, its linguistic skills cannot lead to full 'adult' communication until the machine has learned a 'high level' language, such as BASIC. A specially adapted version of BASIC is now available on cassette from the Microsoft/Johnson Computer Corporation, which will enable Junior Computer operators to fulfil their dreams at long last.

Part one

BASIC still remains the number one computer language. Although it may not be as grammatical and efficient as other languages (such as COMAL or PASCAL), its popularity shows that it meets the essential requirements of computer users all over the world. Thanks to Microsoft, who developed an excellent version of BASIC for the KIM computer some time ago, the Junior Computer can now be made bilingual, its 'mother tongue' being machine language of course. Even with the addition of a BASIC vocabulary, machine language still plays an important role in various routines and timing processes etc., so there is no question of it being completely replaced. That is a relief for readers who have stayed up to all hours getting to grips with the machine language instruction set!

The KB9 BASIC by Microsoft is a nine digit 8k BASIC on cassette. Since this was originally developed for the KIM, it will have to be modified before it will run on the Junior Computer. Contrary to what might be expected, this is a straightforward operation that takes a mere fifteen minutes or so. This is nothing compared to the thousands of man-hours involved in developing the Microsoft BASIC. Only 31 of the eight thousand memory locations need to have their contents altered. Now to discover what ingredients are required to 'cook up' a BASIC on the Junior Computer.

The ingredients

First of all, what about the hardware? Obviously, the computer will have to be a fully extended version. In addition, 16k of RAM has to be located in the address range $2ØØØ . . . $5FFF, either in the form of two RAM/EPROM cards each containing 8k of RAM, or the 16k dynamic RAM card described in the Elektor April 1982 (E84) issue.

Although the extensions were fully described in Book 3, it may be an idea to briefly recap on a few main points here, as this is really a very basic part of the BASIC facility! The extra bus board memory should also contain the three jump vectors situated in the address range $FFFA . . . $FFFF. Appendix 3 in Book 3 mentions two ways in which to include these vectors without the need for an expensive RAM/EPROM card. In the Elektor April 1982 issue a mini EPROM card is introduced which provides yet another option.

As far as the software requirements are concerned, both the printer monitor (PM) and the tape monitor (TM) routines must be available. The former contains the input/output subroutines RECCHA ($12AE), PRCHA ($1334) and RESTTY ($14BC) which serve to start the Junior Computer BASIC. The latter contains the main cassette routines DUMP ($Ø9DF) and RDTAPE ($ØBØ2). Then, of course, the KB-9 BASIC cassette (not KB-6 nor KB-8!!) will have to be acquired, together with all the necessary documentation. Other requirements include a cassette recorder, an ASCII keyboard, a printer and/or a video display terminal and an understanding of programming in the BASIC language. Anyone who wishes to brush up on their BASIC knowledge should read the crash course published in the March . . . June 1979 issues of Elektor or SC/MP Book 2.

The recipe

- Switch on the Junior Computer and start up the PM routine. Place the KB-9 cassette in the tape recorder:

```
RST 1 Ø Ø Ø GO RES
G1 (CR)
```

- Start the recorder in the play mode at the beginning of the tape. The program number (ID) of KB-9 is Ø1. Reading in the instructions etc. takes several minutes, after which the computer reports 'READY'. Remove the cassette from the recorder as it is advisable to store the Junior BASIC on a separate cassette and preserve the KB-9 version in its original form.
- Using the PM routine, alter the contents of 31 memory locations, as indicated in the first section of the accompanying table. Start by checking the 'old' data at the locations concerned. Any discrepancies will mean that you have been landed with the wrong version of BASIC!
- Place a new cassette in the recorder. Start at the beginning, reset the counter and depress the record and play buttons. After about ten seconds enter:

```
SB1, 2ØØØ, 4261 (CR)
```

It only takes a matter of minutes for the Junior BASIC to be recorded. The program number will now be B1.

- As soon as the Junior BASIC is stored on cassette, the message 'READY' will appear on the printer or the video screen. Let the tape continue for a further ten seconds before depressing the stop key.
- 18 locations on page 1A (PIA RAM) need to be loaded with six LOAD and SAVE instructions. The address area concerned is $1AØØ . . . $1A11;

Table 1

The KB-9 to Junior BASIC conversion table
(based on the KB-9 cassette, # 4065 © 1977 by Microsoft Co.; version V1.1).

I. The interpreter

a. ID = B1 instead of Ø1.

b. 1. Address $2457	should contain AE	instead of 5A;	
2. Address $2458	should contain 12	instead of 1E;	
3. Address $26DD	should contain 8Ø	instead of 4Ø;	
4. Address $26DE	should contain 1A	instead of 17;	
5. Address $2746	should contain 79	instead of F9;	
6. Address $2747	should contain 1A	instead of 17;	
7. Address $274D	should contain 7Ø	instead of F5,	
8. Address $274E	should contain 1A	instead of 17;	
9. Address $275Ø	should contain 71	instead of F6,	
10. Address $2751	should contain 1A	instead of 17;	
11. Address $2757	should contain 72	instead of F7;	
12. Address $2758	should contain 1A	instead of 17;	
13. Address $275A	should contain 73	instead of F8;	
14. Address $275B	should contain 1A	instead of 17;	
15. Address $275E	should contain 1A	instead of 18,	
16. Address $2791	should contain 7Ø	instead of F5;	
17. Address $2792	should contain 1A	instead of 17;	
18. Address $2794	should contain 71	instead of F6;	
19. Address $2795	should contain 1A	instead of 17;	
20. Address $2799	should contain 79	instead of F9;	
21. Address $279A	should contain 1A	instead of 17;	
22. Address $27A4	should contain Ø9	instead of 73;	
23. Address $27A5	should contain 1A	instead of 18;	
24. Address $27B9	should contain FA	instead of ED;	
25. Address $27BA	should contain ØØ	instead of 17;	
26. Address $27BC	should contain FB	instead of EE;	
27. Address $27BD	should contain ØØ	instead of 17,	
28. Address $2A52	should contain 34	instead of AØ;	
29. Address $2A53	should contain 13	instead of 1E;	
30. Address $2AE6	should contain AE	instead of 5A;	
31. Address $2AE7	should contain 12	instead of 1E.	

II. Additional instructions on page 1A

a. ID = B2.

b. 1. Address $1AØØ	contains 2Ø;
2. Address $1AØ1	contains DF;
3. Address $1AØ2	contains Ø9,
4. Address $1AØ3	contains 2Ø;
5. Address $1AØ4	contains BC;
6. Address $1AØ5	contains 14,
7. Address $1AØ6	contains 4C:
8. Address $1AØ7	contains 48;
9. Address $1AØ8	contains 23;
10. Address $1AØ9	contains 2Ø;
11. Address $1AØA	contains Ø2;
12. Address $1AØB	contains ØB;
13. Address $1AØC	contains 2Ø;
14. Address $1AØD	contains BC,
15. Address $1AØE	contains 14;
16. Address $1AØF	contains 4C;
17. Address $1A1Ø	contains A6;
18. Address $1A11	contains 27.

details of the contents of these locations can be found in the second half of the table. This data is given the program number B2 and is again stored on cassette.

- Depress the record and play keys once more and enter:
  SB2, 1AØØ, 1A12 (CR).
- After the 'READY' message, the cassette recorder can be stopped.

Now it is time to check whether the Junior BASIC is correctly stored in memory. This can be done with the aid of the 'question and answer' game following the BASIC start address ($4Ø65). It is always a good idea to enter a test program. The cassette commands can be verified by writing a BASIC program, storing it on cassette (SAVE), erasing the program area (NEW) and then reading the program in again from cassette (LOAD). Once the Junior BASIC has met with approval, the same procedure can be used to test the Junior BASIC cassette. For this, the Junior Computer is switched off for a while and then on again, after which the two programs (B1 and B2) are loaded from cassette.

Ready to serve

By now the operator is all set to dish up the Junior Computer BASIC. Do not forget to read the manual supplied with the cassette. This consists of the 'Microsoft Introduction', 'Dictionary' and 'Usage Notes'. Although the contents are rather concise, to the point of being cryptic, all the necessary information is provided. As far as the software is concerned, only one or two actual addresses are mentioned.

The following remarks, however, should make things a bit clearer:

1. After entering:
   RST 1 Ø Ø Ø GO RES (RUBOUT)
   GB1 (CR)
   READY (depress stop key)
   GB2 (CR) (depress stop key again)
   READY

the Junior BASIC can now be started. A cold start entry takes place at address $4Ø65.
4Ø65 (SP) R
The program must be started by way of PM and not by way of the original monitor routine, as otherwise the input/output parameters will be incorrectly defined. In any case, PM is indispensable for reading in data.

2. The Junior BASIC utilises the following memory range on page zero: $ØØØØ . . . $ØØDC and $ØØFF. Thus one of the locations (MODE) belonging to the original monitor is used. This merely serves to start PM.
3. The start address for a warm start entry is $ØØØØ. In the KIM the warm start entry allows the computer to return to BASIC after writing or reading a BASIC program to or from cassette. In the case of the Junior Computer things are slightly different (see point 9). Here, the warm start entry may be used to return to BASIC from PM. The jump from BASIC to PM occurs either as a result of a non-maskable interrupt (NMI) or because the BREAK key on the ASCII keyboard was depressed. The BRK jump vector points to the label LABJUN ($1Ø5F) of the PM routine. After printing the text 'JUNIOR', the computer jumps to the central label RESALL of PM (see Chapter 14 in this book). In the event of an NMI, RESALL is reached at the end of the STEP initialisation routine ($14CF).
4. The ST key may be used during PM to examine the contents of various memory locations, such as the ones on page zero (see point 2) for instance. A warm start entry heralds the return to the BASIC program.
5. Supposing the operator is executing a BASIC program (RUN) making use of the Elekterminal (up to 16 lines on the display) and the BASIC program turns out to contain more than 16 lines. This is what should be done:

RUN (CR)
BRK (while 16th line is being printed)
examine result
(SP) R The computer prints
OK
Start the program again: RUN(CR)
enter the 16th and following 14 lines, etc.

6. When starting the Junior BASIC by way of a cold start entry, the operator will be requested to state the 'TERMINAL WIDTH'. If the Elekterminal is being used, this is set at 64 (CR).
7. The ASCII keyboard does not provide a '↑' nor a '^' key necessary for power functions, where A↑4 corresponds to A^4. What is required is an

ASCII key which generates the code $5E. This can be improvised by sacrificing another key. One contact is connected to row x7 and the other to column y9 in the keyboard matrix (pins 32 and 22 of IC1). Only two keys are suitable: the 'PAGE' key at the far right in the top row and the 'ESC' key at the far left in the second row. The latter affords the most elegant solution, as the ESC function is preserved (a matter of combining it with the Shift key). Interrupt the two connections x5 and y10 (without actually cutting the wires!) and link the ESC key to pins 22 and 32 of IC1. Further details on the ASCII keyboard are provided in Book 3 (Chapter 12).

8. In order to start the Junior BASIC by way of a fresh cold start entry during a computer session, the program (file B1) will have to be loaded from cassette all over again. This is necessary as a relatively large section of data block B1 is reserved as the first section of the BASIC work area if any trigonometric functions are required. After the cold start entry, the computer will request the operator to specify its task. Whether trig functions are to be performed or not, the computer must be informed by way of a cold start entry, (once B1 has been loaded again).

N.B. In file B1 ($2ØØØ. . . $426Ø), locations $4Ø41 . . . $426Ø are added to the user work space if the operator wishes to utilise the trigonometric functions (depress the Y key); locations $3F1F . . . $426Ø are added to the user work space if the trig functions are not required (N key); locations $3FD3 . . . $426Ø are added if the ATN function (A key) is cancelled.

The first memory location is loaded with ØØ (BOF: Beginning Of File). Now that 16k of RAM has been added, the user work space will cover the following ranges:

$4Ø42 . . . $5FFF (8126 bytes) when Y is depressed;
$3F2Ø . . . $5FFF (8416 bytes) when N is depressed and
$3FD4 . . . $5FFF (8236 bytes) when A is depressed.

9. Thanks to the Junior Computer subroutine system, reading and writing BASIC programs to and from cassette (SAVE and LOAD) is much easier than with the KIM BASIC. The only snag is that this occupies the second file, B2. After SAVE has been entered, the BASIC program is stored on tape (where ID = FE). After a while, the 'OK' message will appear followed by an empty line. After LOAD (CR) is entered, a BASIC program is read from tape (where ID = FF, so make sure the required BASIC program is stored *before* this!). A little later 'LOADED' is printed. This is not followed by the message OK and the computer does not start a new line. In other words, the video screen will display 'LOADEDLIST' if the entered program is to be checked.

Any questions?

Here are the answers to several questions which are likely to be asked:

1. Elektor Publishers Ltd. cannot comply with requests for a copy of the notes accompanying the Microsoft/Johnson BASIC cassette, as this would be an infringement of copyright.
2. The source listing of the KB-9 costs a few thousand dollars. Not surprisingly, Elektor is not in a position to sell it to readers.
3. The Junior BASIC is derived from the KIM BASIC, about which plenty of literature is available.

Part Two

Polishing up the Junior BASIC

When the Microsoft BASIC appeared on the market for the 6502, several batches of the CPU did not contain the ROR instruction at that particular time. That is why it was excluded (out of necessity!) from the multiplication and addition/subtraction routines. It was replaced by a 'macro instruction', a set of six two byte instructions. Nowadays, however, the new 6502 chips all include the ROR facility, so that the slow, tedious macro instructions can be omitted altogether. This cuts down the total time it takes to multiply nine digit numbers with a floating decimal point by 37% and saves a lot of memory space.
In the case of the Junior BASIC B1 and B2 can now be combined into a single file, which means 68 bytes are available for other purposes.
Before modifying anything be sure to preserve the original KB-9 BASIC and the B1 and B2 files on cassette . . . just in case!, or at least until Junior Computer owners are absolutely satisfied that the modified BASIC works.

Table 2*

III. Modifications to the addition and subtraction routines:

```
37C3: 18
       Ø    1    2    3    4    5    6    7    8    9    A    B    C    D
37DØ:       76   Ø2   76   Ø3   76   Ø4   68   6A   C8   DØ   E8   18   6Ø
```

IV. The following B2 instructions fill the 'vacancies' left by the macro instruction set:

```
       Ø    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
37DØ:                                                                       2Ø   DF
37EØ:  Ø9   2Ø   BC   14   4C   48   23   2Ø   Ø2   ØB   2Ø   BC   14   4C   A6   27
```

37FØ . . . 38Ø1 are empty (18 bytes)

V. Modifications to the multiplication routine:

```
       Ø    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
38CØ:                 66   73   66   74   66   75   66   76   66   BD   98   4A   DØ
38DØ:  D6   6Ø
```

38D2 . . . 39Ø3 are empty (50 bytes)

VI. New absolute addresses for the 'ex' B2 instructions
see I. The interpreter)

- lines 1 . . . 14 (I. The interpreter) remain unchanged
- line 14a is included after line 14:
  14a. address $275D should contain DE instead of ØØ
- lines 15, 22 and 23:
  15. address $275E should contain 37 instead of 1A (18 in KB-9)
  22. address $27A4 should contain E7 instead of Ø9 (73 in KB-9)
  23. address $27A5 should contain 37 instead of 1A (18 in KB-9)
- All the other lines and addresses remain unchanged, paragraph II in Table 1 is therefore superfluous.

* *The modifications indicated in this table are optional.*

So much for the theory . . .

Table 2 shows how to modify the B1 file. Start by loading KB-9 into the Junior Computer memory. Then modify the contents of addresses $37C3, 37D∅ . . . 37DD, the ROR instructions used by the addition and subtraction programs. After $37DE, the 'old' instruction set may be substituted for B2 instructions. The ones contained in page 1A are stored in the $37DE . . . 37EF address range (see section II of table 1). Then change the multiplication routine instructions: $38C3 . . . 38D1 (Table 2, section V).

All that remains to be done is to adjust the absolute addresses belonging to the 'old' B2 instructions in the manner shown in Table 2, section VI.

B1 and B2 will now constitute a new file which could be christened BB, for example, to avoid confusion. Readers will find the alterations speed up the Junior BASIC considerably and improve its overall performance.

The division programs are not affected, as they include the ROL instruction and very little would be gained by changing the addition/subtraction operations.

Literature:

BYTE, May 1981, Faster BASIC for the Ohio Scientific,
J. Sauter, page 236
BYTE, September 1981, Byte's bugs, page 110
Elektor, April 1982, E84, 'BASIC on the Junior Computer'
Elektor, April 1982, E84, 'Dynamic RAM card'