CHAPTER 6

TANBUG
V1

The TANBUG monitor program is located in 1K bytes of read only memory (ROM) at the top of the address space i.e. pages 252-255. It contains facilities to enter, modify, run and debug programs. This chapter of the manual gives full details of the command facilities and subroutines available to the user.

TANBUG will only operate in the memory map of the microtan system, it is not a general purpose 6502 software package and has been specifically written for microtan. Locations F7F7, F7F8 and F7F9 are reserved for a jump to an expansion monitor ROM which is positioned on the expansion board, more about this later. Locations 2ØØ-3FF i.e. pages 2 and 3 are the visual display memory - TANBUG writes to these locations whenever a command is typed to the monitor. Locations BFFØ-BFF3 are the addresses of the peripheral attachments, e.g. keyboard, graphics function flip-flop etc. Locations 1ØØ-1FF i.e. page 1, are used as the stack by the microprocessor. Since the stack is of the push down variety it follows that the whole of the are will not be used as stack storage in the majority of programs. TANBUG requires to use locations 1FØ-1FF as stack storage (only 16 locations). The rest of this area is free for user programs. Locations 4Ø-FF are also available as user RAM, the preceeding locations Ø-3F being reserved for use by TANBUG. User programs which do not use the stack may therefore be loaded anywhere i.e. the area 4Ø-1EF. For user programs which do use the stack, the user must calculate how many stack locations are required and reduce the upper limit accordingly.

TANBUG contains coding to automatically identify whether the keypad or full ASCII keyboard is connected to the keyboard socket. This coding is executed every time a reset is issued, and thereafter a sequence of code particular to the keyboard type in use is executed. Reset must therefore always be issued after changing the keyboard type.

When using an ASCII encoded alphanumeric keyboard monitor commands are typed in as shown in this chapter. There is

however, no reset key on an ASCII keyboard, one must be fitted as shown in the chapter describing assembly of the microtan kit. TANBUG drives this type of keyboard in the interrupt mode.

The keypad is used somewhat differently, its layout being shown below.

SHIFT	DEL LF	SP CR	RST
M C	G D	S E	N F
P 8	ESC 9	B A	L B
 4	@ 5	C 6	R 7
 0	 1	. 2	, 3

TANBUG interrogates the keypad for a depressed key, then translates the matrix encoded signal into an ASCII character which it puts up on the visual display just as if the equivalent key were depressed on an ASCII encoded keyboard. Because of the limited number of keys it has been necessary to incorporate a shift function on the keypad. So, to obtain the character P for example, the user presses and releases SHIFT, then depresses and releases P. The SHIFT key contains a self cancelling facility - if the user presses SHIFT twice in succession the pending shift operation is cancelled. So as an example, using the two keys SHIFT and 8 the operations SHIFT P yields P on the display. SHIFT SHIFT P yields 8 on the display. Other special purpose keys on the keypad are RST, which issues a reset to the microtan, and DEL which deletes the last character typed. Repeated deletes erase characters back to the beginning of the line.

From now on in this chapter the microtan will be treated as having one type of keyboard only, since all functions required can be derived by depressing the appropriate key or keys on whatever is used - keyboard or keypad.

Having described some of the background to TANBUG it is now

possible to describe the commands and syntax of TANBUG i.e. how to use it. An example is shown later on. All numerical values of address, data and monitor command arguments are in hexadecimal. The symbol <CR> means on depression of the carriage return key, <SP> the space key or bar, <ESC> the escape key (ALT on some keyboards) and <LF> line feed. In all examples, text to be typed by the user will be underlined, while TANBUG responses will not. ▮ indicates the cursor. <ADDR> means a hexadecimal address, <ARG> means hexadecimal data and <TERM> means one of the terminators <CR>, <SP>, <ESC> or <LF>.

All commands are of the form

<COMMAND><TERM>
or <COMMAND><ARG><TERM>
or <COMMAND><ARG>, <ARG><TERM>
or <COMMAND><ARG>, <ARG>, <ARG><TERM>

where <COMMAND> is one of the mnemonic commands and <ARG> is a hexadecimal argument applicable to the command being used. The required argument is defined for each command. It should be noted at an early stage that the longest argument will contain 4 hexadecimal characters. If more are typed all but the last 4 are ignored. As an example consider the memory modify command M1234ØØ78 <CR>. In this case location ØØ78 will be modified or examined as all but the last 4 characters are ignored.

<TERM> is one of the terminating characters <CR>, <SP>, <LF> or <ESC>. In fact TANBUG accepts any of the "control" characters (HEX code Ø-2Ø) as terminator. TANBUG will reply with a ? if an illegal command is encountered.

Starting the monitor TANBUG:

Press the RST key on the keypad or the reset key or button connected to the microtan. TANBUG will scroll the display and respond with

TANBUG
▮

Note that on initial power up the top part of the display will be filled with spurious characters. These will disappear as new commands are entered and the display scrolls up. On subsequent resets the previous operations remain displayed to facilitate

debugging.

Memory modify/examine command M:

The M command allows the user to enter and modify programs by changing the RAM locations to the desired values. The command also allows the user to inspect ROM locations, modify registers etc. To open a location type the following

M <ADDR> <TERM>

TANBUG then replies with the current contents of that location. For example to examine the contents of RAM location 1ØØ type M1ØØ<CR> TANBUG then responds on the display with

M1ØØ,ØE,

assuming the current contents of the location were ØE.

There are now several options open to the user. If any terminator is typed the location is closed and not altered and the cursor moves to the next line scrolling up the display by one row. If however, a value is typed followed by one of the terminators <CR>, <LF> or <ESC> the location is modified and then closed. For example using <CR>

M1ØØ,ØE,FF

location 1ØØ will now contain FF. If however <SP> is typed, the location is re-opened and unmodified.

M1ØØ,ØE,FF
MØ1ØØ,ØE,

This facility is useful if an erroneous value has been typed. The terminators <LF> and <ESC> modify the current location being examined, then opens the next and previous locations respectely i.e. using <LF>

M1ØØ,ØE,FF
MØ1Ø1,AB,

and using <ESC>

M1ØØ,ØE,
MØØFF,56,

Using <LF> makes for very easy program entry, it only being necessary to type the initial address of the program followed by its data and <LF> , then responding to the cursor prompt for subsequent data words.

NOTE that locations ↑FE and ↑FF should not be modified. These are the stack locations which contain the monitor return addresses. If they are corrupted TANBUG will almost certainly "crash" and it will be necessary to issue a reset in order to recover.

List command L:

The list command allows the user to list out sections of memory onto the display. It is possible to display the contents of a maximum of one hundred and twenty consecutive memory locations simultaneously. To list a series of locations type

L <ADDR>, <NUMBER><TERM>

where <ADDR> is the address of the first location to be printed and <NUMBER> is the number of lines of eight consecutive locations to be printed. TANBUG pauses briefly between each line to allow the user to scan them. For example, to list the first 16 locations of TANBUG (which resides at FCØØ-FFFF) type LFCØØ,2<CR>.
The display will then be

```
LFCØØ,2
FCØØ  A2  FF  9A  E8  86  17  2Ø  B7
FCØ8  FF  8D  F3  BF  A2  ØE  BD  DF
```

If zero lines are requested (i.e. <NUMBER> = Ø) then 256 lines will be given.

Go command G:

Having entered a program using the M command and verified it using the L command the user can use the G command to start running his own program. The command is of the format G <ADDR><TERM>. For example, to start a program whose first instruction is at location 1ØØ type G1ØØ <CR>. When the user program is started the cursor disappears. On a return to the monitor it re-appears.

The G command automatically sets up two of the microprocessors internal registers

a) The program counter (PC) is set to the start address given in the G command.
b) The stack pointer (SP) is set to location IFF.

The contents of the other four internal registers, namely the status

word (PSW), index X (IX), index Y (IY) and accumulator (A), are taken from the monitor pseudo registers (described next). Thus the user can either set up the pseudo registers before typing the G command, or use instructions within his/her program to manipulate them directly.

Register modify/examine command R:

Locations 15 to 1B within the RAM reserved for TANBUG are the user pseudo registers. The user can set these locations prior to issuing a G command. The values are then transferred to the microprocessors internal registers immediately before the user program is started. The pseudo register locations are also used by the monitor to save the user internal register values when a breakpoint is encountered. These values are then transferred back into the microprocessor when a P command is issued, so that to all intents and purposes the user program appears to be uninterrupted.

The R command allows the user to modify these registers in conjunction with the M command. To modify/examine registers type R <CR> and the following display will appear (location 15 containing ØØ say).

```
R
MØØ15,ØØ,
```

Now proceed as for the M command.

Naturally the M command could be used to modify/examine location 15 without using the R command - the R command merely saving the user the need to remember and type in the start location of the pseudo registers. Pseudo register locations are as follows.

Location	Function
15	Low order byte of program counter (PCL)
16	High order byte of program counter (PCH)
17	Processor status word (PSW)
18	Stack pointer (SP)
19	Index X (IX)
1A	Index Y (IY)
1B	Accumulator (A)

Two typical instances of the use of the R command are:-

a) Setting up PSW, IX, IY and A before starting a user program.

b) Modifying registers after a breakpoint but before proceeding with program execution (using the P command) for debugging purposes.

Note that when modifying registers in case (b) care must be taken if PCL, PCH or SP are modified, since the proceed command P uses these to determine the address of the next instructions to be executed (PCL, PCH) and the user stack pointer (SP).

Single instruction mode S:

Single instruction mode is a very powerful debugging aid. When set TANBUG executes the user program one instruction at a time, re-entering the monitor between each instruction and printing out the status of all of the microprocessor's internal registers as they were after the last instruction executed in the user program. The S command is used in conjunction with the proceed command P and the normal mode command N. Examples are given in the description of the P command.

Normal mode command N:

The N command is the complement of the S command and is used to cancel the S command so that the microprocessor executes the user program in the normal manner without returning to the monitor between each instruction. Reset automatically sets the normal mode of operation.

Proceed command P:

The P command is used to instruct TANBUG to execute the next instruction in the user program when in single instruction mode. Pseudo register contents are transferred into the microprocessors internal registers and the next instruction in the users program is executed. The monitor is then re-entered. P may also be used with an argument thus P <NUMBER> <CR> where NUMBER is less than or eqiual to FF. In this case the program executes the specified number of instructions before returning to the monitor.

Each time the monitor is re-entered after execution of an instruction or instructions, the status of the microprocessor internal registers

as they were in the user program are printed across the screen in the following order:

Address of next instruction to be executed.
Processor status word.
Stack pointer.
Index register X.
Index register Y.
Accumulator.

Note that these are in the same order as the pseudo registers described earlier.

Whenever the user program is entered, the cursor is removed from the display. Whenever the monitor is entered, the cursor returns to the display as a user prompt. While in the monitor between user instructions, any monitor command can be typed. A program must always be started by the G command, then P used if in single instruction mode. A P command used before a G command is issued is likely to cause a program "crash" and should not be attempted.

As an example consider the simple program which repeatedly adds 1 to the accumulator.

Address	Data	Mnemonic	Comment
1ØØ	69	ADC 1	: add 1 to acc.
1Ø1	Ø1		
1Ø2	4C	JMP 1ØØ	
1Ø3	ØØ		
1Ø4	Ø1		

Set the single instruction mode and start the program. The user may wish to initially set the accumulator to ØØ by using the M command.

S
G1ØØ
Ø1Ø2 2Ø FF ØØ ØØ Ø1

TANBUG then responds with the characters shown above.

Ø1Ø2 is the address of the next instruction to be executed.
2Ø is the processor status word value.
FF is the low byte value of the stack pointer. The high byte is always set to 1, the stack is therefore pointing at location 1FF.

ØØ is the value of the index X register.
ØØ is the value of the index Y register.
Ø1 is the value of the accumulator. It is a 1 as 1 has been added to the accumulator and it is assumed that the user cleared the accumulator before starting the program.

Since the cursor has re-appeared, TANBUG is ready for any monitor command. For example, registers or memory locations can be modified, or the program may be re-started from scratch by typing G1ØØ<CR> again. If the user wishes to continue then type P<CR>. The resulting display is

```
S
G1ØØ
Ø1Ø2  2Ø  FF  ØØ  ØØ  Ø1
P
Ø1ØØ  2Ø  FF  ØØ  ØØ  Ø1
```

Since the instruction at location 1Ø2 was "Jump to 1ØØ", the status print out shows that this has indeed occurred. Registers, since they were not modified by any program instruction, remain unchanged. To proceed further type P<CR> again.

```
S
G1ØØ
Ø1Ø2  2Ø  FF  ØØ  ØØ  Ø1
P
Ø1ØØ  2Ø  FF  ØØ  ØØ  Ø1
P
Ø1Ø2  2Ø  FF  ØØ  ØØ  Ø2
```

The add instruction has been executed again, so the accumulator has incremented by 1 to become 2. Now typing P4<CR> gives a display.

```
S
G1ØØ
Ø1Ø2  2Ø  FF  ØØ  ØØ  Ø1
P
Ø1ØØ  2Ø  FF  ØØ  ØØ  Ø1
P
Ø1Ø2  2Ø  FF  ØØ  ØØ  Ø2
P4
Ø1ØØ  2Ø  FF  ØØ  ØØ  Ø4
```

TANBUG allowed execution of 4 instructions before again returning to the monitor. The 4 instructions were 2 add instructions and 2 jump instructions thus giving the accumulator the value 4.

By typing N<CR> then P<CR> removes the single instruction mode and causes the program to proceed. It now does not return to the monitor but continues to race around this small program loop continually adding and jumping back. There is no way to exit from this trivial program except by a microprocessor reset or, if using an alphanumeric keyboard, by typing ESC.

It can be seen that the S and P commands are particularly useful when tracing a program which contains instructions that transfer program control e.g. jumps, branches and subroutines, since these commands allow the user to interrogate the order of execution of his/her program.

Breakpoint command B:

A breakpoint is a complementary debugging aid to single instruction mode. Instead of stepping singly through all instructions in a program, the breakpoint facility allows the user to specify the address at which he requires the monitor to be re-entered from his/her program. As an example, consider a long program in which a fault is suspected to exist near the end. It would be very tedious and time consuming to single step through the program to the problem area. A breakpoint can be set just previous to where the fault is suspected to exist and the program started with the G command. Normal execution occurs until the breakpoint is reached, then the monitor is re-entered with the same status print-out as for single instruction mode. Any monitor commands can then be used and the program continued.

The format of the breakpoint command is

B <ADDR>, <NUMBER> <CR>

where <ADDR> is the address of any instruction OPCODE (but not argument), <NUMBER> is any number from Ø-7 defining one of 8 breakpoints. B CR removes all breakpoints. As an example consider the following program

```
1ØØ  E8          LOOP: INX
1Ø1  C8                INY
1Ø2  69  Ø1            ADC#1
1Ø4  4C  ØØ  Ø1        JMP LOOP
```

Firstly set index X, index Y and the accumulator to ØØ using the R command. To set breakpoint Ø at the jump instruction and start the program type B1Ø4,Ø<CR>. The display will then be

```
B1Ø4,Ø
G1ØØ
Ø1Ø4  3Ø  FF  Ø1  Ø1  Ø1
```

The jump instruction was reached and the breakpoint re-directed control back to TANBUG. If it were required, single instruction mode could be set for further debugging. However, assume that we wish to execute the loop again by typing P<CR>.

```
B1Ø4,Ø
G1ØØ
Ø1Ø4  3Ø  FF  Ø1  Ø1  Ø1
P
Ø1Ø4  3Ø  FF  Ø2  Ø2  Ø2
```

The proceed command P has gone once through the breakpoint and then re-entered the monitor. If P<NUMBER><CR> was typed it would have proceeded through the breakpoint <NUMBER> times.

Up to 8 breakpoints can be set at 8 different locations. The B<CR> command removes all breakpoints. A single breakpoint can be removed by setting its address to Ø. For example, imagine a breakpoint is set as follows; B1Ø2,2, and it is subsequently wished to remove it but leave any others unaltered; type BØ,2<CR> to remove it.

Caution. The breakpoint system works by replacing the users instruction with a special instruction (BRK) whose opcode is ØØ. Replacement is carried out when G or P is typed. On return to the monitor the original opcode is replaced. It is therefore possible to corrupt the user program under some circumstances. The following points should therefore be observed:

a) Breakpoints must only be set at the opcode part of a user instruction and nowhere else.

b) If the user program utilises the BRK instruction as part of the user code, then the user must have his own special interrupt routine and cannot use breakpoints.

c) If breakpoints are set in the user program and a reset is issued while the microprocessor is executing the user program rather than the monitor, the breakpoints are lost and those locations at which breakpoints were set in the user program will be corrupted. These locations must be re-entered using the M command before restarting the user program.

d) Setting more than one breakpoint at the same address causes the user program to be corrupted.

e) To use breakpoints, the user must not have modified the interrupt link, i.e. the interrupt code within TANBUG must be executed.

The status of breakpoints may be inspected by using the M command to examine the breakpoint status table. This is located in RAM locations 2Ø-2F and are as follows:

Address	Contents
2Ø	PCL BØ
21	PCH BØ
22	PCL B1
23	PCH B1
24	PCL B2
25	PCH B2
26	PCL B3
27	PCH B3
28	PCL B4
29	PCH B4
2A	PCL B5
2B	PCH B5
2C	PCL B6
2D	PCH B6
2E	PCL B7
2F	PCH B7

For example, typing M2Ø<CR> followed by <LF> gives

M2Ø,ØØ,
MØØ21,Ø1,

This indicates that breakpoint Ø is set to location 1ØØ by taking

the contents of location 2Ø as PCL and of location 21 as PCH. If the breakpoint is set at location Ø then this particular breakpoint is disabled.

Offset command O:

The offset command O is a program writing aid. It calculates branch offsets for the user for incorporation as arguments in branch instructions. Consider the example:-

```
1ØØ  E8          LOOP:  INX
1Ø1  C8                 INY
1Ø2  69                 ADC#1
1Ø3  Ø1                   .
                          .
                          .
12Ø  DØ                 BNE LOOP
121                     (branch argument)
```

To calculate the number to enter into location 121 is quite tedious without a facility such as the O command. It is used with the following format.

O<ADDR. OF BRANCH OPCODE><ADDR. OF DEST.><CR>

and in this case it would be necessary to type O12Ø,1ØØ<CR>. The display would be

```
O12Ø,1ØØ = DE
```

DE is the number that should be entered into location 121 such that if the BNE instruction is true the program counter will jump to the label LOOP.

Note that the maximum branch range is 7F forwards and backwards. If the range is exceeded a ? is displayed.

Copy command C:

The copy command allows copying of the contents of one block of memory to another. Its format is

C<START ADDR. SOURCE><END ADDR. SOURCE><START ADDR. DEST.>

Suppose it is required to copy the block of data in locations FCØØ-FDØØ into a block starting at location 2ØØ. This may be achieved by typing CFCØØ,FDØØ,2ØØ<CR>. The display will be

CFCØØ,FDØØ,2ØØ

As 2ØØ is the starting address of the display memory the user will notice that the top half of the screen has been over written with all sorts of weird and wonderful characters. What this example has done is to take the first 256 bytes of TANBUG and copy them into the top half of the display. The display then scrolled having the top 7 rows filled with these characters.

Breakpoints and the ESC key

If an alphanumeric keyboard is being used, depression of the ESC key (ALT on some keyboards) will cause a re-entry into the monitor from the user program. This is possible because the alphanumeric keyboard is interrupt driven. For example, if the trivial program

```
1ØØ  69        LOOP: ADC#1
1Ø1  Ø1
1Ø2  4C              JMP LOOP
1Ø3  ØØ
1Ø4  Ø1
```

has been started by typing the G command the program continues to loop around continuously with no exit path to the monitor, except by issuing a reset. Instead of a reset the user can press the ESC key, TANBUG responding thus

```
Ø1ØØ  2Ø  FF  Ø1  Ø1 Ø1
```

Using the ESC key has caused a breakpoint to be executed and the monitor invoked. The register print-out above is only typical, the value of each being that when the ESC was depressed. Any monitor command may now be typed, for example P causes the user program to proceed once again.

The ESC facility is most useful in debugging where the user program gets into an unforseen loop where breakpoints have not been set. It enables the user to rejoin the monitor without using reset and losing the breakpoints that have been set.

Notes:
a) The ESC facility is only implemented on interrupt driven keyboards i.e. alphanumeric ASCII keyboards, and is not implemented on the keypad.
b) Interrupts must be enabled for the ESC facility to

operate. TANBUG enables interrupts when entering a user program, therefore do not disable interrupts if the ESC facility is required.

c) The user must not have modified the interrupt jump link. TANBUG's interrupt code must be executed.

USER SUBROUTINES

Certain input/output subroutines are available to the user. Since these rely on a standard display format, this will be described first, followed by the user subroutine descriptions.

Display format

TANBUG uses the bottom line of the display for all text operations. Initially the cursor is at the left hand edge of the screen, and moves gradually to the right as a line is filled. When either a carriage return is output, or the bottom line overflows, the display is scrolled (all lines shift up one row) and the bottom line becomes available for more output. Therefore the cursor always remains on the bottom line. However, there is no reason why users should restrict themselves to this mode of operation unless they intend to use TANBUG's subroutines to control the display in their own programs. It should be noted that the display memory is read/write memory and may be used as a character buffer prior to processing thus saving RAM locations for a user program.

Subroutine POLLKB

Subroutine POLLKB is used to interrogate the keyboard for a typed key. (Appropriate software for the type of keyboard in use is automatically set-up by TANBUG when a reset is issued). On exit from the subroutine the RAM location labelled ICHAR (address ØØØ1) contains the ASCII code of the character typed, whether it is typed on the keypad or on an alphanumeric keyboard. When using the alphanumeric keyboard, interrupts must be in the enabled state. As an example consider the user code

.
.
.
.

```
1)   CLI                    ; enable interrupts
2)   JSR   POLLKB           ; poll the keyboard
3)   LDA   ICHAR            ; load acc. with character
```

The sequence of operations here are

1) Enable interrupts so that alphanumeric keyboard may be interrogated.
2) The program loops around within the POLLKB subroutine until a key is pressed.
3) The program exits from POLLKB with the ASCII code for the key pressed in the location labelled ICHAR. The accumulator is loaded with this value.

Notes: Address of subroutine POLLKB is FDFA. Address of ICHAR is ØØØ1. The registers IX, IY and A are corrupted, therefore the user must save and restore their values if necessary.

Subroutine OUTPCR

This subroutine causes the display to scroll up one line by outputting a carriage return to it. It also re-instates the cursor, which is switched off when a user program is started. This subroutine should be called in a user program prior to any display input or output to clear the bottom line.
Notes: Address of subroutine OUTPCR is FE73. Registers IX and IY are unaffected. Register A is corrupted and must be saved if required.

Subroutine OPCHR

This subroutine is called to output a character held in the accumulator, to the display. The cursor, obliterated on a user program start, is re-instated. As an example, consider the code

```
.
.
.
LDA#3Ø
JSR OPCHR
LDA#31
JSR OPCHR
.
.
.
```

Since 3Ø is the ASCII code for the character "Ø" and 31 is the ASCII code for the character "1", the result (assuming this is the first call to this subroutine) on the bottom line of the display is

Ø1

Repetitive calls of OPCHR will fill the bottom line of the display with the appropriate characters. When the end of the line is reached, OPCHR scrolls the display up one line and then writes characters in the newly vacated bottom line and so on.
Notes: Address of subroutine OPCHR is FE75. Registers IX and IY are unaltered. Register A is corrupted and must be saved if required.

Subroutine HEXPNT

Subroutine HEXPNT takes a binary value from the accumulator and outputs it as two hexadecimal characters on the display. Consider the code

```
PHA             ; save A on stack
JSR OUTPCR      ; scroll display
PLA             ; recover A
JSR HEXPNT      ; output A in hex
JSR OUTPCR      ; scroll display
```

This code will display the contents of the accumulator as two hex characters. For example if the accumulator contained the value 2C the resulting display would be

2C

Notes: Address of subroutine HEXPNT is FFØB. Register IY is unaltered. Registers IX and A are corrupted and must be saved if required.

Subroutine HEXPCK

This subroutine reads hex characters from the bottom line of the display and packs them up into two eight bit binary values,

enabling a sixteen bit word to be assembled. It is useful for incorporation into programs which require numerical keyboard input. Usually POLLKB is used in conjunction with OPCHR to enter data to the display, then HEXPCK called when a carriage return is encountered. The following user code could be used to do this

```
                JSR  OUTPCR      ;  scroll display
        NXTCHR: JSR  POLLKB      ;  wait for character
                LDA  ICHAR       ;  put it in A
                CMP# ØD          ;  carriage return ?
                BEQ  GOPACK      ;  yes, pack it
                JSR  OPCHR       ;  else store in display
                JMP  NXTCHR      ;  get next character
1)      GOPACK: LDY# FF          ;  set IY to first char.
2)              JSR  HEXPCK      ;  pack it
3)
```

In this example the subroutine is used in the following way:

1) Set IY with the character position at which packing is to start. The left most location of the bottom line corrosponds to setting IY to FF The next location corresponds to IY equal to Ø1 etc.
2) Call HEXPCK. Characters are packed until a character other that 0-9 or A-F is encountered; an exit then occurs.
3) Continue into the user code where the values of HXPKL and HXPKH will be read.

For example, packing 1 CR gives HXPKL = 1 and HEXPKH = 0. Packing FEDC CR gives HEXPKL = DC and HEXPKH = DC. Packing FEDCBA CR gives HXPKL = BA and HEXPKH = DC, ie if more than four hexadecimal characters in succession are encountered then the last four are packed. Additionally, flags in the processor status word (PSW) are used to indicate exit conditions. The overflow flag V is set if the first character is a valid hexadecimal character, otherwise it is clear. The zero Z and carry C flags are clear if a non-hexadecimal character is encountered, otherwise they are set. The Y index register holds the number of characters encountered.

Notes: Address of subroutine HEXPCK is FF28. Address of HXPKL is

is ØØ13 and HXPKH is ØØ14. Registers IX, IY and A are all corrupted and must be saved if necessary.

INTERRUPTS

TANBUG uses both the maskable and non-maskable interrupts. However, means have been provided to access the interrupts via both hardware and software. Of necessity user interrupts may, in some cases, place restrictions on certain monitor commands.

The maskable interrupt

When TANBUG is initialised by a reset, certain RAM locations are set up to link through the interrupts for monitor use. These locations are labelled INTFS1, INTFS2, INTFS3 and INTSL1. When a maskable interrupt occurs, the following sequence of events is obeyed (assuming the RAM locations mentioned above have not been modified).

a) The program jumps to INTFS1 in RAM.

b) The locations INTFS1, INTFS2 and INTFS3 contain the instruction JMP KBINT. The program therefore jumps to KBINT which resides in the monitor ROM.

c) The monitor software looks to see what caused the interrupt. If a BRK instruction, then the breakpoint code is executed. If a keyboard interrupt, location ICHAR is updated with the new ASCII character which is read from the keyboard I/O port.

d) If the interrupt is caused by anything other than a BRK instruction then the monitor jumps to location INTSL1.

e) Normally INTSL1 contains an RTI instruction - the program would then return to where it was interrupted.

It can therefore be seen that the user can implement his/her own interrupt service routines in two ways.

1) A fast interrupt response by modifying the locations INTFS1, INTFS2 and INTFS3 to jump to the user interrupt service code. In this case breakpoints and the ESC command cannot be used unless the user program jumps back to the monitor service

routine after executing its own code.

2) A slower interrupt response by modifying INTSL1, INTSL2 and INTSL3 to jump to user service routine, after executing the monitor service routine. The RAM locations INTSL1, INTSL2 and INTSL3 would be modified to contain the instruction JMP USER. This method places no restrictions on monitor commands.

A number of things should be noted when using interrupts:

a) An RTI instruction must always occur at the end of user code to return the program to the point at which it was interrupted, unless the user code jumps back to the monitor service routine.

b) If a reset it issued, the INTFS and INTSL locations are set back to their monitor values by TANBUG, and the user has to reset them.

c) If any microprocessor internal registers are used in the user interrupt service routine, they must be saved before modification and restored before the RTI instruction, i.e. on return to the monitor the registers IX, IY and A must contain the same values as they had on entry to the user routines.

d) The interrupt jump locations should be modified by instructions in the user program at run time and not by the use of the M command. This is because TANBUG software uses keyboard interrupts. If using an alternative link at INTFS1, no breakpoints can be set.

e) Addresses of RAM locations are; INTFS1 = ØØØ4, INTFS2 = ØØØ5, INTFS3 = ØØØ6, INTSL1 = ØØ1Ø, INTSL2 = ØØ11, INTSL3 = ØØ12.

The non-maskable interrupt

The non-maskable interrupt vector is accessed in the same way as explained for the maskable interrupt. The user can obtain access by modifying locations NMIJP, NMIJP1 and NMIJP2. Note that single instruction mode will be inoperative and that breakpoints will be destructive, i.e. they are destroyed when they have been executed once and replaced with the original code. Addresses of RAM locations are; NMIJP = ØØØ7, NMIJP1 = ØØØ8 and NMIJP2 = ØØØ9.

ERROR LINKING

It will be noted that TANBUG displays a question mark whenever an illegal command is typed. In order to allow future expansion of the monitor, an error link to memory external to the monitor ROM's is incorporated.

When an error occurs the following sequence of events is initiated:

a) The program jumps to F7F7.
b) With no expansion board (TANEX) present the address F7F7 (outside TANBUG space) is decoded as address FFF7 (inside TANBUG space).
c) A question mark is printed.

With TANEX present, a special link is incorporated to return the program to the monitor. The user may remove this link and insert an EPROM in the position which includes the address F7F7 containing the code JMP USERCODE at address F7F7, where USERCODE may contain software to deal with any extra commands the user wishes to add to the monitor. Note that this facility will be used by future TANGERINE software.

There are two methods of returning to the monitor from external code:

1) The instruction RTS at the end of the user code returns to the monitor, gives a carriage return then continues looking for commands.
2) The instruction JMP FFF7 returns to the monitor, giving a question mark on the display.

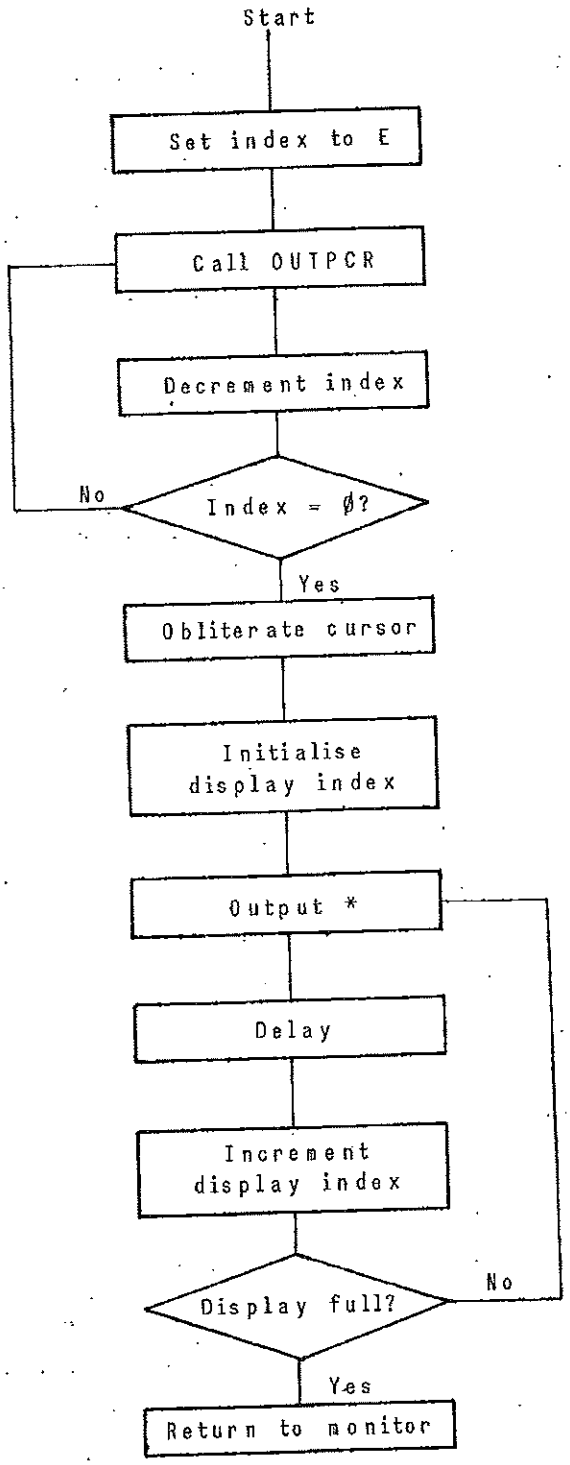
EXAMPLE OF TANBUG's USE

The following, simple example program clears the screen by calling OUTPCR F times, then slowly fills the screen with asterisks. It is used as an example to demonstrate the use of some of TANBUG's commands. Deliberate errors are later written into the program to demonstrate TANBUG's fault finding capabilities.

The first step in writing a program is to produce a flowchart of program execution. The second step is to write the program in assembly language code using the instruction mnemonics. The third

step is to look up and write the op-codes and arguments for each instruction. At this stage the branch code arguments will be left blank and TANBUG's O command used.

The flowchart and program listing now follows.



Example program listing

```
ØØ5Ø  ØØ  ØØ        VDUIND:  Ø               ;display index
ØØ52  AØ  ØF         START:  LDY# F           ;set Y index
ØØ54  2Ø  73  FE     SCRAG:  JSR  OUTPCR      ;carriage return
ØØ57  88                     DEY              ;do E times
ØØ58  1Ø  (arg 1)            BPL  SCRAG
ØØ5A  A9  2Ø                 LDA# 2Ø          ;load A ascii space
ØØ5C  8D  EØ  Ø3             STA  3EØ         ;obliterate cursor
ØØ5F  A9  ØØ                 LDA# Ø           ;set display index
ØØ61  85  5Ø                 STA  VDUIND
ØØ63  A9  Ø2                 LDA# 2
ØØ65  85  51                 STA  VDUIND+1
ØØ67  AØ  ØØ          CONT:  LDY# Ø           ;clear Y index
ØØ69  A9  2A                 LDA# 2A          ;set ascii *
ØØ6B  91  5Ø                 STA (VDUIND),Y
ØØ6D  A2  ØF                 LDX# F           ;delay loop
ØØ6F  AØ  FF                 LDY# FF
ØØ71  88               DECIT: DEY
ØØ72  DØ  (arg 2)            BNE  DECIT
ØØ74  CA                     DEX
ØØ75  DØ  (arg 3)            BNE  DECIT
ØØ77  18                     CLC              ;inc display index
ØØ78  E6  5Ø                 INC  VDUIND
ØØ7A  DØ  (arg 4)            BNE  NOMSB
ØØ7C  E6  51                 INC  VDUIND+1
ØØ7E  A5  51           NOMSB: LDA  VDUIND+1    ;top of display?
ØØ8Ø  C9  Ø3                 CMP# 3
ØØ82  DØ  (arg 5)            BNE  CONT        ;no - continue
ØØ84  A5  5Ø                 LDA  VDUIND
ØØ86  C9  FF                 CMP# FF
ØØ88  DØ  (arg 6)            BNE  CONT        ;double prec. cmp.
ØØ8A  ØØ                     BRK              ;return to monitor
```

Program entry is performed using the M command. For the time being set the branch arguments (arg 1 - arg 6) to ØØ, these can be altered when calculated, using the O command.

Once the program is entered the branch offsets are calculated. The first is arg 1 which has an opcode address of ØØ58 and branches to the label SCRAG at location ØØ54. By typing 058,54<CR> TANBUG prints out the value of arg 1 as FA. This may now be placed in location ØØ59 using the M command. By repeating the exercise for the other five arguments it will be found that location ØØ73 should contain FD, ØØ76 should contain FA, ØØ7B should contain Ø2, ØØ83 should contain E3 and ØØ89 should contain DD.

The program will now run if it has been entered correctly. To start the program type G52<CR> since the first instruction of the program is at location ØØ52. When the screen is full of asterisks the program exits to the monitor. Alternatively, if an alphanumeric keyboard is being used, depression of the ESC key causes an exit to the monitor. If the program does not run correctly then it may be necessary to issue a reset in order to regain control. The program can be listed by typing LSØ,8<CR> yielding a display of

```
L5Ø,8
ØØ5Ø  ØØ  ØØ  AØ  ØF  2Ø  73  FE  88
ØØ58  1Ø  FA  A9  2Ø  8D  EØ  Ø3  A9
ØØ6Ø  ØØ  85  5Ø  A9  Ø2  85  51  AØ
ØØ68  ØØ  A9  2A  91  5Ø  A2  ØF  AØ
ØØ7Ø  FF  88  DØ  FD  CA  DØ  FA  18
ØØ78  E6  5Ø  DØ  Ø2  E6  51  A5  51
ØØ8Ø  C9  Ø3  DØ  E3  A5  5Ø  C9  FF
ØØ88  DØ  DD  ØØ  XX  XX  XX  XX  XX
```

providing the program has been correctly entered (XX indicates any value as these locations are not part of the program). If the program failed to run carefully check the listing from the L command with the program listing and correct any errors with the M command.

Having got the program working it is now possible to introduce a deliberate error to demonstrate the use of breakpoints and the single instruction mode. The error to be introduced is to put the wrong value for the branch argument on the first occurrence of the

instruction BNE DECIT; instead of location 73 containing FD change it to FB. Now the register IY will never be zero and the program will loop here. If the program is started now only one asterisk will be printed and then nothing else will happen. Debugging steps are as follows:

a) Regain control to the monitor by issuing a reset.

b) The first part of the program is being executed correctly as the display scrolls. Furthermore, it is at least getting to location 6B because an asterisk is printed. It would be very tedious to single instruction this far from the beginning because the OUTPCR routine is called sixteen times. Therefore set a breakpoint at location 6D by typing B6D,Ø<CR>.

c) Start the program again by typing G52<CR>. The display scrolls and the status message

```
ØØ6Ø 31  FF  ØF  ØØ  2A
```

is displayed. Control is now back in the monitor.

d) Set single instruction mode by typing S<CR>.

e) Repeatedly typing P<CR> causes single instructions to be executed followed by a status print-out. The following sequence of instructions will be observed.

```
ØØ6F  21  FF  ØF  ØØ  2A
ØØ71  A1  FF  ØF  FF  2A
ØØ72  A1  FF  ØF  FE  2A
ØØ6F  A1  FF  ØF  FE  2A
```

Now if the code were correct the program could not go back to location 6F. In fact, since IY is shown to be FE, the program should have jumped back to location 71. The branch instruction is probably at fault, therefore examine it and its argument using the M command.

```
M72,DØ,
MØØ73,FB,
```

The value in location 73 should be FD, therefore change it by typing FD<CR>.

f) Remove single instruction mode and breakpoints by typing N<CR> then B<CR>.

g) Restart the program by typing G52<CR>. The program should now run correctly.

Note that when using an alphanumeric keyboard, debugging is slightly easier. When the program sticks in a loop ESC can be used to return to the monitor (provided interrupts have not been disabled). Single instruction mode can then be set to determine the loop in which the program was running.

TABLE OF HEX ASCII CODES

ØØ	NUL		
Ø1	Control A	-	Home
Ø2	Control B		
Ø3	Control C		
Ø4	Control D		
Ø5	Control E		
Ø6	Control F		
Ø7	Control G	-	Bell
Ø8	Control H	-	Backspace
Ø9	Control I	-	Horizontal Tab - Cursor Right
ØA	Control J	-	Line Feed
ØB	Control K		
ØC	Control L	-	Page Clear - Form Feed
ØD	Control M	-	Carriage Return
ØE	Control N		
ØF	Control O		
1Ø	Control P		@
11	Control Q		
12	Control R		
13	Control S		
14	Control T		
15	Control U		
16	Control V		
17	Control W		
18	Control X		
19	Control Y		
1A	Control Z	-	Vertical Tab - Cursor Up
1B	S1		Esc
1C	S2		
1D	S3		
1E	S4		
1F	S5		

Note that the codes ØØ - 1F produce special symbols when used in display memory.

TABLE OF HEX ASCII CODES (CONTINUED)

2Ø	Space	4Ø	@	6Ø	'
21	!	41	A	61	a
22	"	42	B	62	b
23	£ or #	43	C	63	c
24	$	44	D	64	d
25	%	45	E	65	e
26	&	46	F	66	f
27	'	47	G	67	g
28	(	48	H	68	h
29	)	49	I	69	i
2A	*	4A	J	6A	j
2B	+	4B	K	6B	k
2C	,	4C	L	6C	l
2D	-	4D	M	6D	m
2E	.	4E	N	6E	n
2F	/	4F	O	6F	o
3Ø	Ø	5Ø	P	7Ø	p
31	1	51	Q	71	q
32	2	52	R	72	r
33	3	53	S	73	s
34	4	54	T	74	t
35	5	55	U	75	u
36	6	56	V	76	v
37	7	57	W	77	w
38	8	58	X	78	x
39	9	59	Y	79	y
3A	:	5A	Z	7A	z
3B	;	5B	[	7B	{
3C	<	5C	\	7C	¦
3D	=	5D	]	7D	}
3E	>	5E	^	7E	~
3F	?	5F	_	7F	■ or Rubout