using elbug

It is almost a year since the article on Elbug, the monitor software program for the Elektor SC/MP μP system was published. The original article concentrated on a description of the various control functions which Elbug provided, and did not examine how the program actually worked. Prompted partly by the many requests from readers, the following article takes a more detailed look at Elbug, describing how some of the more important subroutines function, and how these routines can profitably be incorporated into one's own programs.

(H. Huschitt)

Programming techniques

Writing programs for microcomputers is not difficult, providing one adopts the approach of breaking the program down into a number of smaller units which can be tackled individually. Just as a complex electronic circuit is built up from a number of separate components, so any large program is composed of a number of smaller routines and subroutines. This is also true of Elbug, which contains e.g. a display routine, which ensures that the hexadecimal representation of a data byte appears on the displays, a keyboard routine, which ensures that the correct code is generated when a particular key is depressed, and so on. Subroutines are implemented by jumping from the main program to the start address of the routine in question. At the end of the routine the microprocessor resumes main program execution by jumping back to the address of the main program instruction which follows the subroutine call.

In higher programming languages, such as, e.g. BASIC, there are special instructions, GOSUB (go to subroutine) and RETURN (return from subroutine), for these tasks. Certain microprocessors are also provided with similar instructions, however this is not the case with the SC/MP. The instruction which the SC/MP employs to initiate a subroutine is XPPC (Exchange Pointer with Program Counter). By loading the address of the subroutine in whichever pointer is specified, the above instruction will effect a jump to that routine, since the address in question is loaded into the program counter.

The SC/MP has of course three 16-bit pointer registers in addition to the program counter. Each of these pointers may be used as page pointers, stack pointers or subroutine pointers, however PTR 3 is unique in that, when the SC/MP senses an interrupt request (the enable interrupt line — Sense bit A in the Status Register — goes high) the SC/MP automatically executes an XPPC-3 instruction. Thus, after a valid interrupt, the next instruction executed will be that contained in the address held in PTR 3 (incremented by one). At the end of the interrupt routine the jump back to the main program is similarly effected by means of an XPPC-3 instruction. As a result of this interrupt facility, PTR 3 is conventionally assigned as the subroutine pointer. However, it is of course perfectly feasible to use the other two pointer registers to call subroutines from within the main program.

To implement a subroutine call, the subroutine pointer is actually loaded with the start address of the routine *minus one*. The reason for this is that the SC/MP increments the contents of the program counter *before* it fetches the next instruction. Thus:

```
LDI L(SUBR)-1
XPAL n
LDI H(SUBR)
XPAH n
```

Since the address contained in the subroutine pointer must be incremented in order to obtain the true start address of the subroutine, it is important that this operation does not require a carry from bit 11 to bit 12 of the address since the SC/MP will not perform such a carry. Thus, for example, if the start address of the subroutine is FØØØ, normally the address loaded into the

Table 1.

```
DELAY:
   LDI Ø8          ; load counter with 8
   ST COUNT
LOOP:
   DLY X'FF
   DLD COUNT
   JNZ LOOP        ; execute delay instruction 8 times
   XPPC 3          ; jump back to main program
   JMP DELAY       ; jump to start
COUNT:
   • BYTE          ; RAM byte as counter
```

Figure 1. This figure illustrates the functions assigned to the various locations in Elbug's software stack.

1

ADR	STACK	LDKB	GETHEX	PUTHEX
ØFFF	STAKPT, lower			
ØFFE	STAKPT, higher			
ØFFD	ROUTAD, lower			
ØFFC	ROUTAD, higher			
ØFFB	STFULL			
ØFFA	STDEEP			
ØFF9	STKEFF			
ØFF8	AC			
ØFF7	PTR, lower			
ØFF6	PTR, higher			
ØFF5	SPEED			
ØFF4				
ØFF3				
ØFF2				
ØFF1				
ØFFØ				
ØFEF				Counter
ØFEE				
ØFED				
ØFEC				7-Segm-
ØFEB				Code
ØFEA				
ØFE9		Key, binary		
ØFE8		Key-Code		
ØFE7		7-Segm-Code		
ØFE6				half
ØFE5			Keys	Bytes
ØFE4			binary	
ØFE3				
ØFE2			Byte, higher	ADR, higher
ØFE1			Byte, lower	ADR, lower
ØFEØ	STKBSE		Counter	DATA
ØFDF	AC			
ØFDE	E			
ØFDD	SR			
ØFDC	PTR 1 L			
ØFDB	PTR 1 H			
ØFDA	PTR 2 L	STATUS 1		
ØFD9	PTR 2 H			
ØFD8	PTR 3 L			
ØFD7	PTR 3 H			
ØFD6	ROUTAD L			
ØFD5	ROUTAD H			
ØFD4	AC			
ØFD3	E			
ØFD2	SR			
ØFD1	PTR 1 L			
ØFDØ	PTR 1 H			
ØFCF	PTR 2 L	STATUS 2		
ØFCE	PTR 2 H			
ØFCD	PTR 3 L			
ØFCC	PTR 3 H			
ØFCB	ROUTAD L			
ØFCA	ROUTAD H			
ØFC9	AC			
		STATUS 3 etc		

pointer would be FØØØ - 1 = EFFF. However in this instance the address thereby obtained would be incorrect, since, as stated, there can be no carry from bit 11 to bit 12 and the four highest address bits would remain unaltered (i.e. 'E'). The correct address to enter into the pointer is therefore FFFF.

Whilst the subroutine is being executed, PTR 3 will contain the address of the last instruction executed in the main program, i.e. the return address −1, assuming of course that the contents of the PTR are not altered by the subroutine. Thus an XPPC-3 instruction at the end of the subroutine will effect a return to main program execution. However, the address now held by PTR 3 will be that of the last instruction in the subroutine, which means that a subsequent XPPC-3 instruction would effect a jump to the end of the subroutine and not the start. For this reason the final instruction of almost every subroutine will be a jump back to the start of the routine.

A practical example of the above-described techniques is the delay routine listed in table 1. This routine can be used in the course of main program execution in order to avoid filling a large portion of program memory with delay instructions. If the delay routine is used repetitively, the subroutine call will be structured as follows:

```
.
.
.
JS 3* (DELAY) ; load PTR 3 and make
.               first jump to delay
.               routine
.
XPPC 3        ; second jump to delay
                routine
.
.
XPPC 3        ; third jump to delay
                routine
```

Unfortunately, the process is not quite as simple as might first appear. The contents of the accumulator are altered by the subroutine. Thus if the contents of the AC prior to the jump to delay routine are required later in the main program, they must first be stored somewhere. As long as it is simply the contents of the AC which must be preserved, this does not present any special problems, since they can easily be stored in the extension register. Unfortunately, however, the situation becomes slightly more complicated if the contents of the pointers themselves are altered in the course of a subroutine, since the return addresses to the main program will then be lost.

Thus it is necessary to store the return addresses at the beginning of the subroutine, and then re-enter these into the pointers at the end of the routine, so that an XPPC instruction will effect a return to the main program.

From a programming point of view it is extremely useful to be able to jump from the middle of one subroutine to a second subroutine, i.e. to 'nest' routines inside one another like chinese boxes. However for each jump that is made a return address must be stored, so that it must be possible to 'stack up' the return addresses somewhere in memory in order that they can be retrieved as required. Some microprocessors are provided with an integral on-chip stack, capable of storing up to 12 or 16 return addresses. This is not the case with the SC/MP, however, so that it is necessary to employ a 'software stack'.

Software lifo stack

A software stack is basically a routine which simulates the function of a stack

---

*JS 3 is a symbol for a 'pseudo instruction', i.e. a statement which results in the generation of several machine-language instructions — in this case the loading of PTR 3 and exchanging the contents of PC and PTR 3.

register, by employing a section of read/write memory as a scratch-pad store for the data to be saved.

The advantage of a software stack is that there need be virtually no limit to its *depth*, i.e. the number of return addresses it is capable of storing. In addition there is the possibility of storing the contents of other important registers, such as the AC or extension register, in the stack. The software stack of Elbug utilises the section of RAM between ØFC9 and ØFFF. This section was chosen since it can easily be addressed via the program counter from the beginning of that page of memory (i.e. from ØØØØ). In addition to return addresses the contents of all the CPU registers, with the exception of the PC, are stored on Elbug stack.

In order to store the status of all of the CPU registers 11 bytes of RAM are required. Figure 1 indicates which locations are reserved for this purpose. As can be seen, the stack contains sufficient space to store the status of each CPU register twice. Since Elbug only nests to a level of one subroutine (i.e. one subroutine called by another) this is sufficient. However a particular user's program may require several subroutines to be nested, in which case the stack can be extended downwards from ØFC9 as far as is desired.

The stack is organised on a 'last-in-first-out' (lifo) basis, and employs a 'stack pointer' – usually PTR 2 – to point to the last value pushed onto the stack. A 'stack routine' is required to write the contents of the CPU registers into the stack, and in order to ensure that the stack pointer can be used during a subroutine and the stack address still be preserved, the status of the stack pointer (STAKPT) is itself stored in locations ØFFF and ØFFE at the top of the stack (see figure 1). When Elbug is started, the address ØFEØ is written into these locations; this location represents the 'base' of the stack. The section of stack from ØFFF to ØFEØ is fixed, however below this point the stack can be expanded or contracted as required.

In a user's program which contains a large number of nested interrupts, there exists the danger of the dynamic portion of the stack being extended downwards to the point where it overlaps a user's program stored from ØCØØ onwards. In order to prevent such an eventuality, a stack counter (STKEFF) is maintained, which is incremented or decremented each time a byte is pushed onto or pulled off the stack. In addition, a byte of RAM is reserved which, via the MODIFY routine or the user's program, can be used to specify the number of bytes of status information which may be stored on the stack.

This byte, which effectively determines the depth of the stack, is stored in location ØFFA (STDEEP) – see figure 1. This byte is compared with the contents of the stack counter each time a stack operation is performed, and when the effective stack depth (STEFF) equals

Table 2.

Elbug STACK routines

```
        Ø7ØØ   DISPL    = Ø7ØØ        ; EA of display
        ØFFF   STAKPT   = ØFFF        ; 2 bytes for current contents of stack ptr
        ØFFD   ROUTAD   = ØFFD        ; 2-byte address of subroutine
        ØFFB   STFULL   = ØFFB        ; 'stack-full' flag
        ØFFA   STDEEP   = ØFFA        ; 1 byte to set stack depth
        ØFF9   STKEFF   = ØFF9        ; current stack depth
        ØFF8   AC       = ØFF8        ; scratch-pad for (ac)
        ØFF7   PTR      = ØFF7        ; scratch-pad for (ptr)
        ØFF5   SPEED    = ØFF5        ; speed of cassette routine
        ØFEØ   STKBSE   = ØFEØ        ; stack base
        ØØØØ   . = ØØØØ
               STACK:
ØØØØ    Ø8       NOP
ØØØ1    C415     LDI X'15             ; set cassette speed
ØØØ3    C8F1     ST SPEED             ; to 600 bits/sec
ØØØ5    C4EØ     LDI L (STKBSE)       ; set stack ptr to
ØØØ7    C8F7     ST STAKPT            ; stack base
ØØØ9    C4ØF     LDI H (STKBSE)
ØØØB    C8F2     ST STAKPT-1
ØØØD    C4ØØ     LDI ØØ
ØØØF    C8E9     ST STKEFF            ; set stack counter to Ø
ØØ11    C8E9     ST STFULL            ; stack-full byte = Ø
ØØ13    9Ø3D     JMP $1               ; jump to 'elbug'
               PULL:                  ; pull status off stack
ØØ15    CØE9     LD STAKPT            ; load ptr 1 with current
ØØ17    31       XPAL 1               ; contents of stack ptr
ØØ18    CØE5     LD STAKPT-1
ØØ1A    35       XPAH 1
ØØ1B    C5Ø1     LD @1 (1)            ; load routine-address from
ØØ1D    C8DE     ST ROUTAD-1          ; stack into 'routad'
ØØ1F    C5Ø1     LD @1 (1)
ØØ21    C8DB     ST ROUTAD
ØØ23    C5Ø1     LD @1 (1)            ; load ptr 3 from stack
ØØ25    37       XPAH 3
ØØ26    C5Ø1     LD @1 (1)
ØØ28    33       XPAL 3
ØØ29    C5Ø1     LD @1 (1)            ; load ptr 2 from stack
ØØ2B    36       XPAH 2
ØØ2C    C5Ø1     LD @1 (1)
ØØ2E    32       XPAL 2
ØØ2F    C5Ø1     LD @1 (1)            ; load (ptr 1) from stack
ØØ31    C8C4     ST PTR-1             ; into scratch-pad
ØØ33    C5Ø1     LD @1 (1)
ØØ35    C8C1     ST PTR
ØØ37    C5Ø1     LD @1 (1)            ; load s-register from stack
ØØ39    Ø7       CAS
ØØ3A    C5Ø1     LD @1 (1)            ; load e-register from stack
ØØ3C    Ø1       XAE
ØØ3D    C5Ø1     LD @1 (1)            ; load (ac) from stack into scratch-pad
ØØ3F    C8B8     ST AC
ØØ41    CØB4     LD PTR-1             ; store current contents of stack
ØØ43    35       XPAH 1               ; ptr in 'stakpt' and load ptr 1 from
ØØ44    C8B9     ST STAKPT-1          ; scratch-pad
ØØ46    CØBØ     LD PTR
ØØ48    31       XPAL 1
ØØ49    C8B5     ST STAKPT
ØØ4B    B8AD     DLD STKEFF           ; update stack counter
ØØ4D    CØAA     LD AC                ; load ac from scratch-pad
ØØ4F    3F       XPPC 3               ; return
ØØ5Ø    9ØØ4     JMP PUSH
                  $1:
ØØ52    9Ø4D     JMP START            ; 'jump-assist'
                  $2:
ØØ54    9ØBF     JMP PULL             ; ditto
                  PUSH:               ; push status onto stack
ØØ56    C8A1     ST AC                ; store (ac) in scratch-pad
ØØ58    CØA6     LD STAKPT
ØØ5A    33       XPAL 3               ; store (ptr 3) in scratch-pad and
ØØ5B    C89B     ST PTR               ; load ptr 3 as stack pointer
ØØ5D    CØAØ     LD STAKPT-1
ØØ5F    37       XPAH 3
ØØ6Ø    C895     ST PTR-1
ØØ62    C4FF     LDI L (STAKPT)       ; push (ptr 1) onto stack and
ØØ64    31       XPAL 1               ; load ptr 1 as ram pointer
ØØ65    CFFC     ST @-4 (3)
```

Table 2, continued.

ØØ67	C4ØF	LDI h (STAKPT)	
ØØ69	35	XPAH 1	
ØØ6A	CFFF	ST @-1 (3)	
ØØ6C	Ø1	XAE	; push (e) onto stack
ØØ6D	CBØ3	ST 3 (3)	
ØØ6F	Ø6	CSA	; push (sr) onto stack
ØØ7Ø	CBØ2	ST 2 (3)	
ØØ72	C1F9	LD-7 (1)	; (ac) from scratch-pad onto stack
ØØ74	CBØ4	ST 4 (3)	
ØØ76	32	XPAL 2	; push (ptr 2) onto stack
ØØ77	CFFF	ST @-1 (3)	
ØØ79	36	XPAH 2	
ØØ7A	CFFF	ST @-1 (3)	
ØØ7C	C1F8	LD-8 (1)	; (ptr 3) from scratch-pad onto
ØØ7E	CFFF	ST @-1 (3)	; stack
ØØ8Ø	C1F7	LD-9 (1)	
ØØ82	CFFF	ST @-1 (3)	
ØØ84	C1FE	LD-2 (1)	; routine-address from 'routad'
ØØ86	CFFF	ST @-1 (3)	; onto stack
ØØ88	C1FD	LD-3 (1)	
ØØ8A	CFFF	ST @-1 (3)	; load ptr 3 with routine address and
ØØ8C	37	XPAH 3	; store current contents of stack ptr
ØØ8D	C9FF	ST-1 (1)	; (from ptr 3) in 'stakpt'
ØØ8F	C1FE	LD-2 (1)	
ØØ91	33	XPAL 3	
ØØ92	C9ØØ	ST Ø (1)	
ØØ94	A9FA	ILD-6 (1)	; update stack counter and
ØØ96	E1FB	XOR-5 (1)	; compare with preset stack depth
ØØ98	9CØ4	JNZ $3	
ØØ9A	C4FF	LDI X'FF	; set 'stack-full' flag
ØØ9C	C9FC	ST-4 (1)	
		$ 3:	
ØØ9E	3F	XPPC 3	; jump to subroutine
ØØ9F	9ØB3	JMP $2	

Table 3. LDBYTE routine

		.LOCAL	
		.PAGE	
		LDBYTE:	; routine: fetch one byte from cassette
Ø1D1	C215	LD X'15 (2)	
Ø1D3	1C	SR	; speed: 2 to ram
Ø1D4	CA14	ST X'14 (2)	
		$ 1:	
Ø1D6	C4FF	LDI X'FF	
Ø1D8	Ø1	XAE	
Ø1D9	19	SIO	; give stop bit
Ø1DA	4Ø	LDE	
Ø1DB	94Ø2	JP $ 2	; wait for start bit
Ø1DD	9ØF7	JMP $ 1	
		$ 2:	
Ø1DF	C4FF	LDI X'FF	
Ø1E1	Ø1	XAE	
Ø1E2	C214	LD X'14 (2)	; copyspeed/2
Ø1E4	CAØA	ST 1Ø (2)	
		$ 3:	
Ø1E6	BAØA	DLD 1Ø (2)	; 1/2 bit delay
Ø1E8	9CFC	JNZ $ 3	
Ø1EA	C4Ø8	LDI Ø8	; load bit-counter
Ø1EC	CAØ8	ST 8 (2)	
		$ 4:	
Ø1EE	C215	LD X'15 (2)	; copy speed
Ø1FØ	CAØ9	ST 9 (2)	
Ø1F2	C416	LDI 22	; delay 114 μs (sc/mp 1)
Ø1F4	8FØØ	DLY ØØ	
		$ 5:	
Ø1F6	BAØ9	DLD 9 (2)	; decrement speed
Ø1F8	9CFC	JNZ $ 5	
Ø1FA	19	SIO	; accept bit
Ø1FB	BAØ8	DLD 8 (2)	
Ø1FD	9CEF	JNZ $ 4	; 8 bits accepted
Ø1FF	C215	LD X'15 (2)	
Ø2Ø1	CAØ9	ST 9 (2)	
		$ 6:	
Ø2Ø3	BAØ9	DLD 9 (2)	; decrement speed (1 x 66 μs)
Ø2Ø5	9CFC	JNZ $ 6	; (sc/mp 1)
82Ø7	4Ø	LDE	; load byte in ac
Ø2Ø8	3F	XPPC 3	; return
Ø2Ø9	9ØC6	JMP LDBYTE	; jump for next pass

the preset maximum stack depth (STDEEP), this condition is flagged by loading X'FF into ØFFB (≙STFULL). The STFULL flag can be tested by the user's program, and if desired set, so that subsequent jumps to subroutine are prevented.

The stack routines in Elbug which are responsible for storing the contents of the CPU registers before a subroutine is executed and retrieving same after the subroutine is finished are designated the PUSH and PULL routines respectively. A complete listing for both routines is provided in table 2, whilst figure 2 illustrates the timing sequence of the routines.

The end of the PUSH routine contains the instructions required to effect the jump to subroutine, thus it is important that the start address of the subroutine in question is first stored on the stack for reference. The 16-bit start address (−1) is loaded into locations ØFFD and ØFFC (ROUTAD).

With the aid of Elbug's stack and the PUSH and PULL routines a jump to subroutine can be implemented as follows:

- the start address (−1) of the subroutine is loaded into the appropriate locations (ROUTAD).
  (if the user's program has not yet caused the contents of PTR 2 to be altered, the ROUTAD bytes can be loaded via it. Upon pressing the RUN key and leaving Elbug, PTR 2 is automatically loaded with the address of the stack base (ØFEC). The displacement values X'1C and X'1D will reference the higher and lower ROUTAD locations respectively. If PTR 2 has already been used, then effective addresses can be obtained via PTR 3, since the latter will contain the start address (−1) of PUSH. The relative addresses (displacements) are then X'A8 and X'A7 respectively.
- PTR 3 should be loaded with the start address (−1) of the PUSH routine (ØØ55)
- If the above steps have been taken, the actual subroutine jump can be effected by an XPPC-3 instruction.

The program will now jump to PUSH, causing the current contents of the SC/MP's registers to be stored on the stack, whereupon the subroutine will be executed. This subroutine may use any register, the user need have no fears for their original contents. However it is worth noting that it is impossible to transfer data from the main program to a subroutine via one of the CPU registers (and vice versa).

The above procedure will enable a subroutine to be called and implemented under any circumstances. However there are situations where the process is even simpler.

- If the same subroutine is called by the main programme more than once without a second subroutine being called in between, then one need not load the ROUTAD addresses anew

Table 4.

```
BYTOUT

                .LOCAL
                BYTOUT:          ; copy one byte to cassette
Ø5D8   CAØ7      ST 7 (2)        ; store byte in ram
Ø5DA   C4ØB      LDI 11
Ø5DC   CAØ8      ST 8 (2)        ; load bit-counter
Ø5DE   C4ØØ      LDI ØØ
Ø5EØ   Ø1        XAE
Ø5E1   19        SIO             ; supply start bit
Ø5E2   Ø1        XAE
Ø5E3   BA2Ø      DLD X'2Ø (2)
Ø5E5   C2Ø7      LD 7 (2)
Ø5E7   Ø1        XAE             ; byte to e
                $ 1:
Ø5E8   C4ØB      LDI 11
Ø5EA   8FØØ                      ; delay 70 µs (sc/mp 1)
Ø5EC   C215      LD X'15 (2)     ; copy 'speed'
Ø5EE   CAØ9      ST 9 (2)
                $ 2:
Ø5FØ   BAØ9      DLD 9 (2)       ; decrement speed
Ø5F2   9CFC      JNZ $ 2
Ø5F4   19        SIO             ; shift bit out
Ø5F5   4Ø        LDE
Ø5F6   DC8Ø      ORI X'8Ø        ; add stop bit to byte
Ø5F8   Ø1        XAE
Ø5F9   BAØ8      DLD 8 (2)
Ø5FB   9CEB      JNZ $ 1         ; if bit-counter = Ø, continue
Ø5FD   3F        XPPC 3          ; return
Ø5FE   90D8      JMP BYTOUT      ; jump for next pass
```

Table 5.

```
LDKB routine

                •PAGE            'load keyboard' routine
                •LOCAL
                LDKB:
Ø2ØB   C414      LDI L(PULL)—1   ; prepare ptr 3
Ø2ØD   33        XPAL 3
Ø2ØE   C4ØØ      LDI H(PULL)
Ø21Ø   37        XPAH 3
                LDKB1:           ; label for start address without stack
Ø211   C4Ø1      LDI L(DISPL)+1
Ø213   31        XPAL 1
Ø214   C4Ø7      LDI H(DISPL)    ; prepare ptr 1 and ptr 2
Ø216   35        XPAH 1
Ø217   C4EØ      LDI L(STKBSE)
Ø219   32        XPAL 2
Ø21A   C4ØF      LDI H(STKBSE)
Ø21C   36        XPAH 2
                $ 1:
Ø21D   C1Ø8      LD 8 (1)
Ø21F   94FC      JP $ 1          ; wait for key closure
Ø221   8F1E      DLY 3Ø          ; debounce time — approx. 30 ms
Ø223   C1Ø8      LD 8 (1)
Ø225   CAØ8      ST 8 (2)        ; store keyboard code in ram
Ø227   D4ØF      ANI ØF
Ø229   CAØ9      ST 9 (2)        ; store binary value of key in ram and
Ø22B   Ø1        XAE             ; in e
                $ 2:
Ø22C   C1Ø8      LD 8 (1)
Ø22E   94Ø2      JP $ 3          ; wait for key release
Ø23Ø   9ØFA      JMP $ 2
                $ 3:
Ø232   8F1E      DLY 3Ø          ; debounce time
Ø234   C41F      LDI L (TAB)
Ø236   31        XPAL 1
Ø237   C4Ø1      LDI H (TAB)
Ø239   35        XPAH 1
Ø23A   C18Ø      LD - 128(1)     ; fetch 7-segment code
Ø23C   CAØ7      ST 7 (2)        ; store in ram
Ø23E   3F        XPPC 3          ; return
```

each time, since once loaded they remain unaltered. If a second subroutine is called whose address is within ¼ K of the first, then only the lower order address byte need be loaded (ROUTAD low)

- If the contents of PTR 3 are not altered by the main program between jumps to one or more subroutine, then the jump to the PUSH routine can be realised via an XPPC 3 instruction.
- If both of the above conditions apply — which is not infrequently the case — a single XPPC instruction will save the status of the CPU registers and effect a jump to the subroutine!

To return from a subroutine to main program (or the previous subroutine), the address of the PULL routine (start address ØØ13) is loaded into PTR 3 at the end of the routine in question. If the subroutine does not alter the contents of PTR 3 (again, this will often be the case), the latter will contain the last instruction of the PUSH routine. Since this is in fact a jump to the start address of the PULL routine (see table 2), a single XPPC 3 instruction at the end of the subroutine will cause a return to the main program.

A similar instruction is present at the end of the PULL routine, namely JMP PUSH. This means once PTR 3 has been loaded with the start address (−1) of PUSH, and assuming its contents are not affected by the subroutine, then jumps to and from the subroutine can always be implemented using just an XPPC 3 instruction. The subroutine procedure described above remains valid for external subroutine calls, i.e. interrupt requests. It goes without saying, however, that the interrupt line is not enabled until the ROUTAD bytes and PTR 3 have been loaded. Only then will the XPPC 3 instruction generated by the interrupt cause a jump to subroutine to be implemented.

If several interrupt inputs are used, the software required to recognise the priority of simultaneous interrupt requests must be included in the subroutine. This software was discussed in an earlier article in the SC/MP series (see Elektor 33, January 1977).

Series/parallel and parallel/series conversion routines

Via the extension register and the SIO (Serial Input/Output) instruction the SC/MP offers the user the possibility of serial/parallel and parallel/serial conversion without the need for additional hardware. The appropriate routines are already contained in Elbug, since they are required when transferring data to and from the cassette interface.

The 'load byte' routine (LDBYTE, see table 3) will load a serial data byte, including start and stop bits, via the serial input (SIN) into the extension register. As was explained in part 5 of the series on the SC/MP system (see Elektor 35, March 1978), the rate at which the data is transferred can be varied. This is done by altering the contents of the SPEED-address (OFF 5). Once LDBYTE has been executed the serial data word is available in parallel form in both the AC and extension register. During this routine a stop bit is present continuously at the serial output (SOUT).

The 'byte out' routine (BYTOUT, see table 4) enables a byte to be transmitted in serial form – along with start and stop bits – from the serial output. Once again the transmission rate can be varied with the aid of the SPEED byte. In the case of both the LDBYTE and BYTOUT routines the data is coded in ASCII format, i.e. one start bit, eight data bits and one stop bit. Data presented at the serial input during execution of the BYTOUT routine is ignored, which means that using these routines the SC/MP may only be operated in the half-duplex mode.

The routines can be employed in a variety of applications such as, e.g. to interface to a TTY or telex. The routines are initiated not by the Elbug stack routines, but in the manner illustrated in the case of the delay routine described earlier. The user thus has the possibility of inputting and outputting information via the CPU registers. In the case of the BYTOUT routine it is in fact necessary that the byte to be transmitted be loaded into the AC under main program control.

In addition to PTR 3, which is used in jumping to both routines, PTR 2 is also required. Before the jump to either routine this pointer is loaded with ØFEØ, since it is via this pointer that the SPEED byte is referenced. Both routines leave PTR 1 unaltered.

The keyboard routine

This routine, the listing for which is given in table 5, is designed to scan the keyboard. An interesting feature of the routine is that it has two start addresses:

LDKB = Ø2ØB the start address when called by the stack routines

LDKB1 = Ø211 the start address when called by other than the stack routines.

In the latter case PTR 3 is loaded with the address (−1) LDKB1, and the routine started by an XPPC 3 instruction. PTR 3 must be loaded with the appropriate address prior to each jump, since there is no JMP LDKB1 instruction. In both cases the jump back to the main routine is only implemented after the key has been released.

When called by other than via the stack, at the end of the keyboard routine the binary equivalent of the hex data key which has been pressed is available in the extension register, whilst the corresponding 7-segment code is present in the AC. The code generated by the keyboard hardware is written into address ØFE8.

If the routine is called via the stack, then it is not possible to transfer information out of the keyboard routine, via the SC/MP registers, into the main program. The above information is only available at the RAM locations reserved for this purpose, namely ØFE7 to ØFE9 (see figure 1). If desired, the table of 7-segment code can be used separately. For organisational reasons it is not included in LDKB, but is stored in memory from location Ø11F.

The GETHEX routine

This routine itself calls the above-described LDKB routine four times in order to store the four hexadecimal numbers successively generated by the keyboard. These four hex digits are joined together to form two bytes, which are stored at addresses ØFE1 (lower byte) and ØFE2 (higher byte). The start address of the GETHEX routine is Ø23F (see table 6) and the routine is initiated via the stack routines. In order to jump to the routine from the main program the start address (−1) is loaded into location ROUTAD, and

Table 6. GETHEX routine

```
                    •PAGE
                    •LOCAL
                    GETHEX:
Ø23F   C4Ø6          LDI L (DISPL)+6      ; load ptr 1 with address
Ø241   31            XPAL 1               ; of display 6
Ø242   C4Ø7          LDI H (DISPL)
Ø244   35            XPAH 1
Ø245   C4E7          LDI L (STKBSE)+7     ; load ptr 2 with
Ø247   32            XPAL 2               ; stack base + 7
Ø248   C4ØF          LDI H (STKBSE)
Ø24A   36            XPAH 2
Ø24B   C4Ø4          LDI Ø4               ; load key-counter
Ø24D   CAF9          ST - 7 (2)
                    $ 1:
Ø24F   C455          LDI L (PUSH)−1
Ø251   33            XPAL 3
Ø252   C4ØØ          LDI H (PUSH)
Ø254   37            XPAH 3
Ø255   C4ØA          LDI L (LDKB)−1
Ø257   CBA8          ST - 88 (3)
Ø259   C4Ø2          LDI H (LDKB)
Ø25B   CBA7          ST - 89 (3)
Ø25D   3F            XPPC 3               ; call 'ldkb' (via stack)
Ø25E   C4EØ          LDI L (STKBSE)
Ø26Ø   33            XPAL 3               ; ptr 3 becomes ram pointer
Ø261   C4ØF          LDI H (STKBSE)
Ø263   37            XPAH 3
Ø264   C3Ø7          LD 7 (3)             ; fetch 7-segment code
Ø266   CDFF          ST @-1 (1)           ; write to display 5 (4, 3, 2)
Ø268   C4ØØ          LDI ØØ
Ø26A   C9FF          ST - 1 (1)           ; blank all other displays
Ø26C   C9FE          ST - 2 (1)
Ø26E   C9FD          ST - 3 (1)
Ø27Ø   C9FC          ST - 4 (1)
Ø272   C9FB          ST - 5 (1)
Ø274   C3Ø9          LD 9 (3)             ; binary value of key to ram table
Ø276   CEFF          ST @-1 (2)
Ø278   BBØØ          DLD Ø (3)
Ø27A   9CD3          JNZ $ 1              ; 4 keys?
Ø27C   C48Ø          LDI X'8Ø
Ø27E   C9FF          ST - 1 (1)           ; '•' to displays 0,1
Ø28Ø   C9FE          ST - 2 (1)
Ø282   C3Ø6          LD 6 (3)             ; form higher byte
Ø284   1E            RR
Ø285   1E            RR
Ø286   1E            RR
Ø287   1E            RR
Ø288   Ø1            XAE
Ø289   C3Ø5          LD 5 (3)
Ø28B   58            ORE
Ø28C   CBØ2          ST 2 (3)
Ø28E   C3Ø4          LD 4 (3)             ; form lower byte
Ø29Ø   1E            RR
Ø291   1E            RR
Ø292   1E            RR
Ø293   1E            RR
Ø294   Ø1            XAE
Ø295   C3Ø3          LD 3 (3)
Ø297   58            ORE
Ø298   CBØ1          ST 1 (3)
                    JSPULL:               ; assist-label for return via 'pull'
Ø29A   C4ØØ          JS 3 (PULL)          ; to main program
Ø29C   37
Ø29D   C414
Ø29F   33
Ø2AØ   3F
```

Figure 2. This diagram shows the sequence in which the PUSH and PULL routines of Elbug store and retrieve status information when subroutines are called.

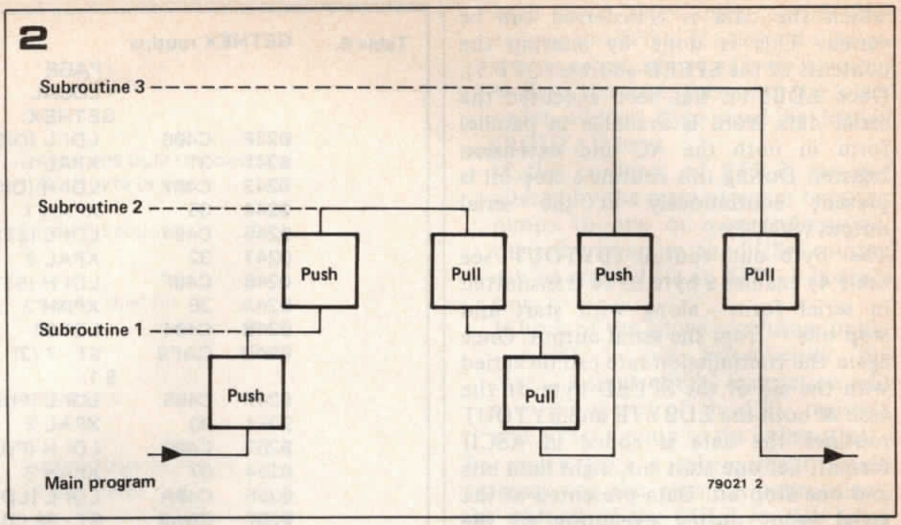

Table 7.

PUTHEX routine

```
              • PAGE              ; puthex routine
              • LOCAL
              PUTHEX:
Ø2A1  C4EØ    LDI L (STKBSE)
Ø2A3  33      XPAL 3              ; load ptr 3 as ram pointer
Ø2A4  C40F    LDI H (STKBSE)
Ø2A6  37      XPAH 3
Ø2A7  C4EØ    LDI L (STKBSE)
Ø2A9  32      XPAL 2              ; prepare ptr 2 and ptr 1 for
Ø2AA  C40F    LDI H (STKBSE)      ; auto-indexed addressing
Ø2AC  36      XPAH 2
Ø2AD  C4E3    LDI L (STKBSE)+3
Ø2AF  31      XPAL 1
Ø2BØ  C40F    LDI H (STKBSE)
Ø2B2  35      XPAH 1
Ø2B3  C403    LDI Ø3              ; load byte-counter
Ø2B5  CBØF    ST ØF (3)
              $ 1:
Ø2B7  C2ØØ    LD Ø (2)            ; fetch first (next) byte
Ø2B9  D40F    ANI ØF              ; bits Ø - 3 to
Ø2BB  CDØ1    ST @-1 (1)          ; ram
Ø2BD  C6Ø1    LD @-1 (2)          ; fetch same byte again,
Ø2BF  1C      SR                  ; and bits 4-7
Ø2CØ  1C      SR
Ø2C1  1C      SR
Ø2C2  1C      SR
Ø2C3  CDØ1    ST @-1 (1)          ; to next ram address
Ø2C5  BBØF    DLD ØF (3)
Ø2C7  9CEE    JNZ $ 1             ; 3rd byte stored?
Ø2C9  C41F    LDI L (TAB)—1
Ø2CB  31      XPAL 1              ; prepare ptr 1 for indirect addressing
Ø2CC  C401    LDI H (TAB)
Ø2CE  35      XPAH 1
Ø2CF  C4Ø6    LDI Ø6              ; load hex-character-counter
Ø2D1  CBØF    ST ØF (3)
              $ 2:
Ø2D3  C6Ø1    LD @-1 (2)          ; fetch first (next) half-byte
Ø2D5  Ø1      XAE
Ø2D6  C18Ø    LD - 128 (1)        ; fetch 7-segment code
Ø2D8  CAØ5    ST 5 (2)            ; load in ram
Ø2DA  BBØF    DLD ØF (3)
Ø2DC  9CF5    JNZ $ 2             ; 6 digits ready?
Ø2DE  C4ØØ    LDI L (DISPL)
Ø2EØ  31      XPAL 1              ; load ptr 1 with address of displays
Ø2E1  C4Ø7    LDI H (DISPL)
Ø2E3  35      XPAH 1
Ø2E4  C4Ø6    LDI Ø6              ; load counter
Ø2E6  CBØF    ST ØF (3)
              $ 3:
Ø2E8  C6Ø1    LD @-1 (2)          ; 7-segment code to
Ø2EA  CDØ1    ST @-1 (1)          ; display 5
Ø2EC  BBØF    DLD ØF (3)
Ø2EE  9CF8    JNZ $ 3             ; ready?
Ø2FØ  9ØA8    JMP JSPULL          ; via assist-label to 'pull'
```

PTR 3 is loaded with the start address (—1) of the PUSH routine, whereupon an XPPC 3 can be executed.
When the routine is finished, the two above-mentioned bytes can be read out of their locations in memory (ØFE1 and ØFE2) to be used later in the program. Care should be taken to ensure that the data is retrieved before the GETHEX routine is called again, otherwise two new bytes will be written into these locations and the previous data will be lost.

PUTHEX routine

The last routine to be examined is also the simplest. The PUTHEX routine does nothing more than convert the contents of memory locations ØFE0 to ØFE2 into the equivalent 7-segment code, and then display the results as a six-digit hexadecimal number. The code for the four lowest-order bits of address ØFEØ appears on display 2 (third from the right), the code for the next four bits on display 3, and so on. The start address of PUTHEX is Ø2A1 (see table 7), and the routine may only be called via the PUSH routine in the fashion described above.
That concludes the discussion of Elbug routines which can be called by a user's program. As one might have imagined, the entire monitor program has not been analysed, since the remaining routines cannot be used outside of Elbug. It is hoped that the above article will not only reveal how the routines which have been discussed can be profitably incorporated into one's own programs, but also that studying these routines will lead the aspiring programmer to an understanding of the various techniques involved, and enable him to tackle longer and more sophisticated programming tasks. ◼