Chapter 9

CPU Registers

9.1 CPU Registers

All of the MSP430 CPU registers can be used with all instructions.

9.1.1 The Program Counter PC

One of the main differences from other microcomputer architectures relates to the Program Counter (CPU register R0) that can be used as a normal register with the MSP430. This means that all of the instructions and addressing modes can be used with the Program Counter too. A branch, for example, is made by simply moving an address into the PC:

```
MOV    #LABEL,PC     ; Branch to address LABEL
MOV    LABEL,PC      ; Branch to the address contained in address LABEL
MOV    @R14,PC       ; Branch indirect, indirect R14
```

Note:

The Program Counter always points to even addresses. This means that the LSB is always zero. The software has to ensure that no odd addresses are used if the Program Counter is involved. Odd PC addresses will result in non-predictable behavior.

9.1.2 Stack Processing

9.1.2.1 Use of the System Stack Pointer (SP)

The system stack pointer (CPU register R1) is a normal register like the others. This means it can use the same addressing modes. This gives good access to all items on the stack, not only to the one on the top of the stack.

The system stack pointer (SP) is used for the storage of the following items:

- ❏ Interrupt return addresses and status register contents
- ❏ Subroutine return addresses
- ❏ Intermediate results
- ❏ Variables for subroutines, floating point package etc.

When using the system stack, remember that the microcomputer hardware also uses the stack pointer for interrupts and subroutine calls. To ensure the error-free running of the program it is necessary to do exact *housekeeping* for the system stack.

Note:

The Stack Pointer always points to even addresses. This means the LSB is always zero. The software has to ensure that no odd addresses are used if the Stack Pointer is involved. Odd SP addresses will end up in non-predictable results.

If bytes are pushed on the system stack, only the lower byte is used, the upper byte is not modified.

```
PUSH      #05h          ; 0005h –> TOS
PUSH.B    #05h          ; xx05h –> TOS
MOV.B     1(SP),R5      ; Address odd byte
```

9.1.2.2 Software Stacks

Every register from R4 to R15 can be used as a software stack pointer. This allows independent stacks for jobs that have a need for this. Every part of the RAM can be used for these software stacks.

EXAMPLE: R4 is to be used as a software stack pointer.

```
MOV    #SW_STACK,R4  ; Init. SW stack pointer
...
DECD   R4            ; Decrement stack pointer
MOV    item,0(R4)    ; Push item on stack
...                  ; Proceed
MOV    @R4+,item2    ; Pop item from stack
```

Software stacks can be organized as byte stacks also. This is not possible for the system stack, which always uses 16-bit words. The example shows R4 used as a byte stack pointer:

```
MOV#SW_STACK,R4     ; Init. SW stack pointer
...
DEC R4              ; Decrement stack pointer
MOV.B               item,0(R4)    ; Push item on stack
...                 ; Proceed
MOV.B  @R4+,item2   ; Pop item from stack
```

9.1.3 Byte and Word Handling

Every memory word is addressable by three addresses as shown in the Figure 9–1:

- ❏ The word address: An even address N
- ❏ The lower byte address: An even address N
- ❏ The upper byte address: An odd address N+1

If byte addressing is used, only the addressed byte is affected. No carry or overflow can affect the other byte.

Note:

Registers R0 to R15 do not have an address but are treated in a special way. Byte addressing always uses the lower byte of the register. The upper byte is set to zero if the instruction modifies the destination. Therefore, all instructions clear the upper byte of a destination register except CMP.B, TST.B, BIT.B and PUSH.B. The source is never affected.

The way an instruction treats data is defined with its extension:

- ❏ The extension .B means byte handling
- ❏ The extension .W (or none) means word handling

EXAMPLES: The next two software lines are equivalent. The 16-bit values read in absolute address 050h are added to the value in R5.

```
ADD    &050h,R5          ; ADD 16–BIT VALUE TO R5
ADD.W  &050h,R5          ; ADD 16–BIT VALUE TO R5
```

The 8-bit value read in the lower byte of absolute address 050h is added to the value contained in the lower byte of R5. The upper byte of R5 is set to zero.

```
ADD.B  &050h,R5          ; ADD 8–BIT VALUE TO R5
```

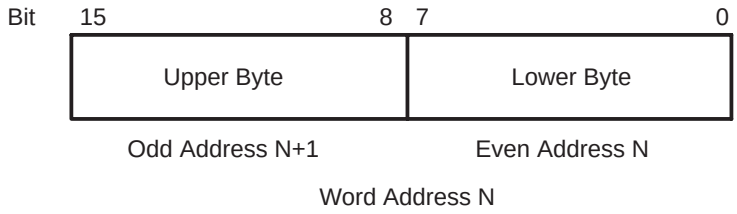


Figure 9–1. Word and Byte Configuration

If registers are used with byte instructions the upper byte of the destination register is set to zero for all instructions except CMP.B, TST.B, BIT.B and PUSH.B. It is therefore necessary to use word instructions if the range of calculations can exceed the byte range.

EXAMPLE: The two signed bytes OP1 and OP2 have to be added together and the result stored in word OP3.

```
MOV.B     OP1,R4         ; Fetch 1st operand
SXT       R4             ; Change to word format
MOV.B     OP2,OP3        ; Fetch 2nd operand
SXT       OP3            ; Change to word format
ADD.W     R4,OP3         ; 16–bit result to OP3
```

9.1.4 Constant Generator

A statistical look at the numbers used with the Immediate Mode shows that a few small numbers are in use most often. The six most often used numbers can be addressed with the four addressing modes of R3 (constant generator 2) and with the two not usable addressing modes of R2 (status register). The six constants that do not need an additional 16-bit word when used with the immediate mode are:

Table 9–1. Constant Generator

NUMBER	EXPLANATION	HEXADECIMAL	REGISTER	FIELD AD
+0	Zero	0000h	R3	00
+1	Positive one	0001h	R3	01
+2	Positive two	0002h	R3	10
+4	Positive four	0004h	R2	10
+8	Positive eight	0008h	R2	11
–1	Negative one	FFFFh	R3	11

The assembler inserts these ROM-saving addressing modes automatically when one of the previously described immediate constants is encountered. But, only immediate constants are replaceable this way, not (for example) index values.

If an immediate constant out of the constant generator is used, the execution time is equal to the execution time of the register mode.

The most often used bits of the peripheral registers are located in the bits addressable by the constant generator whenever possible.

9.1.5 Addressing

The MSP430 allows seven addressing modes for the source operand and four addressing modes for the destination. The addressing modes used are:

Table 9–2. Addressing Modes

ADDRESS BITS		SOURCE MODES	DESTINATION MODES	EXAMPLE
src	dst			
00	0	Register	Register	R5
01	1	Indexed	Indexed	TAB(R5)
01	1	Symbolic	Symbolic	TABLE
01	1	Absolute	Absolute	&BTCTL
10	–	Indirect	–	@R5
11	–	Indirect autoincrement	–	@R5+
11	–	Immediate	–	#TABLE

The three missing addressing modes for the destination operand are not of much concern for the programming. The reason is:

Immediate Mode: Not necessary for the destination; immediate operands can always be placed into the source. Only in a very few cases it is necessary to have two immediate operands in one instruction

Indirect Mode: If necessary, the Indexed Mode with an index of zero is usable. For example:

```
ADD    #16,0(R6)        ; @R6 + 16 –> @R6
CMP    R5,0(SP)         ; R5 equal to TOS?
```

The second previously shown example can be written in the following way, saving 2 bytes of ROM:

```
CMP    @SP,R5           ; R5 equal to TOS? (R5–TOS)
```

Indirect Autoincrement Mode: With table processing, a method that saves ROM space and reduces the number of used registers to one can be used:

EXAMPLE: The content of TAB1 is to be written into TAB2. TAB1 ends at the word preceding TAB1END.

```
      MOV    #TAB1,R5               ; Initialize pointer
LOOP  MOV.B  @R5+,TAB2–TAB1–1(R5)   ; Move TAB1 –> TAB2
      CMP    #TAB1END,R5            ; End of TAB1 reached?
      JNE    LOOP                   ; No, proceed
...                                 ; Yes, finished
```

The previous example uses only one register instead of two and saves three words due to the smaller initialization part. The normally written, longer loop is shown in the following

```
      MOV    #TAB1,R5               ;Initialize pointers
      MOV    #TAB2,R6
LOOP  MOV.B  @R5+,0(R6)             ;Move TAB1 –> TAB2
      INC    R6
      CMP    #TAB1END,R5            ;End of TAB1 reached?
      JNE    LOOP                   ;No, proceed
...                                 ;Yes, finished
```

In other cases it can be possible to exchange source and destination operands to have the auto increment feature available for a pointer.

Each of the seven addressing modes has its own features and advantages:

Register Mode: Fastest mode, least ROM requirements

Indexed Mode: Random access to tables

Symbolic Mode: Access to random addresses without overhead by loading of pointers

Absolute Mode: Access to absolute addresses independent of the current program address

Indirect Mode: Table addressing via register; code saving access to often referenced addresses

Indirect Autoincrement Mode: Table addressing with code saving automatic stepping; for transfer routines

Immediate Mode: Loading of pointers, addresses or constants within the instruction,

With the use of the symbolic mode an interrupt routine can be as short as possible. An interrupt routine is shown that has to increment a RAM word COUNTER and to do a comparison if a status byte STATUS has reached the value 5. If this is the case, the status byte is cleared. Otherwise, the interrupt routine terminates:

```
INTRPT INC    COUNTER      ;Increment counter
       CMP.B  #5,STATUS    ;STATUS = 5?
       JNE    INTRET       ;
       CLR.B  STATUS       ;STATUS = 5: clear it
INTRET RETI
```

No loading of pointers or saving and restoring of registers is necessary. The action is done immediately, without any overhead.

9.1.6 Program Flow Control

9.1.6.1 Computed Branches and Calls

The branch instruction is an emulated instruction that moves the destination address into the program counter:

```
   MOV    dst,PC        ; EMULATION FOR BR @dst
```

The ability to access the program counter in the same way as all other registers provides interesting options:

1) The destination address can be taken from tables: see Section 9.2.5
2) The destination address can be calculated
3) The destination address can be a constant. This is the usual method of getting the address.

9.1.6.2 Nesting of Subroutines

Due to the stack orientation of the MSP430, one of the main problems of other architectures does not play a role here at all. Subroutine nesting can proceed as long as RAM is available. There is no need to keep track of the subroutine calls as long as all subroutines terminate with the *Return from Subroutine* in-

struction RET. If subroutines are left without the RET instruction, some housekeeping is necessary; popping of the return address or addresses from the stack.

9.1.6.3 Nesting of Interrupts

Nesting of interrupts gives no problem at all, provided there is enough RAM for the stack. For every occurring interrupt, two words on the stack are needed for the storage of the status register and the return address. To enable nested interrupts, it is necessary to only include an EINT instruction into the interrupt handler. If the interrupt handlers are as short as possible (a good real-time practice), nesting may not be necessary.

EXAMPLE: The basic timer interrupt handler is woken-up with 1 Hz only, but has to do a lot of things. The interrupt nesting is therefore used. The latency time is 8 clock cycles only.

```
; Interrupt handler for Basic Timer: Wake–up with 1Hz
;
BT_HAN    EINT               ; Enable interrupt for nesting
          INC.B  SECCNT      ; Counter for seconds +1
          CMP.B  #60,SECCNT  ; 1 minute elapsed?
          JHS    MIN1        ; Yes, do necessary tasks
          RETI               ; No return to LPM3
;
; One minute elapsed: Return is removed from stack, a branch to
; the necessary tasks is made. There it is decided how to proceed
;
MIN1   INC       MINCNT      ; Minute counter +1
       CLR       SECCNT      ; 0 –> SECCNT
   ...                       ; Start of necessary tasks
       RETI                  ; Tasks completed
```

9.1.6.4 Jumps

Jumps allow the conditional or unconditional leaving of the linear program flow. Jumps cannot reach every address of the address space. But they have the advantage of needing only one word and only two MCLK cycles. The 10-bit offset field allows jumps of 512 words maximum forward and 511 words, maximum, backwards. This is four to eight times the normal reach of a jump. Only in a few cases, the 2-word branch is necessary.

Eight Jumps are possible with the MSP430; four of them have two mnemonics to allow better readability:

Table 9–3. Jump Usage

MNEMONIC	CONDITION	APPLICATIONS
JMP label	Unconditional Jump	Program control transfer
JEQ label	Jump if Z = 1	After comparisons: src = dst
JZ label	Jump if Z = 1	Test for zero contents
JNE label	Jump if Z = 0	After comparisons: src # dst
JNZ label	Jump if Z = 0	Test for nonzero contents
JHS label	Jump if C = 1	After unsigned comparisons: dst ≥ src
JC label	Jump if C = 1	Test for a set carry
JLO label	Jump if C = 0	After unsigned comparisons dst < src
JNC label	Jump if C = 0	Test for a reset carry
JGE label	Jump if N .XOR. V = 0	After signed comparisons: dst ≥ src
JLT label	Jump if N .XOR. V = 1	After signed comparisons: dst < src
JN label	Jump if N = 1	Test for the sign of a result: dst < 0

Note:

It is important to use the appropriate conditional jump for signed and unsigned data. For positive data (0 to 07FFFh or 0 to 07Fh) both signed and unsigned conditional jumps operate similarly. This changes completely when used with negative data (08000h to 0FFFFh or 080h to 0FFh): the signed conditional jumps treat negative data as smaller numbers than the positive ones, and the unsigned conditional jumps treat them as larger numbers than the positive ones.

No *Jump if Positive* is provided, only a *Jump if Negative*. But after several instructions, it is possible to use the *Jump if Greater Than or Equal* for this purpose. It must be ensured that only the instruction preceding the JGE resets the overflow bit V. The following instructions ensure this:

```
AND    src,dst      ; V <– 0
BIT    src,dst      ; V <– 0
RRA    dst          ; V <– 0
SXT    dst          ; V <– 0
TST    dst          ; V <– 0
```

If this feature is used, it should be noted within the comment for later software modifications. For example:

```
MOV    ITEM,R7      ; FETCH ITEM
TST    R7           ; V <– 0, ITEM POSITIVE?
JGE    ITEMPOS      ; V=0: JUMP IF >= 0
```

Note:

If addresses are computed only the unsigned jumps are adequate. Addresses are always unsigned, positive numbers.

No *Jump if Overflow* is provided. If the status of the overflow bit is needed from the software, a simple bit test can be used (the BIT instruction clears the overflow bit, but its state is read correctly before):

```
OV      .EQU   0100h      ; Bit address in SR
                          ;
        BIT    #OV,SR     ; Test Overflow Bit and clear it
        JNZ    OVFL       ; If OV = 1 branch to label OVFL
    ...                   ; If OV = 0 continue here
```

9.2 Special Coding Techniques

The flexibility of the MSP430 CPU together with a powerful assembler allows coding techniques not available with other microcomputers. The most important ones are explained in the following sections.

9.2.1 Conditional Assembly

For a detailed description of the syntax please refer to *MSP430 Family Assembler Language Tools User's Guide*.

Conditional assembly provides the ability to compile different lines of source into the object file depending on the value of an expression that is defined in the source program. This makes it easy to alter the behavior of the code by modifying one single line in the source.

The following example shows how to use of conditional assembly. The example allows easy debugging of a program that processes input from the ADC by pretending that the input of the ADC is always 07FFh. The following is the routine used for reading the input of the ADC. It returns the value read from ADC input A0 in R8.

```
DEBUG .set  1                ;1= debugging mode; 0= normal mode
ACTL  .set  0114h
ADAT  .set  0118h
IFG2  .set  3
ADIFG .set  4

; get_ADC_value:
;
      .IF   DEBUG=1
      MOV   #07FFh,R8
      .ELSE
      BIC   #60,&ACTL        ; Input channel is A0
      BIC.B #ADIFG,&IFG2
      BIS   #1,&ACTL         ; Start conversion
WAIT  BIT.B #ADIFG,&IFG2
      JZ    WAIT             ; Wait until conversion is ready
      MOV   &ADAT,R8
      .ENDIF
      RET
;
```

With a little further refining of the code, better results can be achieved. The following piece of code shows more built-in ways to debug the written code. The second *debug code*, where debug=2, returns 0700h and 0800h alternating.

```
DEBUG .SET  1                ; 1= debug mode 1; 2= deb. mode 2; 0=
                             ; normal mode
ACTL  .SET        0114h
ADAT  .SET        0118h
IFG2  .SET        3
ADIFG .SET        4

; get_ADC_value:
;
VAR   .SECT       "VAR"'0200h
OSC   .WORD       0700h

      .IF         DEBUG=1     ; Return a  constant value
      MOV         #07FFh,R8
      .ELSEIF     DEBUG=2     ; Return alternating values
      MOV         #0F00h,R8
      SUB         OSC,R8
      MOV         R8,OSC
      .ELSE
      BIC         #60h,&ACTL  ; Input channel is A0
      BIC         #ADIFG,&IFG2
      BIS         #1,&ACTL    ; Start conversion
WAIT  BIT         #ADIFG,&IFG2
      JZ          WAIT        ; Wait until conversion is ready
      MOV         &ADAT,R8
      .ENDIF
      RET
```

Conditional assembly is not restricted to the debug phase of software development. The main use is normally to get different software versions out of one source. For every version only the necessary software parts are assembled and the unneeded parts are left out by conditional assembly. The big advantage is the single source that is maintained.

An example of this is the MSP430 floating point package with two different number lengths (32 and 48 bits) contained in one source. Before assembly the desired length is defined by an .EQU directive. See Section 5.6, *The Floating Point Package* for details.

9.2.2 Position Independent Code

The architecture of the MSP430 allows the easy implementation of position independent code (PIC). This is a code, which may run anywhere in the address space of a computer without any relocation needed. PIC is possible with the MSP430 because of the allocation of the PC inside of the register bank. The addressability of the PC is often used. Links to other PIC blocks are possible only by references to absolute addresses (pointers).

EXAMPLE: Code is transferred to the RAM from an outside storage (EPROM, ROM, or EEPROM) and executed there at full speed. This code needs to be PIC. The loaded code may have several purposes:

- ❏ Application specific software that is different for some versions
- ❏ Additional code that was not anticipated before mask generation
- ❏ Test routines for manufacturing purposes

9.2.2.1 Referencing of Code Inside Position Independent Code

The referenced code or data is located in the same block of PIC as the program resides.

Jumps

Jumps are position independent anyway: their address information is an offset to the destination address.

Branches

```
        ADD   @PC,PC            ; Branch to label DESTINATION
        .WORD DESTINATION–$     ; Address pointer
```

Subroutine Calls

```
; Calling a subroutine starting at the label SUBR:
;
SC      MOV   PC,Rn             ; Address SC+2 –> Rn
        ADD   #SUBR–$,Rn        ; Add offset (SUBR – (SC+2))
        CALL  Rn                ; SC+2+SUBR–(SC+2)) = SUBR
```

Operations on Data

The symbolic addressing mode is position independent. An offset to the PC is used. No special addressing is necessary

```
        MOV   DATA,Rn           ; DATA is addressed
        CMP   DATA1,DATA2       ; symbolically
```

Jump Tables

The status contained in Rstatus decides where the SW continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n). Range: +512 words, –511 words

```
        ADD   Rstatus,PC        ; Rstatus = (2x status)
        JMP   STATUS0           ; Code for status = 0
        JMP   STATUS2           ; Code for status = 2
        ...
        JMP   STATUSn           ; Code for status = 2n
```

Branch Tables

The status contained in Rstatus decides where the SW continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n). Range: complete 64K

```
       ADD   TABLE(Rstatus),PC       ; Rstatus = status
TABLE .WORD STATUS0–TABLE            ; Offset for status = 0

       .WORD STATUS2–TABLE           ; Offset for status = 2
       ...
       .WORD STATUSn–TABLE           ; Offset for status = 2n
```

9.2.2.2 Referencing of Code Outside of PIC (Absolute)

The referenced code or data is located outside the block of PIC. These addresses can be absolute addresses only (e.g. for linking to other blocks or peripheral addresses).

Branches

Branching to the absolute address DESTINATION:

```
       BR    #DESTINATION            ; #DESTINATION –> PC
```

Subroutine Calls

Calling a subroutine starting at the absolute address SUBR:

```
       CALL  #SUBR                   ; #SUBR –> PC
```

Operations on Data

Absolute mode (indexed mode with status register SR = 0). SR does not loose its information!

```
       CMP   &DATA1,&DATA2           ; DATA1 + 0 = DATA1
       ADD   &DATA1,Rn
       PUSH  &DATA2                  ; DATA2 –> stack
```

Branch Tables

The status contained in Rstatus decides where the SW continues. Rstatus steps in increments of 2. The table is located in absolute address space:

```
       MOV   TABLE(Rstatus),PC ; Rstatus = status
       ...
       .sect xxx               ; Table in absolute address space
TABLE .WORD STATUS0            ; Code for status = 0
```

```
        .WORD STATUS2           ; Code for status = 2
        ...
        .WORD STATUSn           ; Code for status = 2n
```

Table is located in PIC address space, but addresses are absolute:

```
        MOV   Rstatus,Rhelp     ; Rstatus contains status
        ADD   PC,Rhelp          ; Status + L$1 –> Rhelp
L$1     ADD   #TABLE–L$1,Rhelp  ; Status+L$1+TABLE–L$1
        MOV   @Rhelp,PC         ; Computed address to PC
TABLE   .WORD STATUS0           ; Code for status = 0
        .WORD STATUS1           ; Code for status = 2
        ...
        .WORD STATUSn           ; Code for status = 2n
```

The previously shown program examples can be implemented as MACROs if needed. This would ease the usage and enhance the legibility.

9.2.3 Reentrant Code

If the same subroutine is used by the background program and interrupt routines, then two copies of this subroutine are necessary with normal computer architectures. The stack gives a method of programming that allows many tasks to use a single copy of the same routine. This ability of sharing a subroutine for several tasks is called reentrancy.

Reentrancy allows the calling of a subroutine despite the fact that the current task has not yet finished using the subroutine.

The main difference of a reentrant subroutine from a normal one is that the reentrant routine contains only pure code. That is, no part of the routine is modified during the usage. The linkage between the routine itself and the calling software is possible only via the stack (i.e. all arguments during calling and all results after completion have to be placed on the stack and retrieved from there). The following conditions must be met for reentrant code:

- ❏ No usage of dedicated RAM; only stack usage
- ❏ If registers are used, they need to be saved on the stack and restored from there.

EXAMPLE: A conversion subroutine *Binary to BCD* needs to be called from the background and the interrupt part. The subroutine reads the input number from TOS and places the 5-digit result also on TOS (two words). The subroutines save all registers used on the stack and restore them from there or compute directly on the stack.

```
        PUSH  R7                ; R7 CONTAINS THE BINARY VALUE
        CALL  #BINBCD           ; TO BE CONVERTED TO BCD
        MOV   @SP+,LSD          ; BCD–LSDs FROM STACK
        MOV   @SP+,MSD          ; BCD–MSD  FROM STACK
        ...
```

9.2.4 Recursive Code

Recursive subroutines are subroutines that call themselves. This is not possible with typical architectures; stack processing is necessary for this often used feature. A simple example for recursive code is a line printer handler that calls itself for the inserting of a *form feed* after a certain number of printed lines. This self-calling allows the use all of the existent checks and features of the handler without the need to write it more than once. The following conditions must be met for recursive code:

- ❏ No use of dedicated RAM; only stack usage
- ❏ A termination item must exist to avoid infinite nesting (e.g., the lines per page must be greater than 1 with the above line printer example)
- ❏ If registers are used, they need to be saved and restored on the stack

EXAMPLE: The line printer handler inserts a form feed after 70 printed lines

```
;
LPHAND      PUSH  R4          ; Save R4
      ...
      CMP   #70,LINES         ; 70 lines printed?
      JLT   L$500             ; No, proceed
      CALL  #LPHAND           ;
      .BYTE CR,FF             ; Yes, output Carriage Return
      ...                     ; and Form Feed
L$500 ...
```

9.2.5 Flag Replacement by Status Usage

Flags have several disadvantages when used for program control:

- ❏ Missing transparency (flags may depend on other flags)
- ❏ Possibility of nonexistent flag combinations if not handled very carefully
- ❏ Slow speed: the flags can be tested only serially

The MSP430 allows the use of a status (contained in a RAM byte or register), which defines the current program part to be used. This status is very descrip-

tive and prohibits nonexistent combinations. A second advantage is the high speed of the decision. Only one instruction is needed to get to the start of the appropriate handler (see Branch Tables).

The program parts that are used currently define the new status dependent on the actual conditions. Normally the status is only incremented, but it can be changed to be more random too.

EXAMPLE: The status contained in register Rstatus decides where the software continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n)

```
      ; Range: Complete 64K
      ;
            MOV   TABLE(Rstatus),PC ;Rstatus = status
      TABLE .WORD STATUS0           ; Address handler for status = 0

            .WORD STATUS2           ; Address handler for status = 2
            ...
            .WORD STATUSn           ; Address handler for status = 2n
      ;
      STATUS0     ....              ; Start handler status 0
            INCD  Rstatus           ; Next status is 2
            JMP   HEND              ; Common end
```

The previous solution has the disadvantage of using words even if the distances to the different program parts are small. The next example shows the use of bytes for the branch table. The SXT instruction allows backward references (handler starts at lower addresses than TABLE4).

```
; BRANCH TABLES WITH BYTES: Status in R5 (0, 1, 2, ..n)
; Usable range: TABLE4–128 to TABLE4+126

            PUSH.B      TABLE4(R5)        ; STATUSx–TABLE4 –> STACK
            SXT         @SP          ..... ; Forward/backward references
            ADD         @SP+,PC           ; TABLE4+STATUSx–TABLE4 –> PC
TABLE4      .BYTE       STATUS0–TABLE4    ; DIFFERENCE TO START OF
                                             HANDLER
            .BYTE       STATUS1–TABLE4
            ....
            .BYTE       STATUSn–TABLE4    ; Offset for status = n
```

If only forward references are possible (normal case), the addressing range can be doubled. The next example shows this:

```
; Stepping is forward only (with doubled forward range)
; Status is contained in R5 (0, 1, ..n)
```

```
; Usable range: TABLE5 to TABLE5+254

            PUSH.B      TABLE5(R5)  ; STATUSx–TABLE –> STACK
            CLR.B       1(SP)        ..... ; Hi byte <– 0
            ADD         @SP+,PC            ; TABLE+STATUSx–TABLE –> PC
TABLE5      .BYTE       STATUS0–TABLE5     ; DIFFERENCE TO START OF
                                               HANDLER
            .BYTE       STATUS1–TABLE5
            ....
            .BYTE       STATUSn–TABLE5     ; Offset for status = n
      ;
```

The previous example can be made shorter and faster if a register can be used:

```
; Stepping is forward only (with doubled forward range)
; Status is contained in R5 (0, 1, 2..n)
; Usable range: TABLE5 to TABLE5+254
;
            MOV.B       TABLE5(R5),R6      ; STATUSx–TABLE5 –> R6
            ADD         R6,PC         ..... ; TABLE5+STATUSx–TABLE5 –> PC
TABLE5      .BYTE       STATUS0–TABLE5     ; DIFFERENCE TO START OF
                                               HANDLER
            .BYTE       STATUS1–TABLE5
            ....
            .BYTE       STATUSn–TABLE5     ; Offset for status = n
```

The addressable range can be doubled once more with the following code. The status (0, 1, 2, ..n) is doubled before its use.

```
      ; The addressable range may be doubled with the following code:
      ; The ”forward only” version with an available register (R6) is
      ; shown: Status 0, 1, 2 ...n
      ; Usable range: TABLE6 to TABLE6+510
      ;
            MOV.B TABLE6(R5),R6      ; (STATUSx–TABLE6)/2
            RLA   R6                 ; STATUSx–TABLE6
            ADD   R6,PC              ; TABLE6+STATUSx–TABLE6 –> PC
      TABLE6      .BYTE (STATUS0–TABLE6)/2      ; Offset for Status = 0
            .BYTE (STATUS1–TABLE6)/2            ;
            ...
            .BYTE (STATUSn–TABLE6)/2            ; Offset for Status = n
```

9.2.6 Argument Transfer With Subroutine Calls

Subroutines often have arguments to work with. Several methods exist for the passing of these arguments to the subroutine:

- On the stack
- In the words (bytes) after the subroutine call
- In registers
- The address is contained in the word after the subroutine call

The passed information itself may be numbers, addresses, counter contents, upper and lower limits etc. It only depends on the application.

9.2.6.1 Arguments on the Stack

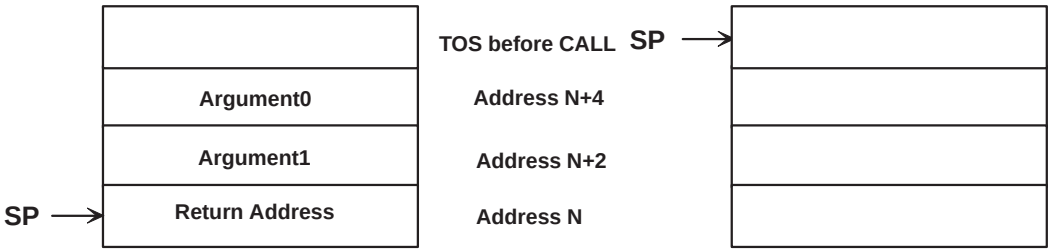
The arguments are pushed on the stack and read afterwards by the called subroutine. The subroutine is responsible for the necessary housekeeping (here, the transfer of the return address to the top of the stack).

- Advantages:
  - Usable generally; no registers have to be freed for argument passing
  - Variable arguments are possible
- Disadvantages:
  - Overhead due to necessary housekeeping
  - Not easy to understand

EXAMPLE: The subroutine SUBR gets its information from two arguments pushed onto the stack before being called. No information is given back and a normal return from subroutine is used.

```
        PUSH  argument0   ; 1st ARGUMENT FOR SUBROUTINE
        PUSH  argument1   ; 2nd ARGUMENT
        CALL  #SUBR       ; SUBROUTINE CALL
        ...
SUBR    MOV   4(SP),Rx    ; COPY ARGUMENT0 TO Rx
        MOV   2(SP),Ry    ; COPY ARGUMENT1 TO Ry
        MOV   @SP,4(SP)   ; RETURN ADDRESS TO CORRECT LOC.
        ADD   #4,SP       ; PREPARE SP FOR NORMAL RETURN
        ...               ; PROCESSING OF DATA
        RET               ; NORMAL RETURN
```

After the subroutine call, the stack looks as follows:



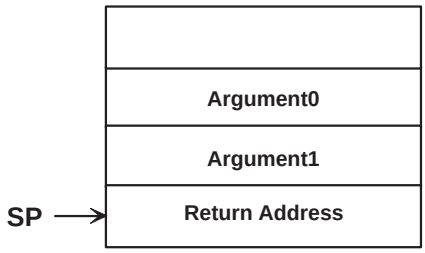
After the RET, it looks like this:

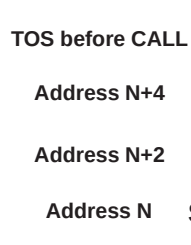
Figure 9–2. Argument Allocation on the Stack

EXAMPLE: The subroutine SUBR gets its information from two arguments pushed onto the stack before being called. Three result words are returned on the stack. It is the responsibility of the calling program to pop the results from the stack.

```
        PUSH  argument0         ; 1st ARGUMENT FOR SUBROUTINE
        PUSH  argument1         ; 2nd ARGUMENT
        CALL  #SUBR             ; SUBROUTINE CALL
        POP   R15               ; RESULT2 –> R15
        POP   R14               ; RESULT1 –> R14
        POP   R13               ; RESULT0 –> R13
        ...
SUBR    MOV   4(SP),Rx          ; COPY ARGUMENT0 TO Rx
        MOV   2(SP),Ry          ; COPY ARGUMENT1 TO Ry
        ...                     ; PROCESSING CONTINUES
        PUSH  2(SP)             ; SAVE RETURN ADDRESS
        MOV   RESULT0,6(SP)     ; 1st RESULT ON STACK
        MOV   RESULT1,4(SP)     ; 2nd RESULT ON STACK
        MOV   RESULT2,2(SP)     ; 3rd RESULT ON STACK
        RET
```

After the subroutine call, the stack looks as follows:





After the RET, it looks like this:

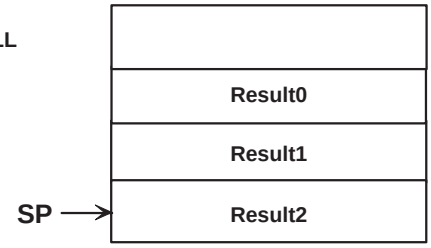


Figure 9–3. Argument and Result Allocation on the Stack

Note:

If the stack is involved during data transfers, it is very important to have in mind that only data at or above the top of stack (TOS, the word the SP points to) is protected against overwriting by enabled interrupts. This does not allow the SP to move above the last item on the stack. Indexed addressing is needed instead.

9.2.6.2 Arguments Following the Subroutine Call

The arguments follow the subroutine call and are read by the called subroutine. The subroutine is responsible for the necessary housekeeping (here, the adaptation of the return address on the stack to the 1st word after the arguments).

- Advantages:
  - Very clear and easily readable interface
- Disadvantages:
  - Overhead due to necessary housekeeping
  - Only fixed arguments possible

EXAMPLE: The subroutine SUBR gets its information from two arguments following the subroutine call. Information can be given back on the stack or in registers.

```
CALL  #SUBR             ; SUBROUTINE CALL
.WORD START             ; START OF TABLE
.BYTE 24,0              ; LENGTH OF TABLE, FLAGS
```

```
        ...                     ; 1st instruction after CALL
SUBR    MOV   @SP,R5            ; COPY ADDRESS 1st ARGUMENT TO R5
        MOV   @R5+,R6           ; MOVE 1st ARGUMENT TO R6
        MOV   @R5+,R7           ; MOVE ARGUMENT BYTES TO R7
        MOV   R5,0(SP)          ; ADJUST RETURN ADDRESS ON STACK
        ...                     ; PROCESSING OF DATA
        RET                     ; NORMAL RETURN
```

9.2.6.3 Arguments in Registers

The arguments are moved into defined registers and used afterwards by the subroutine.

- Advantages:
  - Simple interface and easy to understand
  - Very fast
  - Variable arguments are possible
- Disadvantages:
  - Registers have to be freed

EXAMPLE: The subroutine SUBR gets its information from two registers which are loaded before the calling. Information can be given back, or not with the same registers.

```
        MOV   arg0,R5           ; 1st ARGUMENT FOR SUBROUTINE
        MOV   arg1,R6           ; 2nd ARGUMENT
        CALL  #SUBR             ; SUBROUTINE CALL
        ...
SUBR    ...                     ; PROCESSING OF DATA
        RET                     ; NORMAL RETURN
```

9.2.7 Interrupt Vectors in RAM

If the destination address of an interrupt changes with the program run, it is valuable to have the ability to modify the pointer. The vector itself (which resides in ROM) cannot be changed but a second pointer residing in RAM can be used for this purpose.

EXAMPLE: The interrupt handler for the basic timer starts at location BTHAN1 after initialization and at BTHAN2 when a certain condition is met (for example, when a calibration is made).

```
; BASIC TIMER INTERRUPT GOES TO ADDRESS BTVEC. THE INSTRUCTION
```

```
; ”MOV @PC,PC” WRITES THE ADDRESS IN BTVEC+2 INTO THE PC:
; THE PROGRAM CONTINUES AT THAT ADDRESS
;
      .sect ”VAR”,0200h       ; RAM START
BTVEC .word 0                 ; OPCODE ”MOV @PC,PC”
      .word 0                 ; ACTUAL HANDLER START ADDR.

; THE SOFTWARE VECTOR BTVEC IS INITIALIZED:
;
INIT  MOV   #04020h,BTVEC     ; OPCODE ”MOV @PC,PC
      MOV   #BTHAN1,BTVEC+2   ; START WITH HANDLER BTHAN1
      ...               ; INITIALIZATION CONTINUES
;
; THE CONDITION IS MET: THE BASIC TIMER INTERRUPT IS HANDLED
; AT ADDRESS BTHAN2 STARTING NOW

      MOV   #BTHAN2,BTVEC+2   ; CONT. WITH ANOTHER HANDLER
      ...
;
; THE INTERRUPT VECTOR FOR THE BASIC TIMER CONTAINS THE RAM
; ADDRESS OF THE SOFTWARE VECTOR BTVEC:

      .sect ”BTVect”,0FFE2h   ; VECTOR ADDRESS BASIC TIMER
      .WORD BTVEC             ; FETCH ACTUAL VECTOR THERE
```

9.3 Instruction Execution Cycles

9.3.1 Double Operand Instructions

With the following scheme, it is relatively easy to remember how many cycles a double operand instruction will need to execute. Figure 9–4 shows the number of cycles for all 28 possible combinations of the source and destination addressing modes. All similar addressing modes are condensed.

	Rdst	X(Rdst) SYMBOLIC &ABSOLUT
Rsrc	**1†**	**4**
@Rsrc, @Rsrc+, #N	**2†**	**5**
X(Rsrc), SYMBOLIC, &ABSOLUT	**3**	**6**

†: Add one cycle if Rdst is PC

Figure 9–4. Execution Cycles for Double Operand Instructions

EXAMPLE: the instruction `ADD #500h,16(R5)` needs 5 cycles for the execution.

9.3.2 Single Operand Instructions

The simple and clear scheme of the double operand instructions is not applicable to the six single operand instructions. They differ too much. Figure 9–5 gives an overview.

	SWPB SXT RRx	PUSH	CALL
Rdst	**1**	**3**	**4**
@Rdst	**3**	**4**	**4**
@Rdst+, #N	**3**	**4**	**5**
X(Rdst), SYMBOLIC, &ABSOLUT	**4**	**5**	**5**

Figure 9–5. Execution Cycles for Single Operand Instructions

EXAMPLE: the instruction `PUSH #500h` needs 4 cycles for the execution.

9.3.3 Jump Instructions

All seven conditional jump instructions need two cycles for execution, independent if the jump condition is met or not. The same is true for the unconditional jump instruction, JMP.

9.3.4 Interrupt Timing

An enabled interrupt sequence needs eleven cycles overhead:

- Six cycles for the storage of the PC and the SR on the stack until the first instruction of the interrupt handler is started
- Five cycles for the return from interrupt—by the instruction RETI—until the first instruction of the interrupted program is started.

If the interrupt is requested during the low power modes 3 or 4, then additional two cycles are needed.