3
Introducing the rest of the family

So far, we have been looking at the most basic types of AVR, the 1200 and the Tiny AVRs. I will now introduce some of the differences between these and other AVRs, so that in the subsequent chapters they might appear more familiar. Other models may benefit from extra memory called RAM. The allocation of memory differs in different models, but follows the arrangement shown in Figure 3.1.

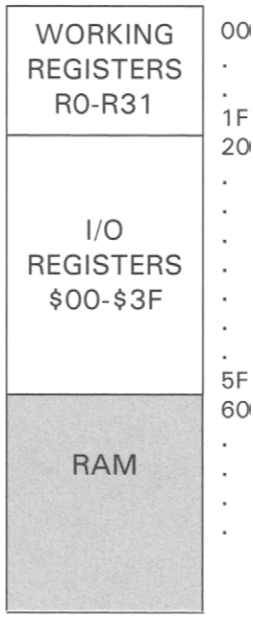

Figure 3.1

The first 32 addresses are the working registers and the next 64 are the I/O registers. So the key difference between those with RAM and those without is the presence of further memory spaces from $60 onwards. These can be accessed using the **ld** and **st** commands already introduced, and with other instructions now available on these more advanced models. A significant

change to the working registers is the introduction of two more 2-byte register pairs. In addition to Z (made up of R30 and R31), there is now Y (made up of R28 and R29), and X (made up of R26 and R27). They can be used in any instruction that takes a Z (e.g. **ld**, **st**, **lpm** etc.).

Whilst there was a dedicated three-level stack on the 1200 and Tiny AVRs, the other models require that you tell them *where in the RAM you want your stack to be*. This means it is potentially as deep as the RAM address space, though obviously you may be wishing to give some of the RAM addresses a more glamorous purpose. What we will do is make the *last address of RAM* the top of the stack. In this way we have what looks like an upside-down stack, as shown in Figure 3.2, which works in exactly the same way as any other stack.

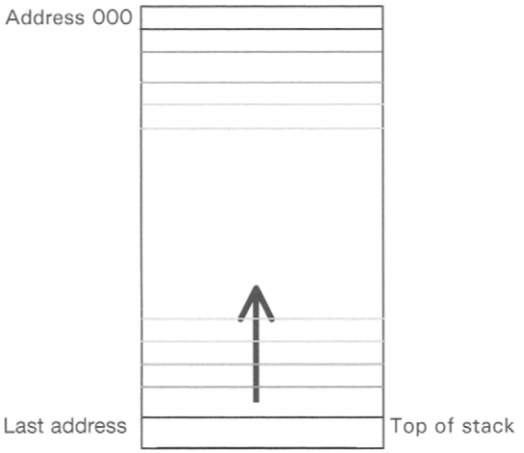

Figure 3.2

The I/O registers **SPL** and **SPH** are the *stack pointer* registers (lower and higher bytes), and so we move into these the last address of the RAM. This is helpfully stored for us in the include file we read at the top of each program as RAMEND. We therefore load the lower byte of RAMEND into SPL and the upper byte into SPH, and thus point the stack to the end of the RAM. The instructions are:

```
ldi     temp, LOW(RAMEND)   ; stack pointer points to
out     SPL, temp           ;    last RAM address
ldi     temp, HIGH(RAMEND)  ;
out     SPH, temp           ;
```

This must take place in the **Init** section, before any subroutines are called. For

devices with only 128 bytes of RAM, RAMEND is only 1 byte long, so the last two lines given above should be omitted.

Another major difference seen in the other models is a greater set of instructions. First, you are given greater flexibility with the **ld** and **st** instructions. You can make the 'long' registers X, Y or Z being used as an address pointer automatically increment or decrement with each load/store operation:

```
ld      reg, longreg+
```

This **load**s the memory location pointed to by a double register (i.e. X, Y or Z) into **reg**, and then adds one to longreg.

```
ld      reg, -longreg
```

This subtracts one from the double register (X, Y or Z), and then **load**s the memory location pointed to by that double register into **reg**. There are analogous instructions for **st**.

We can use this to shorten our multiple register clearing routine. In this case I have chosen to use X and the indirect address pointer, so this routine clears registers up to R25.

```
        clr   XL          ; clears XL
        clr   XH          ; clears XH
ClearLoop: st   XH, X+    ; clears indirect address and increments X
        cpi   XL, 26      ; compares XL with 26
        brne  ClearLoop   ; branches to ClearLoop if ZL = 26
```

Other enhancements to load/store operations include:

```
ldd     reg, longreg+number
```

This **load**s the memory location pointed to by the Y or Z registers into **reg**, and then adds a **number** (0–63) to Y or Z. *(Note: doesn't work with X.)* There is an equivalent instruction for storing, **std**, which works in the same way. There is also a way to *directly address* memory in the RAM:

```
lds     reg, number
```

This **load**s the contents of memory at the address given by **number** into **reg**. The number can be between 0 and 65 535 (i.e. up to 64K). Similarly, **sts** stores the number in a register into a directly specified address.

Indirect jumps and calls are particularly useful and are specified by the number in the Z register:

```
icall   ; calls the address indirectly specified in Z
ijmp    ; jumps to the address indirectly specified in Z
```

Example 3.1 We have a program that has to perform one of five different functions, depending on the number in a register called **Function**. By adding **Function** to the current value of the program counter, and jumping to that address, we can make the program branch out to different sections:

```
            clr     ZH              ; makes sure higher byte is clear
            ldi     ZL, JumpTable   ; points Z to top of table
            add     ZL, Function    ; adds Function to Z
            ijmp                    ; indirectly jumps
JumpTable:  rjmp    Addition        ; jumps to Addition section
            rjmp    Subtraction     ; jumps to Subtraction section
            rjmp    Multiplication  ; jumps to Multiplication section
            rjmp    Division        ; jumps to Division section
            rjmp    Power           ; jumps to Power section
```

Notice that **JumpTable** is loaded into Z, this is translated by the assembler as the program memory address of the line it is labelling. We do this to initialize Z to point to the top of the *branching table* (**rjmp Addition**). Note that loading **JumpTable** is equivalent to loading **PC+3**. The number in **Function** is then added to Z, so that the number in **Function** (between 0 and 4) will make the program jump to one of the five sections.

You will no doubt remember the number of additions and subtractions we had to do to 2-byte numbers in the frequency counter project. Here are two new instructions that may help:

```
adiw    longreg, number
sbiw    longreg, number
```

These **add** or **sub**tract a **number** between 0 and 63 to/from one of the 16-bit registers (X, Y or Z). The 'w' stands for **w**ord (16 bits). If there is an overflow or carry this is automatically transferred onto the higher byte. Hence:

```
subi    XL, 50
sbci    XH, 0
```

```
sbiw    XL, 50
```

The two remaining instructions that are added to the repertoire of the more advanced AVRs are:

```
push    register
pop     register
```

So far we have only been using the stack for the automatic storage of program counter addresses when calling subroutines. Using these instructions, you can **push** or **pop** the number in any working register on to or off of the stack.

Example 3.2 We can use the **push** and **pop** instructions to create a *palindrome detector.* A palindrome is essentially a symmetric sequence (like 'radar', 'dennis sinned' and 'evil olive'). We can massively simplify this problem by also requiring that we are given the length of the input sequence. We can use the length to find the middle of the input. We can also assume that the input is fed (as an ASCII character) into a register called **Input**. ASCII is a way to translate letters and symbols into a byte, so each letter corresponds to a particular byte-long number. So effectively we are looking for the sequence of bytes fed into **Input** to be palindromic (symmetric). We start by *pushing* the number in **Input** on to the stack. We do this for every new input until we reach the half-way point. We then start *popping* the stack and comparing it with the input. As long as each new input continues to be the same as the popped number, the sequence is potentially palindromic. If the new input fails to be the same as the popped number, we reject the input sequence. PinD, bit 0 will pulse high for 1 microsecond to indicate a new input symbol (we need this because we cannot just wait for the input symbol to change, as this would not respond to repeated letters).

First, we assume the length of the word is stored in **Length**. This has to be divided by two to get the half-way point. We will have to make a note if the length is odd or not. This is done by testing the carry flag; if it is high **Length** was odd and we shall set the T bit.

```
Start:    mov    HalfLength, Length  ; divides Length by 2 to get
          lsr    HalfLength          ;    HalfLength
          in     temp, SREG          ; copies Carry flag into T bit
          bst    temp, 0             ;    i.e. sets T-bit if Length is odd
```

Assuming the first input byte is in **Input**, we push it on to the stack and then wait for the pulse on PinD, bit 0. The pulse lasts 1 microsecond, so assuming a 4 MHz clock it must be tested at least once every four cycles. In the segment below, it is tested once every three cycles.

```
FirstHalf: push   Input               ; pushes Input onto stack
Pulse:     sbis   PinD, 0             ; tests for pulse
           rjmp   Pulse               ; keeps looping
```

When a pulse is received (i.e. a new input symbol is ready), the program increments **Counter** which is keeping track of the input number. It compares this number with **HalfLength** and loops back as long as **Counter** is less than **HalfLength**.

```
          inc    Counter             ; counts the input number
          cp     Counter, HalfLength ; compares with half-way value
          brlo   FirstHalf           ; loops back to start and skips one
```

When **Counter** equals **HalfLength** we check the T bit to see if the length of the input is odd or even. If it is odd, we need to ignore the middle letter, so we reset the T bit and loop back to **Pulse** which will wait until the next input is ready. If the length is even we can skip on to test the second half of the input.

```
        brtc    SecondHalf      ; test T bit
        clt                     ; clears T bit
        rjmp    Pulse           ;
```

We have now passed the half-way point in the sequence and now the new input symbols must match the previous ones. The top of stack is popped and compared with the current input. If they are not equal the sequence is rejected.

```
SecondHalf: pop     Input2          ; pops stack into Input2
        cp      Input, Input2   ; compares Input and Input2
        brne    Reject          ; if different reject sequence
```

As before, we then increment **Counter** and test to see if **Counter** equals **Length**. If it does, the testing is over and we can accept the input. If we haven't yet reached the end the program then waits for the input to change, and then loops back to **SecondHalf**.

```
        inc     Counter         ; counts the input number
        cp      Counter, Length ; compares with total length
        breq    Accept          ; end of sequence so accept

Pulse2:     sbis    PinD, 0         ; waits for pulse
        rjmp    Pulse2          ;
        rjmp    SecondHalf      ; loops back when new input is
                                ;    ready
```

You might want to play around with this on the simulator, but don't forget to set up the top of the stack as described at the start of the chapter. You may also want to think about how to remove the need to be given the length of the input sequence. If you want to find out more about this, you may want to find a book on *Formal Languages* and *Parsing*.