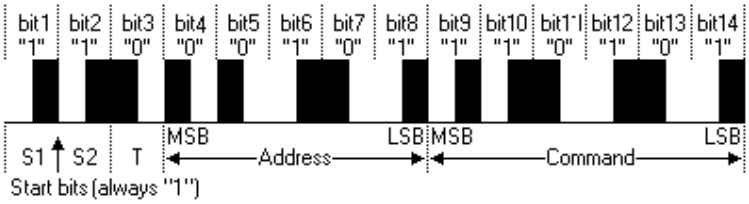
RC5-decoder software for AVR

The RC5 protocol, invented by Philips, is not so difficult to decode, as always if you know how. First of all, in order to understand what has to be decoded, let's have a look at a sample of an RC5 code:



RC5 always starts with two start bits that are logical one.

A logical one for RC5 consists of half a bit period idle time, and half a bit period active time. A logical zero is just the other way around, so half a bit period active time and half a bit period idle time.

The following bit, the third bit, is called the 'toggle bit'. This bit toggles each time a key on the remote is released. This means that a difference can be detected between a key that is pressed for a longer time and a key that is repeatedly pressed.

The next five bits represent the address or system code. These are used to indicate a system. For instance system ID '0' (zero) is used for TV's and system ID '5' is used for VCR's.

Finally, the last 6 bits represent the actual command. Obviously what response occurs after which command is entirely in the hands of the programmer.

Another thing you need to know is that one bit last exactly 1.778 msec. So the whole sequence of one RC5 command will take place in 14*1.778 msec = 24.892 msec or about 1/40 of a second. If one bit lasts 1.778 msec then one half bit will last 889 usec.

Looking at the datasheets, various IR-detectors need anything from about 6 to 10 pulses at 36 kHz modulation to detect IR-activity, this means from 166 usec to 277 usec. So if I would send the RC5-code from the example above, my IR-detector would see activity after 889+277 usec = 1.166 msec. That is after about 65% of the time of the first bit!

The idea now to decode an RC5 command is not to measure the time between rising and falling edges of the signal and then to decide what the logical sequence should be, but to start measuring just after the IR-detector sees activity, so after about ¾ of the first start bit, and then to look at the IR-detector every 889 usecs i.e. every half bit time. Every second half bit is the complement of its first half, so if you know the logical value of the first half, you know what the second half should be. Is it not correct, then either your timing is incorrect or the command received has an error.

So what we do is, as soon as we see activity on the IR-detector, we first wait a short time to get to about ¾ of the first start bit, then we start to measure 889 usecs, take another look at the IR-detector, it should read logical zero for the second start bit, measure 889 usecs, take another look at the IR-detector, it now should read logical one for the second start bit, so second start bit OK, measure 889 usecs, take yet another look at the IR-detector, see a logical one, measure 889 usecs, take another look at the IR-detector and should see a logical zero (toggle bit) and so on. Right until the last bit. If anywhere in this sequence a false value is detected, then there's probably an error in the reception and we can bail out the loop and throw away anything we have received so far and try again.

Right, now have a look at the source code. The AVR device used here is an ATmega16 running at 16 MHz. This device has SRAM (and EEPROM, but we don't use that here) which I use to store information since the decoder routine is only part of a complete program. The RC5-decoder routine also uses Timer2 of the ATmega16 configured for timer overflow interrupt. My IR-detector is an IRM-2636A by Everlight Electronics, but that is not really critical. What is critical is that its output is connected to pin 26 i.e. PINC4. This means, that when you use the ATmega16 'straight out of the box' your machine will not function, since the device is shipped with the JTAG interface enabled, and that uses PINC4 also. So first of all flick the JTAG enable fuse bit. Then the fun can start.

```
;***************************************************************************
;*  Program name      : RC5.asm                                              *
;*  Written by        : Jan van Rijsewijk                                    *
;*  Date              : 17-07-2006                                           *
;*  Device            : Atmel ATmega16 @ 16MHz                               *
;*-------------------------------------------------------------------------*
;*  Purpose           : Decodes RC5 code and stores the result in SRAM in    *
;*                    : variable m_RC5. The first byte will be the actual    *
;*                    : command and the second byte will be the device code  *
;*                    : Upon correct reception bit0 of register Flags will   *
;*                    : be set. Incorrect reception will result in m_RC5 to  *
;*                    : be filled with $FFFF and no flag set.                *
;*                    : The routine uses Timer2 Overflow Interrupt.          *
;*                    : The IR-detector should be connected to PINC4.        *
;*                                                                           *
;***************************************************************************

.nolist
.include "m16def.inc"
.list

; variables in SRAM.

.dseg
m_RC5:                      ; RC5 command received. Second byte represents system code (5 for VCR)
.dw $0000                   ; and first byte represents the actual command

; Register definitions

.def  TEMP     =r16
.def  TEMP2    =r17
.def  Counter1 =r18
.def  Counter2 =r19
.def  Flags    =r20
.def  LowByte  =r24
.def  HighByte =r25

; Interrupt vector table

.cseg
.org $0000
      rjmp     reset              ; Common start/reset

.org OVF2addr
      rjmp     IntOVF2            ; Timer2 Overflow vector

;---------------------------------------------------------------------------
reset:
      ldi         TEMP,low(RAMEND)  ; setup Stack before all else
      out         SPL,TEMP          ; Low BYTE
      ldi         TEMP,high(RAMEND) ; High BYTE
      out         SPH,TEMP          ; setup Stack done

      ldi         Flags,$00         ; reset all flags

;     Setup Timer2 Overflow interrupt

      ldi         TEMP,(1<<TOIE2)
      out         TIMSK,TEMP        ; enable Timer2 overflow

      ldi         TEMP,$00          ; set initial value Timer2
      out         TCNT2,TEMP

      ldi         TEMP,$01          ; prescaler /1 for Timer2, timer starts
      out         TCCR2,TEMP        ; interrupt every 64 usecs. so we have to
      ldi         Counter1,$04      ; setup Counter1 to divide by 4

;---------------------------------------------------------------------------
      sei                           ; Global Interrupt Enable
;===========================================================================

main_loop:
      rjmp     main_loop

;---------------------------------------------------------------------------
```

That's basically the initialization of the program as well as the main program. As you can see the main program does nothing yet, but I'll show a small example at the end.

Then the rest, bit by bit as it were:

```
IntOVF2:
      push     TEMP              ; save usual registers
      in       TEMP,SREG
      push     TEMP
      push     TEMP2
      push     HighByte
      push     LowByte
      push     ZH
      push     ZL

      dec      Counter1          ; decrease counter
      breq     CheckStartBit     ; 64 usec passed? If not return
      rjmp     RetIntOVF2        ; brne is too far away

CheckStartBit:
      .../

      /...

RetIntOVF2:
      pop      ZL                ; and get the rest back
      pop      ZH                ; to the original state
      pop      LowByte           ;
      pop      HighByte          ;
      pop      TEMP2             ;
      pop      TEMP              ;
      out      SREG,TEMP         ;
      pop      TEMP              ;
      reti                       ; Go back
```

This is the main loop of the handling of overflow interrupt2. Since this is an interrupt routine it can be invoked anywhere in the main program, so the wise thing to do is to save the register contents to the stack. The only drawback is that you have to get them of the stack as well again. So what happens is that each 64 usecs a branch is executed to 'CheckStartBit'. Every other time the interrupt is executed a normal return is made.

Now on to the next step:

```
CheckStartBit:
      sbrc     Flags,0           ; jump over if there are no pending
      rjmp     Reload_C1         ; RC5 commands (bit0 must be cleared)

.../

/...

Reload_C1:
      ldi      Counter1,$04      ; restore counter1

RetIntOVF2:
```

A check is made to see if the Flag register (r20) has been cleared. If it has not been cleared, it means that there is still an RC5 command pending, so then again nothing will happen and a normal return is made. The only thing that has to happen is that Counter1 has to be reloaded, before returning. If the Flag register is cleared we can finally have a look at PINC4 to check for activity.

```
CheckStartBit:
      sbrc     Flags,0           ; jump over if there are no pending
      rjmp     Reload_C1         ; RC5 commands (bit0 must be cleared)

      in       TEMP,PINC         ; get PINC and
      sbrc     TEMP,4            ; if there is activity, move on (active LOW)
      rjmp     Reload_C1         ; else return

      ldi      Counter2,$03      ; Wait for about 130 usecs
                                 ; to get to about half the active part
ReloadCount1S:                   ; of the first start bit. That is, if
      ldi      Counter1,$E6      ; it is RC5 ofcourse

DecCount1S:
      dec      Counter1
      brne     DecCount1S

      dec      Counter2
```

```
      brne     ReloadCount1S     ; 129.94 usecs later move on

      rcall    DelayHalfBit      ; If there is activity
                                 ; wait another half bit
      in       TEMP,PINC         ; now C4 should be low here, or it's not RC5
      sbrs     TEMP,4
      rjmp     Reload_C1         ; and so return

      rcall    DelayHalfBit      ; wait another half bit time

      in       TEMP,PINC         ; Now C4 must be high (start bit #2)
      sbrc     TEMP,4            ; yes means '2 start bits received OK'
      rjmp     Reload_C1         ; or else go back

.../

/...

DelayHalfBit:
      ldi      Counter2,$14      ; Wait for half a RC5 bit time

ReloadCount1:
      ldi      Counter1,$EC      ; Counter1 can be used here

DecCount1:
      dec      Counter1
      brne     DecCount1

      dec      Counter2
      brne     ReloadCount1      ; 888.75 usecs later return
      ret
```

What happens now is that as soon as activity is detected on PINC4 a delay loop is started to delay about 130 usec. This means that added to the time the IR detector needs to detect activity (according to the datasheet 10 pulses at 36 kHz which is 277 usec) there is a delay of about 407 usec and that places us about halfway the active time of the first start bit. After that a call is made to delay an additional half a bit time, 889 usecs, so that places us at a quart of time for the second start bit. PINC4 is checked and should be low. Mind the use of high and low here since the output of the IR-detector is active low

Then there's another half bit time delay and again PINC4 is checked. It should be active now. Once that fits together it means that we correctly received two start bits. On the time line we are at three quarters of the second start bit.

Moving right along…

```
      in       TEMP,PINC         ; Now C4 must be high (start bit #2)
      sbrc     TEMP,4            ; yes means '2 start bits received OK'
      rjmp     Reload_C1         ; or else go back

      ldi      HighByte,$06      ; so now we have 2 start bits and room for the toggle bit
      ldi      TEMP2,$06         ; now we read the next 6 bits

GetDeviceByte:
      rcall    DelayHalfBit      ; wait another half bit time

      in       TEMP,PINC         ; Now get the value of PINC4
      sbrc     TEMP,4            ; check for '0' of IRM-2636
      rjmp     C4H_isHigh        ; which means a RC5-0bit should come
                                 ; '1' would mean a RC5-1bit should come
      cbr      HighByte,1        ; set bit0 to 0
      rjmp     C4H_nextHalf

C4H_isHigh:
      sbr      HighByte,1        ; or set bit0 to 1

C4H_nextHalf:
      rcall    DelayHalfBit      ; wait for the second half bit time

      in       TEMP,PINC         ; Now get the value of PINC4
      bst      TEMP,4            ; move bit4 (IRM-2636)
      clr      TEMP              ; ...clean out TEMP in the mean time...
      bld      TEMP,0            ; to bit0
      eor      TEMP,HighByte     ; eor with HighByte and then
      andi     TEMP,0b00000001   ; mask off bit0, which should be 1
      breq     RC5_error         ; if not, there's an error, so go back
```

```
      dec     TEMP2            ; decrease bit counter
      breq    GetCommandByte   ; if all is well, move on (device received)

      lsl     HighByte         ; or else make room for the next bit
      rjmp    GetDeviceByte    ; and go and get it

GetCommandByte:
```

Next, we set up Highbyte with $06 or 0b0000 0110 which means there are two start bits in HighByte and there is room for the toggle bit. Then we step in a loop that reads the next six bits i.e. the toggle bit and the 5 device bits. A check is made to make sure that each second half of a bit is the complement of the first half, or else a branch to RC5_error is made where Highbyte and Lowbyte are filled with $FF. If all is well, the contents of Highbyte is left-shifted to make room for the next bit and the bit counter TEMP2 is decreased and the loop is looped again. Note that the value of the second half of one bit is copied from bit4 to bit0 with the use of the T-register in SREG. This takes less time than right-shifting. You could even omit the 'clr TEMP' statement, since the only bit that matters is bit0.

After this, we have received and stored the two start bits, the toggle bit and the 5 device bits in Highbyte. So the next thing we do is try to receive the actual command consisting of the next 6 bits.

```
GetCommandByte:                ; Next, get the command byte
      ldi     LowByte,$00      ; so now we have the device byte
      ldi     TEMP2,$06        ; and we read the next 6 bits for the
                               ; command byte
LoopCommandByte:
      rcall   DelayHalfBit     ; wait another half bit time

      in      TEMP,PINC        ; Now get the value of PINC4
      sbrc    TEMP,4           ; check for '0' of IRM-2636
      rjmp    C4L_isHigh       ; which means a RC5-0bit should come
                               ; '1' would mean a RC5-1bit should come
      cbr     LowByte,1        ; set bit0 to 0
      rjmp    C4L_nextHalf

C4L_isHigh:
      sbr     LowByte,1        ; or set bit0 to 1

C4L_nextHalf:
      rcall   DelayHalfBit     ; wait for the second half bit time

      in      TEMP,PINC        ; Now get the value of PINC4
      bst     TEMP,4           ; move bit4 (IRM-2636)
      clr     TEMP             ; ...clean out TEMP in the mean time...
      bld     TEMP,0           ; to bit0
      eor     TEMP,LowByte     ; eor with LowByte and then
      andi    TEMP,0b00000001  ; mask off bit0, which should be 1
      breq    RC5_error        ; if not, there's an error, so go back

      dec     TEMP2            ; decrease bit counter
      breq    RC5_complete     ; if all is well, move on (command received)

      lsl     LowByte          ; or else make room for the next bit
      rjmp    LoopCommandByte  ; and go and get it

RC5_complete:
```

It's just the same as the previous loop. What is left to do, is to set the appropriate flag and store the contents of Highbyte and Lowbyte in SRAM:

```
RC5_complete:
      sbr     Flags,1          ; set bit0 in Flags

Write_RC5:
      ldi     ZL,low(m_RC5)    ; Point to 'm_RC5' in SRAM
      ldi     ZH,high(m_RC5)

      st      Z+,LowByte       ; and write RC5-command
      st      Z,HighByte       ; and RC5-device number

Reload_C1:
```

And of course let's not forget the error routine:

```
RC5_error:
      ldi     LowByte,$FF      ; fill both LowByte
```

```
      ldi      HighByte,$FF      ; and HighByte with $FF
      cbr      Flags,1           ; and reset Flags:0 just to be sure
      rjmp     Write_RC5         ; indicating an error state
```

That's all there is to it. For those of you who feel desperation coming on, be advised that I am a relative newbee to Atmel ASM programming and this was one of my first projects. True, I had previous ASM programming experience, but that was over thirty years ago with a Z80 (on 1.8 MHz in a TRS-80 model 2).

Ok, so now we've stored two bytes worth one RC5-command in SRAM, now what?

Well, …

```
main_loop:
      rcall    Delay1sec         ; Delay one second
      sbrc     Flags,0           ; Check for news
      rcall    GetRC5            ; When there's news, get it

      rjmp     main_loop         ; or else just wait another second
```

If we were to extend our main program with something like the above, we would need a one second delay routine obviously…

```
Delay1sec:
      push     TEMP           ; save TEMP
      in       TEMP,SREG      ; as well as SREG
      push     TEMP           ; and the rest
      push     LowByte
      push     HighByte

      ldi      TEMP,$50       ; initial delay value gives about one sec.

ReloadHigh:
      ldi      HighByte,$FF   ; Fill the two other loops

ReloadLow:
      ldi      LowByte,$FF

DecLow:
      dec      LowByte        ; and start counting down
      brne     DecLow

      dec      HighByte
      brne     ReloadLow

      dec      TEMP
      brne     ReloadHigh

      pop      HighByte       ; restore all
      pop      LowByte
      pop      Temp
      out      SREG,TEMP
      pop      TEMP

      ret                     ; and return
```

and of course some sort of command interpreter, that for instance outputs the command to port B.

```
GetRC5:
      push     TEMP              ; save registers
      in       TEMP,SREG
      push     TEMP
      push     ZL
      push     ZH
      push     LowByte

      ldi      TEMP,$FF          ; make all B's output
      out      DDRB,TEMP

      ldi      ZL,low(m_RC5)     ; set pointer to m_RC5 in SRAM
      ldi      ZH,high(m_RC5)
      ld       TEMP,Z            ; and get the whole byte

      cpi      TEMP,$01          ; '1' key pressed?
      brne     No_one

      ldi      TEMP,$01          ; set TEMP to '1'
```

```
      out      PORTB,TEMP        ; transport to PORTB

No_one:
      cpi      TEMP,$02          ; '2' key pressed?
      brne     No_two

      ldi      TEMP,$02          ; set TEMP to '2'
      out      PORTB,TEMP        ; transport to PORTB

No_two:
      cpi      TEMP,$03          ; '3' key pressed?
      brne     No_three

      ldi      TEMP,$00          ; set TEMP to '0'
      out      PORTB,TEMP        ; transport to PORTB, lights out

No_three:
      ldi      Flags,$00         ; reset Flags

      pop      LowByte           ; restore registers
      pop      ZH
      pop      ZL
      pop      TEMP
      out      SREG,TEMP
      pop      TEMP
      ret
```

True, we haven't looked at the device byte at this time, but you'll be able to switch a LED or two on or off. Easily extendable.

Here's the whole caboodle without the noise and have fun with it:

```
;***************************************************************************
;*  Program name      : RC5.asm                                            *
;*  Written by        : Jan van Rijsewijk                                  *
;*  Date              : 17-07-2006                                         *
;*  Device            : Atmel ATmega16 @ 16MHz                             *
;*-------------------------------------------------------------------------*
;*  Purpose           : Decodes RC5 code and stores the result in SRAM in  *
;*                    : variable m_RC5. The first byte will be the actual  *
;*                    : command and the second byte will be the device code *
;*                    : Upon correct reception bit0 of register Flags will  *
;*                    : be set. Incorrect reception will result in m_RC5 to *
;*                    : be filled with $FFFF and no flag set.              *
;*                    : The routine uses Timer2 Overflow Interrupt.        *
;*                    : The IR-detector should be connected to PINC4.      *
;*                                                                         *
;***************************************************************************

.nolist
.include "m16def.inc"
.list

; variables in SRAM.

.dseg
m_RC5:                      ; RC5 command received. Second byte represents system code (5 for VCR)
.dw $0000                   ; and first byte represents the actual command

; Register definitions

.def  TEMP     =r16
.def  TEMP2    =r17
.def  Counter1 =r18
.def  Counter2 =r19
.def  Flags    =r20
.def  LowByte  =r24
.def  HighByte =r25

; Interrupt vector table

.cseg
.org $0000
      rjmp     reset              ; Common start/reset

.org OVF2addr
      rjmp     IntOVF2            ; Timer2 Overflow vector
```

```
;-----------------------------------------------------------------------
reset:
      ldi         TEMP,low(RAMEND)  ; setup Stack before all else
      out         SPL,TEMP          ; Low BYTE
      ldi         TEMP,high(RAMEND) ; High BYTE
      out         SPH,TEMP          ; setup Stack done

      ldi         Flags,$00         ; reset all flags

;     Setup Timer2 Overflow interrupt

      ldi         TEMP,(1<<TOIE2)
      out         TIMSK,TEMP        ; enable Timer2 overflow

      ldi         TEMP,$00          ; set initial value Timer2
      out         TCNT2,TEMP

      ldi         TEMP,$01          ; prescaler /1 for Timer2, timer starts
      out         TCCR2,TEMP        ; interrupt every 64 usecs. so we have to
      ldi         Counter1,$04      ; setup Counter1 to divide by 4

;-----------------------------------------------------------------------
      sei                           ; Global Interrupt Enable
;=======================================================================

main_loop:
      rcall    Delay1sec         ; Delay one second
      sbrc     Flags,0           ; Check for news
      rcall    GetRC5            ; When there's news, get it

      rjmp     main_loop         ; or else just wait another second

;-----------------------------------------------------------------------

IntOVF2:
      push     TEMP              ; save usual registers
      in       TEMP,SREG
      push     TEMP
      push     TEMP2
      push     HighByte
      push     LowByte
      push     ZH
      push     ZL

      dec      Counter1          ; decrease counter
      breq     CheckStartBit     ; 64 usec passed? If not return
      rjmp     RetIntOVF2        ; brne is too far away

CheckStartBit:
      sbrc     Flags,0           ; jump over if there are no pending
      rjmp     Reload_C1         ; RC5 commands (bit0 must be cleared)

      in       TEMP,PINC         ; get PINC and
      sbrc     TEMP,4            ; if there is activity, move on (active LOW)
      rjmp     Reload_C1         ; else return

      ldi      Counter2,$03      ; Wait for about 130 usecs
                                 ; to get to about half the active part
ReloadCount1S:                   ; of the first start bit. That is, if
      ldi      Counter1,$E6      ; it is RC5 ofcourse

DecCount1S:
      dec      Counter1
      brne     DecCount1S

      dec      Counter2
      brne     ReloadCount1S     ; 129.94 usecs later move on

      rcall    DelayHalfBit      ; If there is activity
                                 ; wait another half bit
      in       TEMP,PINC         ; now C4 should be low here, or it's not RC5
      sbrs     TEMP,4
      rjmp     Reload_C1         ; and so return

      rcall    DelayHalfBit      ; wait another half bit time

      in       TEMP,PINC         ; Now C4 must be high (start bit #2)
      sbrc     TEMP,4            ; yes means '2 start bits received OK'
      rjmp     Reload_C1         ; or else go back
```

```
      ldi      HighByte,$06      ; so now we have 2 start bits and room for the toggle bit
      ldi      TEMP2,$06         ; now we read the next 6 bits

GetDeviceByte:
      rcall    DelayHalfBit      ; wait another half bit time

      in       TEMP,PINC         ; Now get the value of PINC4
      sbrc     TEMP,4            ; check for '0' of IRM-2636
      rjmp     C4H_isHigh        ; which means a RC5-0bit should come
                                 ; '1' would mean a RC5-1bit should come
      cbr      HighByte,1        ; set bit0 to 0
      rjmp     C4H_nextHalf

C4H_isHigh:
      sbr      HighByte,1        ; or set bit0 to 1

C4H_nextHalf:
      rcall    DelayHalfBit      ; wait for the second half bit time

      in       TEMP,PINC         ; Now get the value of PINC4
      bst      TEMP,4            ; move bit4 (IRM-2636)
      clr      TEMP              ; ...clean out TEMP in the mean time...
      bld      TEMP,0            ; to bit0
      eor      TEMP,HighByte     ; eor with HighByte and then
      andi     TEMP,0b00000001   ; mask off bit0, which should be 1
      breq     RC5_error         ; if not, there's an error, so go back

      dec      TEMP2             ; decrease bit counter
      breq     GetCommandByte    ; if all is well, move on (device received)

      lsl      HighByte          ; or else make room for the next bit
      rjmp     GetDeviceByte     ; and go and get it

GetCommandByte:                  ; Next, get the command byte
      ldi      LowByte,$00       ; so now we have the device byte
      ldi      TEMP2,$06         ; and we read the next 6 bits for the
                                 ; command byte
LoopCommandByte:
      rcall    DelayHalfBit      ; wait another half bit time

      in       TEMP,PINC         ; Now get the value of PINC4
      sbrc     TEMP,4            ; check for '0' of IRM-2636
      rjmp     C4L_isHigh        ; which means a RC5-0bit should come
                                 ; '1' would mean a RC5-1bit should come
      cbr      LowByte,1         ; set bit0 to 0
      rjmp     C4L_nextHalf

C4L_isHigh:
      sbr      LowByte,1         ; or set bit0 to 1

C4L_nextHalf:
      rcall    DelayHalfBit      ; wait for the second half bit time

      in       TEMP,PINC         ; Now get the value of PINC4
      bst      TEMP,4            ; move bit4 (IRM-2636)
      clr      TEMP              ; ...clean out TEMP in the mean time...
      bld      TEMP,0            ; to bit0
      eor      TEMP,LowByte      ; eor with LowByte and then
      andi     TEMP,0b00000001   ; mask off bit0, which should be 1
      breq     RC5_error         ; if not, there's an error, so go back

      dec      TEMP2             ; decrease bit counter
      breq     RC5_complete      ; if all is well, move on (command received)

      lsl      LowByte           ; or else make room for the next bit
      rjmp     LoopCommandByte   ; and go and get it

RC5_complete:
      sbr      Flags,1           ; set bit0 in Flags

Write_RC5:
      ldi      ZL,low(m_RC5)     ; Point to 'm_RC5' in SRAM
      ldi      ZH,high(m_RC5)

      st       Z+,LowByte        ; and write RC5-command
      st       Z,HighByte        ; and RC5-device number

Reload_C1:
      ldi      Counter1,$04      ; restore counter1
```

```
RetIntOVF2:
      pop      ZL               ; and get the rest back
      pop      ZH               ; to the original state
      pop      LowByte          ;
      pop      HighByte         ;
      pop      TEMP2            ;
      pop      TEMP             ;
      out      SREG,TEMP        ;
      pop      TEMP             ;
      reti                      ; Go back

;-------------------------------------------------------------------------------

DelayHalfBit:
      ldi      Counter2,$14     ; Wait for half a RC5 bit time

ReloadCount1:
      ldi      Counter1,$EC     ; Counter1 can be used here

DecCount1:
      dec      Counter1
      brne     DecCount1

      dec      Counter2
      brne     ReloadCount1     ; 888.75 usecs later return
      ret

;-------------------------------------------------------------------------------

RC5_error:
      ldi      LowByte,$FF      ; fill both LowByte
      ldi      HighByte,$FF     ; and HighByte with $FF
      cbr      Flags,1          ; and reset Flags:0 just to be sure
      rjmp     Write_RC5        ; indicating an error state

;-------------------------------------------------------------------------------

Delay1sec:
      push     TEMP          ; save TEMP
      in       TEMP,SREG     ; as well as SREG
      push     TEMP          ; and the rest
      push     LowByte
      push     HighByte

      ldi      TEMP,$50      ; initial delay value gives about one sec.

ReloadHigh:
      ldi      HighByte,$FF  ; Fill the two other loops

ReloadLow:
      ldi      LowByte,$FF

DecLow:
      dec      LowByte       ; and start counting down
      brne     DecLow

      dec      HighByte
      brne     ReloadLow

      dec      TEMP
      brne     ReloadHigh

      pop      HighByte      ; restore all
      pop      LowByte
      pop      Temp
      out      SREG,TEMP
      pop      TEMP

      ret                    ; and return

;-------------------------------------------------------------------------------

GetRC5:
      push     TEMP             ; save registers
      in       TEMP,SREG
      push     TEMP
      push     ZL
      push     ZH
      push     LowByte
```

```
      ldi      TEMP,$FF          ; make all B's output
      out      DDRB,TEMP

      ldi      ZL,low(m_RC5)     ; set pointer to m_RC5 in SRAM
      ldi      ZH,high(m_RC5)
      ld       TEMP,Z            ; and get the whole byte

      cpi      TEMP,$01          ; '1' key pressed?
      brne     No_one

      ldi      TEMP,$01          ; set TEMP to '1'
      out      PORTB,TEMP        ; transport to PORTB

No_one:
      cpi      TEMP,$02          ; '2' key pressed?
      brne     No_two

      ldi      TEMP,$02          ; set TEMP to '2'
      out      PORTB,TEMP        ; transport to PORTB

No_two:
      cpi      TEMP,$03          ; '3' key pressed?
      brne     No_three

      ldi      TEMP,$00          ; set TEMP to '0'
      out      PORTB,TEMP        ; transport to PORTB, lights out

No_three:
      ldi      Flags,$00         ; reset Flags

      pop      LowByte           ; restore registers
      pop      ZH
      pop      ZL
      pop      TEMP
      out      SREG,TEMP
      pop      TEMP
      ret

;-----------------------------------------------------------------------------
```