Modular 6502 SBC with emulated CPU

Modular 6502 SBC with emulated CPU

Overview

The minimum system consists of an ATMega16 or ATmega32, an SRAM IC and some components to provide a clock source and an RS232 interface. Non volatile program storage can be added as a serial EEPROM. Examples in the source code showcase, how parallel IO can be added from simple latches to 65xx IO ICs.

The emulation provides:

- some of the ATMega's internal IO capabilities
  - RS232
  - I²C/TWI
  - SPI
  - Timers
  - Interrupt pins as IRQ and NMI inputs
- proper sequencing of tags for parallel IO devices
  - glue logic replaced by emulation IO modules and simple registers
  - 1 MHz phase 2 output to drive 65xx IO circuits
- a debugger & monitor outside of the emulated CPUs code space
  - alter and display registers and memory
  - program control – start, stop, breakpoint and single step
  - trap undefined opcodes
- loading or booting code images to RAM from
  - RS232 (monitor console)
  - serial I²C or SPI EEPROM

The achievable 6502 equivalent clock speed is approximately 2 MHz at 16 MHz ATMega clock.

In order to combine the required IO blocks for your own SBC design, some AVR assembler knowledge is required. The emulator source code contains extensive comments about configurable items and the usage of emulated registers in the IO page.

Hardware

Main block – CPU, SRAM and RS232

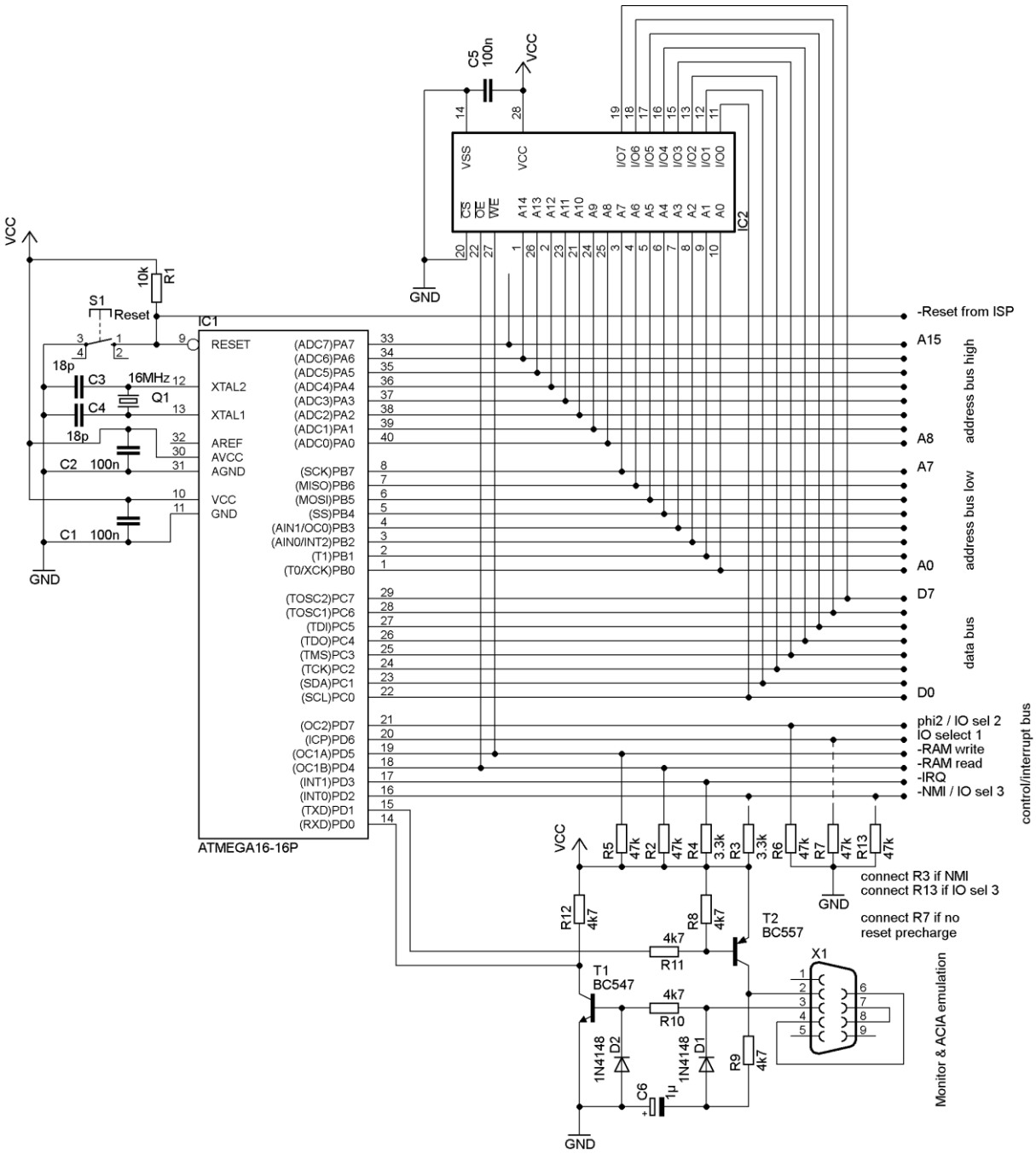
Already a complete system, as you can run programs by loading them through the monitor's console.

The RS232 connection is accessible through a limited ACIA emulation. The ACIA is hardwired to the same parameters in use by the monitor. The default is 8n1 and 38400 Baud, but can be changed in the emulator's source. The emulated ACIA has 256 Bytes of buffer each for both TX and RX and is connected to the IRQ input if the ACIA is programmed to enable both or either TX and RX interrupts.

Instead of the suggested simple transistor RS232 interface you may use the usual MAX232. I have used the simple interface many times up to 115kBaud and it has always worked.

SRAM is attached to the respective data, address and control busses. You can choose larger or smaller memory as required. There is no glue logic and therefore RAM is mirrored across the whole 64k address range, except the IO page which is internally mapped to IO devices. Please note that instruction fetches will always read from RAM even if it is the address of an IO page. The pull-up resistor on -RAM write should prevent unintentional RAM writes during hardware reset.

Add an in system programming (ISP) connector as needed (see SPI section).

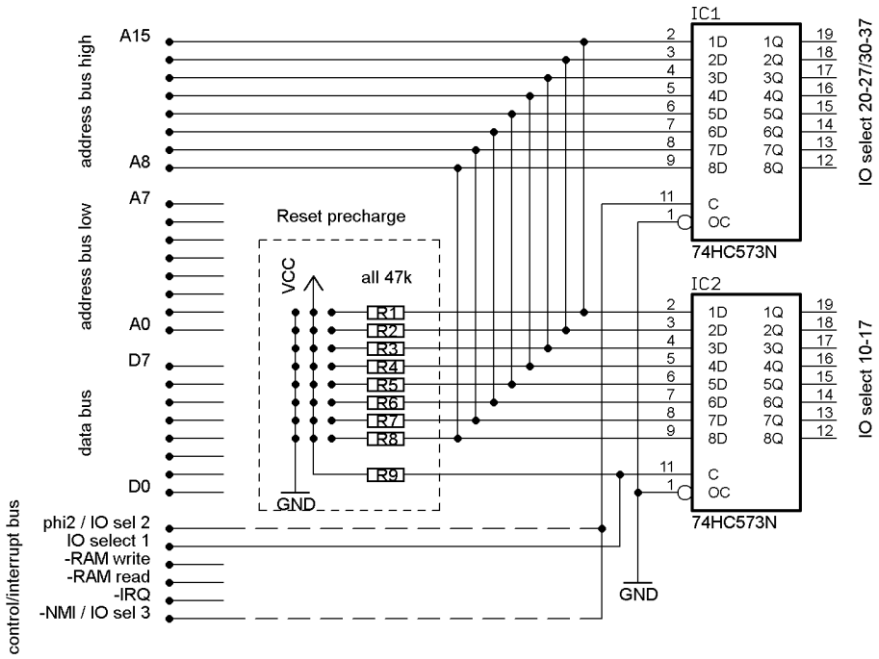


Adding parallel IO is simple. Just hook up the data pins to the data bus and the tags to IO select and let the emulation wiggle the tags. Since there are not many pins left, some pins have to be reused in more complex configurations.

IO select expansion

Most of the time one wants to have more IO control lines similar to the output of some glue logic. Simple registers are used for that. The emulation decodes IO addresses and asserts data strobe and other control signals to the IO devices.

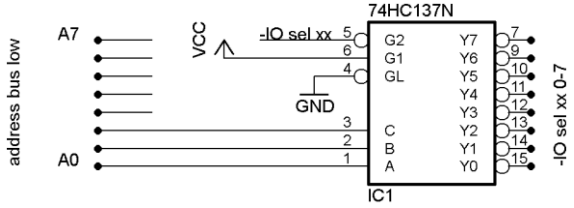
The address bus high is used for this purpose as we already know that we are on an IO page. The address bus low remains valid and can be used to decode individual IO addresses. A read/write signal is applied to bit 7.



Of course you can just use 1 register, if that provides enough control lines. You may even connect a third register if neither NMI input nor phi2 output are needed.

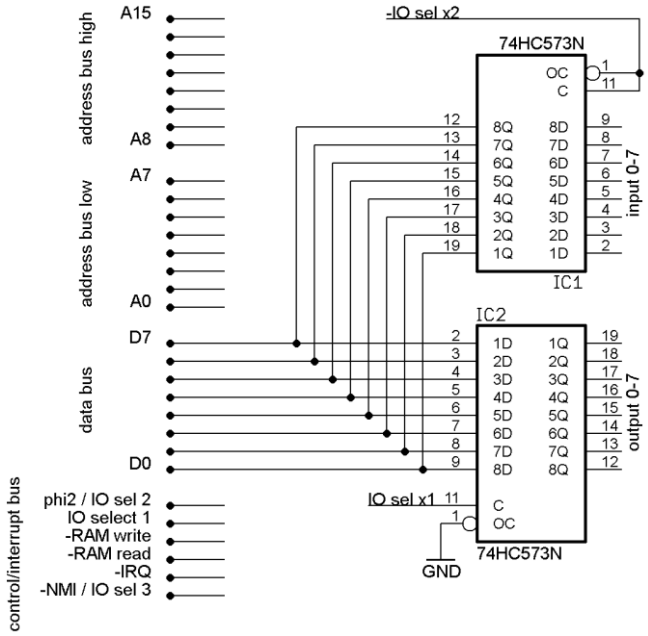
The optional 47k pull up/down resistors should initialize the IO select lines during the initial reset (tri state of the AVR pins) to an inactive state. The goal is to prevent multiple IO devices from writing to the data bus and potentially destroying their output drivers especially in a battery powered system, where the brown out detector could keep the ATMEGA in reset due to low power. It can also be used to generate a reset to external devices or tri state output latches including the second and third IO expansion register.

The IO selects can be further qualified by the lower address bits. The example below could create individual low strobes for 8 addresses. A 74HC138 can be used alternatively as latching is not required. Use a 74HC237/238 to create high strobes.

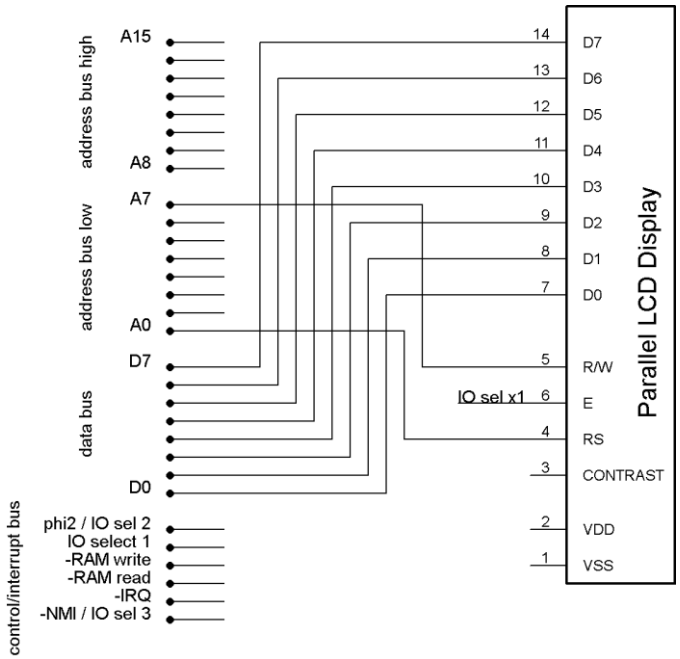


Parallel IO

The simplest type of parallel IO one can connect are registers. Please note, that the input register latches the data upon enabling its output to the data bus. The output registers output enable could be used to assert a reset condition. May use 74HCT573 for 3.3V & TTL compatible input.



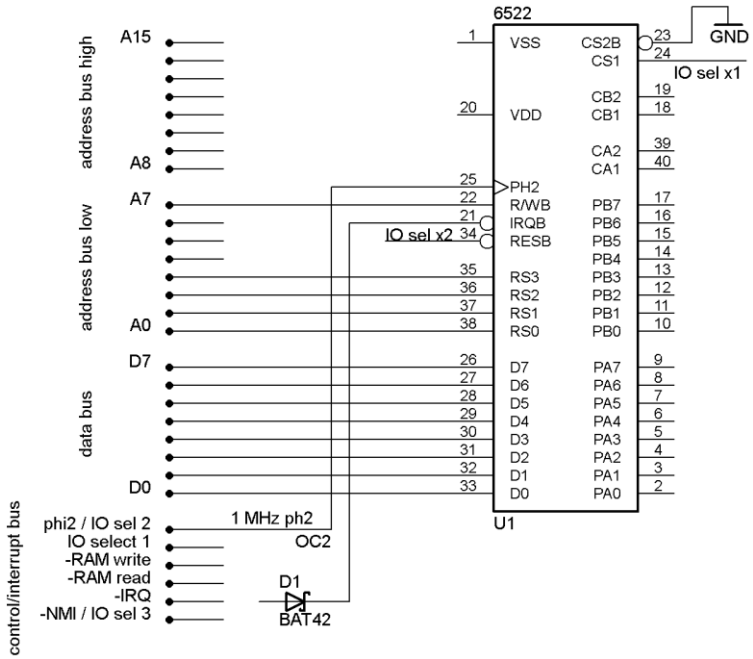
Example of a character LCD connected to the data bus over a short distance only. Address bus low is used as follows: A0 selects between data and command mode, A7 selects read/-write.



In the next example you see a typical 65xx IO device connected to the busses. Again A7 is driving R/W. A second IO select is used to allow a reset of the IO device. This can be common with other devices requiring a reset.

These Chips require a steady phase 2 clock to drive their internal timers and shift registers. A phase 2 clock can be generated on the AVR timer 2 output compare pin as an alternative usage of IO-select 2.

Original NMOS devices may have a very low high output level. They barely reach the high voltage threshold of 0.6*Vcc required by an AVR. To avoid problems, you may add some pull-up resistors on the data bus or use a TTL compatible bus transceiver like the 74HCT245.



IRQB can be connected to either NMI or IRQ or even left unconnected. If the IRQB output is push/pull you need a schottky diode to isolate the high driving transistor from the IRQ pin as this would prevent other open drain interrupts on the same pin including internal interrupts driving the IRQ pin low.

Serial IO

An AVR provides basically 3 serial IO communication methods: RS232, SPI & TWI/I²C. As seen in the basic SBC's diagram, RS232 is already utilized for the monitor/debugger but can be shared for application IO. The other two methods must share their IO-pins with the emulated 6502 busses. As a consequence it is necessary to separate these IO methods from the busses using some logic components.

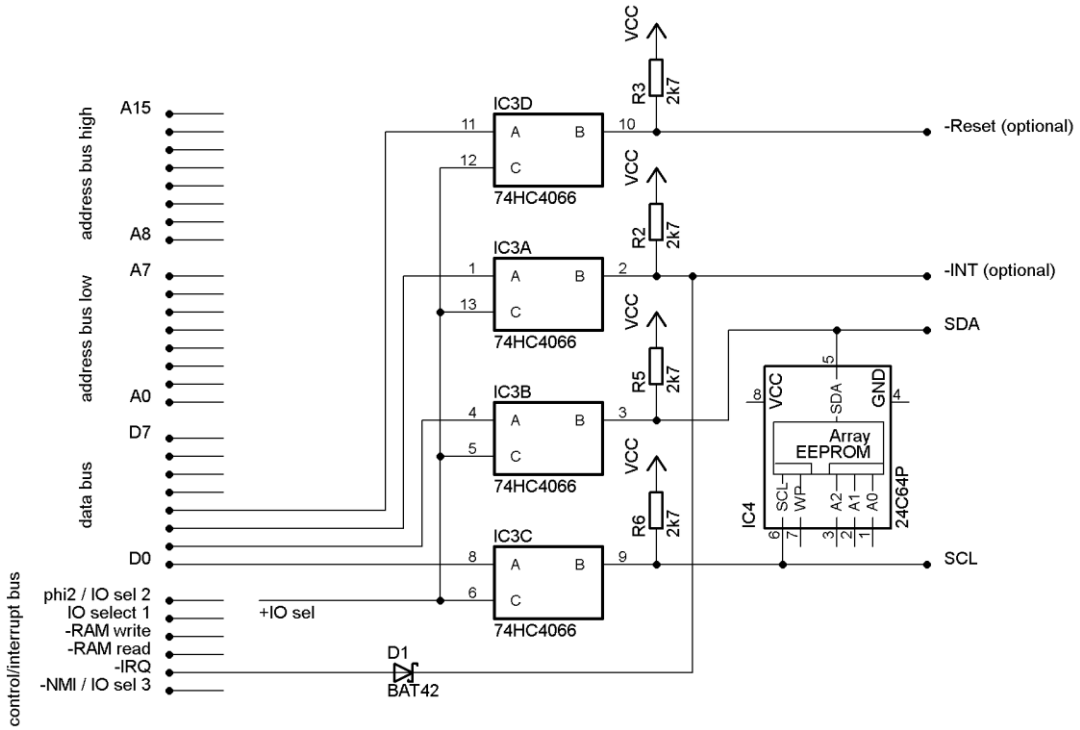
During transfer of a byte over SPI or TWI all bus activity is halted. This is permissible for most applications. However, there may be some applications that do not tolerate the required halt of instruction execution mainly because of interrupts being delayed until a byte is completely transferred over the serial bus. These cases could benefit from an alternate method using a second AVR as an independent IO-controller.

TWI/I²C master

On an I²C bus even the clock is bidirectional (clock stretching). Simple tri state gates wouldn't be sufficient, so bilateral switches are used for this task. The EEPROM in this example can be used by the monitor to load the emulator with a ROM image.

In this setup the TWI bus will only be disconnected while the master is not driving either SCL or SDA low. On read the bus is disconnected before sending an Ack, on write the disconnect is done after Ack has been received.

An optional interrupt circuit and input can be added to allow the I²C device to signal certain conditions like data waiting. A Reset output can be added if your I²C devices support them. The reset can then be driven by the application setting the reset command in the I²C control register.



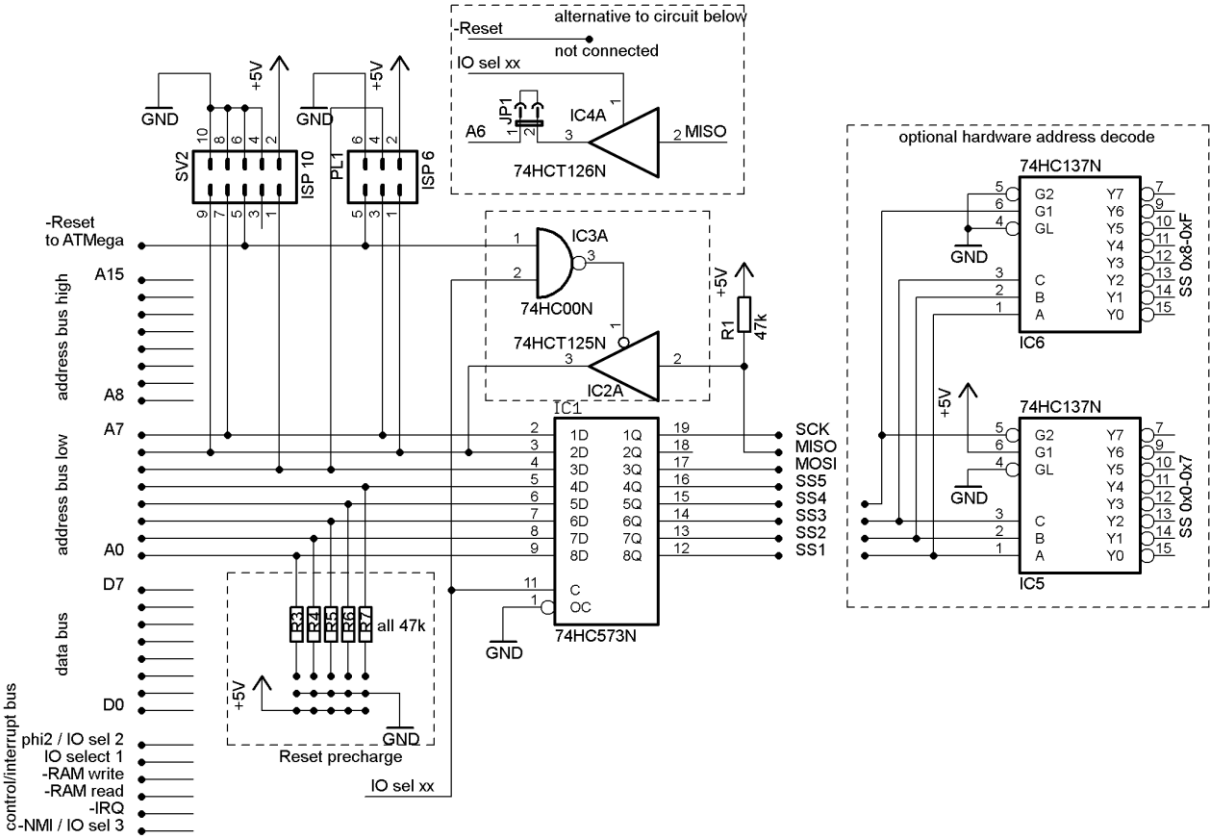
SPI master

A more complex SPI device like an SPI EEPROM or SD card requires the slave select signal to be held active for more than one byte. At the same time the clock and slave select signal must be frozen and MISO must be disconnected between bytes while the emulator is using port B as the lower address bus. The MISO input is 3.3V compatible when using HCT type bus buffers. If that is not required you may use HC types.

Since ISP also uses the SPI interface MISO must be disabled for all other devices than the ISP to allow programming of the AVR. Of course one could simply disconnect MISO by jumper instead of the extra NAND with reset.

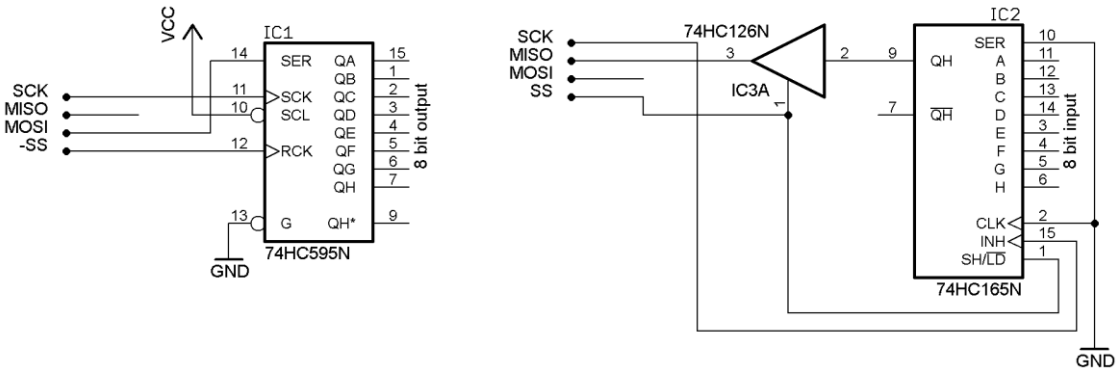
All slaves should be deselected during reset. For this reason the SPI IO select signal should be activated while the ATMEGA is tri stating its pins during a reset. The slave selects are then pre-charged by the push/pull resistors as required.

The slave select pins are set by software and an inverted phase may be duplicated on a second IO pin to avoid a hardware inverter where necessary. A hardware address decode is available alternatively. The example only shows 74HC137 being used but the latch is not required and 74HC138 can be used as well. 74HC237/238 can be used for slaves to be selected by a high input.



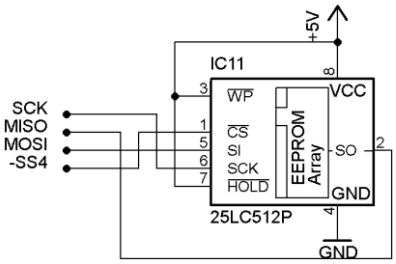
Connecting SPI devices

Simple shift registers can be connected to the SPI channel. If there are no other devices on SPI you can connect one of the shifter circuits below directly to the port B SPI pins and the SPI IO select avoiding the extra hardware described above.

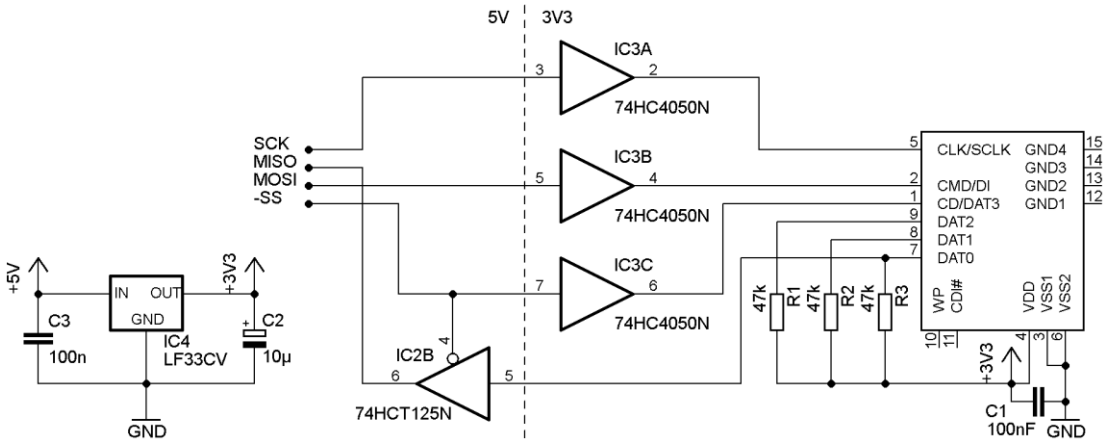


To make the input shifter's slave select active low a 74HC(T)125 can be used combined with an inverter (from remaining 74HC00 gates) to the shift/-load input of the 74HC(T)165.

You can also connect an SPI compatible EEPROM. A 64k EEPROM can be used by the monitor to store & load binaries.



Another option is to connect an SD card socket. Shown below is an SD card interface with proper 5V to 3.3V conversion. The 74HC4050 Vcc pin must be connected to 3.3V.



Customizing the emulator source code

Prior to assembling the emulator in AtmelStudio, the source code must be configured to match your hardware and feature requirements. For simplicity, all changes only go into the 6502_Emu_config.inc file. This should make it possible to retain the customized configuration over most of the firmware upgrades in the future.

CPU and SRAM hardware

Defining the AVR type

You may select ATMEGA16 or ATMEGA32. The emulator may require more than 16kB flash and therefore may not fit in an ATMEGA16 device, if you attempt to built a feature rich system.

```
; ATMega platform - ATMega16 is minimum - may run on ATMega32 without modification
.IFDEF   SIGNATURE_000  ;already defined per default device in Atmel Studio >4
   .NOLIST
   .INCLUDE "m16def.inc"
   ;.INCLUDE "m32def.inc"
   .LIST
.ENDIF
```

In AtmelStudio >4 the device dependant defines are automatically selected by the device type set in the EMU_6502_asmproj file, if [Tool chain] [General] [Include File] is set to ($IncludeFile). If ($IncludeFile) is removed the defines are selected as specified in the source code.

Defining the clock speed

The clock is defined in Hz and must match the crystal for the ATMEGA. It is not recommended to use the internal RC oscillator of the device, as it does not provide the clock stability necessary for the UART.

```
;AVR clock speed
.equ  Osc_Hz            = 16000000    ;16 MHz
```

The cycle time is calculated automatically based on the frequency of the clock and must not be changed.

Defining the SRAM wait time

You may modify the timing for the external SRAM chip. However, in most cases at least three clock cycles of the wait time are used for various background activities like incrementing the PC, checking for interrupts, decoding the opcode, checking for IO page address, jumping to common operation and more. Therefore decreasing the wait time may not speed up the emulator significantly. At 16MHz a clock cycle is 62.5ns. So the default wait time of 180ns generates 3 wait cycles (actually 187.5ns). This will allow for SRAM chips with up to 120ns delay between address input valid and data output valid.

```
;external SRAM waittime between applying address and data stable
.set  data_valid_ns     = 180
```

Defining the RS232 port

The RS232 port is used to control the emulator by the built in monitor & debugger. In addition it can be used by a program running on the emulator, if an IO address is assigned (see Emulator IO feature) and the debugger / monitor is inactive.

Baud rate

The Baud rate is automatically checked against the oscillator frequency and will be rejected, if the resulting Baud rate from the clock divider is off by more than 1%. In this respect the maximum standard Baud rate for a 16MHz crystal is 38400. A higher Baud rate will require a typical Baud rate crystal with a frequency of 14.7456MHz for example. To be able to send status line updates by the debugger every half second, the Baud rate should not be less than 2400.

```
;Baudrate
.equ BAUD  = 38400
```

Flow control

A software flow control can be defined for the RS232 input buffer. This is useful, when large amounts of ASCII data is to be downloaded to a program running on the emulator. A typical example would be a large program listing to be transferred to EhBASIC. Since EhBASIC must first make room for any inserted program line, it would be unable to receive a large continuous stream of basic code without the flow control option.

Caution! When the RS232 port is connected to a USB adapter, more conservative settings for flow control are required. I have tested flowhi=128 and flowlo=64 with Tera Term. PUTTY would still overflow the input buffer of the emulator with these settings. You may have to experiment with lower Baud rates or delay settings of the terminal program.

If the program needs to send or receive raw binary data through the RS232 connection, flow control should be disabled. Simply change the equates to comments.

```
;RS232 software flow control watermarks, disabled if undefined
;  the buffer max is 255 bytes
;  RX only to prevent overruning applications input capacity
.equ flowlo = 128       ;low watermark, send XON if less buffer used
.equ flowhi = 192       ;high watermark, send XOFF if reached
```

The terminal program on the PC side must also have software flow control enabled to make it work.

Emulator CPU features

Defining the emulated CPU type

Two emulations are selectable. The classic NMOS 6502 CPU or the newer CMOS 65C02. Uncomment the equate for the CMOS core to enable the 65C02 extra instructions. The CMOS emulation includes the BBR, BBS, RMB, SMB, STP & WAI instructions. However, STP & WAI will not reduce power consumption of the ATMEGA. STP will bring up the debugger in stopped state.

```
;select emulator core CMOS 65C02, NMOS 6502 if undefined
;.equ cmos_core = 1
```

Defining the interrupt disable mode

The behavior of the emulator in respect to the interrupt disable bit in the processor status register can be configured. The default mode is virtual and allows background interrupts for the monitor and debugger to proceed even when the interrupt disable bit is set. The disable is only relevant for an external IRQ input or any internal interrupt reassigned to the emulated machine. Therefore this interrupt scheme does not guarantee atomic operation of 6502 instructions during interrupt disable.

The real mode allows full control of all interrupts and therefore instructions executed under interrupt disable are atomic. There are a few disadvantages:

1. adds extra ATMEGA clock cycles to instructions modifying the interrupt disable bit in the processor status register, see ** note in the 65(C)02 Emu speed calc document
2. loss of control, if programmer forgets to do CLI, requires HW reset

```
;interrupt disable mode real, virtual if undefined
;.equ irq_dis_real = 1
```

Defining write protected RAM

RAM can be protected against writes from the emulated CPU at and above a certain page address (pseudo ROM). This feature adds 2 ATMEGA clock cycles (1/4 6502 clock cycle) to every modify instruction and every store instruction with a non ZP address. Default is disabled.

If multiple mirrors of the same RAM exist (SRAM is less than 64k), then the protect address must be set against the lowest RAM mirror.

```
;ROM load & write protect page
;.equ rommap    = 0xc0
```

Emulator IO features

Defining the IO page

256 addresses (1 page) will be reserved for IO registers. These registers can only be accessed through absolute operand addresses on the emulated CPU. Neither the debugger nor instruction fetches will be able to access IO. Both will access the RAM instead. Code and IO space can co-exist in the same address page. ZP is not a valid IO page.

```
;internal IO access page (RS232, Timers, Latches) - disabled if undefined
.equ iomap     = 0xbf
```

IO can be disabled, if you change the equate to a comment.

Defining IO select

On the emulator, three pins are available to generate chip selects to IO circuits. These are PD6, PD7 and PD2. Some may be in use for other functions.

1. PD6 is always IO select 1
2. PD7 is phase 2 clock output for 65xx IO chips, optional IO select 2
3. PD2 is NMI input, optional IO select 3

```
; enable phi2 output on portd7, disabled if undefined
.equ phi2_ena  = 1
; enable NMI input on portd2, disabled if undefined
.equ nmi_ena   = 1
```

To enable one of the optional IO select pins, its special function must be disabled by changing the equate to a comment.

Defining IO select expansion registers

Since only 1-3 pins may not be enough to generate chip selects for all external IO circuits, IO selects can be expanded by standard 8-bit latches (74HC573). This is done by declaring their inactive value and it is recommended to place a comment for the usage of each pin. During IO access the expansion latch is loaded using the address bus bits 8-15.

```
.equ  ios1_default = 0b10000001        ;IO select pin 1 expansion default
;io select bits 10 = -74HC573 input latch
;               11 = 74HC573 output latch
;               12 = HD44780 compatible character LCD
;               13 = 65xx IO
;               14 = I2C connect (74hc4066)
;               15 = XMEM select
;               16 = SPI select
;               17 = -IO reset
```

In the example above IO select pins 10 & 17 must be strobed low to activate, all other pins are active high. The definition for ios1_default contains the inactive (reset) values for each pin.

The hardware may support forcing a proper reset condition by weak pull up or pull down resistors (47k) on the inputs of one latch and a pull up on its gate input. When the ATMEGA goes high-Z, the reset values are present on the output of the latch. Further IO expansion latches can only be reset by the emulation software (definition for IOSx_default).

Templates are available for the other 2 potential IO select expansions and their chip enables are called 20-27 & 30-37 if activated.

Defining IO selects for built-in modules

Some IO options are pre-coded in the emulator's IO section and are enabled by the following equates. Serial bus modules to drive I2C and SPI require extra hardware connected to the defined IO selects. The LCD select requires a HD44780 compatible module.

```
; enable SPI module and define IO select signal for SPI, disabled if undefined
.equ spi_sel   = 16     ;IO select register 1, pin 6

; enable I2C module and define IO select signal for I2C, disabled if undefined
.equ i2c_sel   = 14     ;IO select register 1, pin 4

; enable LCD module and define IO select signal for LCD, disabled if undefined
.equ lcd_sel   = 12     ;IO select register 1, pin 2
```

SPI details

Same as with the IO select expansion registers, there is a SPI slave select latch to be configured to its idle (reset) value. Only the 5 lower bits are used to select a slave. The upper 3 bits are used as clock and data bits of the SPI.

The hardware may support forcing a proper reset condition by weak pull up or pull down resistors (47k) on the inputs of the SPI latch and by a high state from the respective IO select on its gate input. When the ATMEGA goes high-Z, the reset values are present on the output of the latch.

```
;reset state, must be defined
.equ spi_idle  = 0b01101   ;all slaves inactive
;SS bit 0      = -SD card
;SS bit 1      = 74HC165 parallel input shifter (QH via 74HCT126)
;SS bit 2      = -74HC595 parallel output shifter
;SS bit 3      = -25LC512 SPI EEPROM 64kB
;SS bit 4      = SD card (MISO via 74HCT126)
```

The address in this case is translated to the appropriate bit combination according to the following translation macro.

```
;oplow BCD to SS pin table
.macro   spi_virt_ss
;  x0 = SD card on bit0 (-cs) & bit4 (+oe)
;  x1 = 74HC165 on bit1 (+sh/-ld +oe)
   .db 0b10001^spi_idle,0b00010^spi_idle
;  x2 = 74HC595 on bit2 (+rck)
;  x3 = 25LC512 EEPROM on bit 3 (-cs)
   .db 0b00100^spi_idle,0b01000^spi_idle
;  x4 inactive
;  x5 inactive
   .db spi_idle,spi_idle
;  x6 inactive
;  x7 inactive
   .db spi_idle,spi_idle
.endmacro
```

This requires the virtual address translation to be set.

```
;enable spi virtual address translation, disabled if undefined
.equ spi_vat = 1
```

As an alternative the slave selects will not be driven by the SPI latch directly, but can be decoded in hardware from the lower address bits. In that case spi_vat should remain undefined (masked as a comment). In that case you also need

```
;.equ spi_idle  = 0xf       ;unused address for hardware translation
```

Defining IO reset behavior

One of the IO select outputs can be configured to provide a reset for external IO circuits.

```
;a reset pin can be defined and must reside on an IO-select expansion register
;this pin is strobed low during reset if defined
.equ  io_reset_pin = 17
```

Custom subroutines can be defined to reset the contents of registers. Built in modules will reset automatically and 65xx IO chips can be reset via the reset signal defined above.

```
;definition for io modules with a reset vector
.macro   io_modules_reset
   rcall ltch_rs        ;reset 74HC573
.endmacro
```

Defining a serial EEPROM as program storage

```
;64kB serial EEPROM reserved for non volatile program storage
;eep_adr >= 8 = I2C slave adr.; eep_adr < 8 = SPI device adr.
;none if undefined
   ;.equ  eep_adr = 0xa0       ;I2C EEPROM
   .equ  eep_adr = 3          ;SPI EEPROM
```

If an SPI EEPROM is selected, the respective select signal should be disabled in the virtual address translation for an SPI address (see SPI details). Writes to an I^2C EEPROM are automatically disabled for emulated programs.

Defining a diagnostic register

This enables an IO address as a way to internally test the external IRQ and NMI inputs and to force a reset of the ATMEGA.

Caution! Enable at your own risk.

```
; self diag register enable, allow force IRQ, NMI, RESET
; should stay undefined to prevent you from shooting yourself in the foot
;.equ ena_diag  = 1
```

Defining address decode

The upper 8 bits of the IO address decode have already been defined in the iomap constant. Further decoding is done in groups of 16 addresses each. There is a separate table for reads and writes.

```
; IO address table, reserve 0x10 addresses at a time
;     larger chunks can be defined by pointing sequential addresses
;     to the same entry point
; read address table
ior_tab:
   rjmp  ir_exit     ;0x
   rjmp  ir_exit     ;1x
   rjmp  ir_exit     ;2x
   rjmp  ir_exit     ;3x
   rjmp  ir_exit     ;4x
   rjmp  ir_exit     ;5x
   rjmp  ir_exit     ;6x
   rjmp  ir_exit     ;7x
   rjmp  ir_exit     ;8x
   rjmp  spr_data    ;9x   SPI data register
   rjmp  io65xx_rd   ;Ax   65xx IO chip RAM
   rjmp  io65xx_rd   ;Bx   65xx IO chip
   rjmp  ltch_rd     ;Cx   74HC573 latch(es)
   rjmp  lcd_rd      ;Dx   character LCD
   rjmp  iorE0       ;Ex - subdecoded 0xe0 - 0xef
   rjmp  iordreg     ;Fx - internal emulated registers

; write address table
iow_tab:
   rjmp  iw_exit     ;0x
   rjmp  iw_exit     ;1x
   rjmp  iw_exit     ;2x
   rjmp  iw_exit     ;3x
   rjmp  iw_exit     ;4x
   rjmp  iw_exit     ;5x
   rjmp  iw_exit     ;6x
   rjmp  iw_exit     ;7x
   rjmp  iw_exit     ;8x
   rjmp  spw_data    ;9x   SPI data register
   rjmp  io65xx_wrt  ;Ax   65xx IO chip (RAM)
   rjmp  io65xx_wrt  ;Bx   65xx IO chip
   rjmp  ltch_wrt    ;Cx   74HC573 latch(es)
   rjmp  lcd_wrt     ;Dx   character LCD
   rjmp  iowE0       ;Ex - subdecoded 0xe0 - 0xef
   rjmp  iowtreg     ;Fx - internal emulated registers
```

Some entries are broken down to each address by sub decode tables, some are decoded in hardware, if the IO circuits have address inputs. If more than 16 addresses are needed, multiple

entries may point to the same IO label. Unused entries point to ir_exit & iw_exit. Labels of inactive built in modules are redirected to ir_exit & iw_exit. Addresses can be reassigned by swapping their respective RJMP labels in both read & write decode tables with an unused location.

Follow the labels iorE0: and iowE0: to find an example on how to decode the lower 4 address bits.

```
iorE0:                  ;read decode for 0xe0 - 0xef
   io_adr_dec  iorE0_tab

iowE0:                  ;write decode for 0xe0 - 0xef
   io_adr_dec  iowE0_tab
```

It just needs the tables it refers to, so more address decoders can be made by replicating the example decoder with other labels.

Coding access to IO circuits

There are also examples on how to drive parallel IO in the configuration file. Again, theses examples can be replicated with different labels to accommodate more IO chips on the data bus. The new labels replace unused entries in the address decoder tabs.

Fuse settings

When programming the ATMEGA for the first time, you must also set the correct fuse bits. Refer to your flash tool's documentation on how to set AVR fuses and to the ATMEGA manual, memory programming, fuse bits. The settings are:

- JTAGEN must be disabled. Otherwise port C (the data bus) will not work.
- BOD must be enabled, 4.0V. This is required to protect the internal EEPROM's content from corruption during low power conditions.
- Clock source - external crystal >8MHz: CKOPT = 0, CKSEL3..1 = 111 (see ATMEGA manual, system clock, crystal oscillator).
  Caution - dead ATMEGA ahead! Do not set external oscillator if you have a crystal.
- Set the Startup time according to Atmel's recommendations for crystal and BOD: CKSEL0 = 1, SUT1..0 = 01.
- EESAVE must be enabled to protect emulator startup settings and stored programs during firmware upgrades.

Caution! Enable means programmed = 0, disable means unprogrammed = 1.

Fuse high byte 0xC1

OCDEN	JTAGEN	SPIEN	CKOPT	EESAVE	BOOTSZ1	BOOTSZ2	BOOTRST
1	1	0	0	0	0	0	1

Fuse low byte 0x1F

BODLEVEL	BODEN	SUT1	SUT0	CKSEL3	CKSEL2	CKSEL1	CKSEL0
0	0	0	1	1	1	1	1

Changed versus default fuses.

Using the IO registers

Shared RS232

The ATMEGA's RS232 port is already in use for the built in debugger/monitor of the 6502 emulator. The very same port can also be used by programs running on the emulator. Bit rate, frame size and buffer parameters are fixed in the ATMEGA's source code and cannot be altered dynamically.

The default RS232 register definitions are as follows:

```
iomap   equ $bf00       ;IO page (A15:8)
ser_dat equ iomap+$f0   ;RS232 RX/TX data register, read/write
ser_msk equ iomap+$fe   ;RS232 interrupt enable mask register, read/write
ser_flg equ iomap+$ff   ;RS232 buffer flag register, read/write
```

RS232 data register

Reading the data register returns a null character, if the monitor /debugger is active or if the receive buffer is empty. Otherwise data is returned.

Writing the data register will cause the store or modify instruction to wait, while the monitor/debugger is active or the transmit buffer is full. Otherwise data is written to the transmit buffer. Even with the write waiting, an IRQ or NMI will still be processed if enabled. A RTI opcode will return to the waiting instruction.

RS232 flag register

Bit	Name	Description
7	TDRE	Transmit data register empty, transmit buffer can accept characters - read only
6	RDRF	Receive data register full, data waiting in receive buffer - read only
0 - 5		*Timer flag bits see Timer section, not used for RS232*

TDRE

The TDRE flag indicates that the transmit data buffer can except at least one more byte. It is cleared when the transmit data buffer is full or the monitor/debugger is active. Writing the TDRE flag bit is ignored.

RDRF

The RDRF flag indicates that the receive buffer is filled with at least one byte. It is cleared when no byte is remaining to be read from the receive buffer or the monitor/debugger is active. When RDRF is 1, a subsequent read of 0x00 is guaranteed to actually have been received on the RS232 port and is not a fill byte. Writing the RDRF flag bit is ignored.

RS232 interrupt mask register

Bit	Name	Description
7	ITDRE	IRQ on transmit data register empty enable
6	IRDRF	IRQ on receive data register full enable
0 - 5		*Timer flag bits interrupt enable see Timer section, not used for RS232*

ITDRE

Writing this bit to one enables the IRQ input to be set low when TDRE is one. The IRQ input is cleared when the TDRE flag is zero or ITDRE is written to zero.

IRDRF

Writing this bit to one enables the IRQ input to be set low when RDRF is one. The IRQ input is cleared when the RDRF flag is zero or IRDRF is written to zero.

Timer

Some of the internal timers are made available for use by the emulated program. A fixed 10ms periodic and countdown timer and a 16 bit timer is available. The fixed and countdown timer is based on the ATMEGA's timer 0. The 16 bit timer is based on the ATMEGA's timer 1. Timer 2 of the ATMEGA is reserved to generate phase 2 output for 65xx IO chips. No direct input or output modes are available for timer 1 & 0.

The default timer register definitions are as follows:

```
iomap   equ $bf00       ;IO page (A15:8)
tim_cdn equ iomap+$f1   ;10ms tick timer countdown register, read/write
t1_adr  equ iomap+$f4   ;timer 1 register address, read/write
t1_dat  equ iomap+$f5   ;timer 1 register data, read/write
tim_msk equ iomap+$fe   ;timer interrupt enable mask register, read/write
tim_flg equ iomap+$ff   ;timer flag register, read/write
```

Tick countdown register

When writing the tick countdown register with a value of 1 - 255 (0xFF), the value in the register is decremented every 10ms until it reaches zero. When the value reaches zero the TCDN flag is set in the timer flag register and an IRQ may be triggered if enabled in the timer interrupt enable register.

When reading the tick countdown register, a value of zero signals the expiry of the stored countdown.

Timer flag register

Bit	Name	Description
6-7		*RS232 flag bits see RS232 section, not used for timers*
5	ICF1	Timer 1 ICR compare match
4	OCF1A	Timer 1 OCRA compare match
3	OCF1B	Timer 1 OCRB compare match
2	TOV1	Timer 1 overflow
1	TCDN	Tick countdown timer 0 expired
0	TICK	Fixed 10ms tick timer 0

Writing a one to a timer flag bit will clear the flag and will release the corresponding IRQ if enabled and set.

ICF1, OCF1A, OCF1B

These flag bits are set when their corresponding register match the timer counter value.

TOV1

This flag bit is set when the timer reaches the top, bottom or maximum depending on the timer operation selected in the timer control register.

TCDN

This flag bit is set when the 10ms count stored in the tick countdown register has expired (a read returns 0x00).

TICK

This flag is set every 10ms (9.984ms at 16MHz).

Timer interrupt mask register

Bit	Name	Description
6-7		*RS232 interrupt mask bits see RS232 section, not used for timers*
5	IICF1	Timer 1 ICR compare match
4	IOCF1A	Timer 1 OCRA compare match
3	IOCF1B	Timer 1 OCRB compare match
2	ITOV1	Timer 1 overflow
1	ITCDN	Tick countdown timer 0 expired
0	ITICK	Fixed 10ms tick timer 0

Writing a one to the timer interrupt enable bit will enable an IRQ, if the corresponding flag bit is one.

Timer 1 registers

The timer 1 control, count and compare registers are accessed through an address and a data register. To read or write a timer 1 register, its address must be set first by writing to the timer 1 address register. For 16 bit register reads the low byte (even register address) and for writes the high byte (odd register address) must be accessed first. Reading or writing the data register switches automatically between high and low byte.

Start address:			
Read	**Write**	**Register**	**Description**
0x00		ICR1L	Timer 1 ICR compare register
	0x01	ICR1H	
0x02		OCR1BL	Timer 1 OCR B compare register
	0x03	OCR1BH	
0x04		OCR1AL	Timer 1 OCR A compare register
	0x05	OCR1AH	
0x06		TCNT1L	Timer 1 counter
	0x07	TCNT1H	
0x08	0x08	TCCR1	Timer 1 counter control register

ICR1, OCR1B, OCR1A

The compare registers will set their respective flags when the stored value matches the actual timer count. They can also be used to configure a counter top value, at which the timer wraps to 0x0000 (clear timer on compare mode) or reverses its count direction (up down count).

TCNT1

Writing the TCNT register presets the count of the timer. The 16 bit counter is written when the second (low) byte is stored. Reading the TCNT represents the count when the first (low) byte is read. In both cases the high byte is stored in an internal temporary register.

TCCR1

The upper nibble of the TCCR is used to control the timers operation modes. The lower nibble sets the clock prescaler options of the timer.

TCCR	Timer operation				*	Clock prescaler		
Bit	7	6	5	4	3	2	1	0

*Prescaler reset. Writing this bit to a one will reset the timer 1 prescaler. Always 0 on read.

TCCR 210	**Clock prescaler**	**Frequency @ 16 MHz**
000	No clock (counter stopped)	0 Hz
001	System clock/1 (no prescaling)	16 MHz
010	System clock /8	2 MHz
011	System clock /64	250 kHz
100	System clock /256	62.5 kHz
101	System clock /1024	15625 Hz
110	Not valid	undefined
111	Not valid	undefined

TCCR 7654	**Timer operation**	**Top**	**Update of OCR1x**	**TOV1 flag set on**
0000	Normal (up only)	0xFFFF	immediate	0xFFFF
0001	Up down, 8 bit	0x00FF	0x00FF	0x0000
0010	Up down, 9 bit	0x01FF	0x01FF	0x0000
0011	Up down, 10 bit	0x03FF	0x03FF	0x0000
0100	Up only, CTC (clear timer on compare match)	OCR1A	immediate	0xFFFF *
0101	Up only,8 bit	0x00FF	0x00FF	0x00FF
0110	Up only,9 bit	0x01FF	0x01FF	0x01FF
0111	Up only,10 bit	0x03FF	0x03FF	0x03FF
1000	Up down	ICR1	0x0000	0x0000
1001	Up down	OCR1A	0x0000	0x0000
1010	Up down	ICR1	ICR1	0x0000
1011	Up down	OCR1A	OCR1A	0x0000
1100	Up only, CTC (clear timer on compare match)	ICR1	immediate	0xFFFF *
1101	- undefined	-	-	-
1110	Up only	ICR1	ICR1	ICR1
1111	Up only	OCR1A	OCR1A	OCR1A

* Timer flags cannot be set by overflow or compare match, if they are out of range of the normally generated counter values (between 0x0000 and top).

IRQ

When an IRQ is received, the source of the interrupt can be determined by polling all enabled interrupt sources for their respective interrupt flag bits. As an alternative and faster method a vector register is provided by the emulator to distinguish, whether the source of an IRQ was external or which internal resource caused the interrupt.

The default IRQ vector register definitions is as follows:

```
iomap   equ $bf00       ;IO page (A15:8)
irq_vec equ iomap+$fd   ;irq vector register, read only
```

IRQ vector register

The IRQ vector register is read only.

Vector	Name	Interrupt source	Recommended action
0x00	VEXT	External input to the -IRQ pin	Poll external IO chips for pending interrupt
0x02	VTICK	10ms timer	Write one to the respective flag in the timer flag register
0x04	VTCDN	10ms tick countdown expired	
0x06	VTOV1	Timer 1 overflow	
0x08	VOCF1B	Timer 1 OCRB register match	
0x0A	VOCF1A	Timer 1 OCRA register match	
0x0C	VICF1	Timer 1 ICF register match	
0x0E	VRDRF	RS232 RDRF flag	Read the RS232 data register
0x10	VTDRE	RS232 TDRE flag	Write the RS232 data register or clear ITDRE

Using the 65C02 emulation an indexed indirect jump "JMP (vector_table,X)" can be performed using the above vectors as index. When only the 6502 base instruction set is available, the indexed vector -1 can be pushed to the stack followed by an RTS.

Data bus IO

8 bit IO to devices (latches, 65xx IO ICs) on the emulators data bus can be achieved by simply writing or reading the designated address space and will automatically generate the configured strobe or chip select signals.

Character LCD

A special driver exists to drive a HD44780 compatible character LCD. With this driver the busy condition of the LCD is automatically checked and a write or read of the LCD's registers will be kept waiting until the LCD is no longer busy. An interrupt during the wait time is allowed to execute if enabled and the waiting LCD IO will continue after the interrupt service ends (RTI).

A secondary set of LCD register addresses is provided to bypass the busy check. When these are used the program is responsible of waiting the maximum required busy time before issuing another IO to the LCD. Typically during re-initialization of the LCD the non wait registers should be used for LCD commands that do not support setting of the busy flag.

The default character LCD register definitions are as follows:

```
iomap   equ $bf00       ;IO page (A15:8)
lcd_cmd equ iomap+$d0   ;lcd command register, read/write
lcd_dat equ iomap+$d1   ;lcd data register, read/write
lcd_rcd equ iomap+$d2   ;lcd raw command register (no busy wait), r/w
lcd_rdt equ iomap+$d3   ;lcd raw data register (no busy wait), read/write
```

I²C

The I²C interface is master only, no interrupt capability. It shares its resources with the data bus (port D) of the emulator. Therefore normal CPU processing is stopped, while a byte is transferred on the I²C pins. The I²C interface is isolated from the data bus in normal operation by a 74HC4066 analogue switch.

Although the hardware TWI pins are used on the ATMEGA, the I²C interface is driven by bit-banging. Because normal operation of the emulated CPU is stopped during a byte transfer and the I²C speed is limited to 400 kHz, the ATMEGA provides enough processing power to support I²C in software only.

Please note that during the transfer of a byte, emulation interrupts are delayed until after the transfer is completed. This may include delays by clock stretching of the I²C device.

The default I²C register definitions are as follows:

```
iomap   equ $bf00       ;IO page (A15:8)
i2c_cmd equ iomap+$f2   ;i2c command register, write only
i2c_sta equ iomap+$f2   ;i2c status register, read only
i2c_dat equ iomap+$f3   ;i2c data register, read/write
```

I²C Command Register

The I²C Command Register is write only

Code	Name	Description
0x00	Stop	Send Stop, Nak & Stop if reading
0x01	Start 100kHz	Next data write is I²C device address, reset if stuck *, low speed
0x02	Start 400kHz	Next data write is I²C device address, reset if stuck *, high speed
0x03	Reset	Reset I²C devices, optional pulse reset pin low *

* reset requires ~ 60µs and may cause lost interrupts, extra delay for a hardware resettable slave may be required

I²C Status Register

The I²C Status Register is read only

Bit	Name	Description
7	Int	Interrupt from I²C, optional input
6	Ack	Acknowledge after Write, device present & ready
5	Stuck	Reset was unable to obtain control of the interface – SCL or SDA are stuck low
4	0	Unused
3	Read	Current operation is read, Read Ack outstanding
2	Speed	100kHz if clear, 400kHz if set
1	Start	Next data write is I²C device address
0	Stop	I²C stopped

I²C Data Register

Accessing the I²C data register initiates the data transfer with the I²C device. After a start command was issued, the first byte written is the I²C slave address (bits 7-1) and the r/w designator (bit 0, write=0, read=1).

After writing the I²C data register, the acknowledge bit in the I²C status register should be checked. When reading the data register, acknowledge is automatically send to the I²C device.

I²C IO Sequence

I²C Op	Application *Op Register(bit or code)*	Interface
Start	Write Command(Start) Write Data (Address) -> Read Status(Ack) *device present* <-	 If Status(Read) send Nak Status(Read) = Data(0) Send Data Receive Ack *
Write	Write Data -> Read Status(Ack) *device ready* <-	Send Data Receive Ack *
Read	 Read Data <-	If not 1st Read send Ack receive Data
Stop	Write Command(Stop) ->	If Status(Read) send Nak Send Stop
Query Interrupt (optional)	 Read Status(Int) <-	Receive –Int
Reset	Write Command(Reset) ->	Send 9 Naks (0x1ff) Send Stop Optional drive reset pin low

* Whenever a Nak is received, the interface automatically sends a stop to the device and sets status(stop).

Writing to an I²C device

1. Write 0x01 or 0x02 (start 100/400 kHz) to the I²C command register
2. Write I²C device address for write to the I²C data register
3. Read the I²C status register - verify ack received, device responding
4. Write data byte to the I²C data register
5. Read the I²C status register - verify ack received, device accepts more data
6. Repeat from 4. for desired number of bytes
7. Write 0x00 (stop) to the I²C command register

Reading from an I²C device

1. Write 0x01 or 0x02 (start 100/400 kHz) to the I²C command register
2. Write I²C device address for read (+1) to the I²C data register
3. Read the I²C status register - verify ack received, device responding
4. Read data byte from the I²C data register
5. Repeat from 4. for desired number of bytes
6. Write 0x00 (stop) to the I²C command register

Instead of the stop command another start can be issued (repeated start condition). If the device does not respond in step 3, the program may have to wait for devices to become ready again (for example an EEPROM busy executing its internal write cycle).

SPI

The SPI interface is master only, no interrupt capability. It shares its resources with the lower address bus (A7:0, port B) of the emulator. Therefore normal CPU processing is stopped while a byte is transferred on the SPI pins. The SPI interface is isolated from the address bus in normal operation. The MOSI, SCK and slave select signals are latched by a 74HC573 latch and MISO is disconnected by a 74HC(T)125 bus buffer.

The ATMEGA's SPI hardware is used and its valid control bits are merged into a single command register together with a slave select control to determine, whether slave select will be preserved between bytes. Refer also to the ATMEGA16 or ATMEGA32 manual for a detailed description of the SPI hardware bits.

Please note that during the transfer of a byte, emulation interrupts are delayed until after the transfer is completed.

The default SPI register definitions are as follows:

```
iomap   equ $bf00       ;IO page (A15:8)
spi_cmd equ iomap+$f6   ;spi command register, write only
spi_lrd equ iomap+$f6   ;spi last read data register, read only
spi_dat equ iomap+$90   ;spi data registers base, read/write
; SPI device list
spi_sd  equ spi_dat     ;SD card slot
spi_165 equ spi_dat+1   ;74HC165 parallel input shift register
spi_595 equ spi_dat+2   ;74HC595 parallel output shift register
spi_eep equ spi_dat+3   ;25LC512 EEPROM
```

SPI command register

The SPI command register is write only.

Bit	Name	Description
7	---	unused
6	SSHLD	Slave select hold
5	DORD	Data order
4	SPI2X	Double SPI speed
3	CPOL	Clock Polarity
2	CPHA	Clock Phase
1	SPR1	SPI clock rate select 1
0	SPR0	SPI clock rate select 0

SSHLD - slave select hold

- When the SSHLD bit is zero, the addressed slave will be selected only for one byte at a time.
- When the SSHLD bit is written to one, the addressed slave remains selected from the next data byte until the SSHLD bit is written to zero again.

DORD - data order

- When the DORD bit is zero, the most significant bit of the data byte will be transmitted first (shifting to the left).
- When the DORD bit is one, the least significant bit of the data byte will be transmitted first (shifting to the right).

CPOL - clock polarity, CPHA - clock phase

CPOL	CPHA	Leading Edge	Trailing Edge
0	0	Sample on rising edge	Shift on falling edge
0	1	Shift on rising edge	Sample on falling edge
1	0	Sample on falling edge	Shift on rising edge
1	1	Shift on falling edge	Sample on rising edge

Recommended CPOL/CPHA:

- 74HC595: 0/0 or 1/1, 8MHz
- 74HC165: 0/0 or 1/1 (not 1/0 or 0/1 because of the 4 gates SCK to MISO delay chain!), 8MHz
- 25LC512: 0/0 or 1/1, 8Mhz
- SD card slot: to be tested

SPI2X, SPR1, SPR0 - SPI clock rate

SPI2X	SPR1	SPR0	SCK frequency	@ 16MHz
0	0	0	$f_{osc}/4$	4 MHz
0	0	1	$f_{osc}/16$	1 MHz
0	1	0	$f_{osc}/64$	250 kHz
0	1	1	$f_{osc}/128$	125 kHz
1	0	0	$f_{osc}/2$	8 MHz
1	0	1	$f_{osc}/8$	2 MHz
1	1	0	$f_{osc}/32$	500 kHz
1	1	1	$f_{osc}/64$	250 kHz

SPI data registers

Accessing the SPI data registers initiates the data transfer with the addressed slave (lower 4 bits of the SPI data register address). When data is written to an SPI data register, the incoming data is temporarily stored in the SPI last read data register. When data is read from an SPI data register, a dummy byte of $FF is written to initiate the data transfer.

SPI last read data register

The SPI data register is read only. This register can be accessed after a write to any of the SPI slaves and contains the byte received from the slave during the last write. This allows the utilization of the full duplex capability of the SPI interface.

SPI IO sequence

Reading or writing to an SPI device

1. Write parameters to the SPI command register
2. Read data byte from or write data byte to the SPI data register for the desired slave
3. Repeat 2. for desired number of bytes
4. Write SSHLD=0 to the SPI command register

Step 1 (writing the SPI command register) can be omitted, if SSHLD should remain 0 and all other parameters remain unchanged. However, the command register should be written at least once after a reset.

Step 4 can be omitted, if SSHLD is already 0.

Read and write an SPI device simultaneously (full duplex)

1. Write parameters to the SPI command register
2. Write data byte to the SPI data register for the desired slave
3. Read data byte from the SPI last read data register
4. Repeat from 2. for desired number of bytes
5. Write SSHLD=0 to the SPI command register

DMA

Direct memory access is available for I²C, SPI and the emulator's non-volatile program storage serial EEPROM on I²C or SPI. The dedicated EEPROM is only accessible using the special DMA commands. It is not visible using the I²C or SPI registers. Same as with the individual byte transfers for I²C and SPI, the emulated CPU's processing is halted until the whole DMA block has been transferred. However, the DMA block transfer is interruptible between each transferred byte except when the dedicated EEPROM is accessed. The DMA transfer will commence after the interrupt service returns by RTI.

In addition to the I²C and SPI definitions, the default DMA registers are defined as follows:

```
dma_cmd equ iomap+$f7   ;dma command register, write only
dma_sta equ iomap+$f7   ;dma status register, read only
dma_dat equ iomap+$f8   ;dma data register, read/write
```

DMA command register

The DMA command register is write only.

Cmd.	Name	Description
0x00	Set SPI Set EEP	Set DMA parameters for SPI: slave address, RAM address, byte count Set EEPROM save or load parameters: program #, RAM address, byte count
0x01	Set I²C	Set DMA parameters for I²C: RAM address, byte count
0x02	Write SPI	Write a data block from RAM to an SPI slave
0x03	Read SPI	Read a data block from an SPI slave to RAM
0x04	Write I²C	Write a data block from RAM to an I²C slave
0x05	Read I²C	Read a data block from an I²C slave to RAM
0x06	Save EEP	Save a binary image from RAM to serial EEPROM
0x07	Load EEP	Load a binary image from serial EEPROM to RAM

DMA status register

The DMA status register is read only and in general contains the last command issued. For reads and writes 0x10 is added to signal command completion, otherwise the command is still in progress. Command in progress can only be seen from an interrupt service routine reading the status. The next instruction after writing the command register always sees completion. If a command could not be executed because of an invalid setup sequence, the status register contains 0xFF.

DMA data register

The DMA data register is used to setup the DMA parameters. No other DMA command is allowed until all required parameters have been set.

	SPI r/w	I²C r/w	save to EEPROM	load from EEPROM
command	0x00	0x01	0x00	0x00
data byte	slave address	---	program number	program number
data word	RAM start address	RAM start address	RAM start address	RAM start address
data word	byte count	byte count	RAM end address	---

When accessing I²C devices, the addressing of the I²C slave must be done before a DMA transfer can be initiated. When loading a program image from the serial EEPROM, the byte count is determined by the size of the stored image. The byte count can be read from the DMA data register after the program was loaded.

SPI & I²C IO

Standard IO and DMA IO can be mixed. For I²C, IO must be mixed as the start of an I²C device must be done with standard IO prior to moving data with the DMA feature.

DMA setup

1. Write 0x00 or 0x01 (Set SPI/ I²C) to the DMA control register
2. Only SPI: Write slave address to the DMA data register
3. Write start address low byte to the DMA data register
4. Write start address high byte to the DMA data register
5. Write byte count low byte to the DMA data register
6. Write byte count high byte to the DMA data register

DMA read or write commands will fail, if the above sequence is incomplete or omitted.

Access SPI

1. Write parameters to the SPI command register
2. Perform setup DMA with command 0x00
3. Write 0x02 or 0x03 to the DMA command register to write or read a data block
4. Repeat 3. for desired number of blocks with the same count and incrementing RAM address or repeat from 2. to set new DMA parameters for the next block
5. Write SSHLD=0 to the SPI command register

At any time you can read or write single bytes before, between and after DMA blocks by accessing the SPI data register.

Access I²C

1. Write 0x01 or 0x02 (start 100/400 kHz) to the I²C command register
2. Write I²C device address (+1 if read) to the I²C data register
3. Read the I²C status register - verify ack received, device responding
4. Perform DMA setup with command 0x01 skipping step 2 of the DMA setup.
5. Write 0x04 or 0x05 to the DMA command register to write or read a data block
6. Read only: Read the I²C status register - verify ack received, device accepts more data
7. Repeat 5. for desired number of blocks with the same count and incrementing RAM address or repeat from 4. to set new DMA parameters for the next block
8. Write 0x00 (stop) to the I²C command register

At any time you can read or write single bytes before, between and after DMA blocks by accessing the I²C data register.

serial EEPROM IO sequence

A program like EhBASIC running inside the emulator may also use the attached serial EEPROM as non volatile program storage. Messages regarding success of the DMA command will be posted directly to RS232 output. The status register is set to command + 0x10 if the load or save succeeded, 0xFF otherwise.

You should use a designated subset of program numbers to identify whether a program is a binary image of an EhBASIC program. Trying to load a program saved for the emulator itself will result in an "incompatible format" error message.

Save to non volatile program storage

1. Write 0x00 (set EEP) to the DMA command register
2. Write program number to the DMA data register
3. Write start address low byte to the DMA data register
4. Write start address high byte to the DMA data register
5. Write end address low byte to the DMA data register
6. Write end address high byte to the DMA data register
7. Write 0x06 (save EEP) to the DMA command register

A save command will fail, if the required 5 data bytes (program number, start address & end address) have not been set.

Load from non volatile program storage

1. Write 0x00 (set EEP) to the DMA command register
2. Write program number to the DMA data register
3. Write start address low byte to the DMA data register
4. Write start address high byte to the DMA data register
5. Write 0x07 (load EEP) to the DMA command register
6. Read the DMA data register, end address low byte
7. Read the DMA data register, end address high byte

A load command will fail, if the required 3 data bytes (prog number & start address) have not been set.

Show directory, delete program, set autoload at power on

Use the monitors EEPROM utility to perform these functions. The monitor can be invoked from any application by writing a one into bit 7 of the diag register. To return to your program type X <CR> at the monitors prompt.

Diagnostic register

The diagnostic register has limited practical use for programs on the emulator. You can bring up the debugger/monitor or query whether it is active and whether the version of the IO module is DMA capable. All other bits are only available, if they are enabled in the source code configuration.

The default diagnostic register definition is as follows:

```
iomap   equ $bf00       ;IO page (A15:8)
emu_dig equ iomap+$fc   ;diagnostic emulator flags, read/write
```

Bit	Name	Write	Read
7	DDEB	Start debugger/monitor	Debugger/Monitor is running
6	DDMA	---	Has DMA (IO version >= 0.83)
5		---	0
4		---	0
3		---	0
2	*DRES*	*Force reset*	*0*
1	*DNMI*	*Force NMI*	*Forced NMI pending*
0	*DIRQ*	*Force IRQ*	*Forced IRQ pending*

Debugger/Monitor command reference

Connect a terminal

You can connect to the debugger/monitor through a serial RS232 connection set to 38400 Baud with 8 data bits, no parity and 1 stop bit. The terminal or terminal emulation should understand escape sequences such as reverse video and save & restore cursor position. The Baud rate and other parameters of the USART may be changed in the source code of the emulator.

I am using PuTTY and TeraTerm and both work well for me. I haven't tested any other terminal or terminal emulation, so it may or may not work with your favorite option.

Start the debugger/monitor

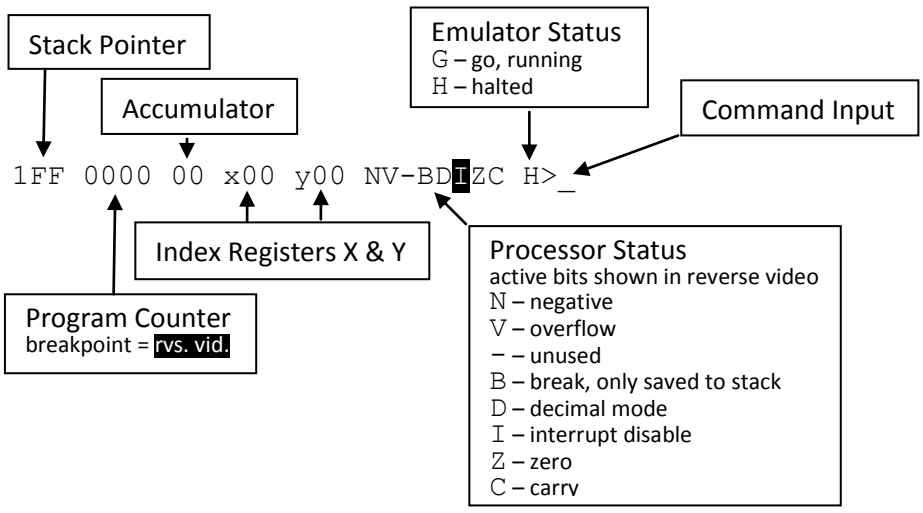
Either a manual reset (external pushbutton reset) or sending a break from the terminal brings up the prompt. At the same time the currently running program is halted.

In addition, a reset loads the reset vector into the program counter. For debugging purposes, the register content at reset time is shown. Since instruction execution might have been in progress, the registers may not have been completely written by the ongoing instruction, even if the PC already points to the next instruction.

Power on or brown out may also get you to the prompt. However, the monitor may be configured to autoload and run a program from an attached I²C EEPROM.

In PuTTY you can send a break by right clicking on the window title bar, select special command, break.

The prompt



When a program is running, the register display is updated every ½ second.

Edit the commandline

All input is converted to upper case except when enclosed in double quotes. The commandline can hold a maximum of 48 characters. The following keys have special functions:

- <CR> enters the commandline
- <BS> deletes the last character
- <ESC> clears the commandline and redraws the prompt

If the monitor rejects the command just entered, it places a question mark after the first character it doesn't understand. The rest of the line is cleared. You can then continue to edit the same command.

Load a program

Format: `L` no parameters

The program must be in Intel hex format and only accepts function 0 and 1 records. After typing `L<CR>` the monitor responds:

```
1FF 0000 00 x00 y00 NV-BDIZC H>L
Loading, <ESC> to abort
```

Now the Intel hex file must be transferred to the monitor. In PuTTY you can open the file in an editor, select all content and copy to clipboard (<CTRL>A, <CTRL>C). Then just right click inside the terminal window.

While loading the program, you will see the load address being displayed briefly. At the end you should see: `Load OK` If the closing record contained an address higher than page zero the address is loaded into the program counter.

You may also see an error message, if there is a problem with the Intel hex format of the file you are trying to load. The error messages are self explanatory.

Reset the emulator

Format: `R` no parameters

If the program you just loaded has a reset vector, the reset address of the program is transferred to the program counter by a reset. In addition, the reset command clears all registers, sets interrupt disable and performs a reset of external hardware as defined in the file 6502_Emu_IO.asm.

Start and stop a program (go & halt)

Format: `G` no parameters

Format: `H` no parameters

In order to run a program after its start address has been put into the program counter or to continue a program after it has been halted enter `G` (go). To stop the currently running program enter `H` (halt).

Exit the monitor

Format: `X` no parameters

If the program you want to run requires access to the RS232 interface, you must exit the monitor. Exit will also start the program if it is not already started.

Alter a register

Format: `Arbb` or `APwwww` `r` = register, `bb` = byte, `wwww` = word

Byte or word parameters can be separated with spaces (`AA FF<CR>` to alter the accumulator to FF). Bytes must be entered as two hex digits while words are automatically filled with leading zeros (`AP1<CR>` loads the PC with 0001).

Valid registers are:

R	Register	parameter
P	program counter	word
A	Accumulator	byte
X	index register X	byte
Y	index register Y	byte
F	processor status (flags)	byte
S	stack pointer	byte

Display memory

Format:

	`Mwwww`	`wwww` = RAM address
	`M`	display same address again
	`M+`	display next
	`M-`	display previous

The memory display command shows 256 bytes of data including the selected address in the first line of output. For practical reasons, the display begins with a location dividable by hex 20. IO addresses are never shown or changed. A shadow address of RAM appears instead.

Hit `<CR>` to display more.

Write memory

Format: `Wwwww bb…bb` `wwww` = RAM address, `bb…bb` = bytes of data

In this case, the address word must be separated from the data bytes by one or more spaces. Bytes may be separated by spaces but don't have to be. However, they must have an even number of digits.

Type `M<CR>` to see the changes you just made.

Display code (disassemble)

Format:

	`Dwwww`	`wwww` = RAM address
	`D+`	disassemble next

Disassembles 10 opcodes. Hit `<CR>` to display more.

Single step

Format: `<TAB>`

For this function to work, the emulator must be in a halted or breakpoint state. Every time you hit the `<TAB>` key, 1 instruction is executed and shown. Hold down `<TAB>` for continuous execution.

```
1FC 3DB1 80 x58 yFF NV-BDIZC LDA 57
1FC 3DB3 B1 x58 yFF NV-BDIZC SBC 0201,X
1FC 3DB6 D1 x58 yFF NV-BDIZC H>
```

In this example you can see the LDA 57 at 3DB1 has loaded B1 into the accumulator. Then the SBC 201,X at 3DB3 has subtracted the byte in 0259 from it and the result was D1 with overflow cleared. The next instruction to be executed is at 3DB6.

If you have configured the interrupt mode to be real (.equ irq_dis_real = 1) in the source code of the emulator, single step will not show portions of the program that are running with the interrupt disable bit set in the processor status register.

Breakpoints

Up to ten breakpoints can be set. The emulator will stop and invoke the monitor, if an opcode is about to be executed at one of the breakpoint addresses. The register display will show the program counter in reverse video to indicate a breakpoint stop. To continue, you may use the single step key <TAB>, the start (G) or the exit (X) command.

Breakpoints interact with other commands as follows:

- The memory display (M) command will show a breakpoint byte location in reverse video.
- The disassemble (D) command will show the opcode's mnemonic in reverse video.
- All load (L, EL) commands will automatically clear all breakpoints.

Caution: Under the hood the breakpoint utility uses the 0xdb opcode (STP on later 65C02s). Activating a breakpoint for a non opcode location like byte 2 or 3 of an instruction or for data may lead to data corruption or unexpected behavior of the program.

Set a breakpoint

Format: `BSaaaa` `aaaa` = breakpoint address

Breakpoint addresses must be chosen with care. See caution above.

Breakpoint information

Format: `BI`

Lists active breakpoints by the slot number they occupy.

Clear breakpoints

Format: `BCs` `s` = slot number
`BCA` clear all slots

Clears the specified breakpoint slots.

EEPROM utility

These commands are only available, if the I²C or SPI bus and an I²C or SPI EEPROM are present and configured in the emulators' assembler source. It allows a 64kB I²C or SPI EEPROM to be used as non volatile program storage.

Programs in the EEPROM are identified by a hex number from 00 to FE. FF is a special number and means none or free. The EEPROM is divided into 256 slots of 256 bytes each. A slot allocation table is stored in the AVRs' internal EEPROM.

With a record size of 32 bytes per record in the Intel hex output, the size of a program in EEPROM is about 10% larger than the actual number of bytes in the program. Unlike a binary image, empty space as defined in the source does not occupy space in the EEPROM.

EEPROM save

Format: `ESbb` `bb` = program number

It works pretty much the same way as the load function of the monitor. After starting the command, you get: `Saving, <ESC> to abort` Copy the Intel hex formatted assembler output to the clipboard and drop it onto the terminal window. At the end, you should see: `Save OK` If not, you should get a meaningful message about what the problem is.

Saving to EEPROM with an existing program number replaces the program on the EEPROM. It is an in place replacement and therefore does only need extra free slots if the new version is bigger. But be aware, that if the save is aborted due to an error, neither the old nor the new version of the program remain on the EEPROM.

EEPROM load

Format: `ELbb` `bb` = program number

This loads the program with the number you specified from EEPROM to RAM. The program counter can be set either from the closing records' address if valid, by a reset from the reset vector or by altering the PC manually. Then enter go or exit to start the program.

If the program was not saved with the ES command (saved from EhBasic for example), the load is rejected with an "incompatible format" error message.

EEPROM delete

Format: `EDbb` `bb` = program number

Of course, you can delete programs from the EEPROM.

EEPROM autoload

Format: `EAbb"abc"` `bb` = program number, `FF` = none
`"abc"` = initial string for program

Setting the autoload option to a valid (existing) program number will configure the monitor to automatically load this program during power on. If the program has a start address in the reset vector, the program will be started and the monitor will exit. To disable the autoload option, enter FF as the program number.

An initial string can be sent to the autoloaded program. It must be enclosed in double quotes. The backslash is an escape character allowing entry of special characters:

- \\ single backslash
- \r carriage return
- \q double quote

Enter an empty string (“”) to disable this option.

You may specify each parameter individually or combined in a single command. An example for EhBASIC:

```
1FE 3AEB 80 x58 yFF NV-BDIZC H>EA00”C\rLOAD $80\rRUN\r”
```

This will automatically load program 00 (EhBASIC) into the emulation on power on, then command EhBASIC to coldstart, use default memory size, load program 0x80 and run it.

EEPROM information

Format: `EI` no parameters

Shows stored programs, free slots and the status of the autoload option. You get:

```
1FE 3AEB 80 x58 yFF NV-BDIZC H>EI
EEPROM autoload none - CB slots free
prog#/slots  00/35
1FE 3AEB 80 x58 yFF NV-BDIZC H>
```

You have to figure out yourself, what the number of slots means in bytes. Hint: a slot is 256 bytes (at least for now) and the slots count in hexadecimal numbers.