Description of the Test Software
Microcontrollers project, FILS

AT Mega 16 7.3728MHz

The purpose of the test software:
- to test your newly assembled board
- to show you (by examining the source) how the functions specific to a microcontroller are implemented in this particular combination of uC and compiler: how to set a pin as input or output, how to read and write binary values to a pin, how to use the timers and watchdog, how to read/write using the serial port. Also, a "project" (containing .c, .h and specific files) for the CodeVisionAVR compiler is demonstrated. Students are advised to try and modify this software first, before trying to create an entirely new project from scratch.

The test software has the following functions:

- when a character is received on the serial port (9600,8,N,1), the character code +1 is sent back (a becomes b, etc)
- when the "?" character is received, the version number is sent back
- the LED at port D.6 blinks using the timer 1
- at each press of the key at port D.5, the blinking frequency is changed between 1Hz and 4Hz.

The project file (.prj)

In the menu Project->Configure->C Compiler there are specific values to be set according to our board, such as, the quartz frequency which is needed to calculate delay values, and the options supported by the printf() and scanf() functions: since we are compiling for a uC with a very limited amount of Flash and RAM, some options which are standard in a x86 compiler can be omitted; for instance, if we select only "int, width, precision", then the long and float types will not be printed (%f and %l modifiers in the format string of printf() will simply not print anything).

Other files

Large projects should be partitioned in multiple files, grouping functions based on their purpose. It is a good idea to prefix the name of a function with the name of the file in which it is defined; for instance, the function Init_initController() is in init.c. Functions without prefixes are in main.c.

func.h

This file holds function prototypes (the standard way of defining what the parameters of a function are in C) for all functions defined in all files outside main.c, and also **extern** declarations of global variables used outside main.c. Failure to define them here will generate an "undefined" error in the compiler.

defs.h

This file is #included by the main program and by other files. Some numerical constants are defined here, also the names of the devices using a certain port (so we can use the name LED instead of PORTD.6, and so on). The version number of the program is also defined here.
Note that #defines do not occupy memory, they are just replacements that the compiler use in a special phase before the actual compilation (called preprocessing), so, if something is defined and not used, it is simply ignored.

main.c

Lines 1-15
Header files are included here. Between <> are system headers, which can be found in the CodeVision installation folder, and between "" are local headers defined by the programmer and located in the current folder. Don't forget to include the mega16.h header (or the one for your particular processor) where the definitions of all processor-specific registers can be found – this is how the compiler knows which particular AVR processor you use; a different AVR processor may have less registers, for instance only PORTA and PORTB, or a different number of timers, etc.

Lines 16-127
This relatively long section is automatically generated by CodeWizard and so it can be ignored; in short, it defines the interrupt-driven serial functions putchar() and getchar(); all other higher level functions like printf() depend on these 2 functions.

Lines 130-136
The timer1 interrupt service routine (ISR) is defined (Timer1 registers are initialized in init.c). When the event "output compare match A" happens, i.e. when the timer's counter reaches the value OCR1A, this interrupt is called. In our case, the sole instruction performed is to invert the state of the LED. The effect is that the LED blinks with a period determined by the value in OCR1A multiplied by the timer's period, which in turn is the main clock period multiplied by the prescaler value. See more in the documentation about a timer's operation.

Lines 138-150
Beginning of the main() function; some initializations must be done here, such as enabling interrupts, initializing default values of some registers, etc (by calling init_initController()) a.s.o.
After initialization, an infinite loop (while(1)) is entered; this is one difference between an "usual" C program written for Windows/Linux, which normally will end at some point, and a microcontroller program, which cannot "end" since there is no operating system to take over. So our program must run forever.

Line 152
The *wdr* instruction is called (using the #define wdogtrig() in defs.h), which means *watchdog reset*. The watchdog, common in many microcontroller applications, must be reset every 2 seconds at most (or some other programmable interval) by issuing this instruction at least that often. If the software fails to reset the watchdog, it is assumed that the software has "locked up", either because of a bug or because of a glitch on the power supply which can corrupt program/data memory, and the watchdog, which runs independently inside a protected circuit of the processor, will issue a processor reset. So, no program can "lock up" longer than 2 seconds.
Take care while waiting in loops, for instance when waiting for input for the user, when using delays, etc. – you must still reset the watchdog or it will reset the processor! (in some special cases, you may decide to disable the watchdog; see the datasheet for how this can be done).

Lines 153-160
The *rx_counter* variable holds the number of serial characters available (and not yet read by the user program). The getchar() function reads a character in the variable *temp*, to be processed and analyzed later. Every call of getchar() returns a different character (if available), you must store the received characters, you cannot read the same character more than once with getchar() !
If the "?" character is received, the version number is printed, else the next character is issued. It is a good idea for any program to define a way to retrieve the version number using a particular serial command; this way, you know if you have loaded the correct version.

Lines 162-167
Reading a key (which is as simple as comparing with 0 the value from PINX.Y) is here *debounced*. Debouncing is a specific term which means – preventing multiple reads because a contact that "bounces". Since the processor runs at 7MHz, very short openings and closing of the contact, which can happen at the microsecond level, caused by imperfections of the electromechanical parts in the contact, can be interpreted as multiple keypresses. Thus, we read the value, wait for a short time, and read it again – only if it's still 0 we consider this a valid keypress.
Also, we use another *while* loop, for waiting for the user to release the key, similarly to the way mouse clicks usually work – the action is performed when the key is released, not at the initial press.

Lines 168-171
Each keypress changes between 2 sets of values for the Timer1 registers – the values in init.c and the same values divided by 4. The effect is that the timer period alternates between 1 second and 1/4 seconds.

init.c

Most of the code generated by CodeWizard was manually moved here for clarity (normally, CodeWizard generates code in a single .c file).

Lines 12-30
For each port the initial value (PORTx) and direction (DDRx) are set. See the documentation for the meaning of these registers. We must observe the schematic when initializing ports, in our case, the key must be defined as input, and the LED as output. Also, we must activate the pull-up resistor on the input (by writing 1 to the respective PORT bit). Unused ports can safely be left as inputs (the default value).
Note that you can either read/write a single bit or all 8 bits of the port at once. Some examples:

`PORTA.5 = 1` sets bit 5 to 1
`PORTA |= 0b00100000` does the same, since the "OR" operation can only set to 1, not clear to 0, a certain bit; performing OR with 0 has no effect, thus leaving the rest of the bits unchanged.

`PORTA.5 = 0` clears bit 5 to 0
`PORTA &= 0b11011111` does the same, since the AND operation with 0 clears a bit to 0; performing AND with 1 has no effect.

`if(PIND && 0b00100000)` …. this condition is true if PIND.5 is 1, regardless of the other bits of the port. Note that for the same port, some pins may be used as inputs and others for outputs, without any interference between them. Remember: logical AND is "&&", arithmetic AND is "&".

Note that, if reading a port, the compiler will *not* generate an error if you mistakenly try to read value PORTX.Y instead of PINX.Y – this is because it's perfectly allowed to read the "set" value of the port, but the result is the value you have set yourself, not the value applied externally by the key. So, this will return a constant value and will not detect actual keypresses.

Lines 31-76 and 87
Timers 0,1,2 are initialized here, see additional info in the documentation for how to compute the values of the timer registers, and what means each. In this case, timer 1 is used in CTC mode and the timer interrupt is enabled (the actual interrupt routine must be then explicitly written, this happens in main.c).

Lines 89-99
The USART port is initialized in Asynchronous mode with the communication parameters 9600 bps, 8 data bits, 1 stop bit, no parity (9600,8,N,1). The computations are performed using Code Wizard, once the exact quartz frequency is entered in the Project Properties. Changing the quartz to a different frequency without also changing this part will cause the serial port to be unable to communicate with the PC.

Lines 107-112
The watchdog timer is initialized for a 2 second watchdog interval. The watchdog register is written twice with 2 different values; this is not an error, it is a special access mode which is guaranteed not to happen by chance. Then, in the main program, you *must* call the wdogtrig() instruction no later than 2 seconds, or the processor will reset.

What about the bootloader?

Everything described above refers to the application program. The bootloader program is resident in Flash memory (it is programmed separately with a special programmer) and is given control when the processor is reset (no matter what the reset cause is). Then, the bootloader checks if the key at Pin D.5 is pressed and if so, communication is initiated via the serial port; after a the program transfer ends, the bootloader jumps to the beginning of the application program, thus ending its control over the processor, unlike an operating system which is always at last partly in control. So, the resources (in particular, the pin D.5 and the serial port) are not shared between the bootloader and the application, which means the application has complete control over them when it runs, while the bootloader has the same when it's its turn to run.
If the key is not pressed at the time of reset, the bootloader detects this and jumps to the application immediately.

Note that the bootloader uses special instructions (available only as machine opcodes) in order to write to the Flash memory, so a "normal" program written in C cannot overwrite itself and also cannot overwrite the bootloader. Thus, no matter what you do in your C program and what you send on the serial port, you can always load another application using the bootloader. In rare cases, if the bootloader tries and fails to load an application, the application memory may become corrupted and it will appear that the processor does nothing. The solution is simply to try and reload the application – try with a "known good" application such as this test software, if you ever suspect the processor has gone bad.