Title:	An embedded systems Simple Operating System
Author:	grilialex
Language:	C
Platform:	Windows, GCC, AVR, Xilinx
Technology:	OS, Embedded Systems, Hardware programming
Level:	Intermediate
Description:	A simple O/S for embedded systems for rapid application
Section	Hardware
SubSection	Hardware Programming
License:	CDDL

Summary

An Embedded System simple Operating System Framework that allows rapid development of applications build for AVR family but can be ported to other architectures easily enough. In this article I will describe the concepts and the structure of this OS and also I will provide an example application in order for the reader to understand the simplicity of building new stuff easily.

Introduction

Embedded Systems is a very interesting field. You can do things using hardware and software that everybody gets impressed when they see them, unless the beauty is well hidden. Along with their problems, limitations and special requirements, if you repetitively build such systems, you have some common denominator on each design. I always used a system tick timer and a Uart for example. This is why I choose to build an OS core framework that would allow me to build faster applications without being too complex. This OS does not do pre-emptive multitasking. Instead it is a round-robin co-operative system ie. Each task either does something or exits if it is waiting new data (no blocking). It is very simple with very low memory footprint and also you can remove or add components very easily, ready to use it in your next project. With this OS in hand I could develop my small applications very fast because when I already had the basics ready to run the only piece missing was my pure application: What I needed to do, which many times was a one or two pages program. I could write it and have it up and running in a few days…

Background

When I started to work with embedded systems I wrote assembly language and we had microcontrollers with EPROM for program memory and very little RAM to use of. In every project I almost had to use a system tick timer and not a few times a UART to communicate with the host PC. Around the end of the 90's with the introduction of AVR I switched to ISP (In-System Programming). No more bulky EPROM erasers, easy programming in a few seconds etc. However back then, memory continued to be limited. During my porting of 8051 codes to AVR I needed a few things. A system tick timer and a Uart. My next problem was debugging. Although there was an AVR simulator from ATMEL I could not test my application when I had to take inputs from external environment. So in order to debug more easily I build a very small footprint monitor: just read/write ports, memory and exercise external peripherals. This debugger was an integral part of any new build. Later on I moved to C and thus I re-write once more most of my code to C. Also as I added more and more peripherals and program memory was precious I began configuring my tiny kernel to add or remove components with #defines. The end result was to have a platform that allowed me to build fast my applications. I just have all the infrastructure support and I was focused on building the actual application. I only used simulator for particular pieces of codes (more to see what the actual C statement do). During debugging I use my monitor, printf to serial port and of course multimeters, oscilloscopes and … LEDs(!) from the hardware side if I need to.

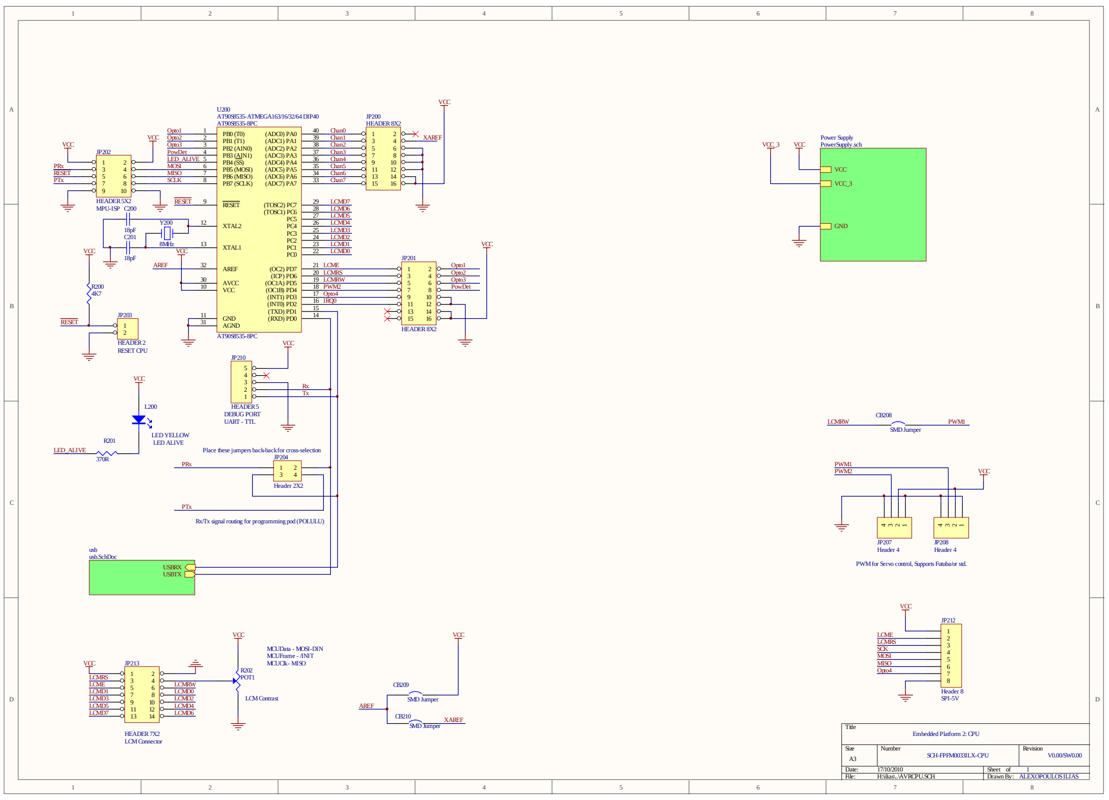
Also as I am using GCC for compiling I do not use an IDE for AVR development. I have make files for building and configuring my builds and I use my favorite editor to do my coding. So this framework can be ported to any microcontroller theoretically (ie. PIC, ARM etc). In fact I have ported variants to ARM and ColdFire processors/controllers. Of course you are free to use your favorite IDE.

As memory was limited and I did not need the complexity of pre-emptive multitasking the philosophy of this OS is that each task checks if it has any input to process, if there is something to do it just executes otherwise it returns to the round-robin main. The advantage is that this is very scalable and you don't worry for complex things, it is very efficient for RAM and also there is no context switch so you save execution time. The drawback of course is that the worst case execution (all tasks execute) should be small enough (preferably less than the system tick period), but this depend from your application! You might be able to break this rule for once in a while. However I am avoiding doing that and I believe for most projects the timing of this loop should not be problem.

Description

<u>Aim of the project</u>

Aim of the project was to have a platform that was scalable and allowed rapid application development. I finally wrote it in C so it is portable in a way. You have to write the main peripherals for each new processor which is the main pain. However after having the core up and running you can benefit from this structure to have it as a base project where you can build your new applications. As I build this for AVR initially I name it “AVRILOS”: AVR ILias Operating System. I am assuming that you have an AVR hardware ready. For your reference and because various I/O are mapped to specific ports (although you can easily change them) I provide my basic schematic which again is replicated (like AVRILOS) over projects more or less with additions for my particular problem.



Tools

The tools I use for AVRILOS are:

1. (SW) WinAVR (AVR GCC for windows).
2. (SW) Atmel AVR Studio (for Simulation).
3. (SW) Your preferable Editor.
4. (SW) Terminal Program (ie. Terminal, PuTTY).
5. (SW) Programmer software (AVRDude is already included in WinAVR but you might use AVREAL32 if you want).
6. (HW) Hardware Board where you controller lives!
7. (HW) Programmer Dongle.
8. (HW) USB/RS232-TTL Serial Level Converter for connecting to Monitor.

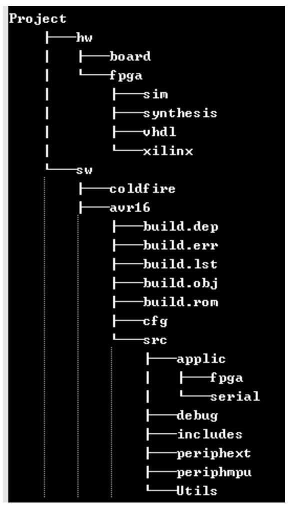
Optionally I have used some extra tools:

1. (SW) CVTFPGA (for integrating serial bitstreams of Xilinx Spartan FPGAs to my code, more on this later)
2. (SW) Hexbin3
3. GNUWIN32 (for makefiles if I don't use the WinAVR ie. Other compiler packages like MPLAB)
4. (SW) Python and Python Wx for building host applications.
5. (HW) Oscilloscope (recommended)
6. (HW) Multimeter (nothing less than that!)
7. Anything else you can imagine and fits.

AVRILOS

Directory Structure

The directory structure is as follows:



```
Project
    ├──hw
    │   ├──board
    │   └──fpga
    │       ├──sim
    │       ├──synthesis
    │       ├──vhdl
    │       └──xilinx
    └──sw
        ├──coldfire
        ├──avr16
            ├──build.dep
            ├──build.err
            ├──build.lst
            ├──build.obj
            ├──build.rom
            ├──cfg
            └──src
                ├──applic
                │   ├──fpga
                │   └──serial
                ├──debug
                ├──includes
                ├──periphext
                ├──periphmpu
                └──Utils
```

There are two major directories, HW and SW.

1. HW is the directory where all my hardware development is done. This includes board schematic & PCB files as well as FPGA design.
2. SW is the software directory which contains directory name of the processor used, so I know which processor is used in each project after many years. Also I might put here host software (in another directory called host).

Let’s concentrate on this directory structure of avr16. AVR16 refers to ATMega16 AVR. You could name it as you wish. Because the compilers have a tendency to generate many intermediate files I don’t want these to interfere with my source files. So I have included three directories specifically for this purpose:

1. build.dep: Here the dependencies of the C files are placed. These are auto-generated by the makefile.
2. build.err: Here I instruct the compiler to put any error files in order to track them if I need them.
3. build.obj: Here I place the object files of each module. Also the final .elf file. From the MAP file I can find the memory location of any variable and use it in my monitor for inspection.
4. build.lst: Listing file of each module.
5. build.rom: final programming files ready to be used for the device programming.

The root makefile (named makefile) resides on the root of avr16.
The cfg directory contains all the satellite makefiles. These files contains the configuration options, compiler commands, etc.

The src directory is what you are expecting for. The source files. There we have:

1. Applic: The main file Kernel.c which contains the “scheduler”. Initialization and main loop are here. Also my specific application c files are placed here as well.
2. peripheint: Internal peripherals of the microcontrollers. These are timer, uart etc.
3. periphext: External peripherals outside of the microcontroller. These might be smart cards, LPC flash, SPI devices etc.
4. utils: Contains many type conversion tools (hex2bin, bin2hex), delays etc. Of course you might use the stdlib sprintf instead if you have enough memory space available.
5. debug: Here I have my monitor debugger and in addition an extended debug file which I have functions individual enabled or disabled for external peripherals that I might use. If I don’t use them then I just disable the corresponding functions and I save some memory space.
6. include: Here I have global definitions and settings. Also pin/port allocation for each peripheral used if this is programmable.

Description of Kernel

The Kernel.c contains the initialization code and the main loop. During then startup the Kernel executes the various initializations of each module/peripheral. There is no one initialization file for all modules. Instead each module modifies its own bits in its I/O registers with Read & Modify instructions. This way each peripheral does not interferes with the others unless there is a port conflict. This allows more modular design and easy addition/removal of modules.

The scheduler does a round-robin execution of the tasks we need to. Each task checks if should be triggered due to some flag like the SysTick timer (In periphint/Timer0.c):

if ((v_SysStat & (1 << b_SysTick)) != 0)

Checks to see that the timer interrupt have flagged b_SysTick (Bit) in v_SysStat (Variable), if no just exits. If yes executes all its timer functions it needs to. Another case would be to test if there are any new data in the serial port (like the applic/serial/SerApp.c) does and exit if nothing is there.

The modules that I almost always have here are SysTick (performs software timers for mS, Led Alive etc), debugger (monitor for debugging), SysADC which captures all 8 channels in sequence and thus the application just read the memory locations for ADC data, SerApp which is a small serial command application. Also we have the capability to run the application either on every main loop or we might select to run the application every n-times (which is when the LED flashes):

if ((v_SysStat & (1 << b_AppTick)) != 0)

This allows the application to use a simple counter as delays or timeouts because the action of modifying (increment/decrement) is done every SysTick. Of course with a slight modification you can have a different pace for LedAlive and the application.

Finally for low power application I have included a sleep command at the end of the loop. If there is no interrupt active the system will go to low power mode until an event happens.

So in order to make a new task (application, device etc) you need to know that the new element should not block the system waiting for too-long. Specifically the execution delay of all the functions should not exceed the SysTick timing. If it does you have too alternatives: Either increase the Systick interval or reduce the blocking time.

In my experience I haven’t had the need to trim these timing for any of my projects until now.

Another concept I am using is that my interrupt routines are minimal. For example the timer0 interrupt just sets a flag and exits. The main loop will execute the SysTick (deferred handler) which in turn will do all the hard work. Of course the interrupts might be more complex like the serial interrupt which puts the data in the FIFO. But the idea is to avoid any major processing at the interrupt level. Thus the possibility of interrupts blocking will be minimal.

Also I use simple producer-consumer communication between interrupts and deferred handlers. I check that each variable modification in the background is not affected by the interrupt with atomic operations or by unidirectional actions (write/read-only variables). These will be visible in the Uart module.

Description of Modules

Module: SysTick

The SysTick module does all the major timing functions. It flashes the Alive LED, it triggers the ADC for starting a new conversion, triggers the LCM end of delay, triggers the keyboard scan function and also has the software timers.

These triggers are simple flags (bits) residing on v_SysStat. You can change easily the Application time interval by modifying includes/ifc_time.h constants in the beginning of the file:

```
// Alive Led Indications Set
#define   c_ALIVEOK_ms      250   /* Alive LED When Ok */
#define   c_ALIVESER_ms      500   /* Alive LED When Serial Error */

// Timed Tasks interval
#define c_AppInterval_ms       8
#define c_ADCInterval_ms       32
#define c_KEYInterval_ms       16
```

The corresponding activation bit definitions are stated in includes/settings.h:

```
/*************** SysStat Register ******************/
#define b_SysTick                              0
#define b_AppTick                              2
#define b_ADCTick                                        3
#define b_LedAlive                             5
```

I have omitted the flags that are not activated by the Systick timer. Clearing of the flags is done by the Kernel.c main loop. Keyboard scanning and LCD actions are directly called by the Systick so there are no flags in v_SysStat for these.

As these constants are referred in millisecond units you can change them at your desire. The CPU clock frequency is defined in the cfg/hw.in makefile and all timings should immediately comply, unless you need a different prescaler (set later on same file as CLK/64).

```
/* Select Clock Source for T0 */
#define c_T0CLK   c_CLK64
#define     c_T0DIV                    64
```

Also you can provide dynamic error indications using the f_SystickSetErrLevel function. This function modifies the flash LED (Alive) interval (in Ticks) so you can notify the user that something is wrong/different changing the flash interval of the LED Alive. For example in case of serial communication error you can call from your application:

```
f_SystickSetErrLevel(c_ALIVESER_ticks);
```

c_ALIVESER_ticks is defined later on include/ifc_time.h and is derived by the corresponding constant ALIVESER_ms in the beginning of this same file.

Now what happens when do you need a timer interval to do general timeouts or what ever? Easy! The SysTick provides a programmable (MAXSWTIMERS) number of soft-timers plus some additional definitions for easy reference:

```
// Maximum SW Timers (mS)
#define c_MAXSWTIMERS4
#define c_SwTimer0     0   /* Timer Activation (0: Stop, >0: Run) */
#define c_SwTimer1     1
#define c_SwTimer2     2
#define c_SwTimer3     3
```

These counters count in mS. In order to start a timer you write:

```
buf_SwTimer[SwTimer0] = 10;
```

This starts SW timer 0 with a timeout value of 10 mS.

To see if the timer expired:

```
If (buf_SwTimer[SwTimer0] == 0) Action_Timer0_finished();
```

To prematurely stop the timer:

```
buf_SwTimer[SwTimer0] = 0;
```

Additionally you may need small delay functions (just a few microseconds). In these cases you can find some blocking functions in utils/delay.c. These can be used during initialization where you waiting for a peripheral to startup. For example I use such a delay during FPGA code configuration at startup. Or alternatively you can use a microsecond delay on an SPI component.

Module: Uart

The UART is my favorite peripheral. As most of my applications do not directly need the UART, in my hardware I do not add the level shifter from TTL-RS232 level to connect to my PC. Given that many times I hand made my boards I do not like to spend components, wires and time for functionality that is useful only during development (for my monitor or application that is). So what I do, is just put on a standard (my standard of course) pin-header the Tx/Rx signals along with power and ground and I use an adapter (dongle) to connect to my PC. This dongle could be a simple level shifter (and that’s why I need power at the pin-header) or now where every computer uses USB ports I use a USB-serial converter (for example my Polulu AVR programmer provides this additional functionality). Now every board has this pin-header with 3-4 wires and I am done.

The UART has simple I/O functionality. For initialization you call the f_ConfigSerial. Baudrate is defined in the makefile (hw.in). The rest of the settings are standard 8N1. If you need to change them you have to change the source of this function in uart.c.

For the hardware level we support a number of processors and we use software FIFO for both Tx/Rx. The size of the FIFO is defined in includes/settings.h:

```
/*************** UART Buffer Sizes ******************/
#define c_RXBUFLEN        16 // 16
#define c_TXBUFLEN        64  // 16
```

As you see the Tx FIFO is larger. I do this in order to be able to respond with a large Tx string from my application without blocking the system. If the FIFO was smaller than my larger string sent to the host then the Put String function would wait (and block there) until it could write all the data to the FIFO.

The main interface functions are:

```
bool f_Uart_PutChar(INT8 c);
INT16  f_Uart_GetChar(void);
bool f_Uart_PutStr(INT8 s[]);
```

f_Uart_PutChar: Put a character to Tx in the TxFIFO unless the FIFO is full. Return code signifies that (1: success, 0: Fail). It does not block the system.

f_Uart_GetChar: Checks if a character is available and if yes it removes it from the Rx FIFO and returns it to the application (-1 [0xFFFF]: Fail, 0x00XX Character received). It does not block the system.

f_Uart_PutStr: Send a null terminated string to serial port. Always successful. This function may block the system if the Tx FIFO is smaller than the string.

If you need a printf function you can use an sprintf function to a buffer that would be sent to f_Uart_PutStr. Or you can build your own printf function as well.

In order to minimize footprint I use simple conversion functions that resides in utils/typeconv.c. A

Module: Debugger

Although I refer to the "debugger" in this article what I actually mean is a monitor. The debugger/monitor I have built does not take execution control, does not step instructions or does any other fancy things that you would do with a normal debugger. It is more like a monitor. You can change variables at run time (without interrupting program execution), you can inspect memory, port, I/O or write these peripherals. Also you can exercise other devices like SPI, LPC or FPGA if you have added this functionality in the monitor.

The basic functionality is summarized below:

Command & Format	Summary
R XXXX	Read Byte at Addr XXXX
W XXXX YY	Write Byte YY at XXXX
V XXXX	See hex 4 bytes starting at XXXX
A XXXX	see ASCII 4 bytes starting at XXXX
1/2/3/4	PINA/B/C/D

B XX YY	Write Port PortX(01-04), DDRX(11-14) YY
b XX	Read Port PortX(01-04), DDRX(11-14)
Q 0X	Read Analog Port 0X (X: 0-7)
I XXXX	Inspect Data in EEPROM at addr XXXX
P XXXX YY	Write byte YY in EEPROM at addr XXXX
U ?????????	User Command
L 00XX	Read LCM addr XX
C XX	Write LCM command XX
D XX	Write LCM data XX
S XXXXXXXX	Read 4 bytes at LPC Bus Addr. XXXXXXXX-X+4 (32 bits)
s XXXXXXXX YY	Write LPC Byte at address XXXXXXXX (32 Bits)
t XXXXXXXX YY	Write a byte at LPC FLASH SST49F020/A (with write protection)
*	Revert To Serial APP. Disable Debugger
P	Read SPI DR, SR
F 00XX	Read FPGA reg XX (Custom Commands, depend on FPGA Code)
f 00XX YY	Write FPGA Byte YY at reg XX (Custom Commands, depend on FPGA Code)

Note: All numbers are in Hex format and should be in Capital (case sensitive).

For example in order to inspect a variable at RAM location 0x10 you type at your serial terminal:

```
R 0010
```

If you need to modify the contents of this memory location with the value of 0x55:

```
W 0010 55
```

If you need to inspect PIN A of AVR you just type ‘1’ and the pin status of Port A is returned.

Variables address can be found at the /build.lst/kernel.map file. For example you can search v_SysStat address and find something similar to the following:

```
*(COMMON)
 COMMON  0x00800160       0x2 build.obj/kernel.o
         0x00800160              v_SysStat
         0x00800161              v_StatReg
```

So the RAM address for AVR is 0x160.

You can provide you own commands on dbgext.c, but you need to modify the base debugger.c file as well to get the input and process the new commands. What I prefer to do is to implement the extra commands to the separate dbgext.c while I add the command recognition and parsing on the debugger.c. I then enable or disable with the HW.in makefile module definition the corresponding command when I need it. The functionality of the extra commands exists always in the debugger.c which calls the dbgext.c functions.

Also there is an empty user command (‘U’) which you may implement differently on each application without making more complex dbgext.c modifications.

The debugger module does not use sprintf or stdio libs, so they have minimum footprint. On processors with more memory you may implement a better and more capable monitor but a small footprint monitor can be used anywhere.

<u>Other modules</u>

I have included the LCM and the keymat 4x4 modules. However these modules are partially tested and may not work always properly. Especially the LCM_char module was based from Joerg Wunsch's code for HD44780 controller. In another article we will discuss the FPGA SPI programming and the flow in order to produce the C-code from the .bit file.

System setup

In order to allow the make files to work you need to adjust the env.in file. Here you need to identify the compiler executables, the programmers etc. You need to have the executables in your path or otherwise you might need to add the absolute path to your tools.

Cfg/Srcobj.in defines which source/object files will be used depending on definitions of hw.in file.

Cfg/Srcdef.in translates the hw.in makefile variable definitions to the C-files used #define pre-processor variables.

Finally cfg/hw.in configures all the hardware parameters (ie crystal clock frequency, baud rate, active modules etc).

In order to compile the code you open a command prompt at the root directory where the makefile resides. Then you type "make" or "make all". In order to do a new build or when you modify the hw.in file you need to begin a new clean build: "make clean" and then "make".

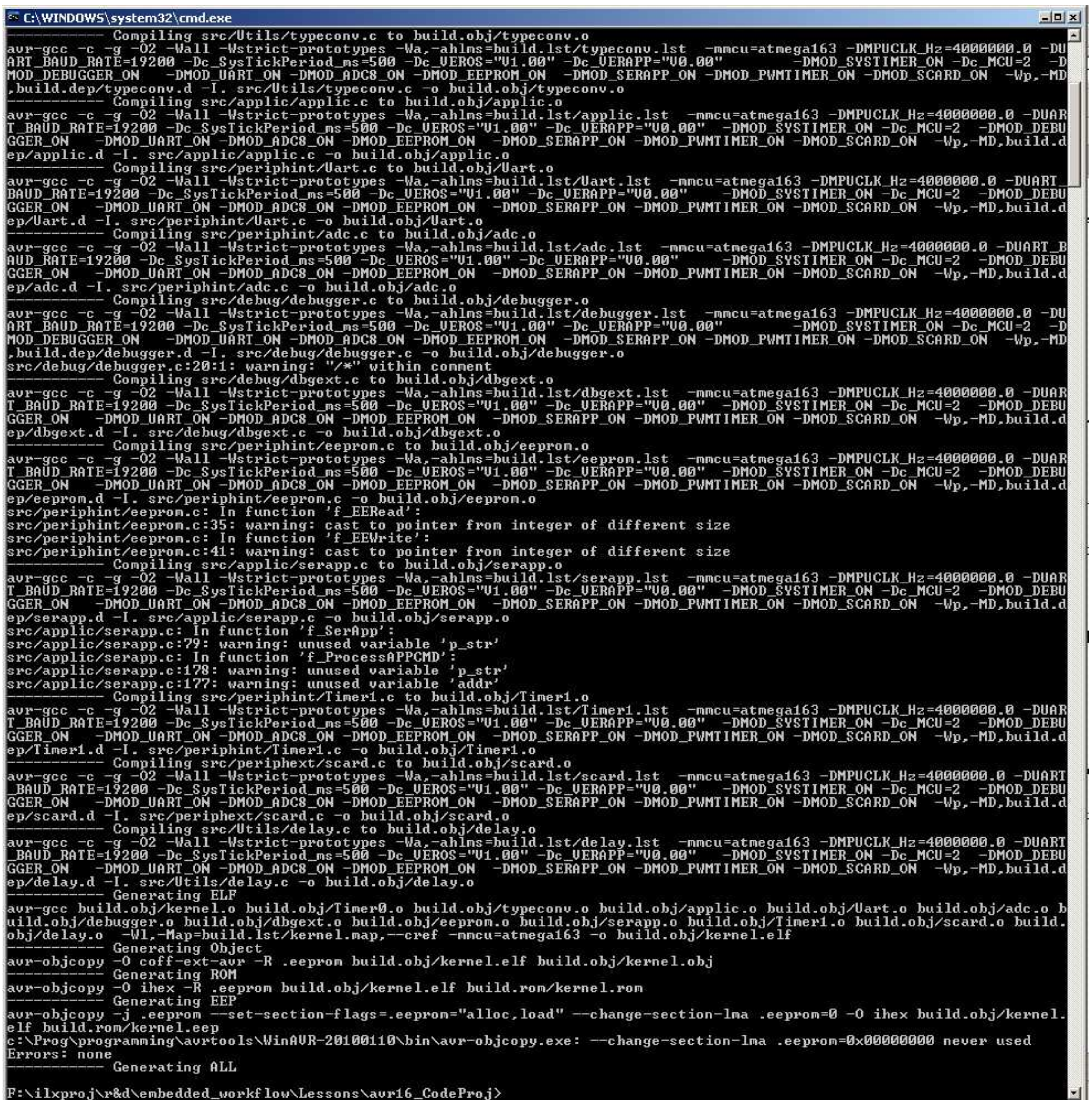
You can see various messages during compilation. In case of error the compiler stops and states in which lines have problems in a particular file.

When the rom files are ready you can type either "make prog" or "make progsp" to download the program to your target. The prog option is used with AVReal programmer while progsp uses the avr-dude which is the standard gcc tool for programming.

Typing "make size" you can review the object size.
Typing "make list" you can see the available options.

Hardware pin-out control is all set to the includes/settings.h file. All my hardware devices are using define macros that are referring to this file. So I have all of my I/O in one place and for each hardware I have to change only this file. Of course there are limitations for specific peripherals like UART or SPI where functionality is hardwired.

In the picture below you can see an example compilation.



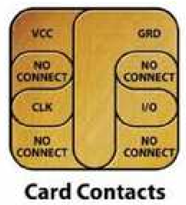
```
------------ Compiling src/Utils/typeconv.c to build.obj/typeconv.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/typeconv.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DU
ART_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"        -DMOD_SYSTIMER_ON -Dc_MCU=2  -D
MOD_DEBUGGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD
,build.dep/typeconv.d -I. src/Utils/typeconv.c -o build.obj/typeconv.o
------------ Compiling src/applic/applic.c to build.obj/applic.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/applic.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUAR
T_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"  -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/applic.d -I. src/applic/applic.c -o build.obj/applic.o
------------ Compiling src/periphint/Uart.c to build.obj/Uart.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/Uart.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUART_
BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"    -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/Uart.d -I. src/periphint/Uart.c -o build.obj/Uart.o
------------ Compiling src/periphint/adc.c to build.obj/adc.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/adc.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUART_B
AUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"     -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/adc.d -I. src/periphint/adc.c -o build.obj/adc.o
------------ Compiling src/debug/debugger.c to build.obj/debugger.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/debugger.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DU
ART_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"        -DMOD_SYSTIMER_ON -Dc_MCU=2  -D
MOD_DEBUGGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD
,build.dep/debugger.d -I. src/debug/debugger.c -o build.obj/debugger.o
src/debug/debugger.c:20:1: warning: "/*" within comment
------------ Compiling src/debug/dbgext.c to build.obj/dbgext.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/dbgext.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUAR
T_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"  -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/dbgext.d -I. src/debug/dbgext.c -o build.obj/dbgext.o
------------ Compiling src/periphint/eeprom.c to build.obj/eeprom.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/eeprom.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUAR
T_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"  -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/eeprom.d -I. src/periphint/eeprom.c -o build.obj/eeprom.o
src/periphint/eeprom.c: In function 'f_EERead':
src/periphint/eeprom.c:35: warning: cast to pointer from integer of different size
src/periphint/eeprom.c: In function 'f_EEWrite':
src/periphint/eeprom.c:41: warning: cast to pointer from integer of different size
------------ Compiling src/applic/serapp.c to build.obj/serapp.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/serapp.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUAR
T_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"  -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/serapp.d -I. src/applic/serapp.c -o build.obj/serapp.o
src/applic/serapp.c: In function 'f_SerApp':
src/applic/serapp.c:79: warning: unused variable 'p_str'
src/applic/serapp.c: In function 'f_ProcessAPPCMD':
src/applic/serapp.c:178: warning: unused variable 'p_str'
src/applic/serapp.c:177: warning: unused variable 'addr'
------------ Compiling src/periphint/Timer1.c to build.obj/Timer1.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/Timer1.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUAR
T_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"  -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/Timer1.d -I. src/periphint/Timer1.c -o build.obj/Timer1.o
------------ Compiling src/periphext/scard.c to build.obj/scard.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/scard.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUART
_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"   -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/scard.d -I. src/periphext/scard.c -o build.obj/scard.o
------------ Compiling src/Utils/delay.c to build.obj/delay.o
avr-gcc -c -g -O2 -Wall -Wstrict-prototypes -Wa,-ahlms=build.lst/delay.lst  -mmcu=atmega163 -DMPUCLK_Hz=4000000.0 -DUART
_BAUD_RATE=19200 -Dc_SysTickPeriod_ms=500 -Dc_VEROS="V1.00" -Dc_VERAPP="V0.00"   -DMOD_SYSTIMER_ON -Dc_MCU=2  -DMOD_DEBU
GGER_ON   -DMOD_UART_ON -DMOD_ADC8_ON -DMOD_EEPROM_ON  -DMOD_SERAPP_ON -DMOD_PWMTIMER_ON -DMOD_SCARD_ON  -Wp,-MD,build.d
ep/delay.d -I. src/Utils/delay.c -o build.obj/delay.o
------------ Generating ELF
avr-gcc build.obj/kernel.o build.obj/Timer0.o build.obj/typeconv.o build.obj/applic.o build.obj/Uart.o build.obj/adc.o b
uild.obj/debugger.o build.obj/dbgext.o build.obj/eeprom.o build.obj/serapp.o build.obj/Timer1.o build.obj/scard.o build.
obj/delay.o  -Wl,-Map=build.lst/kernel.map,--cref -mmcu=atmega163 -o build.obj/kernel.elf
------------ Generating Object
avr-objcopy -O coff-ext-avr -R .eeprom build.obj/kernel.elf build.obj/kernel.obj
------------ Generating ROM
avr-objcopy -O ihex -R .eeprom build.obj/kernel.elf build.rom/kernel.rom
------------ Generating EEP
avr-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" --change-section-lma .eeprom=0 -O ihex build.obj/kernel.
elf build.rom/kernel.eep
c:\Prog\programming\avrtools\WinAVR-20100110\bin\avr-objcopy.exe: --change-section-lma .eeprom=0x00000000 never used
Errors: none
------------ Generating ALL

F:\ilxproj\r&d\embedded_workflow\Lessons\avr16_CodeProj>
```

Example Application

Our example Application will implement a lock. The key will be a simple smart card and will control a servo for locking. The input of the system is a valid smart card (with proper serial number) and the output is a PWM signal to control the R/C servo for locking/unlocking.

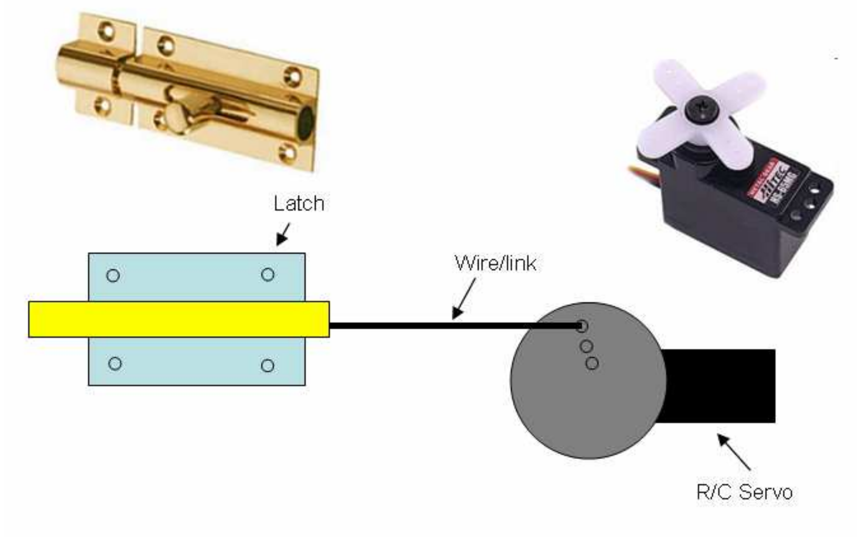
A more detailed information about smart cards you can find at:
http://support.gateway.com/s/Mobile/SHARED/FAQs/1014330Rfaq21.shtml

In this application I am using simple PROM type smart cards used for telephone applications (card-phones). In the pictures below you can see card-readers (passive elements for connecting card contacts to PCB) and the pin out of the smart cards.

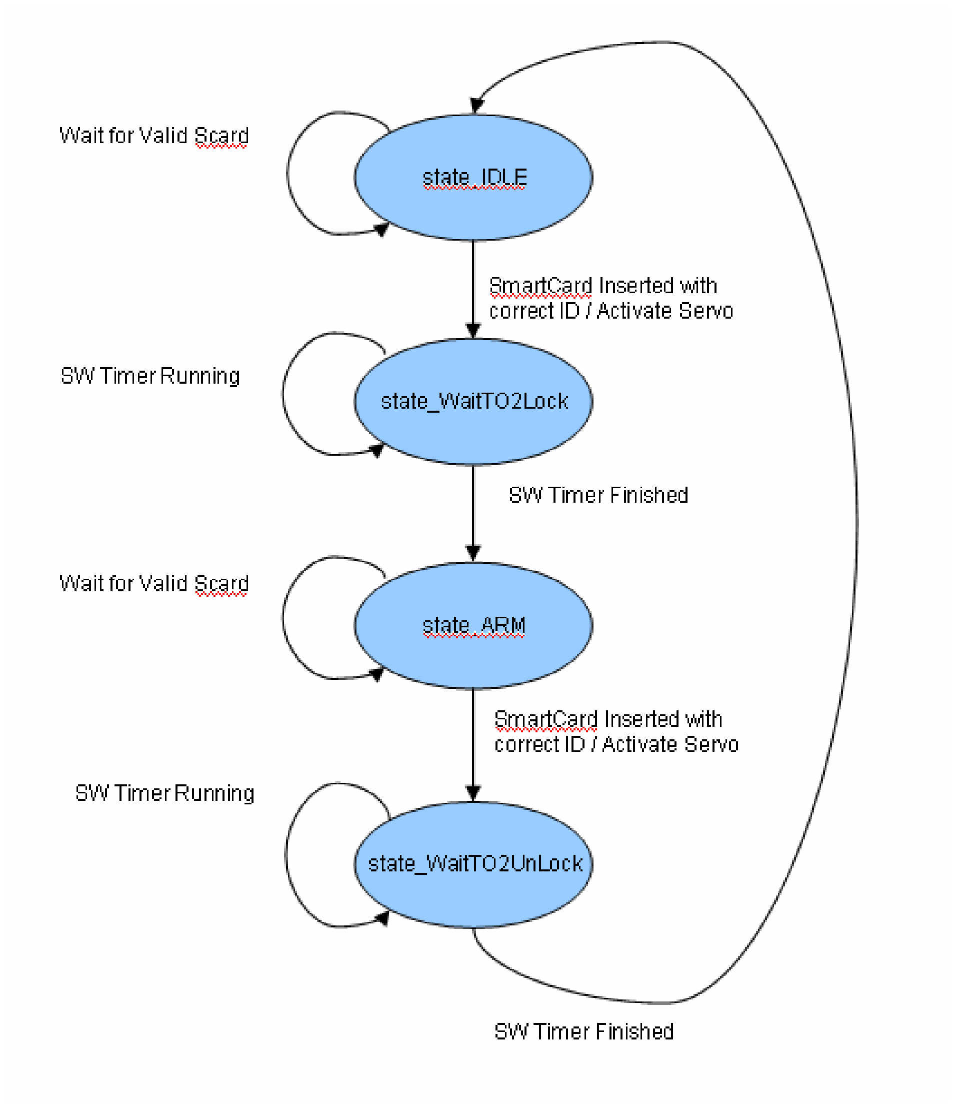


Card Contacts

On the diagram below you can see the mechanical configuration of a servo lock.



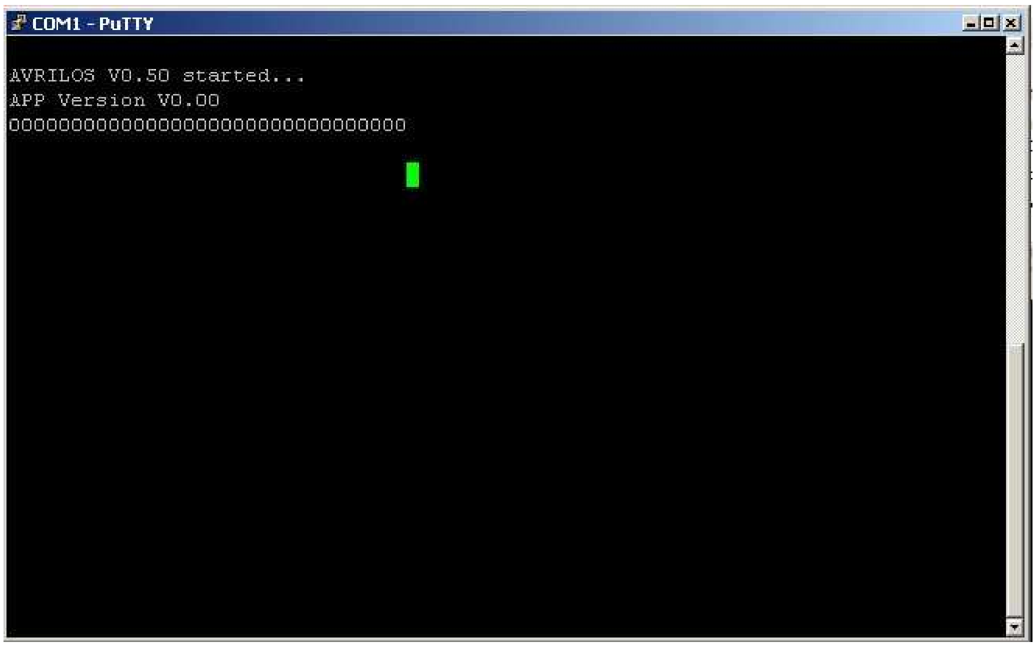
The Finite State Diagram below shows how the application logic control is implemented.



There are four main states. Initially the system goes at the IDLE state. Until a valid smart card is inserted the system waits there. After the insertion of a valid card, which is predefined in the code, the system activates the servo and waits for a predetermined time until the R/C servo reaches its final position at the WaitTO2Lock state. Then it enters the ARM state and waits again a valid smart card in order to do the opposite, activating the R/C servo to its initial position. After that (WaitTO2UnLock) it goes to the initial IDLE state.

We will use the debugger to find out which PWM values are the optimum start & stop positions for the servo (lock/unlock). Depending of how you have implemented the mechanical assembly of the lock you will need different PWM values for the two terminal positions. With the use of the debugger we will change the two variables that control these positions and thus determine with trial and error the correct PWM values. This small example will show how to use a small portion of the debugger/monitor to develop your application.

A boot screen in example is shown below:



```
AVRILOS V0.50 started...
APP Version V0.00
00000000000000000000000000000000
```

The zeroes after the OS/APP statement are from the smart card module (no card reader connected in my hardware).

The minimum and maximum PWM values are stored in two variables:

v_ServoLock,
v_ServoUnLock

We go to the file build.lst/kernel.map and we do a search of "v_ServoLock":

```
v_ServoLock        0x1           build.obj/applic.o
0x00800171                       v_ServoLock
v_ServoLock                      build.obj/serapp.o
                                 build.obj/applic.o
```

We find three instances. The first is the size of the variable. The second is the address (with offset 0x800000). The last is the module where the variable is used.
The important for us is the second. So v_ServoLock is at address 0x171 and it is one byte.

We enter the serial application writing at the terminal "* <Enter>". Then we try the direct key actions 'n' & 'm' exercising the min-max values of the servo. Then we enter the debugger by "& <Enter>".

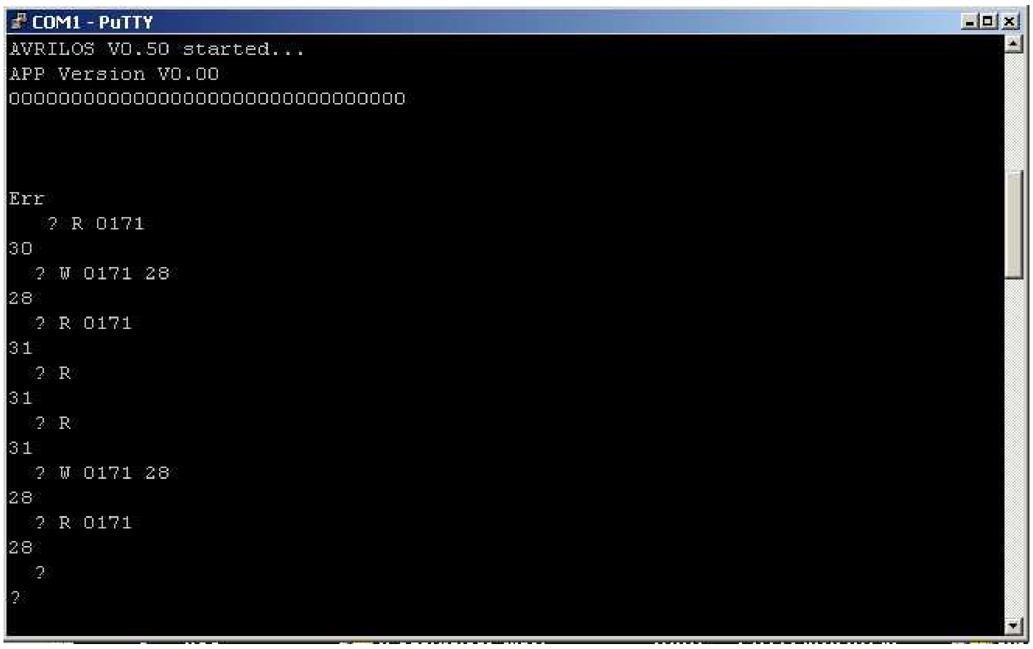
Then we read the value of v_ServoLock:

```
> R 0171
 32
>
```

The returned value 32 is in hex which is 50 dec which is the initial value of the variable. Now let's decrease this value to 40 dec -> 0x28.

```
> W 0171 28
> R 0171
 28
>
```

Here is a similar example shown below:



COM1 - PuTTY

```
AVRILOS V0.50 started...
APP Version V0.00
00000000000000000000000000000000



Err
   ? R 0171
30
  ? W 0171 28
28
  ? R 0171
31
  ? R
31
  ? R
31
  ? W 0171 28
28
  ? R 0171
28
  ?
?
```

Re-enter the application with “* <Enter>” and retry the ‘m’ command. The maximum terminal position of the servo should be changed! You can play with these and find the best values that you can hardcode to your application. Even better you can use the EEPROM to store and use from your program for such settings.

You can do the same with v_ServoUnlock and trim this value as well.

This is a small example of how you can do some real-time tests without re-program your controller. You can test the pin and port I/O in order to see if the controller “see” the proper values on its ports or if any port seems floating (ie a bit would toggle randomly when you touch the pin).

Serial Application

The serial application is another example task that is used to control directly the application. For example it works seamlessly with the debugger. Entering ‘*’ on the debugger it reverts to serial application and the opposite. Entering ‘*’ to the serial application you can revert to the debugger.

The serial application shown supports two different modes. Direct key execution and shell execution. For simplicity the shell commands are single keys (like the debugger commands) followed maybe by parameters and terminated by the <Enter> key (0x0D).

The direct key execution mode is actions that you do while the key is pressed on your terminal. For example we have some commands that control the servo position so you can see that your hardware works. For example pressing the ‘c’ key you rotate slowly the servo to one direction while pressing ‘v’ you rotate the servo the other way around. The function which controls key actions is “f_ProcessAPPKey”.

The shell function which provides more shell like interface is “f_ProcessAPPCMD”. For example entering “q <Enter>” you do the same with pressing ‘c’. That way you can have two-fold control in your application and have flexibility of control through your terminal.

One important aspect is that upon entering the serial application main function we check if the debugger bit is set. If it is we skip execution as the debugger is active. At the same time “f_ProcessAPPCMD” have to support the asterisk “*” command in order to revert to the debugger. You can use the ampersand “&” character to return to debugger if you want to jump to the debugger mode blindly without knowing the previous state (toggle with ‘*’ between debugger and application).

Conclusion

In this article I presented a simple framework for building embedded systems faster and easier. Of course there are easier ways of developing embedded systems (like using basic interpreters) but this is a very scalable system which the convenience of easy development does not compromise performance or capabilities. I began building this system before 15 years on 8051 assembly, I then wrote it in AVR assembly and afterwards I re-wrote the whole framework in C. I have done various modifications over time and here it is. I provide the source code to the public in order to help more people explore the beauty of embedded systems. I hope also that people may contribute their own peripherals and extensions making a larger library of components. This will make development of new systems for the rest of the people easier.

For any questions: avrilos@ilialex.gr

History
Version 1.0, Initial Release.