Atmel ATmega Multi Tasking Kernel

Reuben Carter

October 5, 2013

1 Introduction

The kernel was developed using an ATmega 1280 Arduino[1] as a development platform. The code was written in C where possible, however for some parts it was necessary to use inline assembler. The Arduino environment uses the avr-gcc compiler and AVRDUDE programmer. There is an Arduino boot-loader uploaded into the ATmega's flash memory and starting at the first memory address in all arduino platforms. This boot-loader allows code to be loaded into the flash memory over the ATmega's on-board USART. The Arduino platform also has an FTDI[2] USB to TTL level serial converter, which allows communication over USB to the USART[3]. The arduino is an easy platform to start programming the ATmega series processors, and code written under the arduino environment can run on most ATmega processors without the arduino boot-loader.

2 Kernal Tick

At the heart of the kernel there is a periodic tick. At regular intervals the kernel must interrupt the current executing code and take control of the processor, vital to creating granularity within an operating system. This is achieved using an interrupt. The ATmega processor series contain internal programmable timer/counters which can be set to cause an interrupt under various conditions. Many of the ATmega series contain a 16 bit timer (Timer/Counter 1) which this kernel makes use of to provide the main tick interrupt.

In order to use this timer to generate a regular interrupt with a specific period, the values of certain internal registers must be set. The timer must be set to increment, and when a certain value has been reach, an interrupt must fire and the timer reset. This is setup by setting the values of the timer control regester (TCCR1A/B), the output compare register (OCR1A), and the interrupt mask register (TIMSK1). The timer control register A/B is initially set to zero, then TCCR1B is set to control the timers prescaler (on bits 0 to 2) and to turn on the clear timer on compare match mode (CTC, bit 3 to 4), which causes the timer to reset when a specified value has been reached. The value to compare, is set by writing it to the output compare register (OCR1A). The interrupt mask register (TIMSK1) controls which interrupt fires, and is set

[1] http://www.arduino.cc/
[2] http://www.ftdichip.com/
[3] Universal asynchronous receiver/transmitter

so that the output compare A match interrupt is enabled (by writing to bit 1). This means that when the value of the counter is equal to the value stored in OCR1A, an interrupt vector is executed.

The increment of the timer is based on the system clock, however the system clock is very fast. In the arduino platform the system clock is 16Mhz leading to a minimum frequency of 244Hz for the 16 bit timer to reach its maximum value. To overcome this limitation, the timers clock source can be set to a prescaled system clock signal. A prescalar value of 1024 has been chosen, giving more manageable time periods for the interrupt. This is done by writing the binary value 101 to the prescaler bits of the TCCR1B register.

The value to write to the OCR1A register (the maximum value of the timer) is calculated from the system clock, the prescaler valuer, and the desired time period of the kernel tick. Take the core frequency as f_c, the prescaler value as p, the OCR1A value as c, and the desired tick period as t. The following relationship can be found.

$$c = t\frac{f_c}{p} - 1 \tag{1}$$

By default the kernel tick time is set to 1ms, by setting f_c to 16, p to 1024, and t to 1000, as setting f_c to 16 implies that units of 1us are to be used. The following code sets up the timer, the kernel tick interrupt and calculates the value for the output compare register.

```
1  short TimeToTimerCounts(int fTime, int fCoreFreq, int fPrescaler
       )
2  {
3          int res = (fPrescaler / fCoreFreq);
4          return (fTime / res) - 1;
5  }
6
7  void InitTimer1Interupt(short fCounts)
8  {
9          TCCR1A = 0;
10         TCCR1B = 0;
11         OCR1A = fCounts;
12         TCCR1B |= (1 << WGM12);
13         TCCR1B |= (1 << CS10);
14         TCCR1B |= (1 << CS12);
15         TIMSK1 |= (1 << OCIE1A);
16 }
```

The interrupt service routine is executed when the timer interrupt fires. In a normal situation, the ISR macro used in AVR-gcc to define the interrupt service routine, automatically handles saving and restoring the current processor context, and optimizes the save and restore routines to save/ restore only registers that are in use by the ISR. However any automated code placement via AVR-gcc's ISR macro is not welcome during the ISR of the kernel. This is because the kernel handles all context saving and restoring and would conflict with the generated code. To strip the ISR of all generated code, the ISR_NAKED parameter is used. This ensures that when the interrupt fires, only the program counter is pushed to the stack and altered so that its value corresponds to the value of the first instruction within the ISR macro. No other instructions are executed. The following code shows ISR macro usage. The TIMER1_COMPA_vect refers to the comparison interrupt vector from timer1.

```
1  ISR(TIMER1_COMPA_vect, ISR_NAKED)
2  {
3          ......//ISR code
4  }
```

3 Multi-tasking and Context Switching

The kernel among other things is, responsible for managing processes. A process is an independent instance of a program. The job of the multi-tasking kernel is to allow each process to run as if it was the only program running on the processor, and create the illusion of several process running in parallel. It does this by periodically interrupting the currently running code/process, using the timer generated interrupt. Once the process has been interrupted, it's execution context is saved, a different process is chosen by the kernel, and the new processes execution context is restored to the processor. The new processes continues where it was previously interrupted, as the ISR returns.

The ATmega processors have a Modified Harvard architecture. This means that there program memory and data memory is separated, the instructions are stored on flash memory and have a different address bus from the RAM. However this does not mean that the first address in the address space pointer to the first byte in RAM, as the ATmega are memory mapped devices, their registers are mapped to the address space. The RAM can be divided into two types, the heap, and the stack. The stack is a memory structure which can grow and shrink dynamically as needed, by pushing values onto the stack, and popping values off. This is achieved in hardware by using a stack pointer register, which holds the current address of the stack. When a value is pushed to the stack, the value is copied to the memory location pointed to by the stack pointer and then the stack pointer is decremented. The pop operation is the reverse, where the stack pointer is incremented, and then a value is read from the memory location pointed to by the stack pointer. This means that the stack grows down in memory on the ATmega architecture, as with many processors. The C programming language makes use of the stack heavily. when a function is called, the C language makes use of the stack to store program counter values, so a subroutine may be entered and later, the program counter returned. All local variables are also held in the stack, which makes recursive functions and variable scope possible in C.

Usually, when writing a simple single threaded program for an ATmega processor, the stack pointer will be initially set to the address of the end of the RAM. This will be handled by the compiler. There is a single stack for a single program, which grows from the top of the RAM. The heap will be allocated from the bottom of the ram. Within a multitasking environment however, each process must be allocated its own stack in the RAM. This is vital, as it ensures processes don't write over other processes memory. Multiple processes cannot share the same stack, and also remain independent, so each program must have its own stack. This can be achieved by giving each process a memory address for the base address of its stack. This address corresponds to an address in RAM, and must be chosen carefully, so as not to conflict/overlap with another processes stack. memory addresses must be chosen to have a gap larger than the size the stack may grow too.

When the process is interrupted, the processes current execution context is pushed onto the end of its stack. The value of the stack pointer is then saved in the kernel along with the processes stack base address. The ISR then exicutes, making use of the end of the processes stack, so each process must have a stack big enough for processes stack usage, the whole execution context, and the ISR's stack usage. The current stack pointer is used when the process is restored, to load up the context and return the stack pointer to the processes working position.

The ISR is key to the kernels operation. When the ISR fires the SAVE_CONTEXT macro is run first. This macro is heavily based on the context saving macro withing the freeRTOS[4] operating system. The macro pushes the values of every register r0 to r31 onto the stack, and assuming a process is already running, the stack pointer will pointer to the end of that processes stack. however, after pushing the value of r0 onto the stack the value of the status register (__SREG__) is copied to r0. r0 is again pushed to the stack, pushing the value of the status register. After register r1 has been pushed to the stack, it is set to 0, to fit a compiler expectation. After all registers have been pushed to the stack, the 16 bit stack pointers lower and upper 8 bit (__SP_L__ and__SP_H__) registers are copied to general registers r26 and r27. Finally the stack pointer value (stored in r26 and r27) is copied to the currentStackAddress variable, which is a global variable used by the kernel as the current stack pointer value. The whole processors context is saved on the end of the processes stack. The whole SAVE_CONTEXT macro is defined as volatile inline assembly, where the volatile keyword is used to force the compiler to not perform optimization. The value of the currentStackAddress variable is now equal to the address of the top of the processes stack, and saved by the kernel to the appropriate process. The SAVE_CONTEXT macro is shown in the following code.

```
1  push r0
2  in r0, __SREG__
3  cli
4  push r0
5  push r1
6  clr r1
7  push r2
8  push r3
9  //....CODE MISSING: push r4 to push r29
10 push r30
11 push r31
12 in r26, __SP_L__
13 in r27, __SP_H__
14 sts currentStackAddress+1, r27
15 sts currentStackAddress, r26
16 sei
```

The kernel then runs the scheduling algorithm, which chooses a new process to execute. The RunProcess function, responsible for running a new process, or restoring a process which has previously been interrupted, is executed with the chosen process as a parameter. If the process has not been run before, it will have no entry into its stack, and no context to restore. This is checked by the RunProcess function. In this situation the currentStackAddress kernel variable is set to the value of the base stack address given to the process. And the stack pointer is set to the currentStackAddress variable value. This is done using

[4]http://www.freertos.org/

the SET_STACK_ADDRESS macro. This macro is defined as volatile inline assembly code.

```
1  cli
2  OUT __SP_L__, %A0
3  OUT __SP_H__, %B0
4  sei
```

The %A0 and %B0 parameters are the lower and upper bytes of the the address variable (currentStackAddress) which is passed to the inline assembely code using gcc compatible syntax. The two bytes are passed to the lower and upper stack pointer registers. The stack pointer now points at the new processes base stack address.

The kernel contains a function called the baseFunction. This function takes the process to execute as a parameter, and executes it within the function. It acts a container for an executing process, and is use so that when the processes main function returns, the system does not crash. Instead the processes main function returns to the base function, which pauses the process and sits in a while loop. The kernel can then decide on appropriate action. Without this container, the processes main function would return, causing the ret instruction to be executed, changing the program counter value based on the value stored in the stack. However the stack value does not contain a suitable value for an instruction address to load into the program counter. This would cause the whole system to crash.

The baseFunction pointer is loaded into the currentStackValueLong variable, which is a kernel variable containing a long value to push to the stack, and the PUSH_STACK_VALUE_LONG macro is executed.

```
1  cli
2  mov r0, %A0
3  push r0
4  mov r0, %B0
5  push r0
```

This macro pushes the value in the currentStackValueLong variable onto the stack. The baseFunction's address is now the top value in the stack. This means that when a return instruction (ret) is executed, that value will be loaded to the program counter which will now point to the first instruction of the base function. However the function also takes a parameter, which is the address of the process to execute. When a function is written in the C language and compiled, the compiler produces machine code for a subroutine which looks for the function parameters in certain general registers. Therefore the next stage before the return instruction can execute, is to manually pass the base function parameter. This is done by setting the r24 and r25 registers to the lower and upper bytes of the function parameter, just before returning. This is done using the PASS_FUNCTION_ARGUMENT macro, which takes the kernel variable funcArgument (which holds the current function argument to pass to a new subroutine) and copies its to the correct registers.

```
1  cli
2  mov R25, %B0
3  mov R24, %A0
4  sei
```

The RunProcess function would normally call the return (ret) instruction when exiting the function, however the kernel has already loaded the correct

function parameters into the expected places, and pushed the function pointer onto the stack. This means that the kernel ISR has completed its task of saving the current executing context, choosing a new process, and executing for the first time a new process. The iret instruction is needed to return from the interrupt which breaks out of the ISR and sets the program counter value to the first value popped from the stack.

A different response is required from the RunProcess function if the scheduler chooses a process which has already been run and interrupted at some stage. The execution context from the previous execution of the process will have been pushed onto it's stack when the SAVE_CONTEXT macro was called. The stack pointer pointing to the end of the processes stack would also have been saved for that process. This value is copied by the RunProcess function, to the currentStackAddress kernel variable. The RESTORE_CONTEXT macro is then executed. This macro performs the opposite task of the SAVE_CONTEXT macro.

```
1  out __SP_L__, %A0
2  out __SP_H__, %B0
3  pop r31
4  pop r30
5  //...CODE MISSING: pop r29 to pop r4
6  pop r3
7  pop r2
8  pop r1
9  pop r0
10 out __SREG__, r0
11 pop r0
```

The macro first sets the value of the stack pointer, to the end of the processes stack. The macro takes the currentStackAddress variable as a parameter, and copies the lower and upper bytes (A0 B0), to the stack pointers lower and upper byte registers (__SP_L__ and __SP_H__). Each register is then in turn popped from the stack, where the previous stored context is held, and copied to the appropriate register. The context was save by pushing registers r0 o r31 to the stack, therefore the reverse must happen for restoring the context (popped to registers r31 to r0). The status register is also loaded from the correct stack location, using r0 as an intermediate. Finally the RunProcess function executes the iret interrupt return instruction, which breaks from the kernel ISR, and sets the value of the program counter to the next value popped from the stack. This value will be the correct value to return the program counter to the previous instruction memory position, just before the process was previously interrupted. This is because when an interrupt vector fires on the ATmega processor, the program counter is automatically pushed to the stack before being modified. Therefore after the context is restored, the next value on the stack will be the saved program counter value, pushed to the stack when the interrupt fired (when the process was previously interrupted). This concludes the details of the context switching components of the kernel, for both process which have never been executed, and processes which have, and were previously interrupted.

4 Process and Process Lists

The kernel must store information for each process, including the process function pointer(functionPtr, the address of the start of processes code), the address

of the start of the processes stack (stackAddress), the current processes stored context address which will be the end of the processes stack (contextAddress), and the processes current state (state). These variables are vital to the kernel ISR and scheduler. The command line argument buffer (argv) is also stored along with the length of this buffer (argc). This can provide a method to pass arguments to a process when run from the command line. Finally, there is a variables of type Port (stdPort). A program can read and write to a port, which in turn can read and write to other ports. This can provide an abstraction from directly accessing hardware, and a method for processes to talk to each other. These variables are held within a struct type deceleration, which has been type defined with the identifier Process. A typedef has also been used to define the function pointer type ProcessFunctionPtr, as a function with no parameters and returning void.

```
1  typedef void (*ProcessFunctionPtr)();
2
3  typedef struct _process
4  {
5          ProcessFunctionPtr functionPtr;
6          unsigned long stackAddress;
7          unsigned long contextAddress;
8          unsigned char state;
9          Port stdPort;
10         char argv[ARGV_BUFFER_SIZE];
11         unsigned char argc;
12 }Process;
```

Each Process has a state variable, which holds information about the execution state of the process. The state variable is critical to the control of each processes execution. A number of possible states exist, which are defined by the preprocessor and have associated integer values. The PROCESS_ACTIVE state implies that the process is currently executing on the processor. This state is set by the RunProcess kernel function. The PROCESS_IDLE state implies the process is live, but has been previously interrupted. When the current process is interrupted by the kernel ISR, it's state variable is set to PROCESS_IDLE. It will remain in this state until run once again. The PROCESS_STOPPED state implies that the process is not live, and will be treated as if it has never been run before. If a processes state is set to PROCESS_STOPPED, and the scheduler decided to run that process, the RunProcess function will start it as if it has never been run before. New processes have their state variable set to PROCESS_STOPPED so that the RunProcess function treats them correctly as a process that has never been run. However processes with a PROCESS_IDLE state will be restored by the RunProcess function. Changing a processes state to PROCESS_STOPPED while live, has the effect of restarting the process. When it is next scheduled the RunProcess function will treat it as a new process. The final state is the PROCESS_PAUSED state, which causes the scheduler to miss out the process, when deciding the next process to schedule. This has the effect of pausing the process until the state is set to PROCESS_IDLE, allowing it to be resumed, or PROCESS_STOPPED, effectively restarting the process. If an active processes state has been set to PROCESS_PAUSED, when the kernel ISR interrupts the process, the process state is not changed to PROCESS_IDLE as usual. Finally the PROCESS_DEAD state causes the scheduler to remove the process completely.

In order to hold process structs within the kernel and schedule live processes, a data structure is needed. The data structure needs to be dynamic so that processes can be added and removed at arbitrary positions in the structure. This allows processes to be added and removed while the operating system is running. A singly linked list was created to hold process structures. A Typedef has been used to define a ProcessNode.

```
1  struct _processNode
2  {
3          struct _processNode* nextNode;
4          Process process;
5  };
6  typedef struct _processNode ProcessNode;
```

Each node contains a pointer to the next node in the list, and instantiates a process struct. The linked list adds and removes nodes from a static array of nodes (processNode), which are instantiated with a set length, defined by MAX_PROCESS_NUMBER. This is to avoid the use of dynamic memory allocation. The linked list can then be created by taking process nodes from this static heap of nodes, and linking them together by altering their next-node pointer. When a node is discarded from the list, it is added back to the static heap of nodes. The list needs a base node to start the list (baseNode). A pointer to the end of the list is stored (endNode), and a pointer to the start of the heap of nodes (poolStartNode), along with the current length of the list (unsigned char processListLength). The list is started by adding a node to the base node.

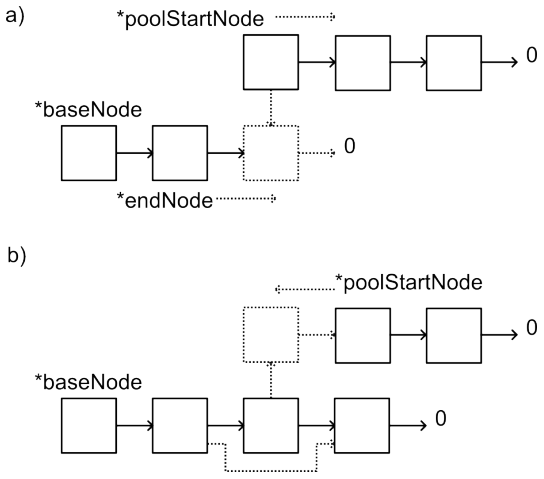

Figure 1: Figure demonstrating the a) push and b) remove operations of the linked list.

The manipulation of the list is done via three functions, InitProcessList, ProcessListPush and ProcessListRemove. The InitProcessList function formats the empty heap of nodes, and sets the initial state for the list variables. First the processListLength variable is zeroed and the endNode variable is set to store the value of the base node of the list (as the list is empty). Then each element of the node heap array is linked to the next by setting the nextNode variable to the next element in the array. The last element's nextnode variable is set to

zero. The process within each of these nodes is set to state PROCESS_DEAD. Finally the poolStartNode variable is set to the address of the first element in the heap array. The process list is now ready for push and remove operations.

The ProcessListPush function takes an empty process node from the heap of nodes and adds it to the end of the list. The node pointed to by the endNode variable is altered such that the nextNode variable points to the first node of the empty node heap. The poolStartNode variable is set to the next node in the heap. Then the endNode variable is set to the endNode's next node variable (which is the new node in the list). Finally the new endNode nextNode is zeroed and the processListLength is incremented. A pointer to the new process node is returned.

The ProcessListRemove function removes a process node from an arbitrary position in the list.

5 Scheduler

The kernel ISR starts with the SAVE_CONTEXT macro. The current processes state is then changed to PROCESS_IDLE if it is currently active, and its context address variable is saved. The scheduler then chooses the next process to run. The Scheduler function returns the address of this new process which is stored in the currentProcessNode variable. Finally the RunProcess function is called with the new process node as an argument.

```
1  ISR(TIMER1_COMPA_vect, ISR_NAKED)
2  {
3          SAVE_CONTEXT();
4          if((*currentProcessNode).process.state == PROCESS_ACTIVE
               )
5          {
6                  (*currentProcessNode).process.state =
                       PROCESS_IDLE;
7          }
8          (*currentProcessNode).process.contextAddress =
               currentStackAddress;
9          currentProcessNode = Scheduler();
10         RunProcess(currentProcessNode);
11 }
```

The scheduler function chooses the next process to run, based on the next process in the process list, and the state of that process. This type of scheduler is know as a round robin scheduler and is the simplest scheduling algorithm. There is significant room for furthering the complexity of this scheduling algorithm by introducing process priority, however for simplicity a round robin scheme was chosen. If the state of that process is PROCESS_PAUSED the process is skipped, and the next process is tested. If the process state is PROCESS_DEAD, the next process is tested and the dead process is removed from the process list by calling the ProcessListRemove function. This is repeated until a process is found, not paused or dead. A set of functions (RemoveProcess, PauseProcess, ResumeProcess and RestartProcess) control the state of a process, while the schedular and run function perform the operation.

```
1  ProcessNode* Scheduler()
2  {
3          ProcessNode* temp;
```

```
4           ProcessNode* tempPrev;
5           temp = (*currentProcessNode).nextNode;
6           if(temp == 0)
7                   temp = baseNode.nextNode;
8           while((*temp).process.state == PROCESS_PAUSED || (*temp)
                .process.state == PROCESS_DEAD)
9           {
10                  tempPrev = temp;
11                  temp = (*temp).nextNode;
12                  if((*tempPrev).process.state == PROCESS_DEAD)
13                          ProcessListRemove(tempPrev);
14                  if (temp == 0)
15                  temp = baseNode.nextNode;
16          }
17          return temp;
18  }
```

6 Further

This simple kernel has room for further development especially in increasing the complexity of the scheduling algorithm and creating a priority scheme for processes. More real-time features could be integrated, such as a FPS scheduling. Also real time tasks could be prioritized over non real-time tasks.