AvrX API 8 April 2008

Note: AvrX is written in assembly language. See avrx.h for info not listed herein such as structure typedefs for messages, timers, etc.

Control Blocks / Structures:

TimerControlBlock

This is a six byte structure consisting of:
- *next - pointer to the next element in the TCB queue
- semaphore - task(s) owning this TCB (?)
- count - 16 bit number of timer ticks

TimerMessageBlock

Similar to the TCB, the TMB sends messages rather than a semaphore. The structure appears to consist of the union of a MessageControlBlock and a TimerControlBlock.

MessageControlBlock

This is to message queues as the TCB is to timer queues. The structure consists of four bytes:
- ***next** - pointer to the next element in the MCB queue
- **semaphore** - task(s) owning this MCB (?)

PID

The size of the Process ID varies depending on whether single-stepping is enabled, which adds two unsigned char values. The standard PID is defined as:
- ***next** - pointer to the next PID in the chain
- **flags/priority** - unsigned char
- ***ContextPointer** - ?

Mutex

Mutex semaphores are a simple linked list of waiting processes. The mutex may have the following values:

SEM_PEND	(0)	// Semaphore is reset waiting for a signal
SEM_DONE	(1)	// Semaphore has been triggered.
SEM_WAIT	(2)	// Any other value is the address of a processID

The primary difference between a **mutex** and a **semaphore** is a mutex is owned by the process which locked it whereas a semaphore is not; a semaphore can be unlocked by another process.

System Objects

A System Object consists of a link (16 bit pointer) and a semaphore and is followed by 0 or more bytes of data.

Macros to simplify declaring AvrX related things:

AVRX_GCC_TASKDEF(start, c_stack, priority) *declare task prototype (GCC)*
AVRX_GCC_TASK(start, c_stack, priority)
AVRX_SIGINT(vector)
AVRX_MUTEX(A) *declare a semaphore*
AVRX_MESSAGE(A) *declare a message queue*
AVRX_EXTERNTASK(start) \ *declare a task's prototype*

```
  CTASKFUNC(start);                 \
  extern TaskControlBlock start##Tcb; \
  extern ProcessID start##Pid
```

Many others are in avrx.h.

TASKS

AvrXInitTask (avrx_tasking.s)
Format: AvrXinitTask(&taskTCB);
Initializes but does not run a task by saving the registers on the appropriate stack and initializing all registers to be used to zero.

AvrXRunTask (avrx_tasking.s)
Format: AvrRunTask(&taskTCB);
Starts a task by first calling AvrXInitTask() followed by calling AvrXResume(). Either this or AvrXInitTask)() must be called by main() to get things going.

AvrXSelf (avrx_priority.s)
Format AvrXSelf();
Returns the process ID (PID) of calling process.

AvrXPriority (avrx_priority.s)
Format: AvrXPriority(&PID);
Returns the current priority of the PID.

AvrXChangePriority (avrx_priority.s)
Format: AvrXChangePriority(&PID, unsigned);
Sets the priority of a process to the second parameter. Returns the original setting.

AvrXSuspend (avrx_suspend.s)
Format: AvrXSuspend(&taskPID);
Marks a PID for suspension and attempts to remove it from the run queue.

AvrXResume (avrx_tasking.s)
Format: AvrXResume(&taskPID);
Gets a task ready to run again by inserting the PID into the run queue.

AvrXTaskExit (avrx_terminate.s)
Format: AvrXTaskExit;
Should be called when a task is complete and will no longer

run. It acts like the "return" in a normal function. Ideally, the task should be idle (not waiting on any semaphores) before this is called.

AvrXTerminate (avrx_terminate.s)
Format: AvrXTerminate(&taskPID);
Similar to the AvrXTaskExit function except that AvrXTerminate is used to terminate another task.

AvrXYield (avrx_reschedule.s)
Format: AvrXYield();
Removes self from run queue and re-queues. If other tasks of the same priority are on the queue, self will queue behind them (round robin)

Example of Declaring and starting a task

```
In the main()
. . . initialize hardware/software

      AvrXRunTask(TCB(demo));




// here is a task named demo, stack size 200, priority 1
AVRX_GCC_TASKDEF(demo, 200, 1)
{
      char ch;    // and so on, as in any function

      puts_P(PSTR("Hello From Demo Task\r\n"));
 . . . the rest of the tasks's code, usually an infinite loop
. . .  with wait for message, or some such.
}
```

SEMAPHORES

AvrXSetSemaphore (avrx_semaphores.s)
Format: AvrXSetSemaphore(&mutex);
Sets a semaphore [**SEM_DONE**]. Wakes up waiting tasks, if any.

AvrXIntSetSemaphore (avrx_semaphores.s)
Format: AvrXIntSetSemaphore(&mutex);
Same as AvrXSetSemaphore above except that this one assumes that code has already switched to the system stack (such as an **interrupt** handler).

AvrXWaitSemaphore (avrx_semaphores.s)
Format: AvrXWaitSemaphore(&mutex);
Causes a task to queue up for a semaphore, implementing a Mutual Exclusion Semaphore (mutex). If used for simple signaling purposes then only one task should wait on the semaphore.

AvrXTestSemaphore (avrx_testsmaphore.s)
Format: AvrXTestSemaphore(&mutex);
A non-blocking call to check the state of a semaphore. Returns one of the following:

SEM_PEND	(0)	// Semaphore is reset waiting for a signal
SEM_DONE	(1)	// Semaphore has been triggered.
SEM_WAIT	(2)	// Any other value is the **address of a processID**

AvrXIntTestSemaphore (avrx_testsmaphore.s)
Format: AvrXIntTestSemaphore(&mutex);
Same as the AvrXTestSemaphore except that it assumes that code has already switched to the system stack (such as an **interrupt** handler).

AvrXResetSemaphore (avrx_resetsemaphore.s)
Format: AvrXResetSemaphore(&mutex);
Forces a semaphore to the **SEM_PEND** (false) state.

TIMERS

Timers use a tick count parameter. The tick count ticks per second as shown in the timer interrupt service routine and its .h header file. The APIs are provide a simple delay, a method to poll a timer, and AvrXStartTimerMessage which sends a message upon timeout. Acronyms:
TCB – Timer Control Block. See avr.h
TMB – Timer Message Block used with AvrXStartTimerMessage(). See avr.h

AvrXStartTimer (avrx_timequeue.s)
Format: AvrXStartTimer(&TCB, unsigned);
This non-blocking API will add the TCB into the timer queue using the second parameter as the wait time (number of timer ticks). The 2nd parameter is the timeout tick count.

AvrXDelay (avrx_timequeue.s)
Format: AvrXDelay(&TCB, unsigned);
This is a blocking version of AvrXStartTimer and has the same parameters. It is implemented by calling AvrXStartTimer followed by AvrXWaitTimer. The 2nd parameter is the timeout tick count.

AvrXCancelTimer (avrx_canceltimer.s)
Format: AvrXCancelTimer(&TCB);
Dequeues a TCB from the TCB chain. Returns a pointer to the removed timer or a 0 on failure.

AvrXWaitTimer (avrx_semaphores.s)
Format: AvrXWaitTimer(&TCB);
Waits on a timer to expire.

AvrXTestTimer (avrx_testsemaphore.s)
Format: AvrXTestTimer(&TCB);
Tests the state of a timer. Returns one of

SEM_PEND	(0)	// Semaphore is reset waiting for a signal
SEM_DONE	(1)	// Semaphore has been triggered.
SEM_WAIT	(2)	// Any other value is the **address of a processID**

AvrXStartTimerMessage (avrx_starttimermessage.s)
Format: AvrXStartTimerMessage(&TMB, unsigned, &MessageQueue);
Starts a timer that sends a message when a timeout occurs. Might be useful is where you want to time out an input message or interrupt. Start a timer that sends a message to *the same message queue* (self) and then does AvrXWaitMessage() on that queue. Messages on the queue will either be the expected message or a timeout message. *AvrXSendMessage* or *AvrXIntSendMessage* can be used with a TMB. The 2nd parameter of this API is the timeout tick count. Note that the TMB is a structure with a union on MCB and TCB, the latter is larger.

AvrXCancelTimerMessage (avrx_canceltimermessage.s)
Format: AvrXCancelTimerMessage(&TMB, &MessageQueue);
Removes a timer message that has been started but is no longer needed. Note: If the timer has already expired, its message, now on the MessageQueue will be removed.

AvrXTimerHandler (avrx_timequeue.s)
Format: AvrXTimerHandler();
This is an ISR for the CPU's timer that must be called within a _Prolog/_Epilog section.

MESSAGES and QUEUES

Acronyms:
MCB – Message Control Block. See avr.h
MessageQueue – Message queue, linked list. Block. See avr.h
See also AvrXStartTimerMessage().

The simplest message has no user data; the address of the MCB that the sender used is meaningful enough to the recipient. To add user data to a message, declare a structure containing an MCB then user data items. Message ACKs by the recipient are optional.

AvrXSendMessage (avrx_message.s)
Format: AvrXSendMessage(&MessageQueue, &MCB);
Places the MCB message on the MessageQueue.

AvrXIntSendMessage (avrx_message.s)
Format: AvrXIntSendMessage(&MessageQueue, &MCB);
This is the same as the previous AvrXSendMessage API except that it is run at the interrupt level, which means that it must be called from within a Epilog/_Prolog wrapper as in an interrupt handler.

AvrXWaitMessage (avrx_message.s)
Format: AvrXWaitMessage(&MessageQueue);
Wait until a message becomes available before returning an address of an MCB.

AvrXRecvMessage (avrx_recvmessage.s)
Format: AvrXRecvMessage(&MessageQueue);
Returns the first item (MCB address) from MessageQueue or a zero if the queue is empty.

AvrXAckMessage (avrx_semaphores.s)
Format: AvrXAckMessage(&MCB);
This allows handshaking to be done through message queues. The sequence would be:
AvrXSendMessage->AvrXWaitMessage->AvrXAckMessage- AvrXWaitMessageAck.

AvrXTestMessageAck (avrx_testsmaphore.s)
Format: AvrXTestMessageAck(&MCB);
Returns the current state of the MCB:

SEM_PEND	(0)	// Semaphore is reset waiting for a signal
SEM_DONE	(1)	// Semaphore has been triggered.
SEM_WAIT	(2)	// Any other value is the **address of a processID**

AvrXWaitMessageAck (avrx_semaphores.s)
Format: AvrXWaitMessageAck(&MCB);
Waits for a particular MCB to be returned.

EEPROM READ/WRITE

AvrX provides some EEPROM access routines that control access to the hardware via a semaphore. This semaphore needs to be "set" once at application startup prior to using the access routines.

```
extern Mutex EEPromMutex;

unsigned char AvrXReadEEProm(unsigned char *);  // one byte
unsigned int AvrXReadEEPromWord(unsigned *);    // 16 bit little-endian
void AvrXWriteEEProm(unsigned char *, char);    // one byte
```

SYSTEM OBJECTS (Internal use)

AvrXSetObjectSemaphore (avrx_semaphores.s)
Format: AvrXSetObjectSemaphore(&mutex);
Similar to AvrXSetSemaphore except that it applies to System Objects.

AvrXIntSetObjectSemaphore (avrx_semaphores.s)
Format: AvrXIntSetObjectSemaphore(&mutex);
Same as above except that it assumes the code is running at the interrupt level.

AvrXWaitObjectSemaphore (avrx_semaphores.s)
Format: AvrXWaitObjectSemaphore(&mutex);
A blocking API which waits on the semaphore in a System Object until it becomes SEM_DONE.

AvrXResetObjectSemaphore (avrx_resetsemaphore.s)
Format: AvrXResetObjectSemaphore(&mutex);
Forces the semaphore to the SEM_PEND state.

AvrXTestObjectSemaphore (avrx_testsemaphore.s)
Format: AvrXTestObjectSemaphore(&mutex);
A non-blocking API that returns the state of the semaphore

DEBUG TOOLS

AvrXStepNext (avrx_singlestep.s)
Format: AvrXStepNext();
Unsuspends a task, adds it to the run queue, then resuspends the task.

AvrXSingleStepNext (avrx_singlestep.s)
Format: AvrXSingleStepNext();
Marks a suspended task for single step support Jams it on the front of the run queue. Returns error if task is not suspended.

OTHER

BeginCritical (avrx.h)
Format: BeginCritical;
This is a simple #define for the asm "cli" command to turn off interrupts.

EndCritical (avrx.h)
Format: EndCritical;
This is a simple #define for the asm "sei" command to enable interrupts.

IntProlog (avrx_tasking.s)
Format: IntProlog();
Pushes entire register context onto the stack, returning a frame pointer to the saved context. If running in user mode upon entry (SysLevel == 0xFF) then switches the stack to the kernel and stores the frame pointer in the current processes PID. Internal use or in ISRs.

Epilog (avrx_tasking.s)
Format: Epilog();
Restore previous context (kernel or user). If task has SingleStep flag set, then generate an interrupt before returning to the task. Internal use or in ISRs.

AvrXSetKernelStack (avrx_tasking.s)
Format: AvrXSetKernelStack(&stack);
Sets AvrX Stack to "newstack" or, if NULL then to the current stack

AvrXHalt (avrx_halt.s)
Format: AvrXHalt;
Effectively halts the processor by disabling interrupts and then doing a "branch self".

SERIAL UART

For WinAVR, this goes in top of your main() for using printf() et al.

```
      initTimer0(); // see timerISR.c
      // uart interrupts go to AvrXBufferedSerial.c  see this .h
      InitSerial0(BAUD(baudrate));  // See BAUD macro.
      fdevopen(put_char0, get_c0);  // Set up standard I/O via UART 0
      // for printf() and printf_P() etc.
```

AvrXBufferedSerial.c uses FIFOs (see next section).

InitSerialIO (serialio.s)
Format: InitSerialIO(baud_rate_value);
Initializes the USART and sets up the baud rate generator

// more often, C library calls are used rather than these:
PrintString (serialio.s)
Format: PrintString(&FlashString);
Prints the string to the USART; returns a pointer to the next character after the NULL.

PushChar (serialio.s)
Format: PushChar();
Set a flag so that the next call to GetChar just returns. In effect pushing the last returned charactor to be retrieved by the next routine.

Other obvious serial commands:
GetChar();
PutCR();
PutSpace();
PutChar(char);

FIFOs (first-in-first-out buffers)

Notes on declaration & usage of AvrXFifo's:

Maximum fifo size is 255 bytes. Most practical systems only need a few bytes to buffer interrupt handlers or protocols (e.g. large enough for the biggest outgoing packet, for example). If this limit is too small, see "Future Expansion" below.

The user can ***declare and initialize*** Fifo's explicitly, in code, or use some handy #define MACROs to do the job. To do this manually follow this template:

Outside of your code (e.g. global data memory)

```
        uint8_t SomeBuffer[sizeof(AvrXFifo) + DesiredBufferSize];
        static const pAvrXFifo FifoName = (pAvrXFifo)SomeBuffer;
```

Or use the macro

```
        AVRX_DECL_FIFO(FifoName, FifoSize);
```

or if external

```
        extern uint8_t SomeBuffer[];
        static const pAvrXFifo FifoName = (pAvrXFifo)SomeBuffer;
```

Or use the macro

```
        AVRX_EXT_FIFO(FifoName);
```

In your code, ***initialize the fifo***:

```
        FifoName->size = DesiredBufferSize;
        AvrXFlushFifo(FifoName);
```

Or use the macro

```
        AVRX_INIT_FIFO(FifoName);
```

AvrX API calls To use the fifo.
"FifoName" below is a pointer. See `AVRX_DECL_FIFO(FifoName, FifoSize);`

```
int retc = AvrXPutFifo(FifoName, data);// Put data in fifo (see Return Values)
int foo = AvrXPullFifo(FifoName); // Get data from fifo (see Return Values)
AvrXWaitPutFifo(FifoName, data);  // Place data in fifo, blocks if full
int foo = AvrXWaitPullFifo(FifoName);  // Get data from fifo, blocks if empty
int ch = AvrXPeekFifo(FifoName); // Return next-out of FIFO or FIFO_ERR
int size = AvrXStatFifo(FifoName); // get number of bytes in FIFO
AvrXFlushFifo(FifoName);          // Flush fifo & release producer.
int foo = AvrXDelFifo(FifoName);  // Remove last item placed in fifo
```

Return Values:

```
FIFO_ERR (-1)   fifo full or empty and operation can't be completed
FIFO_OK  (0)    operation successful
unsigned int    Char data, zero extended for successful removal
```

GCC produces poor [inefficient] AVR code when comparing to literal 0 (there are reasons why this is correct behavior). GCC generates good AVR code when testing results for < 0. e.g.

```
if ((c = AvrXPullFifo(FifoName)) < FIFO_OK)   // rather than ==
 // empty...
else
// Do something with c which is unsigned char data.
```

Because the pointer to the FIFO is declared as a static const value, no actual storage is used. The compiler is smart enough to just use the address of the byte buffer when referencing the FIFO. Optimal code is generated.

Macros (for FIFOs)

Macros hide the need to record the size for initialization, otherwise are identical to the explicit C code. The only caveat is that the init macro needs to be in the same file as the declaration macro. This is also good programming practice to keep declarations and initialization in one file. Also the External declaration macro cannot be used in the same file as the declaration macro.

Synchronization:

AvrXFifo's have built in synchronization between producers (put) and consumers (pull). Producers and consumers can be either tasks or interrupt handlers. Interrupt handlers, of course, cannot use the blocking FIFO API. When blocked, the producer/consumer will be come unblocked when the opposite side either removes or adds a data item (respectively). There is no mechanism to notify an interrupt handler that something has been added or removed from a FIFO. See the example of the buffered serial I/O to see how that is handled. In short, it is up to the task end of the FIFO to manage coordination with the interrupt handler.

Future expansion possibilities:

By using the FIFO code as a starting point some interesting things could be added:

1. FIFO could be defined as word or even arbitrary data size buffers. The size could be declared as multiples of the data and new get/put routines defined that remove the data item. This could be made generic by adding the element size to the FIFO data structure, at the expense of more code for the simple case.

2. Routine addresses could be added to the FIFO structure as call-backs for interrupt handlers when the FIFO fills or empties. This would allow a FIFO to be put between two interrupt handlers. Say a serial to serial buffer. Simple tasks between handlers could be written as entirely kernel context code avoiding the overhead of task switching.

3. FIFO size could be expanded beyond 255 by changing the internal size, in and out from uint8_t to uint16_t and modifying the FIFO code appropriately (internal temporary values). Also, race conditions in loading pointers must be taken into account. With byte pointers CPU reads are atomic and no critical sections are needed. The current byte size data was chosen as a compromise between small system code and decent sized buffers.

CAVEATS:

The FIFO code has not been tested with a buffer size of 255. It should work. There might be race conditions

Theory:

AvrX is a deterministic priority driven fully pre-emptable task scheduler. Boy that was a mouthful.

Terminology

- PID - Process ID, small data structure that contains all information needed to execute a process.

Task Switching

The bulk of the task switching logic is in the two routines _Prolog and _Epilog. There are three data structures that are used:

_RunQueue	Pointer to the first PID in the run queue
_Running	Pointer to the current executing PID
_SysLevel	Count of number of time the kernel has been entered, -1=process stack

_SysLevel indicates whether the system is running user code or kernel code. When _SysLevel = -1 the current stack is a task stack. When an interrupt occurs, or the user code calls a kernel API, _Prolog save the current context on the stack. Then _Prolog increments _SysLevel and then switches to the kernel stack if _SysLevel == 0. Subsequent interrupts, or calls to _Prolog will stack the context on the kernel stack and increment _SysLevel. Switching to the kernel stack involves reading the current SP and storing it in the PID pointed to by _Running. Then the SP is loaded with the top of the kernel stack.

_Epilog unwinds the kernel stack by popping off the previous context, decrementing _SysLevel as it goes, when _SysLevel goes negative it has to swap back to the Process stack. This is when a context switch actually occurs: If _Running == _RunQueue, that means no context switch has been requested and _Epilog simply restores the registers saved by _Epilog (pointed to by _RunQueue). If _Running and _RunQueue are different, that means that a higher priority task became ready to run (either by being queued, or by the current task being blocked e.g. being removed from the _RunQueue). In this case _Epilog, again, just restores the context pointed to by the top of the _RunQueue. The third case is if _RunQueue == 0, or is empty. That means all tasks are blocked and the CPU is idle. In this case _Epilog goes to a special task called the IdleTask. Currently all IdleTask does is put the CPU to sleep waiting for an interrupt.

The designer of a software system might choose to have a low priority task that never blocks and call that the Idle Task. It could halt the CPU or do whatever you want. As long as it never blocks, the internal IdleTask will never get executed.

Queue

When processes are ready to execute, they are placed upon the _RunQueue in inverted priority order. Priority 0 will be inserted at the head of the queue. If there are multiple processes with the same priority, subsequent ones will be inserted behind all other equal processes. As processes run and block they will be re-queued at the end of the like priority task list and effectively be round robin scheduled.

A lower priority process can never interrupt a higher priority process. _Epilog simply looks to the head of the _RunQueue to pick a new process to run. However, if a lower priority process owns a resource needed by the higher priority process, say a semaphore being used to control access to hardware, then the higher priority process will block until the resource have been made available.

Once a process is running, it will run until it is either pre-empted by a higher priority process, or it must block, waiting for some resource.

Suspending a process

Because a "running" process may be blocked on a semaphore, and semaphores are not "known" to the kernel, suspending a process is not as simple as removing it from the Run Queue. AvrXSuspend marks a process for suspension and then attempts to remove it from the run queue. If it succeeds, it then marks it "SUSPENDED" and returns. If it is not successful, it simply returns. Later, when the process becomes eligible to be inserted back into the Run Queue if it is marked for suspension, the procedure _QueuePid will not queue it and will mark it SUSPENDED. By definition, when _QueuePid attempts to put a task on the run queue the task cannot be waiting for anything and the semaphore associated with it will be cleared.

Blocked Tasks

A task is blocked when it is either suspended or queued onto a semaphore. In other words, *not* queued on the RunQueue. Because AvrX is intended for small systems, only a forward linked list is maintained. This conserves SRAM by only needing one pointer. Whenever inserting or deleting something from a queue, AvrX walks the queue to find the object being removed or the insertion point. In general all queues (RunQueue, TimerQueue, Semaphores, MessageQueues and Messages) can have multiple items queued up waiting. In this way a semaphore becomes a Mutex: if it is owned by a process, all other processes will queue up waiting for the owner to "Set" the semaphore, thus releasing the head of the queue. Message queues can have multiple messages waiting for a receiver, or, multiple receivers (tasks) waiting for a message. The RunQueue will have however many tasks are ready to execute, waiting their turn, queued. As tasks block, or exit or are suspended, the next item in the RunQueue is then swapped into the CPU and things continue.

Timer Queue

The timer queue is a special case. It implements a sorted list of adjusted timeouts such that the interrupt handler is only decrementing the first element at all times. When it goes to zero, the code then signals the semaphore inside the TimerQueue element. If a task is waiting on that semaphore, then it is placed onto the RunQueue. There is a further special case of TimerMessages, where instead of signaling the task, the TimerControlBlock is queued onto a message queue

The TimerHandler could have been implemented as a stand-alone task and timers could have been implemented as messages. This actually would be pretty powerful but was not done because of the expense of two context switches (one for the timer handler task and one for whatever task becomes available to run) and the expense of another task context (~40 bytes of sram). AvrX runs three to four tasks comfortably in 512 bytes of SRAM: using one of those tasks for the timer handler seemed too wasteful.

Fifo Support

AvrXFifo support are simple byte oriented FIFOs. FIFOs are allocated as byte arrays with a const pointer of the correct type cast to the array. FIFOs can be static or automatic, or, allocated from the heap. FIFOs include semaphores to implement synchornization between tasks or between interrupt handlers and tasks. The AvrXSerialIO sample code illustrates how this can be used.

Getting Started

The easiest way to get started is to build the kernel with one of the samples or test cases. After that has been verified to work, strip the file down, rename it and add your own code.

The top level file, in all but the most trivial cases, like the test cases, should contain the at least the following sections:

- AvrXTimerHandler interrupt handler
- One or more internal or external TASK definitions
- CPU_Reset routine or main(void)

The last item needs to perform any tasks necessary to prepare the system for running. These are at least the following items, of which the first three are done for you by the C compiler runtime:

- Set the hardware Stack Pointer
- Clear SRAM
- Clear Registers
- Initialize tasks structures (AvrXInitTask or AvrXRunTask)
- Initialize hardware (Timer0 or 1, ports, serialio, whatever.)
- Jump to the routine Epilog().

The last instruction in the startup code needs to be a jump to Epilog. That will start the scheduling by switching the running context to the first item on the _RunQueue. If the monitor is included in your program, it should have the highest priority (0) and will be the first thing to run.

Typically user code would reside in separate files and just the Task Control Blocks (TCB) would be imported into the top level file. For AvrX 2.6 the C macro AVRX_EXTERNALTASK does the importing. Please refer to the header files (avrx.inc or avrx.h) for specific details on how each macro works and where they apply.

The startup code runs under the stack set up by C runtime, or where ever you place it, in the case of assembly. This location is essentially the stack that AvrX will use as the kernel stack. So, everything done in the main() or reset routine is considered to be running in the kernel context. At least one task needs to be prepared to run with AvrXRunTask() before exiting the reset code. If no tasks have been prepared, Epilog() will find the RunQueue empty and will enter the idle task permanently.

Although I have not done this, it would be possible to start up in the idle mode and have an interrupt handler "AvrXRunTask() a task. A more reasonable situation would be to run all your tasks and have them do whatever initialization they require and then block on something (timer, semaphore or message)

Optional Stuff

At a minimum, AvrX is simply task initialization and semaphore support. Single Step, Timer Queue Management and Message Queue Management are all optional services that can be left out if not needed. The resulting kernel is quite small without these services. Here are rough sizing for various kernel functions

Basic tasking and Semaphore Queue Management	~670 bytes
Time Queue Manager:	~236 bytes
Message Queue Manager:	48 bytes
Miscellaneous (Debug Singe Step, Advanced Tasking):	200 bytes
Debug Monitor	~1300 bytes.
Fifo support (written in C)	~300 bytes

Without the debug monitor the total size of AvrX for GCC is only ~600 words (1200 bytes).

There is no fundamental reason that timer support has to be included in your AvrX application. The timer Queue Manager is just one implementation of a mechanism to allow multiple competing tasks to schedule time delays. For simple applications one might simply have an Real Time Clock interrupt signal a semaphore and have a process that encapsulates all the time dependent stuff. With every tick, the process will run, do work and, optionally, set other semaphores to signal other processes that it is time to do work. Alternatively, it could send messages. If your timing requirements are modest, a simple task, or even interrupt routine, might well be more efficient in both code space and processor cycles.

The reason that Message Queue Manager is so small is that many functions are already provided by the basic tasking/semaphore module. Message queues are a really simple concept that derives from the power of queueing semaphores.

Structure of a task

Although not absolutely necessary, a task typically is a routine with an entry, some initialization and then an endless loop. The endless loop typically involves blocking, or waiting, on a semaphore. That might be explicitly as in the case of AvrXWaitSemaphore, or it might be implicit in the case of AvrXWaitTimer or AvrXWaitMessage. These last two items actually bock on a semaphore embedded in the timer or message data structure.

There are some data structures that need to be defined along with the code. Most of the work can be avoided by using handy macros defined in AvrX.inc or AvrX.h, depending upon the version being used.

Below is a simple example of a task that simply blocks on a timer and signals a semaphore. This is code for AvrX 2.5, the GCC compiler.

```
Mutex Timeout;

AVRX_TASKDEF(myTask, 10, 3)
{
    TimerControlBlock MyTimer;

    while (1)
    {
        AvrXDelay(&MyTimer, 10); // 10ms delay
        AvrXSetSemaphore(&Timeout);
    }
}
```

The macro, AVRX_TASKDEF, takes three arguments: the task (procedure) name, the additional stack required above the 35 bytes used for the standard context, and the priority. It builds all required AvrX data structures and declares the C task procedure. Please refer to the file AvrX.h for details.

Structure of an Interrupt Handler

Interrupt handlers have to have a specific name associated with them for the GCC compiler to set the appropriate interrupt vector name. Look at the avr-gcc file sig-avr.h for a list of possible vectors. AvrX completely handles the saving and restoring of the interrupted context so you DON'T want to use the GCC procedure qualifiers SIGNAL or INTERRUPT. Instead use the following:

```
AVRX_SIGINT(SIG_OVERFLOW0)
{
    IntProlog();             // Switch to kernel stack/context
    EndCriticalSection();    // Re-enable interrupts
    outp(TCNT0_INIT, TCNT0); // Reset timer overflow count
    AvrXTimerHandler();      // Call Time queue manager
    Epilog();                // Return to tasks
}
```

IntProlog() does not re-enable the interrupts in an interrupt handler (this is *different* from AvrX 2.3) so if you want to be able to nest interrupts you need to explicitly enable them in your handler. Also, beware, some sources of interrupts are not cleared by servicing the interrupt handler, so you need to clear them BEFORE enabling interrupts or you will endlessly re-enter your code and blow your stack. The serial UART handlers are a good example of this. Check the serial I/O files to see how this is handled.

AvrX 2.3 enables interrupts by default in IntProlog(), so you need to clear any pending interrupts BEFORE calling IntProlog().

Structure of your main() code:

The main() code simply initializes hardware and any tasks and then jumps to Epilog(). Here is the main() for the sample file "messages.c" (Please refer to the actual sample as coding styles have changed since this was originally written)

```
void main(void)                // Main runs under the AvrX Stack
{
    outp((1<<SE) , MCUCR);     // Enable "Sleep" instruction for idle
loop

    outp(TCNT0_INIT, TCNT0);   // TCNT0_INIT defined in "hardware.h"
    outp(TMC8_CK256 , TCCR0);  // Set up Timer0 for CLK/256 rate
    outp((1<<TOIE0), TIMSK);   // Enable0 Timer overflow interrupt

    outp(-1, LED-1);           // Make PORTB output and
    outp(-1, LED);             // drive high (LEDs off)

    AvrXRunTask(TCB(task1));
    AvrXRunTask(TCB(task2));
    AvrXRunTask(TCB(Monitor));

    InitSerialIO(UBRR_INIT);    // Initialize USART baud rate generator

    Epilog();                   // Switch from AvrX Stack to first task
}
```

Determining the size of various stacks

For AvrX 2.6 the task stack needs to be 35 bytes + any additional stack needed by the task code. The C compiler makes pretty heavy use of the stack when calling procedures. It just depends upon how many automatic variables are used in each procedure. The best way to determine proper stack sizing is to allocate lots of stack (say, 70 bytes) and run your application for a while to see how deep the stack gets. Since GCC zeros all memory during startup it is pretty easy to tell. However, to be safe, use the emulator or debug monitor to write a few words of 0xFFFF near the perceived end of the stack to verify exactly how deep it gets. Then, allocate a couple more bytes (at least one or two words) extra to insure you don't run over onto another stack or data structure.

Determining the kernel stack size is a little easier. AvrX doesn't consume much stack in it's normal operation. Perhaps 4-6 bytes at the most. However, the kernel is re-entrant and will stack a full context for each interrupt that is nested. So, you need to multiply the total number of interrupt sources (assuming they are all active at all times) by 35 and add any additional stack used by the interrupt code + 4-6 bytes used by AvrX to get a rough idea of the total stack needed. Of, course, if your application has an interrupt that *doesn't* use AvrX (e.g. no IntProlog()/Epilog()) then it only stacks whatever it uses onto the kernel stack or the user stack. In either case, use the above procedure to determine the exact size: allocate extra space, run your application for a while (exercising the interrupt sources) and see just how deep the stack gets. You might have to deduce the maximum possible depth as some interrupt sources might not happen often enough to cover all possible cases. For high speed stuff (e.g. serial link dumping lots of data, basic timer for the clock) usually all possible combinations will be covered within a short while.