

Dr. Tak's Real Time Kernel (DTRTK) for the Atmel AVR MCUs

Tak Auyeung, Ph.D.

October 25, 2008

1 Introduction

A Real Time Kernel (RTK) is a miniature operating system for very small systems, usually running a monolithic application. How small? The RTK described by this document fits in less than 1.5kB of code on an AVR MCU. What can 1.5kB of code do? Read on and you'll find out shortly!

2 What does an RTK do?

An RTK should perform a few functions. This section attempts to describe what an RTK should do. Before we dig into technical details, we will first describe the terms used in later section.

2.1 Terms

Process. A process is a 'program in execution'. This term is more useful in desktop computers or computers that can run multiple applications concurrently. Two key concept to remember: a process is an *instance* of a program, and that two processes of the same program do not share any variables.

Most MCUs run a single process.

Thread. A process has at least one thread. A thread is the 'context' that is necessary to execute code sequentially as specified by a program's source code. A process can have multiple threads, but all these threads must share all variables and data structures that are either statically allocated or allocated via dynamic memory allocation. On the other hand, threads do *not* share any variables or objects that are stack based.

In *most* implementations, every thread has its own stack. Exceptions include the real-time kernel (RTK.lib) published by Z-World, Inc. at least up to 1998. The author has not checked the latest version of the real-time kernel from Z-World since 1998.

2.2 Multithreading

There is not much for a kernel to do if a program only executes as a single thread. A minimum requirement of an RTK is to somehow support multithreading. Multithreading is the management of resources such that a program appears to execute more than one sequences of code 'concurrently'.

We all know that a processor (especially one with only one ALU) cannot truly perform operations in parallel. So what is the point to "pretend" parallelism? The answer is quite simple: to make coding easier. It is the same argument to write programs in high level lagnagues instead of machine code, even though everything must be translated to machine code before a processor can execute the code.

Let us consider an program that runs on a robotic system. To ensure reliability, it is usual that we have the following tasks to perform:

- Maintain communication with an human interface (remote terminal).
- Motion control.
- Path planning.
- Sensor reading.
- Sensor interpretation.

- Self monitoring and diagnostics.

Each of these tasks is, by itself, a sequence of operations. For example, the overall code to maintain communication is as follows:

```
Initialize
while true do
  wait for query
  interpret query
  formulate reply
  transmit reply
end while
```

Likewise, the overall code to plan a path may appear as follows:

```
initialize
while true do
  wait for change of environment or destination
  recompute best course to reach destination
  use new course
end while
```

As you can see, each individual task is an infinite loop. This makes sense because there is no logical end to each task. Without multithreading, it becomes difficult to translate the pseudocode to actual code. This is because there is no way to express the fact that these individual loops be performed “in parallel”.

With multithreading, each logical task is implemented by a top level subroutine. The top level subroutines can call other subroutines. In other words, we write the following functions for our example:

```
void planPath(void)
{
  while (1)
  {
    // ...
  }
}

void communicate(void)
{
  while (1)
  {
    // ...
  }
}
```

2.3 Round Robin

Round robin describes the behavior of threads sharing the same processor in a circular fashion. Imagine all the threads sit around a round table, and the processor is a server circling the table to serve each thread.

In most round robin implementations, a thread has the ability to voluntarily give up control, so that the next thread has a chance to execute. In our example, we may have the following code in the communication subroutine:

```
while there is in query do
  yield
end while
```

In other words, each time the communication thread has a chance to execute, it checks to see if there is a query to reply. If not, it immediately yield control to the next available thread.

Round robin scheduling assuming threads give up control voluntarily is easy to implement. The costatements in Z-World’s Dynamic C is based on such an assumption. This assumption is useful for threads that do not have hard real-time deadlines. For example, communication with an operator often falls into this category. If it takes a controller half a second to reply, the operator probably will not consider it as a failure.

On the other hand, for operations with “hard real-time constraints”, relying on peer threads to voluntarily give up control is not an option. Consider the control logic for a gripper that catches falling objects. A 50ms delay can lead to failure. We do not want to rely on peer threads to give up control in this case, because a poorly written peer thread may not give up control quickly enough.

This is the limitation of any implementation based on the assumption of threads know when it should give up control. Such implementations are also called “cooperative multithreading” and “cooperative multitasking”.

2.4 Preemptive Thread Change

Multithreading and Round Robin do not imply preemption. In other words, threads may voluntarily ‘yield’ the processor to the next one in certain Round Robin multithreading schemes. A *good*¹ RTK should also allow preemption of threads.

Preemption is the act of an RTK forcing a thread to give up processor resources and passing processor resources to the next thread. This is quite useful for threads that are compute-bound.

Let us consider our example thread that computes the best path to a destination. Most algorithms for such a task do not have “convenient” points to yield. In order for other peer threads to get control, the kernel must force the algorithm to give up control.

For any multithreading implementations using multiple stacks, preemption is relatively easy to add. The easiest way to add preemption is to add the logic to yield in a timer interrupt service routine (ISR). As a timer interrupts, the ISR automatically yields the interrupted thread, and give control to the next peer thread.

This can be done as follows:

- push two more bytes at the entry point of timer ISR
- initialize those two bytes to be the address of the yield subroutine, or to a dummy address with a single `ret` instruction if no preemption should occur

This makes the timer ISR “returns from interrupt” to the yield subroutine, which performs two actions. First, the `reti` instruction reenables interrupt. Second, the “yield” subroutine saves the context (including the return address to the interrupted thread) and switches to the next thread.

2.5 Priority Queues

Instead of having just one round table, an RTK supporting priority queues has multiple round tables. The server (processor) always serves the ‘high priority’ round tables first. If the highest priority round table is empty, the server then moves to the second highest priority round table. The process is repeated until the processor finds a round table with a thread awaiting service.

If all round tables are empty, the processor should monitor the tables so that it can start to serve whenever a thread arrives at a table.

Priority queues are useful when you have things that must be done on time and others that can ‘give or take a little’.

Many cheap² RTKs do not support Round Robin and priority queues together. Some RTKs only allows *one* thread per priority.

In our example, operator communication needs the lowest priority because answering a query late has the least effect on the overall behavior of a robot. Motion control and sensor reading, on the other hand, should be in the highest priority round table because both have hard real-time constraints.

2.6 Time Based Scheduling

The *real time* part of a Real Time Kernel means the kernel is fully aware of the passage of time. In addition, an RTK should also be able to use time to determine which thread to execute. The most basic form of real-time support is to allow a thread be activate at a certain time.

This is useful for operations that only needs to occur at a frequency that is much lower than the timer frequency (in the case of preemptive multithreading) or the natural frequency of cooperative multithreading. For instance, we may only need to perform a self-diagnostic check every minute. The logic of this thread may be as follows:

```
while true do
```

¹‘Good’ is vague, ‘competitive’ or ‘worth paying money for’ may be better adjectives.

²Again, ‘cheap’ is vague. Try half-baked, poorly planned and/or written by non-programmers.

```
    perform self diagnostics
    sleep for one minute
end while
```

The ability to schedule a thread to wake up later saves processing resources because the kernel can compare an internal counter value to the scheduled time very quickly.

2.7 Time Slicing

Time slicing is a natural product when an RTK supports round robin multithreading and preemption. Time slicing typically relies on software events that occur periodically (such as timer interrupts) to provide a timing basis. When a simple logic detects that a thread has been ‘hogging’ the processor for more than a certain amount of time, it invokes the preemption logic to take control from the ‘hogging’ thread and let the next thread start/restart execution.

Time slicing ensures that even if threads do not voluntarily yield to others, processor resources will still be fairly distributed among threads of the same priority.

2.8 Inter-thread Synchronization

Just because threads do share all static variables and objects that are dynamically allocated, you should not assume it is easy to synchronize threads that are executing in parallel. Once preemption is supported, thread synchronization becomes very difficult with just global variables. When inter-thread synchronization is poorly planned or designed, bugs tend to pop up randomly. At random times a program crashes with random symptoms. This makes debugging extremely difficult. When an RTK provides well thought-out and safe inter-thread synchronization, the application programmer can save much development time while producing reliable application software.

One common method to synchronize is through the use of semaphores. In the context of operating system concepts, a semaphore is a “magic” counter that can be initialized, decremented (P) or incremented (V). Conceptually, this is what happens when a semaphore is decremented:

```
if semaphore is zero then
    block thread until semaphore is greater than zero
end if
decrement semaphore value
```

The increment operation does not have any conceptual specialization, it simply increases the value of a semaphore (by one).

In implementation, however, an efficient kernel uses a queue for each semaphore. This way, the “block” operation translates to adding a thread to the tail of a wait queue. As the value of a semaphore increments to non-zero, the thread at the head of the queue is removed and it becomes active again.

It is easy to use a semaphore to ensure mutual exclusion (mutex) of a piece of code. A semaphore is initialized to have a value of 1. The code that needs to be protected (from multiple thread invocation) looks like the following:

```
decrement value of mutex semaphore
perform protected code
increment value of mutex semaphore
```

This way, if one thread is already executing the protected code, all other threads cannot gain entry because the semaphore value is already zero (or less). However, as soon as the thread that successfully execute the code exits the protected code, the semaphore value is incremented, and one other waiting thread gains access to the protected code.

3 Files to Compile and Link

3.1 timer.S

Link with the object code produced by this file for timer-based behavior (for scheduling and time slicing).

3.2 rtk.S

Link with the object code produced by this file for all the real-time kernel functions.

3.3 main.make

This is the main makefile. You need to write a simple makefile for your application that includes this main makefile. Refer to the example section for more details.

4 Downloading and Installing

You can download a snapshot of the related files [here](#).

After you download the distribution file, run the following command in the directory that you want the files installed:

```
tar xzvf rtk.tgz
```

Note that you will need to have the following packages installed to use this RTK:

- `binutils-avr`: binary utilities for the AVR architecture.
- `gcc-avr` (or `avr-gcc`): the GNU C Compiler for the AVR architecture.
- `avr-libc`: standard C libraries for the AVR architecture.

Because `rtk.S` (note the upper-case 'S') is different from `rtk.s` (lower-case 's'), this project can only be used in an OS that is filename case-sensitive. This rules out all DOS or Windows platforms. Even with `winavr` installed, the make utility program still cannot properly process the assembly source files.

5 Important Application Dependent Macros

Several macros are utilized throughout the source files for application dependent RTK configuration. This section describes these macros. In most cases, these macros should be defined in a make file so they are specified as options to compile all object files.

5.1 RTK_NUMQUEUES

This macro defines the number of priority queues. If not defined, it is defined to 4. To override the default, use the `-D` option of the command-line to predefine the macro. For example, if an application requires 2 priority queue, use the following option (as a part to the `avr-gcc` command):

```
-DRTK_NUMQUEUES=2
```

5.2 ISR_FREQ

This macro defines the frequency of the ISR responsible to provide a timing basis for the kernel. It should be defined by the `-D` option in a makefile.

This macro is only useful if `timer.S` is used.

5.3 TICK_FREQ

A macro defined to indicate the kernel tick frequency. Note that this frequency can be lower than the interrupt frequency. This should be defined by the `-D` option in a makefile.

This macro is only useful if `timer.S` is used.

5.4 TIMER_ISR

This macro defines the name of the ISR. For example to use the output compare 2 interrupt of the ATmega128, add this as a compile or assemble option:

```
-DTIMER_ISR=SIG_OUTPUT_COMPARE2
```

Use the same naming convention as the signal names.

This macro is only useful if `timer.S` is used.

5.5 `TIMER_RTK_PREEMPT`

When this macro is defined, the timer ISR defined in `timer.S` includes code to perform time slicing.

This macro is only useful if `timer.S` is used.

5.6 `TIMER_APP_CALL`

If this macro is defined, it is the name of an application defined function that attaches to the timer ISR. If this macro is not defined, `timer.S` assumes there is no application defined logic to execute in the timer ISR. Note that the application defined function executes at the same frequency as the ISR itself. Also, the application defined function should be marked as a `signal` as follows:

```
void __attribute__((signal)) timerAppCall(void)
```

This ensures that the registers are saved and restored correctly.

The matching definition in the makefile should look like the following:

```
-DTIMER_APP_CALL=timerAppCall
```

This macro is only useful if `timer.S` is used.

6 The API

This RTK implementation includes all the features describe in the concept section. This section describes the API (application program interface) that application programmers should use.

6.1 `struct Thread`

This is a type (very similar to a class in C++) that defines objects that represent threads.

6.2 `struct Queue`

This is a type that defines objects that represent queues of threads. (In other words, `struct Queue` is a plan to make a round table.)

6.3 `extern struct Queue rtkQueues[RTK_NUMQUEUES]`

This is an array of queues. The first item represents the queue of threads of the highest priority. `RTK_NUMQUEUE` is a macro definition of the total number of levels of priority. It is defined to 4 by default.

6.4 `unsigned char rtkSliceScaler`

This is an unsigned character (8-bit value) that specifies how many kernel ticks should happen before the time slicer forces a context switch. Initialize this before the timer is activated.

If this variable is not initialized, but `TIMER_RTK_PREEMPT` is defined and `timer.S` is used, the default number of RTK ticks is 256 (because of 0 wraparound). In other words, if an application does not initialize this variable, a thread is forced to yield every 256 RTK ticks.

De not define `TIMER_RTK_PREEMPT` (see subsection 5.5 if you do not want to have time slicing).

6.5 `unsigned char rtkSliceCounter`

This is the count down counter that tracks the number of remaining kernel ticks to the next context switch. It should be initialized to the same value as `rtkSliceScaler`. Initialize this before the timer is activated.

If this variable is not initialized, the first preemption occurs 256 RTK ticks from the enabling of the timer interrupts.

6.6 void rtkInitialize(struct Thread *t, struct Queue *q)

This function (similar to a method in C++) allows an application to initialize the RTK. The initial thread (whoever calls `rtkInitialize`) must declare an `struct Thread` object and pass its address as the first parameter `t`. The second parameter, `q`, is the address of the priority queue that the initial thread belongs to.

Do not call another other `rtkThread...` functions before calling `rtkInitialize`.

6.7 void rtkThreadAdd(struct Thread *t, struct Queue *q, void *SP, void (*start)(void *), void *param)

This function adds threads in addition to the one initialized by `rtkInitialize`. Parameter `t` is a pointer to an `struct Thread` object. Each thread must have its own *unique* `struct Thread` object. Parameter `q` is the priority queue to which this thread belongs to. `SP` points to the starting point of the stack for this thread. Parameter `start` is a function that marks the starting point of this new thread. `param` is a void pointer that is ‘passed’ to the parameter of ‘start’. This allows different instances of the same thread function receive different parameters.

If you declare the stack for this thread as `char stack1[64]`, you should pass `stack1+sizeof(stack1)` as parameter `SP`.

Note that this call may immediately cause the calling thread to yield to the new thread if the new thread has a higher priority.

6.8 void rtkThreadSuicide(void)

This function does not return, it kills the current thread and let the RTK figures out which other thread should get control next.

6.9 void rtkThreadKill(struct Thread* thread, struct Queue* queue)

This function removes a thread from a particular queue. The caller *must* correctly identify which queue to remove the thread from.

6.10 struct Tick

A ‘tick’ is the smallest time unit for an RTK. In other words, all time must be specified as whole multiples of ticks. The current implementation assigns 6 bytes for counting ticks.

6.11 struct Tick *rtkTickCountGet(struct Tick *t)

This function takes a snapshot of the RTK’s reference tick count and makes a copy to the object pointed to by parameter `t`. This function returns the same pointer as the parameter.

6.12 struct Tick *tickAdd4(struct Tick *t, unsigned long u4)

This function adds the value of an unsigned long integer (parameter `u4`) to the tick count of the `struct Tick` object pointed to by parameter `t`.

6.13 struct Thread *rtkCurrentThread

This is a pointer to the `struct Thread` object that represents the thread that is currently executing.

6.14 void rtkThreadSchedule(struct Thread *t, struct Tick *t)

This function schedules the thread represented by a `struct Thread` object pointed to by `t` to be reactivated at the time specified by the `struct Tick` object pointed to by `t`.

The thread being scheduled must be a thread that is ‘ready’ and not already ‘scheduled’ or locked by a semaphore (see later sections). You *can* use `rtkCurrentTask` as parameter `t`, which causes the calling thread becomes inactive until the specified time.

6.15 void rtkThreadYield(void)

This function allows a thread to give up the processor voluntarily.

6.16 void rtkSemaphoreInitialize(struct Semaphore *s, char initValue)

This function allows a semaphore (pointed to by `s`) be initialized to an initial value of `initValue`.

6.17 void rtkSemaphoreP(struct Semaphore *s)

This function decrements the value and requests the semaphore pointed to by `s`. If the semaphore is ‘not available’³, the call blocks the calling thread and places the calling thread onto a queue of awaiting threads of the semaphore.

6.18 void rtkSemaphoreV(struct Semaphore *s)

This function increments the value and releases the semaphore pointed to by `s`. If there are awaiting threads, the thread that has waited for the longest becomes ready again. This may cause the current thread be suspended so that a released thread of a higher priority may execute.

This function should not be used in an ISR because it triggers a context switch. If a context switch happens in an ISR, the stack space used by the ISR is not freed.

6.19 void rtkSemaphoreVNoCS(struct Semaphore *s)

This function is almost the same as `rtkSemaphoreV`, except that it does not cause any context switch even the operation unblocks a thread of a higher priority.

This function should be used instead of `rtkSemaphoreV` if an ISR is to unblock a waiting thread.

Because no context switch is immediately triggered by this function, it always returns normally. Then it is up to one of the following mechanisms to trigger a context switch to an unblocked thread:

- The current thread calls `rtkThreadYield()`. In this case, if the unblocked thread has a higher priority, the context switch follows the call immediately.
- The timer (that links to the RTK) interrupts. In this case, an unblocked thread needs to wait for the next timer interrupt.

6.20 int rtkSemaphoreTryP(struct Semaphore *s)

Similar to `rtkSemaphoreP`, execute this function returns 1 if the semaphore is successfully requested, and it immediately returns 0 if the semaphore is not available. This function call does *not* block the calling thread.

7 Examples

7.1 onethread

This is one of the sample programs in the package. It is not very useful because all it contains is the skeleton of a program that initializes the real time kernel.

```
#include "rtk.h"

struct Thread mainThread;

int main(void)
{
    // ... code before rtk is active
    rtkInitialize(&mainThread, &rtkQueues[0]);
    // ... code after rtk is active
    // not exciting because with one thread, there isn't much you need to use
```

³If a semaphore has a value less than 1, it is not available.

```

// from the RTK
while (1)
{
}
}

```

The file `onethread.make` is the makefile for this program. It is very minimalist, also.

```

MMCU = -mmcu=atmega128
COMMONOPT = -g $(MMCU)
ASFLAGS = $(COMMONOPT) -D__ASM__
CFLAGS = $(COMMONOPT) -O3 # don't forget, otherwise defaulted to 8515!
      # the mmcu option also controls how the linker
# links (which linker script file to use)
LDFLAGS = -g $(MMCU)

MAIN_SRCFILE = onethread.c
C_SRCFILES = $(MAIN_SRCFILE)
ASM_SRCFILES = rtk.S

include main.make

```

The most critical part are explained here:

- `MMCU=-mmcu=atmega128`
This line specifies that our processor is a `atmega128`.
- `COMMONOPT = -g $(MMCU)`
This line specifies that the compiler compiles with debugger symbols included. It is useful for debugging purposes. It also specifies that the processor used is an `atmega128`.
- `CFLAGS = $(COMMONOPT) -O3`
This line specifies that the compiler use the common options as defined in the previous item. It also specifies to optimize the output of the compiler. `-O3` is a very high level of optimization.
- `LDFLAGS = -g $(MMCU)`
This lines specifies the linker options. It specifies to retain debugger symbols.
- `MAIN_SRCFILE = onethread.c`
This line defines the file that contains the `main` function (entry point of the whole program).
- `C_SRCFILES = $(MAIN_SRCFILE)`
This line defines *all* the C source files of the project. It refers to the `MAIN_SRCFILE` because that, too, is a source file written in C.
- `ASM_SRCFILES = rtk.S`
This line defines all the assembly source files of the project.
- `include main.make`
This line specifies that all other rules are defined in `main.make`.

7.2 onethread_sch

This program only uses one thread, but it is a bit different from the previous one.

```

#include <avr/io.h>
#include <avr/ina90.h>
#include <avr/signal.h>
#include "timer.h"
#include "rtk.h"

```

```

static struct Thread mainThread; // static because it doesn't need to
                                // leave this file

// initialize timer2 to interrupt at 5kHz, enable output compare interrupt
static void
setTimer2(void)
{
    output(TIMSK, (input(TIMSK) & 0x3f) | 0x80);
    output(TCNT2,0);
    output(OCR2, 49);
    output(TCCR2,0x0b);
}

// be sure to specify the following when you compile:
// -DTICKFREQ=1000 makes the tick frequency matches milliseconds
// -DISR_FREQ=5000 makes the system know that the ISR interrupts at 5kHz
// -DTIMER_ISR=SIG_OUTPUT_COMPARE2 makes the system know that the ISR is
//                               defined elsewhere
// this file needs to link with rtk.S and timer.S
// because rtk.S provides the RTK implementation, while
// timer.S provides the preemption ability and time tracking ability

int main(void)
{
    struct Tick wakeTime;
    // ... code before rtk is active
    rtkInitialize(&mainThread, &rtkQueues[0]);
    // ... code after rtk is active
    // let's start up the timer to provide timing
    DDRG |= 0x18;
    PORTG |= 0x18;
    setTimer2();
    _SEI(); // enable interrupts
    while (1)
    {
        PORTG ^= 0x18;
        // ... do something
        rtkTickCountGet(&wakeTime); // take a snapshot of the current time
        tickAdd4(&wakeTime, 100); // make the wake time 100 ticks from now
        // while (rtkTime < wakeTime) rtkThreadYield();
        rtkThreadSchedule(rtkCurrentThread, &wakeTime);
        // now wait 100 tick
        // when we "return" from rtkThreadSchedule, it's 100 ticks later
    }
}

```

This program sets up PORTG to control two LEDs. This part is only useful if two LEDs are connected to pin 3 and pin 4 of the MCU (via current limiting resistors). The main subroutine calls `setTimer2` to initialize timer2 to interrupt 5,000 times per second.

`_SEI()` enables global interrupt. This is necessary because when `main` gets control, global interrupt is disabled. Until `_SEI()` is executed, the timer does not interrupt.

The most critical parts of this program is explained as follows:

- `rtkTickCountGet(&wakeTime);`
This call takes a snapshot of the kernel tick count. The snapshot is stored in `wakeTime`, which is a variable of `struct Tick` type.

- `tickAdd4(&wakeTime, 100);`
This call adds a 32-bit unsigned integer to the tick count stored in `wakeTime`.
- `rtkThreadSchedule(rtkCurrentThread, &wakeTime);`
This call asks the real-time kernel to schedule the current thread (the only one in this program) at the tick count specified in `wakeTime`.

This sequence effectively schedules the thread to execute 100 kernel ticks later. One can imagine that `rtkThreadSchedule` does not return until 100 ticks has passed.

Now, let us take a look at the make file for this program.

```
MMCU = -mmcu=atmega128
COMMONOPT = -g $(MMCU) \
            -DTICK_FREQ=1000 -DISR_FREQ=5000 -DTIMER_ISR=SIG_OUTPUT_COMPARE2
ASFLAGS = $(COMMONOPT) -D__ASM__
CFLAGS = $(COMMONOPT) -O3 # don't forget, otherwise defaulted to 8515!
        # the mmcu option also controls how the linker
# links (which linker script file to use)
LDFLAGS = -g $(MMCU)

MAIN_SRCFILE = onethread_sch.c
C_SRCFILES = $(MAIN_SRCFILE)
ASM_SRCFILES = rtk.S timer.S

include main.make
```

The most notably differences of `COMMONOPT` between this make file and the previous one are listed as follows:

- `-DTICK_FREQ=1000`
This line specifies that the kernel tick frequency is 1kHz. In other words, one kernel tick is one millisecond.
- `-DISR_FREQ=5000`
This line specifies that the actual interrupt frequency is 5kHz. Note that the interrupt frequency is determined by how `timer2` is configured. If this definition does not match the actual frequency, then all timing basis will be changed.
- `-DTIMER_ISR=SIG_OUTPUT_COMPARE2`
This line specifies that the name of the timer interrupt routine is `SIG_OUTPUT_COMPARE2`. This macro is chosen because `setTimer2` configures the timer to interrupt whenever it overflows the output compare value.

In addition, this make file include `timer.S` as a modules. This is because `timer.S` provides the actual timer ISR code.

8 Advanced Topics

This section is about cool stuff that you may want to do with this RTK.

8.1 Interrupt Service Routines

Since the AVR core does not support a “protected” mode for ISRs, it is relatively difficult to interface ISRs to the RTK cleanly. We’ll look into a possible solution.

For our discussion, let use use the ISR(s) for an UART as examples.

8.1.1 ISR Philosophy

Keep it short. Because interrupt is disabled during the execution of an ISR, it is critical to keep an ISR short in terms of execution time. In the case of an UART, when a byte is received, we only want to do the following:

- grab the byte from the data registers

- stick it into a queue

It doesn't take long to do this.

Defer complexity to threads. But what if there is a lot of complex logic to process a byte just received? For example, checksum computation, address matching and etc. I suggestion is to do all that in a regular thread at top priority.

How does the ISR communicate with the handler thread? Use semaphores.

8.1.2 An Example

The pseudocode of an UART receiver interrupt is simplified as follows (not to handle error flags):

```
grab the byte
if fifo has space then
    add the byte to the fifo
    V the receiver fifo semaphore
else
    do nothing, maybe to set a flag to indicate the problem
end if
```

The matching handler thread may look like the following:

```
loop
    P the receiver fifo semphoare
    grab a byte from the fifo
    process it (can be leeeengthy)
end loop
```

8.1.3 Implementation

The pseudocode from the previous section looks good. It is clean and clearly separates the ISR from the handler thread. At the same time, the interface between the ISR and the handler thread is quite simple and clear.

There is one problem.

`rtkSemaphoreV` can potentially trigger a thread context change. If the thread being interrupted by the UART receive event has a lower priority than the handler thread, the call to `rtkSemaphoreV` initiates a context switch.

The problem is that the interrupted thread is now suspended half way through an ISR. The problem is not with the I-bit (which is cleared when the ISR gets control). Afterall, a context switch resumes the value of the I-bit. The problem is that potentially, we can have registers of half-finished ISR instances saved on the stack for all threads except those in one priority queue (the highest priority queue). This is a waste of space, although it is still "correct" given enough stack space, and that there is nothing else to do after `rtkSemaphoreV` in the ISR.

8.1.4 Tak's solution

My solution is somewhat ugly: resort to assembly programming. But wait, I have justifications:

- it is speed and stack space efficient
- it discourages overly complex ISRs
- did I mention efficiency?

Here's a quick review of how I do it:

```
SIG_UART_RECV: ; uart receiver ISR
push r31
in r31, __SFR_IO_ADDR(SREG) ; SREG
ori r31, 0x80 ; pre-reenable interrupt in the SAVED version
push r31
push r30
...
push r24
; this concludes saving all the useful registers, note that
```

```
; the registers are saved in the same order as the entry
; point of rtkSemaphoreV
```

The previous code saves the registers so we can use them in the ISR. The important trick to note is how I set the I-bit in the saved copy of SREG. This allows me to do tricks later on.

Next, we write code to check if there is room in the UART receive FIFO. If there is enough room, we put the received byte into the FIFO, then perform the following:

```
rjmp rtkSemaphoreVCont
```

The ISR is not ended with a `reti`! `rtkSemaphoreVCont` is an alternative entry point to V a semaphore. This label is defined immediately after the registers are saved in the subroutine `rtkSemaphoreV`. In other words, we make the ISR just an alternative front for `rtkSemaphoreV`. Two things can happen:

- No thread of a higher priority is triggered. This means `rtkSemaphoreV` does a normal return. The return code restores all the registers from the stack and does a normal `ret`. However, because we “doctored” the saved SREG earlier, interrupt is re-enabled when we return.
- A thread of a higher priority is triggered. This is actually the simple case. A thread context switch happens, and the SREG is restored from the triggered thread.

Is this approach ugly? Oh, yes, absolutely! It makes it impossible to write ISRs in C. On the other hand, this approach promotes efficiency in the ISR. If you have complicated stuff to do when a device interrupts, do it in the handling thread and code it in C.

9 FAQ

9.1 What environment/tool set is needed?

This RTK is meant to work with the `avr-gcc` toolchain in Linux. It is never meant to be portable to the IAR compiler. `avr-gcc` in Windows does not work out of the box, either, due to the lack of case-sensitive filenames. See other FAQs for tips to get this RTK to work in Windows.

9.2 I really need to use this in Windows with WinAVR, is it possible?

Everything is possible. You need to do the preprocessing as a separate step. Use `avr-cpp` to preprocess `rtk.S` and `timer.S`, and be careful not to clobber the original source files. For example, you can do the following (for `timer.S`):

```
avr-cpp -mmcu=atmega128 -DTICK_FREQ=1000 -DISR_FREQ=5000 -DTIMER_ISR=SIG_OUTPUT_COMPARE2 timer.S > timer.ss
```

After this step, `timer.ss` does not need any further preprocessing, so `avr-as` should be able to assemble the object file. This means you need to change the make file a little bit, and you may not be able to use the automatically generated dependency files anymore.

In essence, it is possible, but you can no longer directly use the default make files from the distribution.

9.3 How does this RTK differ from AVRX?

Size: about the same. This RTK compiles/assembles to about 1.5kB of flash and 100 bytes of SRAM.

Preemptive nature: both kernels support preemption.

Scheduling: both kernels support scheduling.

Priority levels: both kernels support priority levels. This RTK support round-robin scheduling for threads of the same priority level. AVRX supports only one thread per priority level.

Semaphores: both kernels support semaphores. This RTK supports generalized semaphores (P and V operations) with arbitrary values. Both kernels use queues for waiting.

Message Queues: AVRX has it, this RTK does not.

Single step debugging: AVRX has it, this RTK does not.

Tool-chain: AVRX supports both GCC and IAR-C, this RTK only supports GCC.

Time slicing: This RTK supports it, AVRX has no mentioning of it.

Multiple thread instances of same entry point: This RTK support it, not clear whether AVRX support it or not.

9.4 What is the selling point of this RTK?

I don't know, I just use it in my own consulting projects.

To me, the main selling point is that I have full control over it! I get to implement the kernel the way I see fit. I'm sure most other RTKs for the AVR offer similar, if not better, features compared to this RTK.

9.5 What is the future of this RTK?

It'll continue to evolve as I need more features. The last feature that I implemented was time slicing.

9.6 Is this a competitive product?

No. I am not trying to compete with anyone. Features are implemented as necessary, not to keep up with the Joneses.

9.7 Is this RTK free?

Yes, but I ask that you respect my copyright notice. I.e., when you need to distribute the files, please do not remove or alter my copyright lines. If you make changes to it, please feel free to add your copyright notice in addition to mine.

9.8 Is this RTK related to any Z-World/RabbitSemiconductor product?

No, absolutely not. Although I was employed by Z-World from 1990 to 1998, this RTK is completely independent from any Z-World code that I know of.