RTOS4AVR User Manual

Contents

Introduction

I’ve worked on several firmware projects that used real-time operating systems (RTOSes). I think they’re pretty neat, and I wanted to try writing one for myself. This project is the result.

I chose to use an 8-bit AVR microcontroller for this project because they are fairly easy to program and use, and because they have better than average documentation. I chose the ATmega328P because I had a few of them on hand.

Since the goal of this project was to learn, I wanted to keep the code as simple and clean as possible. By choosing to write it for a single model from the AVR family, and to not make any of the features of the RTOS configurable or optional, I was able to avoid a lot of visual clutter.

I’m publishing this project on GitHub mostly to show off, but I also hope it might be useful to someone either to use in a project or just to learn from. Because of the simplicity of the code, it should be easy to read and understand, and therefore also easy to modify and maintain.

Copyright Notice

The following copyright license applies to this document as well as all source code and other files supplied with the project.

Audience

Knowledge of the following will help you to understand this manual:

- The C programming language
- What assembly language is
- What an interrupt is
- What the processor stack is and how it is used
- 8-bit AVR microcontrollers
- What an RTOS is

Overview

This RTOS is preemptive and event-driven. It can manage up to 8 tasks, one at each of 8 different priority levels. It provides message queues, counting semaphores and event flags for communication and synchronisation between tasks and/or interrupt service routines (ISRs).

Tasks

All your application code goes in either ISRs or tasks. A task is a function that gets its own stack and a unique priority. All tasks must be registered with the RTOS before the RTOS is started. A task function must never return.

A task can be in one of two states: ‘ready’ or ‘blocked’. Ready tasks are those that are ready to run, while blocked tasks are waiting for some event. If a task is moved from ready to blocked, we say it “gets blocked.” If a task is moved from blocked to ready, we say it “gets unblocked” or we say it is “made ready.”

The task that is allowed to run on the CPU core is always the ready task that has the highest priority, and we call it the ‘running’ task. If a task with a higher priority than the current running task is made ready, the higher priority task becomes the running task.

Enabled interrupts always have higher priority than tasks; the running task runs on the CPU core only when there is no interrupt being handled.

There is an external interrupt request pin for the AVR called INT0. This RTOS uses the INT0 ISR to perform task switches, and the corresponding interrupt to trigger task switches. This means that the INT0 pin must be used as an output by the RTOS, so it is unavailable for use in your project.

If a task switch is triggered while the INT0 interrupt is disabled, the task switch will not occur until INT0 is re-enabled. In particular, if a task switch is triggered by an ISR while all interrupts are disabled, the task switch will not occur until interrupts are re-enabled at the end of the ISR. The RTOS relies on this behaviour; do not call any RTOS function that could trigger a task switch from an ISR with interrupts enabled, and once such an RTOS function has been called from an ISR, do not re-enable interrupts within the ISR.

There is no explicit ‘idle’ task provided or required as in some RTOSes, and it is possible for there to be no task in the ready state; in this case there is no running task and the microcontroller is kept in its ‘idle’ sleep mode until a task is made ready by an ISR.

Delays and Blocking

Timer/Counter0 - a timer built into the AVR - is used to generate a ‘tick’ interrupt every 1ms. The RTOS keeps a 32-bit global count of the number of ticks that have occurred since the RTOS was started. The count overflows to zero approximately every 50 days.

A task can choose to be blocked until a timeout expires, after which the task is made ready again. The timeout is specified either as expiring after a certain number of ticks occur or when a specified global tick count is reached. Note that “after a certain number of ticks occur” is not quite the same thing as “after a duration of a certain number of milliseconds.” Although each tick follows the previous one after exactly 1ms, the very first tick of the timeout period will occur between 0 and 1ms after the task sets the timeout, since the timer is always running independently. This means that a timeout set to expire after n ticks will occur between (n-1) and n milliseconds after the timeout is set.

The RTOS provides management of message queues, semaphores and event flags. These allow tasks and ISRs to communicate and synchronise with each other. Some actions on these items aren't always possible right away. When the action isn't possible right away, a task attempting the action can choose (i) to give up right away; (ii) to be blocked until the action can be performed; or (iii) to be blocked until the action can be performed or a timeout expires. Below, where such a choice is allowed we will say that the task may "optionally block" (on the action.)

It is possible that more than one task will optionally block waiting to perform the same action on the same item. In that case, whenever the action becomes possible, the highest priority such task will be the one to be unblocked and perform the action. The other tasks will remain blocked waiting to perform the action.

An ISR attempting an action that isn't possible right away must always fail immediately; that is, the ISR must never request to optionally block on the action.

Below, where I say an action 'happens immediately' I mean that it both is possible and is performed right away.

Message Queues

A message queue allows items to be safely passed between tasks and/or ISRs. The items passed are C pointers, so they may (but need not) point to larger items as needed. Each queue is created with a fixed length circular buffer which may hold from 0 to 255 items. Each queue starts out empty, i.e. with nothing in its buffer.

'Sending' on a queue is an attempt to insert one message into the queue. Sending to a queue happens immediately if either there is room in the buffer or there is another task blocked waiting to receive from the queue. If sending is not possible right away a sending task may optionally block.

'Receiving' from a queue is an attempt to remove one message from the queue. Receiving from a queue happens immediately if either there is a message in the buffer or there is another task blocked waiting to send to the queue. If receiving is not possible right away a receiving task may optionally block.

Messages are received from a queue in the order in which they are sent to the queue.

Note the special case of a queue with a 0-length buffer (i.e. a queue without a buffer.) Sending on such a queue can only happen immediately if there is another task blocked waiting to receive from the queue, and receiving on such a queue can only happen immediately if there is another task blocked waiting to send on the queue.

Semaphores

A semaphore is a simple 8-bit counter which decrements each time a task or ISR 'takes' it, and increments each time a task or ISR 'gives' it. Each semaphore is initialised with an initial count from 0 to 255.

Taking a semaphore happens immediately if the count is not zero. If taking is not possible right away, the taking task may optionally block.

Giving a semaphore always happens immediately.

A semaphore whose count is zero will never be decremented; however giving to a semaphore with a count of 255 will cause the count to overflow to zero.

Event Flags

A set of event flags is a set of 8 bits each of which is called a flag. The initial states of the bits are set when the set of flags is initialised. A task or ISR can individually 'set' (to 1) or 'clear' (to 0) flag bits; both of these actions happen immediately.

A task or ISR can try to 'take all' of a subset of the bits; this means that only when all of the bits in the subset are 1s then all of those bits are cleared to 0s and the take succeeds. This only happens immediately if all the bits in the subset are 1s when the take is attempted.

A task or ISR can try to 'take any' of a subset of the bits; this means that only when at least one of the bits in the subset is a 1 then the least-significant of such bits is cleared to 0 and the take succeeds. This only happens immediately if at least one of the bits in the subset is a 1 when the take is attempted.

For a task attempting a ‘take all’ or a ‘take any’, if taking is not possible right away, the task may optionally block.

Usage

Source Files

`src/rtos.h`
Contains prototypes for most of the RTOS functions, and definitions of data types and constants used with those functions. In particular it defines the structure types RTOS_QUEUE, RTOS_SEM and RTOS_FLAGS which are used to keep all state information for queues, semaphores, and sets of event flags, respectively.

`src/rtos.c`
Contains most of the RTOS code.

`src/rtos_asm.s`
Contains assembly code definition of the ISR that handles the INT0 interrupt, which is used to perform task switches.

`src/timer0.c`
Contains code to initialise Timer/Counter 0 to generate a tick interrupt every 1ms.

`src/timer0.h`
Contains function prototypes for functions in timer0.c, and the definition of the `RTOS_HANDLE_TICK()` macro, which is needed in rtos.c as the signature for the tick ISR function definition.

Compiling

This RTOS was developed using the avr-gcc compiler and avr-libc. I use the versions of these included with Atmel Studio 7 (Version: 7.0.2397)

`avr-gcc.exe --version`
	reports: `avr-gcc (AVR_8_bit_GNU_Toolchain_3.6.2_1778) 5.4.0`

In avr/version.h:
```
	#define __AVR_LIBC_VERSION_STRING__ "2.0.0"
```

I compile on the command line. I don’t use the Atmel Studio GUI.

I pass the following switches to avr-gcc when compiling - start with these to get the code compiling without errors:

```
        -Wall
        -std=c99
        -O3
        -DF_CPU=8000000
        -mmcu=atmega328p
```

For the rtos_asm.s file I also pass this switch:

```
        -x assembler-with-cpp
```

so that avr-gcc will preprocess the file before passing the result to the assembler. Note that you will need to change ‘`-DF_CPU=8000000`’ to match the frequency of your CPU core clock, in Hz.

You might also find it helpful to look at `build.bat`, which is the build script I use on Microsoft Windows 10.

Startup

Startup code is all the code that runs before you start the RTOS.

Your startup code will likely do any needed initialisation of the microcontroller hardware and of your application code.

Your startup code should also register all your tasks with the RTOS and initialise all your message queues, semaphores and event flag sets.
Use the `rtosTaskRegister()` function to register each task.
Use the `rtosQueueInit()` function to initialise each message queue.
Use the `rtosSemInit()` function to initialise each semaphore.
Use the `rtosFlagsInit()` function to initialise each set of event flags.

The last thing your startup code should do is call the rtosStart() function, which never returns. The RTOS will start the highest priority task running, and thereafter will manage your tasks such that the running task is always the highest priority task that is ready.

Do not enable INT0 or the tick interrupt at any point before `rtosStart()` is called. `rtosStart()` will take care of enabling them both.

RTOS Functions

`void rtosTaskRegister(void (*task)(void), int priority, uint8_t *pStack)`
Use this function to register each of your task functions with the RTOS before starting the RTOS. INT0 and the tick interrupt must both be disabled when this function is called.

Parameters

> **`void (*task)(void)`** is a pointer to the task function, which must be a function that takes no arguments and that never returns. (If you need one of your tasks to stop running permanently, you can use `rtosDelay()` with `RTOS_MODE_BLOCK`.)
> **`int priority`** is the priority you want to give the task. Must be an integer from 0 to 7, with 0 being the lowest priority and 7 the highest. Do not try to register more than one task at a given priority.
> **`uint8_t *pStack`** is a pointer to the *highest-addressed* byte in the stack space that you want the task to use. Each task must be given its own stack. The stack should be a contiguous block of memory whose size is at least:
>
> > (35 bytes) + (max task stack depth) + (max interrupt stack depth)
>
> where:
>
> > (max task stack depth) is the maximum number of bytes of stack that your task will use;
> > (max interrupt stack depth) is the maximum number of bytes of stack that any of your ISRs will use, including the tick ISR `RTOS_HANDLE_TICK()`. Do not include `INT0_vect` stack here, as that is covered by the (35 bytes). If you are allowing interrupts to nest, then use the maximum stack depth needed for any combination of nested ISRs that could occur - again not including `INT0_vect`.

`void rtosQueueInit(RTOS_QUEUE *pQ, void **pBuffer, uint8_t length)`
Use this function to initialise a queue. This must be done before any other `rtosQueue*()` function is used on the queue. A simple way to ensure this is to call `rtosQueueInit()` before starting the RTOS, since all other `rtosQueue*()` functions must not be called until after the RTOS is started.
The queue starts out empty.

Parameters

> **`RTOS_QUEUE *pQ`** points to the queue to initialise.

`void **pBuffer` points to the start of a memory block to use for message storage. The memory block must be big enough to hold `length` pointers. If `length == 0`, `pBuffer` is ignored and can be `NULL`.
`uint8_t length` is the number of pointer messages that the queue can hold.

`void rtosSemInit(RTOS_SEM *pS, uint8_t initialCount)`
Use this function to initialise a semaphore. This must be done before any other `rtosSem*()` function is used on the semaphore. A simple way to ensure this is to call `rtosSemInit()` before starting the RTOS, since all other `rtosSem*()` functions must not be called until after the RTOS is started.

Parameters

`RTOS_SEM *pS` points to the semaphore to initialise.
`uint8_t initialCount` is the initial count to set in the semaphore.

`void rtosFlagsInit(RTOS_FLAGS *pF, uint8_t initialValue)`
Use this function to initialise a set of event flags. This must be done before any other `rtosFlags*()` function is used on the flags. A simple way to ensure this is to call `rtosFlagsInit()` before starting the RTOS, since all other `rtosSem*()` functions must not be called until after the RTOS is started.

Parameters

`RTOS_FLAGS *pF` points to the event flags set to initialise.
`uint8_t initialValue` is a bitmap giving the initial values of the flag bits.

`void rtosStart(void (*startTicker)(void))`
Use this function to start the RTOS. Once this function is called, the RTOS manages the running of your tasks such that the running task is always the highest priority task that is ready.

`rtosStart()` must be called with the global interrupt enable status bit equal to 0 (i.e. interrupts disabled), and with INT0 disabled through the EIMSK register.

Do not call `rtosStart()` from an ISR.

`rtosStart()` never returns.

Parameters

`void (*startTicker)(void)` is a function to start the regular generation of tick interrupts; that is, calling this function causes `RTOS_HANDLE_TICK()` to start being called at the tick frequency. This function must enable the tick interrupt, but must not set the global interrupt enable status flag. The timer0.c file is provided to allow the use of the microcontroller's Timer/Counter0 for tick generation at 1kHz. To use it, pass `Timer0Init` as the `startTicker` parameter. The `RTOS_HANDLE_TICK()` definition in timer0.h makes the `RTOS_HANDLE_TICK()` function in rtos.c into an ISR to handle the tick interrupt from Timer/Counter0.

`uint32_t rtosTickCountRead(void)`
This function returns the global tick count, i.e. the number of ticks that have occurred since the RTOS was started or since the global tick count last overflowed to 0.

`void rtosDelay(uint32_t timeout, RTOS_MODE mode)`
A task can use `rtosDelay()` to have the RTOS put the task into the blocked state for a specified time. Once unblocked, the next time the task runs it returns from the call to `rtosDelay()`.

Do not call `rtosDelay()` before the RTOS is started.

Do not call `rtosDelay()` directly from an ISR.

When `rtosDelay()` returns, the global interrupt enable status bit will be 1.

Parameters

`uint32_t timeout` is used in conjunction with the mode parameter to determine how long the task is blocked - see below.

`RTOS_MODE mode` must be one of the `RTOS_MODE_*` constants defined in `rtos.h`, and determines how `rtosDelay()` behaves:

`mode == RTOS_MODE_NO_BLOCK`
`rtosDelay()` returns immediately, without blocking the task.

`mode == RTOS_MODE_BLOCK_DURATION`
The RTOS puts the task into the blocked state until `timeout` tick interrupts have occurred since the call to `rtosDelay()`.

`mode == RTOS_MODE_BLOCK_UNTIL`
The RTOS puts the task into the blocked state until the global tick count reaches `timeout`, *unless* it would take more than `0x7fffffff`

tick interrupts to reach that value in the global tick count; in the latter case `rtosDelay()` returns immediately without blocking the task. The reason for this `0x7fffffff` cutoff behaviour is to prevent a task from being blocked unintentionally in the following scenario:

1. The task decides to block until the global tick count reaches some value `timeout;`
2. Before the task calls `rtosDelay()`, one or more higher-priority tasks or ISRs run, and the global tick count counts past `timeout;`
3. The first task calls `rtosDelay()` with `timeout`. In this case the task actually meant to block until a time that is now already in the past. The cutoff of `0x7fffffff` allows `rtosDelay()` to detect this and return immediately. (Of course we are assuming here that the first task was not blocked for longer than `0x7fffffff` ticks in step 2.)

`mode == RTOS_MODE_BLOCK`
The RTOS puts the task into the blocked state, without arranging to unblock the task at some future time. Some other `rtos*()` functions, before calling `rtosDelay()`, will arrange for some other event to cause the task to be unblocked. Otherwise, a call to `rtosDelay()` with RTOS_MODE_BLOCK causes the task to be blocked forever.

`Important Notes on Optionally Blocking Functions`
Many of the functions below have, as their last two parameters:
`uint32_t timeout` and `RTOS_MODE mode`.
These functions perform actions on which a task can optionally block.

Do not call an optionally blocking function before the RTOS is started.

When an ISR calls an optionally blocking function, it must pass `RTOS_MODE_NO_BLOCK` as the mode parameter, and the global interrupt enable status bit 'I' in the status register must be 0. 'I' will remain 0 throughout the function and upon return, so no interrupts will be serviced during the function.

When a task calls an optionally blocking function, if the function can perform its action right away then it does so and returns. If the action isn't possible right away, the mode parameter determines what the function does:

`mode == RTOS_MODE_NO_BLOCK`
The function returns immediately, without blocking. (The action fails.)
`mode == RTOS_MODE_BLOCK_DURATION`

`mode == RTOS_MODE_BLOCK_UNTIL`
The task is blocked in the function with a timeout set exactly the same way as it would be in `rtosDelay()` with the same `mode` and `timeout`. If the action becomes possible before the timeout expiry, then the timeout is canceled, the action is performed, and the task is made ready. Otherwise, the task is made ready when the timeout expires.
`mode == RTOS_MODE_BLOCK`
The task is blocked in the function until the action is possible, then the action is performed and the task is made ready.

If an optionally blocking function does block, the global interrupt enable status bit 'I' will be 1 when the function returns. Otherwise, the 'I' flag may be changed to 0 during the function, but its value will be restored upon return to the value it had when the function was called.

Note that if more than one task all get blocked in the same function, each time the action becomes possible only the blocked task with the highest priority will be unblocked and perform the action.

```
int rtosQueueSend(
  RTOS_QUEUE *pQ, void *pMessage, uint32_t timeout, RTOS_MODE mode)
```
This function attempts to send a message on a queue, optionally blocking until the message can be sent or a timeout expires. Returns nonzero if the message is sent, zero if the message could not be sent.

Parameters

`RTOS_QUEUE *pQ` points to the queue on which to send the message.
`void *pMessage` is the message to send. Note that only the pointer `pMessage` is sent through the queue. Any value for `pMessage` is allowed, including `NULL`.
`uint32_t timeout, RTOS_MODE mode` are used for optional blocking.

This function may block or may cause another task to be made ready, possibly causing a task switch.

See the important notes on optionally blocking functions.

`int rtosQueueReceive(`
`  RTOS_QUEUE *pQ, void **ppMessage, uint32_t timeout, RTOS_MODE mode)`
This function attempts to receive a message on a queue, optionally blocking until a message can be received or a timeout expires. Returns nonzero if a message is received, zero if no message could be received.

Parameters

`RTOS_QUEUE *pQ` points to the queue on which to receive a message.
`void **ppMessage` points where the received message (of type `void *`) should be put.
`uint32_t timeout, RTOS_MODE mode` are used for optional blocking.

This function may block or may cause another task to be made ready, possibly causing a task switch.

See the important notes on optionally blocking functions.

`int rtosSemTake(RTOS_SEM *pS, uint32_t timeout, RTOS_MODE mode)`
This function attempts to take from (i.e. decrement) a semaphore, optionally blocking until it is available or a timeout expires. Returns nonzero if the semaphore is taken, zero if the semaphore could not be taken.

Parameters

`RTOS_SEM *pS` points to the semaphore to take.
`uint32_t timeout, RTOS_MODE mode` are used for optional blocking.

This function may block, causing a task switch.

See the important notes on optionally blocking functions.

`void rtosSemGive(RTOS_SEM *pS)`
This function gives to (i.e. increments) a semaphore. This function does not prevent the semaphore's count from overflowing to zero.

Parameters

`RTOS_SEM *pS` points to the semaphore to give.

If an ISR calls `rtosSemGive()`, the global interrupt enable status bit 'I' in the status register must be 0. 'I' will remain 0 throughout the function and upon return, so no interrupts will be serviced during `rtosSemGive()`.

If a task calls `rtosSemGive()`, the global interrupt enable status bit may be changed to 0 during the function, but its value will be restored upon return.

This function may cause another task to be made ready, possibly causing a task switch.

`uint8_t rtosFlagsTake(RTOS_FLAGS *pF, uint8_t flags, uint8_t all, uint32_t timeout, RTOS_MODE mode)`
This function attempts to 'take all' or 'take any' of a subset of the flags in a set of event flags, optionally blocking until it can do so or until a timeout expires. Returns the flag bit(s) taken, or returns zero if it couldn't get the requested bits.

Parameters

> **`RTOS_FLAGS *pF`** points to the set of event flags.
> **`uint8_t flags`** is a bitmap; the 1 bits indicate the positions of which flag bits are being requested. Must be nonzero.
> **`uint8_t all`** indicates whether to 'take all' (`all != 0`) or to 'take any' (`all == 0`) of the requested flags.
> **`uint32_t timeout, RTOS_MODE mode`** are used for optional blocking.

This function may block, causing a task switch.

See the important notes on optionally blocking functions.

`void rtosFlagsSet(RTOS_FLAGS *pF, uint8_t flags)`
This function sets the given flag bits in a set of event flags.

Parameters

> **`RTOS_FLAGS *pF`** points to the set of event flags.
> **`uint8_t flags`** is a bitmap; the 1 bits in the bitmap indicate which flag bits should be set to 1s.

If an ISR calls `rtosFlagsSet()`, the global interrupt enable status bit 'I' in the status register must be 0. 'I' will remain 0 throughout the function and upon return, so no interrupts will be serviced during rtosFlagsSet().

If a task calls `rtosFlagsSet()`, the global interrupt enable status bit may be changed to 0 during the function, but its value will be restored upon return.

This function may cause one or more tasks to be made ready, possibly causing a task switch.

`void rtosFlagsClear(RTOS_FLAGS *pF, uint8_t flags)`
This function clears the given flag bits in a set of event flags.

Parameters

`RTOS_FLAGS *pF` points to the set of event flags.
`uint8_t flags` is a bitmap; the 1 bits in the bitmap indicate which flag bit(s) should be cleared to 0s.

The global interrupt enable status bit may be changed to 0 during `rtosFlagsClear()`, but its value will be restored upon return.

Source Code Design

This section is meant to help you understand the source code of the RTOS.

Notation and Shorthand

For a binary integer, I number the bits right to left starting from 0, and I call this number the "bit position" for a given bit. For example, 0x01 = (binary)'00000001' has a 1 in bit position 0, and 0x24 = (binary)'00110100' has 1s in bit positions 2, 4 and 5.

I use 'iff' to mean "if and only if."

RTOS and Task State

The RTOS keeps track of its state, and the states of the tasks, using the variables described below. Most of these variables are declared in rtos.c, and are global. I would like to make them all 'static' to keep them from being accessed by non-RTOS code, but can't because some need to be accessed from rtos_asm.s.

`uint8_t curPrio` holds the priority of the current running task, an integer from 0 to 7.

`uint8_t running` is treated as a bitmap, where at most one bit position ever contains a 1. The bit position of the 1 is the priority of the current running task.

Example: if the current running task is the task at priority `curPrio == 6`, then `running` has a 1 in bit position 6; i.e. `running == 0x40`.

`uint8_t bitId[8]` is an array of 8-bit bitmaps. `bitId[x]` contains `(1<<x)` if there is a task registered with priority x, else `bitId[x]` contains 0. This is used to check whether there is any task at priority x. It is also used to determine the value with a 1 only at bit position x without having to do a bit shift operation.
Example: Suppose tasks have been registered only at priorities 0, 4 and 6. Then we will have

```
    bitId[0] == 0x01
    bitId[1] == 0
    bitId[2] == 0
    bitId[3] == 0
    bitId[4] == 0x10
    bitId[5] == 0
    bitId[6] == 0x40
    bitId[7] == 0
```

`uint8_t ready` is treated as a bitmap, where the bit at position x is a 1 iff the task at priority x is ready to run. Note that `ready` is the logical OR of the `bitId[]`s corresponding to all the ready tasks.
Example: if the tasks at priorities 2 and 7 (only) are ready, then `ready == 0x84`.

`uint32_t tickCounter` holds the 32-bit global tick count.

`uint8_t *pStacks[8]` is an array used to save the stack pointers of tasks that are not running. When the task with priority x is not running, `pStacks[x]` points to the next free entry on that task's stack.

Blocked State of Tasks

`uint8_t timeoutIds` is treated as a bitmap, where the bit at position x is a 1 iff the task at priority x has set a timeout. Such a timeout would be set by either a direct call to `rtosDelay()` or a call to an optionally blocking function for which the task chose a delay and in which the task then blocked.

`BLOCKING_SLOT blockingSlots[8]` is an array of structures, where `blockingSlots[x]` corresponds to the task at priority x. When a task is blocked, its entry in `blockingSlots[]` contains information on what event(s) it is waiting on, and is used to pass results back to the task when the task gets unblocked. The members of `BLOCKING_SLOT` are:

> **`uint32_t wakeTime`** is used iff a timeout is set for the task (see `timeoutIds`.) When the global tick count increments to the value in `wakeTime`, the timeout for the task expires.
>
> **`uint8_t timedOut`** is used when the task becomes unblocked. Iff nonzero, it indicates to the task that it became unblocked because of the timeout expiry.
>
> **`item.q.pMessage`** (has type `void *`) is used iff the task blocks waiting to send or receive on a queue. If the task is waiting to send, pMessage holds the message to send. If waiting to receive, any received message will be put in pMessage to be passed back to the task.
>
> **`item.f.flags`** (has type `uint8_t`) is used iff the task is blocked waiting to take event flag(s). It is set, before the task is blocked, to the value of the parameter `flags` that was passed to the `rtosFlagsTake()` function. The 1 bits in `flags` correspond to the event flags that the task requested to take.
>
> **`item.f.all`** (has type `uint8_t`) is used iff the task is blocked waiting to take event flag(s). It is set, before the task is blocked, to the value of the parameter `all` that was passed to the `rtosFlagsTake()` function. If nonzero, it indicates that the task wants all of the requested flags, otherwise it indicates that the task wants any one of the requested flags.
>
> **`uint8_t *pWaitingOn`** is used iff the task is blocked waiting on an item that is a queue, a semaphore, or a set of event flags. If used, it points to a bitmap in the structure representing the item's state, and the bit position corresponding to the task's priority is set to 1 in that bitmap.

When a task is not blocked, its bit in `timeoutIds` is 0 and `pWaitingOn` is `NULL` in its blocking slot.

When a task is blocked, iff it has set a timeout then its bit in `timeoutIds` is 1, and iff it is waiting on an item then `pWaitingOn` points to the appropriate bitmap in the item.

Items for Inter-Task Communication and Synchronisation

The RTOS manages message queues, semaphores and event flags. These items are used to allow communication and synchronisation between tasks and/or ISRs. Each of these items' state is kept in a structure defined in rtos.h.

RTOS_QUEUE
This structure holds the state of a message queue. Its members are:

> **void **pNextEmpty** If the queue has no buffer then `pNextEmpty` is `NULL`. Otherwise `pNextEmpty` points, in the circular buffer, to the location following the location that a message was last written into the buffer (or the lowest-addressed location if no message has yet been inserted).
> **void **pFirstFull** is `NULL` if the queue has no buffer or if the queue is empty, otherwise it points, in the circular buffer, to the location of the oldest message that has not yet been read out of the buffer.
> **void **pBuffer** is unused if the queue has no buffer, otherwise `pBuffer` points to the lowest-addressed location in the buffer.
> **void **pBufferEnd** is unused if the queue has no buffer, otherwise `pBufferEnd` points to the address of the first byte after the buffer space in memory. It is used to detect when either `pNextEmpty` or `pFirstFull` has incremented past the end of the buffer in memory and therefore needs to wrap back to `pBuffer`.
> **uint8_t sendersWaiting** is treated as a bitmap, where the bit at each position x is a 1 iff the task with priority x is currently blocked waiting to send on the queue. For each such task, the `pWaitingOn` member of its blocking slot points to `sendersWaiting`.
> **uint8_t receiversWaiting** is treated as a bitmap, where the bit at each position x is a 1 iff the task with priority x is currently blocked waiting to receive from the queue. For each such task, the `pWaitingOn` member of its blocking slot points to `receiversWaiting`.

Recall that the messages have type (`void *`), so the buffer is an array of entries of that type.

If the queue has a buffer, it is managed as a circular FIFO, so:

- Messages are removed from the buffer in the order that they were inserted.
- When a message is inserted it is written at `pNextEmpty` and then `pNextEmpty` is incremented, wrapping to `pBuffer` at the end of the buffer space. If it reaches `pFirstFull` the buffer is full.
- When a message is removed it is read from `pFirstFull` and then `pFirstFull` is incremented, wrapping to `pBuffer` at the end of the buffer space. If it reaches `pNextEmpty` there are no messages left in the buffer, so pFirstFull is set to `NULL`.

When the last unused entry is written into the buffer, `pNextEmpty` advances to point at the same entry as `pFirstFull`. If there is no buffer, `pNextEmpty` and `pFirstFull` are both always `NULL`. Thus a message can be written into the buffer iff `pNextEmpty != pFirstFull`.

`RTOS_SEM`
This structure holds the state of a semaphore. Its members are:

`uint8_t count` holds the semaphore count.

`uint8_t takersWaiting` is treated as a bitmap, where the bit at each position x is a 1 iff the task with priority x is currently blocked waiting to take from the semaphore. For each such task, the `pWaitingOn` member of its blocking slot points to `takersWaiting`.

`RTOS_FLAGS`
This structure holds the state of a set of event flags. Its members are:

`uint8_t flags` is treated as a bitmap, each bit being a flag

`uint8_t takersWaiting` is treated as a bitmap, where the bit at each position x is a 1 iff the task with priority x is currently blocked waiting to take from the set of event flags. For each such task, the `pWaitingOn` member of its blocking slot points to `takersWaiting`.

Atomic Operations

Sometimes it is necessary for several variables to be accessed atomically, meaning that while accesses of those variables are taking place, no other code can access the variables. I won't go into detail here on why this is so; for more information I suggest researching 'concurrency' and 'shared memory.'

To make their code atomic, some RTOS functions disable interrupts. Several macros are used in `rtos.c` to simplify the disabling and enabling of interrupts. Each of these macros is a memory barrier, to prevent the compiler from moving memory accesses into or past the macro (in either direction). The macros are:

`rtos_CLI_MB()` disables interrupts by executing the '`cli`' assembly instruction to clear the global interrupt enable flag to 0.

`rtos_SEI_MB()` enables interrupts by executing the '`sei`' assembly instruction to set the global interrupt enable flag to 1.

`rtos_SAVE_SREG_AND_CLI_MB()` saves a copy of the status register in a variable, then disables interrupts by executing the '`cli`' assembly instruction. The variable is declared in the macro, name supplied to the macro.

`rtos_RESTORE_SREG_MB()` sets the status register to the value in a supplied variable. This will either enable or disable interrupts, depending on whether the value has a 1 or 0 in the bit position of the global interrupt enable flag in the status register.

Another macro used by the RTOS code is **`rtos_MB()`**, which is just a memory barrier. It is used where there is no need to enable or disable the interrupts, but there is a need to prevent memory accesses being moved past the location of the macro.

`avr-libc` provides the macros `cli()` and `sei()` for changing the global interrupt enable flag. I have chosen not to use these macros because I've read that some of the `avr-libc` developers think they should not be memory barriers, so I'm afraid they might change those macros to not be memory barriers in the future. `avr-libc` also supplies `_MemoryBarrier()` which I could have used instead of `rtos_MB()`, but I wanted to match the naming of my other macros.

`avr-libc` also provides the macro `ATOMIC_BLOCK()` to make sure that a block of code executes atomically. I have not used that either.

Task Switching

Recall that task switches are performed by the ISR that handles the `INT0` interrupt, `INT0_vect`. You can find this ISR in rtos_asm.s, written in assembly. It does the following:

1. Saves the status register and all of the CPU core registers on the stack (which is still the old running task's stack)
2. Saves the stack pointer in the pStacks[] entry for the old running task
3. Waits, if necessary, until there is a task ready to run. While waiting, keeps the CPU in idle sleep mode
4. Chooses the highest priority task that is ready to run to be the new running task, and changes `running` and `curPrio` accordingly
5. Loads the stack pointer with the value from the pStacks[] entry for the new running task
6. Restores the status register and all of the CPU core registers from the stack (which is now the new running task's stack

Where possible, INT0_vect enables interrupts so that other interrupts can be serviced.

Recall that the function `rtosTaskRegister()` in `rtos.c` is used to register all the task functions before the RTOS is started. `rtosTaskRegister()` sets up the task's stack and entry in `pStacks[]` to appear as they would after step 2 above if the task had been running and the `INT0` interrupt had occurred right at the start of the task. `rtosTaskRegister()` also sets the task's bit in `ready` to 1. When the RTOS is started, `rtosStart()` jumps to step 3 above in `INT0_vect`, at which point the state of the system appears exactly as it would if a task switch was triggered with all tasks ready.

Code in `rtos.c` triggers task switches using the macro **`rtos_TASK_SWITCH_TRIGGER_MB()`**, and the code always ensures that interrupts are disabled before this macro is reached. The macro toggles pin PD2 to trigger the INT0 interrupt. After toggling the pin, the macro sets the status register to the value in a supplied variable. This will either enable or disable interrupts, depending on whether the value has a 1 or 0 in the bit position of the global interrupt enable flag in the status register.

If the value written to the status register by `rtos_TASK_SWITCH_TRIGGER_MB()` causes interrupts to be enabled, the task switch will occur before the end of the macro. This is ensured by the 3 '`nop`' assembly instructions at the end of the macro.

If the value written to the status register by `rtos_TASK_SWITCH_TRIGGER_MB()` leaves interrupts disabled, the task switch will not occur until interrupts are next enabled after the macro.

I originally intended to use the `PCINT2` interrupt for task switches. Then I noticed this in the datasheet: "*When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served.*" There are 4 interrupts with higher priority than `PCINT2`, so if they were all triggered before interrupts were enabled by `rtos_TASK_SWITCH_TRIGGER_MB()` then the task switch would not occur within the macro unless I added 4 more '`nop`'s to the macro. To allow the macro to be as short as possible I decided to use `INT0` instead of `PCINT2`. `INT0` is the highest priority interrupt other than `RESET`, so it allows the macro to be as short as possible.

Notes on Porting to Other 8-bit AVR Parts

Here are some things that may need to be considered when porting this RTOS to another 8-bit AVR. This list almost certainly doesn't cover everything.

- This RTOS is written for an AVR with 2-byte addresses. For an AVR with different sized addresses, rtosTaskRegister() will need to be modified to put the correct number of address bytes on the stack for each task.
- This RTOS is written for an AVR with 32 CPU core registers (`r0` to `r31`). For an AVR with a different number of CPU core registers, `rtosTaskRegister()` will need to be modified to put the correct number of bytes on the stack for each task, and `INT0_vect` will have to be modified to save and restore all the CPU core registers.
- If you want to port `rtos_TASK_SWITCH_TRIGGER_MB()` for use with another AVR part, there may be several interrupts with higher priority than INT0 on your AVR, requiring more 'nop's to be added to the macro.
- Depending whether Timer/Counter 0 is available, you might need to find another way to generate tick interrupts.
- INT0_vect was written assuming that the I/O registers `EIMSK` and `EIFR` are valid targets for the '`cbi`' and '`sbi`' assembly instructions. This is not the case for all AVRs. See the notes in `<avr/sfr_defs.h>` provided with `avr-libc`.
- INT0_vect was written assuming that the I/O registers `SMCR`, `SREG`, `SPL` and `SPH` are valid targets for the '`in`' and '`out`' assembly instructions. This is not the case for all AVRs. See the notes in `<avr/sfr_defs.h>` provided with `avr-libc`.

References

1. avr-gcc ABI details at https://gcc.gnu.org/wiki/avr-gcc
2. ATmega48A/PA/88A/PA/168A/PA/328/P DATASHEET
   Rev:Atmel-8271J-AVR-ATmega48A/48PA/88A/88PA/168A/168PA/328/328P-Datasheet_11/2015
3. Microchip AVR Instruction Set Manual
   Rev.0856F - 05/2008
4. https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Extended-Asm.html