CCS inc.
Real Time Operating System (RTOS) Documentation V2.1

James Hill

Contents

1. Overview

1.1 Introduction to the CCS RTOS

The Custom Computer Services, Inc. (CCS) Real Time Operating System (RTOS) is an easy way to quickly create microcontroller applications that require multiple tasks to be run at consistent time intervals. The programmer can easily specify certain functions to be tasks run by the RTOS at specified times. The compiler will generate all necessary code based on the programmer's timing specifications. Along with the ability to schedule tasks, the RTOS also gives the programmer the ability to disable and enable tasks, communicate between tasks, handle limited resources, and keep track of task statistics.

This document is intended to give the reader a better idea of how the RTOS manages the tasks so the the reader can better develop software using the RTOS.

1.2 General Program Format

An RTOS program consists of several task functions and a call to rtos_run(). Each task function is managed by the internal code of rtos_run() and each is presented in more detail in sections 2 and 3, respectively. Listing 1 shows the basic layout of an RTOS program.

Listing 1

```
// preprocessor directives
#include <18F452.h>
#use delay(clock=20000000)
#use rs232(baud=9600,xmit=PIN_C6,rcv=PIN_C7)
#use rtos(timer=0,minor_cycle=100ms)

// function declarations
#task(rate=1000ms,max=100ms)
void The_first_rtos_task ( );

#task(rate=500ms,max=100ms)
void The_second_rtos_task ( );

#task(rate=100ms,max=100ms)
void The_third_rtos_task ( );

// more function declarations

// function implementations
void The_first_rtos_task ( )
{
   // task code
}
```

```
void The_second_rtos_task ( )
{
   // task code
}

void The_third_rtos_task ( )
{
   // task code
}

// more function implementations

void main ( )
{
   // initialization code for other resources
   rtos_run ( );
}
```

Listing 1 presents the layout of a simple RTOS program. This program contains three RTOS tasks and makes a call to the rtos_run() function in the main line. Most simple RTOS programs will follow this format and simply require that both the #use RTOS() and #task preprocessor directives be used along with the rtos_run() function. It is recommended that the #task preprocessor directive be used before the function declaration and not the function implementation because many of the other RTOS functions use task names as parameters and the compiler will generate errors if the actual function hasn't been declared in the source file yet. #use RTOS() directive informs the compiler that it should expect to see the #task directive and that it should test the rates and run times for each task against its minor cycle. It also tells the compiler which timer to use. The #task directive informs the compiler that the following function should be compiled as an RTOS task. It also provides the rate at which the task should run along with the maximum time the function is expected to take to execute. The rtos_run() function initializes and begins the operation of the RTOS. The program will act like a regular C program until the rtos_run() function is called.

2. Tasks

Each RTOS task is a C function that takes no parameters and returns no value. It is declared as a standard C function with the exception that the #task preprocessor directive must appear before either the function header or the actual function. The #task directive alerts the compiler that the following function will require special memory allocations, return instructions, and that timing information will need to be calculated for this function once all tasks have been found. Listing 2 demonstrates the recommended declaration of an RTOS task.

Listing 2

```
#task(rate=1s,max=20ms,queue=5)
void task_name();

// more function headers

// source code

// the function implementation
void task_name() {

     // task code
}
```

It is important to place the #task directive before the function header because placing it before the actual function implementation may cause compiler errors if other RTOS tasks reference functions that the compiler has not yet located.

2.1 Task Timing

Each task declaration requires the specification of at least two values, the task rate and the task max value. The task rate tells the compiler how often the task should be run. The max value tells the compiler the maximum amount of time the the task is ever expected to run. These two values are used by the compiler to schedule the tasks and generate the assembly code that will implement that schedule. Listing 3 provides an example of two task declarations.

Listing 3

```
#task(rate=1s,max=20ms)
void task_one();

#task(rate=1s,max=40ms)
void task_two();
```

Listing 3 declares two RTOS tasks. The first will run every second and should never run for more than 20ms. The second will also run every second but should never run for more than 40ms. **It must be stressed that the compiler does not generate the max run time.** The max run time is the programmer's best guess as to the maximum

time the task will ever need to run. The compiler uses both of these values to design a schedule in which neither task is required to run at the same time. For example, the compiler would probably schedule the two tasks in listing X to run at 0.5s intervals; that is, task one would run at time 0s, task two would run at time 0.5s, task one would run again at 1.0s, and task two would run again at 1.5s. Generally, the compiler can build a schedule in which no two tasks will be required to run at the same time; however, in the case that two processes are required to run at the same time, the compiler will generate assembly to run the two tasks with no delay in between.

2.2 Task Control Block

The compiler gives each task a specific number of RAM locations for holding important task information. The amount of RAM required depends on the two factors, whether statistics are used and whether a queue is needed. If no statistics are used and no queue is needed, the task control block will require seven bytes of memory. Listing 4 shows the contents of this control block format and the order in which the contents appear in RAM.

Listing 4

Address	Contents
X+0 bytes	Task State
X+1 byte	Minor Cycles Per Run Low
X+2 bytes	Minor Cycles Per Run High
X+3 bytes	Minor Cycle Counter Low
X+4 bytes	Minor Cycle Counter High
X+5 bytes	Task Address Low
X+6 bytes	Task Address High

The Task State is used by the rtos_run() function to determine whether or not the task is enabled or disabled. If the task is disabled, this RAM locations will hold a 0x80. If the task is enabled, this RAM location will hold a 0x00. The next four bytes are used by the rtos_run() function to keep track of how many minor cycles have passed. This is compared to the number of minor cycles that are required for the task to run next. Once the minor cycle counter has passed the minor cycle per run value, the task is run. Finally, the task address is used by the rtos_run() to call back into tasks that have been yielded via the RTOS_yield() function.

If task statistics are enabled, eight more bytes will be added to the task control block. Listing 5 shows the added contents of the process control block.

Listing 5

Address	Contents
X+7 bytes	Total Time 0
X+8 bytes	Total Time 1
X+9 bytes	Total Time 2
X+10 bytes	Total Time 3
X+11 bytes	Minimum Time Low

X+12 bytes	Minimum Time High
X+13 bytes	Maximum Time Low
X+14 bytes	Maximum Time High

The first four bytes are concatenated into one 32-bit value that represents the total time that the task has been allowed to run. The Minimum Time and Maximum Time values represent the minimum time the task has ever taken to run and the maximum time the task has ever taken to run.

If the task in question has a queue for inter-task communication, then that task will have at least three more bytes associated with it. The number of extra bytes added by the queue can be determined by adding two to the length of the queue. The format of the queue bytes is shown in Listing 6.

Listing 6

Address	Contents
X+15	Queue Index 1
X+16	Queue Index 2
X+17	Queue byte 1
X+18	Queue byte 2
X+19	Queue byte 3
...	
...	
...	

The first two values are used to index into the queue. These are used by the RTOS_msg_poll(), RTOS_msg_read(), and RTOS_msg_send() functions for adding and retrieving data from the queue.

2.3 Task Control

Because the RTOS does not use interrupts, it has been implemented as a cooperatively multitasking operating system. This means that because the actual operating system has little-to-no control over the tasks as they run, it is the programmer's responsibility to make sure that tasks yield to the operating system at reasonable times so that no process overuses the microcontroller. The main functions for controlling the RTOS are rtos_run(), RTOS_terminate(), RTOS_enable(), RTOS_disable(), RTOS_yield(), RTOS_wait(), RTOS_await().

rtos_run() - This function is the actual starting point of the operating system. The RTOS does not actually perform any tasks until this function has been called. Inside of the rtos_run() function is an infinite loop that acts as the task manager for all of the tasks. This loop determines which process needs to run next and waits the needed amount of time before allowing that function to run. rtos_run() also acts as the RTOS initialization function by loading the task control blocks with the needed initialization information such as the function address and the queue index values. The only way to exit the rtos_run() function is by calling the RTOS_terminate() function.

RTOS_terminate() - RTOS_terminate() is an unconditional jump to the return code contained in the rtos_run() function. This code will exit the RTOS and allow the original program to continue operation.

RTOS_enable() - RTOS_enable() sets the most significant bit of the task state variable stored in the task control block. This signals to the rtos_run() function that this task should be run when it's turn arrives.

RTOS_disable() - RTOS_disable() clears the most significant bit of the task state variable stored in the task control block. This signals to the rtos_run() function that this task should not be run when its turn arrives.

RTOS_yield() - RTOS_yield is one of three methods for halting task operation and returning to the rtos_run() function. This function stores the address of the next operation to be run in the task address value contained in the task control block and then jumps back into the RTOS_run loop.

RTOS_wait() - RTOS_wait() is one of three methods for halting task operation. It takes as its parameter one variable that acts as a semaphore. If the value of the semaphore is greater than 0, the resource is assumed to be available and the RTOS_wait function decrements it to claim that resource. If the value of the semaphore is equal to 0, then RTOS_wait() returns control to the rtos_run() function. This will prevent the task from continuing operation until the resource becomes available while allowing other tasks to continue operation.

RTOS_await() - RTOS_await() is one of three methods for halting task operation. It takes as its parameter an expression that must evaluate to true in order for the task to continue operation. As long as the expression evaluates to false, RTOS_await() will return control to rtos_run().

2.4 Task Queue

The programmer can give each task a queue of any length desired. The queue contents and index pointers used to access it are all stored at the end of task control block. Listing 7 shows how the #task preprocessor directive can be used to specify a queue.

Listing 7

```
#task(rate=1s,max=20ms,queue=5)
void task_name();
```

In listing X, task_name has been given a queue containing 5 bytes. The queue can be accessed by the RTOS_msg_poll(), RTOS_msg_read(), and RTOS_msg_send() functions.

RTOS_msg_poll() - RTOS_msg_poll compares the two index values contained in the process control block and waits for them to become different signifying that something

has been added to the queue.

RTOS_msg_read() - RTOS_msg_read returns the values stored at the index value pointing to the beginning of the queue and then increments that index value to point to the next value stored in the queue.

RTOS_msg_send() - RTOS_msg_send takes as its parameter the name of the task to send the information to and the information to send. It then places the information in the specified tasks queue and adjusts the index of that queue accordingly.

2.5 Task Statistics

There are three different statistics that can be obtained for each task. These are the total time the task has run for, the minimum time the task took to complete, and the maximum time the task took to complete. The first value is stored in a 32-bit integer while the second two values are both stored in 16-bit integers. There are two functions that can be used to obtain statistical information; those are RTOS_overrun() and RTOS_stats().

RTOS_overrun() - RTOS_overrun() takes either no parameters or the name of the task to check. If no parameter is specified, a value of true is returned if any tasks overran their maximum time and false if no tasks overran their maximum time. If a task name is passed in as the parameter, RTOS_overrun() will return the same information but only for that task.

RTOS_stats() - RTOS_stats() takes the name of a task and a pointer to a structure containing a 32-bit integer and three 16-bit integers. It places in this structure the total number of clock cycles that have passed since the task was first run, the minimum time (in clock cycles) the task took to complete, the maximum time (in clock cycles) the task took to complete, and the number of clock cycles per microsecond. The structure that could be used to retrieve this information is shown in listing 8.

Listing 8

```
struct stats {
     int32 Total_time;
     int16 Min_time;
     int16 Max_time;
     int16 Ticks_per_us;
}
```

3. RTOS Operation

3.1 Minor Cycles

In order for the CCS RTOS to schedule tasks to occur at the correct time, it must know the smallest amount of time that evenly divides into all of the desired task rates. This amount of time is called the minor cycle and represents the longest time that any task will run. The minor cycle is critical for the RTOS to keep track of what task needs to run and task statistics. The compiler will therefore generate an error if any task rate is not a multiple of the minor cycle. Listing 9 shows an example of several task declarations and then determines the minor cycle that should be used.

Listing 9

```
#task(rate=1s,max=10ms)
void task_one();

#task(rate=2s,max=20ms)
void task_two();

#task(rate=1s,max=5ms)
void task_three();

minor cycle = 20ms
```

The programmer must specify the minor cycle using the #use RTOS() preprocessor directive. The parameters that this directive takes are the timer to be used, the minor cycle, and statistics. The first parameter, timer, is the desired timer to be used in timing the minor cycle. Some timers may not have the resolution to accommodate certain minor cycles and it is therefore the programmer's responsibility to choose the appropriate timer. The second parameter is the minor cycle that is to be used which can be specified in seconds (s), milliseconds (ms), or microseconds (us). The final parameter tells the compiler that statistical information should be kept for each task. Listing 10 shows two examples of #use RTOS()

Listing 10

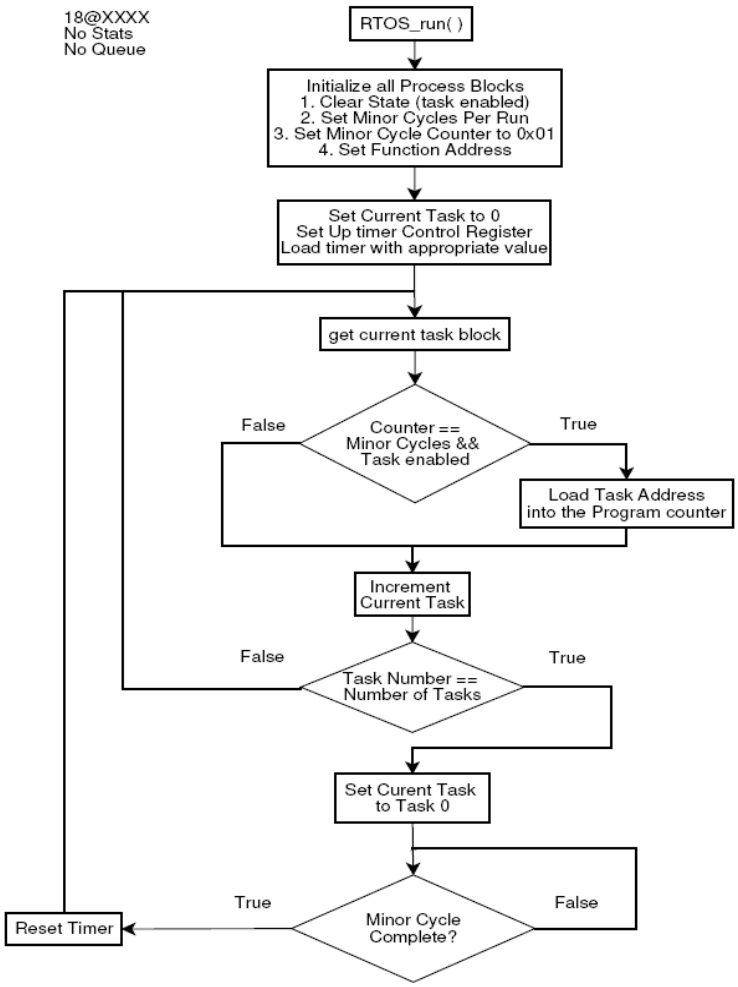
```
// use timer 0 with a minor cycle of 20 milliseconds
// and no statistics
#use RTOS(timer=0,minor_cycle=20ms)

// use timer 1 with a minor cycle or 1000 microseconds
// and keep statistical information about each task
#use RTOS(timer=1,minor_cycle=1000us,statistics)
```

3.2 rtos_run()

rtos_run() acts as the task manager for the RTOS. This function is in charge of determining which tasks should run, when tasks should run, and how long each task

has run. rtos_run() views all tasks as being in an array. Every time a minor cycle is completed, rtos_run() iterates through the list of tasks and increments the Minor Cycle Counter value stored in each task control block. If that increment causes the Minor Cycle Counter value to equal the Minor Cycles Per Run value, the task will be run. Upon returning from the task, rtos_run() will continue the iteration through the tasks. Figure 1 presents this flow for the rtos_run() generated with no statistics and no Queue.

Figure 1



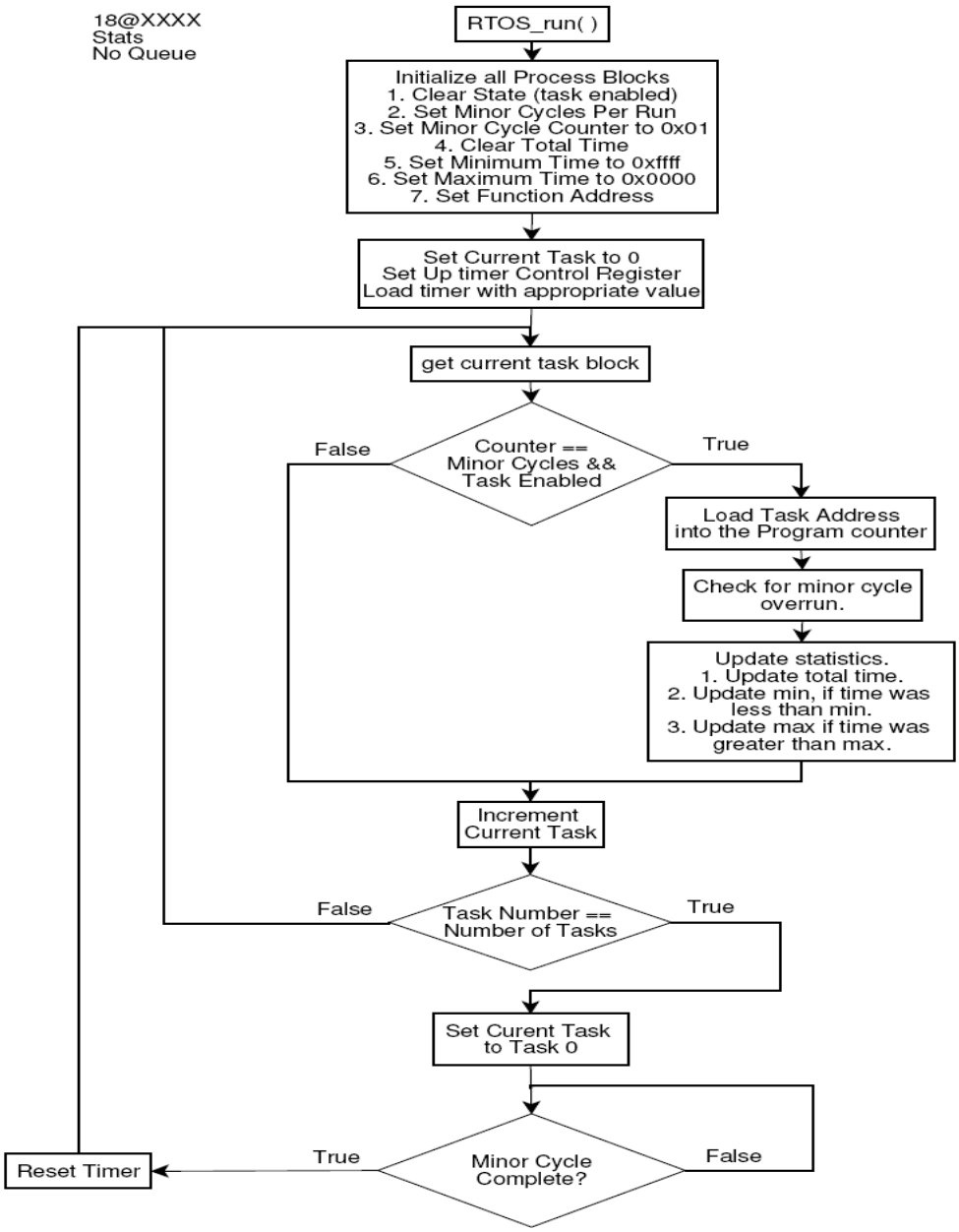
rtos_run() begins by initializing all of the task control blocks. It first clears the state byte which enables the process. It then sets the number of minor cycles that must occur for the process to run. This value is calculated by the compiler. Once the number of minor cycles has been set, the actual minor cycle counter is initialized to 0x01. Finally, the starting address of the function is set. This value is also determined by the compiler. Once all of the task control blocks have been initialized, a global variable representing the current running process is cleared to zero, indicating that the current task is task zero, and the chosen timer is set such that it will overrun on a minor cycle. The value placed in the timer is calculated by the compiler. After the timer has been set, the initialization phase is complete and the actual looping begins.

The first step in the actual rtos_run() loop is to attain the address of the current task's task control block. Once that has been loaded into the microcontrollers indirect file register, the loop checks to see if the task is enabled and if the tasks Minor Cycle Counter is equal to its Minor Cycles Per Run value. If both conditions are met, the current value held in the task's Function Address block is loaded into the program counter and execution jumps to that function. When the task returns, the execution continues along the same path that would have occurred if one or both of the conditions had not been met. In this case, the global value holding the current task is incremented. If the current task is less than the number of tasks, the program execution continues by loading that task's control block and performing the same routines. If the current task is equal to the number of tasks, the current task is reset to zero and rtos_run() begins a loop that checks to see whether or not the timer has overrun. Once the timer overruns, a minor cycle has been completed and the task loop must be restarted. The timer is then reset to run down another minor cycle and the first task control block address is loaded.

3.3 rtos_run() with statistics

When the “statistics” option is set in the #use RTOS() preprocessor directive, extra assembly code is added to the rtos_run() function which is executed on the return of the task. Figure 2 shows the new flow of rtos_run().

Figure 2



The first change is in the initialization routine. Along with clearing the task state and setting the Minor Cycles Per Run value, the Total Time and Maximum values are cleared and the Minimum Value is set to 0xffff. If statistics are checked before the task is ever called, the value of the maximum run time will be 0x0000 and the value of the minimum run time will be 0xffff. These values are set to make calculating the first maximum and minimum value simpler. Had these values not been set this way, an extra check would have been required to determine whether or not the task had run yet. A simpler explanation is that any value will be greater than 0x0000 and less than 0xffff.

Upon returning from the task, rtos_run() first checks to see if the the timer has overrun. If it has, this means that a minor cycle was completed while the task was still running and therefore the task overran its alloted time. If the timer did overrun, then bit four(4) of the status value for that task will be set. The function then proceeds to add the time that the task took to complete to the total time and to check whether it is less than the minimum time or greater than the maximum time. Changes will be made to the maximum and minimum values if the elapsed time happened to be greater than or less than the respective value. After these two checks, the function continues execution in the same manner as it would if the statistics were not used.

3. rtos_run() with queue

The only addition that the queue makes to the rtos_run() function is the initialization of the two queue index values. These are both set to zero.

4. Inter-task Communication

4.1 Task Queue

The task queue allows a task to receive “messages” from other tasks. The size of each task queue is user defined and the actual queue is managed by the RTOS. A more in-depth look at the task queue can be found under task section 2.4, task queue.

4. Semaphores

The RTOS provides functions for dealing with semaphores. The programmer must declare the variable that will represent the semaphore and make it global so that all of the tasks can access it. The RTOS assumes that if a semaphore is non-zero, then the resource that it represents is available. If the semaphore is zero, then the resource that is represents is considered to be unavailable and the task must wait for it. The reason for this organization is that certain resources may be usable by more than one task at a time. If a resource could be used by two different tasks, then the semaphore would be initialized with the value of two. The first task that used that resource would decrement the semaphore to one alerting other tasks that only one more task can access that resource. If that task finished with the resource, it would increment it back to its initial value. If another task decremented the semaphore while the first was using the resource, the semaphore would contain the value zero and no other tasks could use the resource.

A more in-depth look at functions used to control semaphores can be found under task section 2.3, task control.