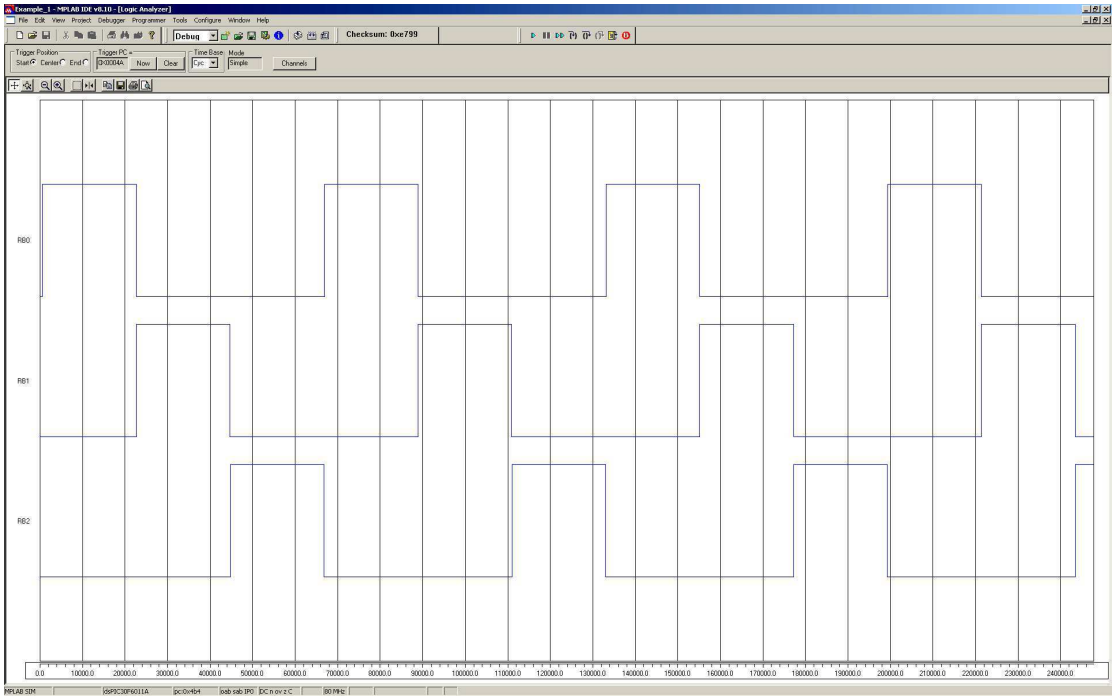

A Multitasking kernel for Microchip Pic30, Pic33 and Pic24 (V3.3)

Here is a lightweight kernel that supports true, time-sliced multitasking using a round robin scheduler. It implements event flags, semaphores and messages. Many programmers will find these basic control structures provide enough functionality for their software. The design aim was to keep the kernel small, fast and easy to use, nonetheless, this multitasking subsystem is not a toy, but a powerful tool that can be used to enhance the performance of your programs.

The Multitasking Kernel.

The software was developed using Mplab IDE version 8.6 and Mplab C30 version 3.25. These were the latest releases at the time of writing. The IDE and student version of the C compiler are available as a free download from Microchip.
The kernel is designed to be used with code written in the C programming language. A multitasking kernel should use minimum resources and task switch in the fastest possible time. In an attempt to achieve these goals, the kernel is written in assembler. The 'dsPic30F Programmers Reference Manual' is available in hardcopy or can be downloaded as a pdf from Microchip. This is essential reading to fully understand the programmer's model and instruction set of the 16-bit microcontrollers.
The program consists of two files, 'multitask.s' contains the core assembler code, and 'multitask.h' defines the C interface to the kernel.

Using The Kernel.

To write a multitasking program you include 'multitask.h' in your C file and add 'multitask.s' to the projects 'Source Files' list.
The kernel needs to steal one timer from the system. This implementation uses Timer 1 for task timing. Timer 1 is sometimes used as a real time clock function. If this is the case, it is quite easy to change timers to one that is not being used.
All instructions operating on the chosen timer have been defined as macros to help with changing timers. Appendix A contains instructions on changing timers.
The Timer 1 interrupt priority is set to 1. Priority level 1 is reserved for use by the kernel. The micro's peripheral nested interrupts can be used as normal. The priority levels for the peripherals should be set to two or greater.

Tasks.

A task is a program running on the microcontroller. A task executes as if it owns the entire CPU. A task is akin to the 'main' function of a C program. The multitasking kernel manages the running of multiple tasks.
A task is not called nor does it return. It should be programmed as a never ending loop using a 'while(True)' statement or a 'for(;;)' ever loop.

Task Template.

```
void task_name(void)
   {
   Local variables;

   for(;;)
      {
      Executable code goes here.
      }
    }
```

All tasks must follow this format.

'Multitask.h' blow by blow.

Two defines set the timing of the multitasking program. These values will be set up to suit the oscillator frequency of the system.

```
/*--- Frequency of Instruction Cycle Clock in Megahertz (Fcy) ---*/

#define FREQ_FCY 10.0
```

This value is the frequency of the internal instruction cycle clock, Fcy.
Include the decimal point as this value is used in the macro to calculate the value for the timer period register.

The 16-bit family has numerous Oscillator options that can be configured to set the speed of the processor oscillator, Fosc.
Fcy is derived from the final set speed of Fosc.

For a dsPic30f, Fcy = Fosc / 4;
For a dsPic33, Pic24, Fcy = Fosc / 2;

```
  /*--- Task time slice in mili Seconds ---*/

#define TIME_SLICE   5.0
```

This value sets up the task switching rate or the time slice duration for each task. This value can be adjusted to suit the response time required by the program. Typical values for this will range from 1 to 25, giving task time slices of 1mS to 25mS.

The maximum time slice that can be set is related to Fcy and the timer prescaler.
With an Fcy of 20MHz, max time slice is (1/Fcy) * 8 * 65536 = 26.2144mS.
If you needed to increase this with a high speed processor, you could change the timer prescaler to 64 in the header file and source code file.

```
/*--- Macro to calculate Timer period (8:1 Prescale value) ---*/

#define PRESCALE  8.0
#define TMR_PERIOD (((TIME_SLICE)*(1000.0))/((PRESCALE)/(FREQ_FCY)))
```

This macro uses the above defines to calculate the value that is loaded into the Timer Period register. This sets the interrupt frequency of the timer to the defined time slice. Unless you wish to change the timer pre-scale value, there is nothing to do here.

```
/*--- System configuration parameters --*/

#ifndef MULTITASK_H
#define MULTITASK_H

/*--- Processor Include files ---*/

#if defined (__dsPIC30F__)
   #include <p30Fxxxx.h>
#elif defined (__dsPIC33F__)
   #include <p33Fxxxx.h>
#elif defined (__PIC24H__)
   #include <p24Hxxxx.h>
#elif defined (__PIC24F__)
   #include <p24Fxxxx.h>
#else
   #error Selected processor not supported
#endif
```

The above lines of code contain the include guard and the include files required by the program. The include guard prevents multiple includes when a program is made up of several C source files.

```
#include "PosixTypes.h"
```

This is a useful file to include in all programs, it is a typdef for the basic numerical types. In the program files you will notice that these types have syntax highlighting. This is a feature of Mplab IDE. There is a file in the include directory called ‘keywords.txt’ that lists the words to be highlighted. If you right click in one of your source files in the IDE and choose ‘Properties’, or choose ‘Edit’ from the menu and ‘Properties’ then click on the ‘Text’ tab. In the ‘Choose Colors’ there is an option for ‘User File Defined.’ Here you can select the color for your highlighting, exit that then use the browse button to select your ‘User Defined Color File’ and click ‘Apply’ This will turn on highlighting for all the words listed in your ‘keywords.txt’ file.

```
/*--- Function pointer type define ---*/

typedef void(*Taskptr)(void);
```

This defines a function pointer type that is used to pass the address of the task in the CreateTask call.

```
/*--- Multitask function prototypes ---*/

void CreateTask(Taskptr task, uint16_t stack_size);
```

This function creates the task. The first argument is a function pointer to the task. The second argument is the size of the task stack.

```
void Multitask(uint16_t tmr_period);
```

After all tasks have been created, this function is called to start up the multitasking kernel. The argument is the previously defined TMR_PERIOD that sets the interrupt rate and task time slice duration.

void TaskSleep(uint16_t count);

This function puts the task to sleep and suspends it from running. Each time it's turn to run comes up in the scheduler, count is decremented. When count reaches zero, it resumes running.

void TaskYield(void);

Calling this function forces an early task switch. This is called if a task becomes idle and can give up the remainder of its time slice to the processor.

void DisableInterrupts (void);

This function disables interrupts. This is used when entering a critical section in your code and you need the task to complete before an interrupt or task switch occurs.

void EnableInterrupts (void);

This function enables interrupts. These two functions are always used together. It is used to resume tasking and interrupts when leaving a critical section.

/*--- Event flag functions ---*/

void WaitForEvent(uint16_t event);

This function sets an event flag, or a combination of event flags and then suspends itself from running. When the event fires and the flag or flags are cleared, it resumes running. The system supports 16 event flags.

void TriggerEvent(uint16_t event);

This function is called to trigger an event and clear the event flag. It can be used as a means of inter task communication or called from a system interrupt.

/*--- Semaphore functions ---*/

void SetSemaphore(uint16_t *sem);
void ClearSemaphore(uint16_t *sem);

These two functions work in conjunction with each other to control access to a resource or a non re-entrant function.

The Re-entrancy Requirement.

Any function that might be called from or shared by two or more tasks must be fully re-entrant, or access to it must be serialised. A re-entrant function is one that uses no global or static data and calls only re-entrant functions. This is easy to see. A task calling a function that uses global or static data might be interrupted while inside the function by a task switch. The new task might also call the function and change the value of the global or static data. When the first task is resumed, the data might not have the value the task assumed it had before. This may lead to system crashes, or worse, subtle, hard to find bugs.
Access to non re-entrant functions must be controlled. The same argument can be applied to hardware resources that cannot be accessed by two or more tasks simultaneously. The process of controlling access to a resource is called serialisation. For historical reasons, the flags used to control serialisation are called Semaphores.

```
/*--- Message passing functions ---*/

void CreateMessage(uint8_t MsgID, uint8_t message_size);
Bool MessageWrite(uint8_t MsgID, void *message);
Bool MessageWaiting(uint8_t MsgID);
Bool MessageRead(uint8_t MsgID, void *message);
Bool ByteWrite(uint8_t MsgID, uint8_t index, uint8_t byte);
uint8_t ByteRead(uint8_t MsgID, uint8_t index);
```

These functions work in conjunction with each other to provide a message passing method for inter task communication. A message can be of any type or structure, providing each task knows what form the message takes.

```
/*--- Debug and Trace functions ---*/

#ifdef __DEBUG
void TraceStack(uint16_t *StackTop);
void TraceTask(uint16_t TaskID, volatile uint16_t *port, uint16_t port_bit);
#endif
```

These functions are used during code development as a debugging aid to trace task stack usage and to visually see when task are running using the Logic Analyzer window. On the Main menu of Mplab IDE in the speed button section, there is a drop down combo box that allows you to switch between ‘Debug’ and ‘Release’ when compiling your project. When set to ‘Debug’, the pre-processor symbol `__DEBUG' is defined. This conditionally compiles these functions. Setting it to ‘Release’ disables them, reducing final code size.

```
#endif /* Include gaurd */

/*--- End of File ---*/
```

Example Programs.

To demonstrate the use of the kernel, 5 small projects have been created.
The files are in a zip file which will create the correct directory structure, the examples should then compile and run ok, otherwise use the Project\Build Options to point the compiler to the correct directories.
All examples make use of MPLAB SIM as the debugger, so no hardware is required to use the examples. Use the ‘Build All’ speed button or Project option and not the ‘Make’ button when compiling the examples. Make will not rebuild ‘multitask.s’ for the processor variant and the linker will complain,

Start MPLAB and open the project ‘example_1.mcp’. This example simulates a dsPic30f6011A and demonstrates how to create tasks. Select MPLAB SIM as the debugger and select ‘Debug’ build.

Example 1

```
#include <p30f6011a.h>
#include "multitask.h"

/*--- Configuration fuses ---*/

_FOSC(CSW_FSCM_OFF & XT_PLL8)
_FWDT(WDT_OFF)
```

After including the processor header file and setting up the configuration fuses we come to this code,

```
/*--- Trace maximum stack usage ---*/

#ifdef __DEBUG
#define NUM_TASKS 5U
uint16_t UsedStack[NUM_TASKS];
#endif
```

One of the more tricky aspects of creating a task is to decide on the optimum stack size for the task. Setting it too small will result in a system crash and setting it too large will result in wasted unused memory. If Debug build is chosen, this array will be defined for use by a stack trace function which tracks the maximum stack size used during program execution. This value is defined as a global so it can be seen in the watch window. To enable real-time watch updates, from the Mplab menu select, Debugger \ Settings \ Animation / Realtime Updates' and enable in the checkbox. After running the program, the maximum stack depth used during execution can be examined.
The NUM_TASKS define should be set to the number of tasks in the program. Failure to do this will result in an illegal address access trap

```
/*--- Task function prototypes ---*/

void task_0(void);
void task_1(void);
void task_2(void);
void task_3(void);
void task_4(void);
```

These are the function prototypes of the tasks we will create in the example program. A task is not called so takes no arguments and does not return a value. It is assumed that more meaningful names would be used for the tasks in a non-trivial program. Tasks have an ID number that is sequentially assigned by the kernel in the order of task creation starting at 0.

```
/*--- Local function prototypes ---*/

void init_io(void);
void Delay(uint16_t delay);
```

Here would be the function prototypes that make up the rest of the program.

```
/*--- Global variables ---*/

uint16_t var0 = 0;
uint16_t var1 = 0;
uint16_t var2 = 0;
uint16_t var3 = 0;
uint16_t var4 = 0;
```

These are global variables so that they can be seen being updated in the watch window when real time updates have been enabled in the Mplab Sim debugger.

```
/*--- Program Entry ---*/

int main(void)
{
init_io();
```

The micros peripherals could be initialized here. In this example, we just set Port B to outputs for use by the Logic Analyser.

IMPORTANT!
If initialising peripherals that generate interrupts, initialise them after creating the tasks and before calling 'Multitask(TMR_PERIOD);' that starts the program.
No interrupts should occur before all the tasks have been created and the multitasking system is started.

```
CreateTask(task_0, 56);
CreateTask(task_1, 56);
CreateTask(task_2, 56);
CreateTask(task_3, 56);
CreateTask(task_4, 56);
```

IMPORTANT!
The sequential creation of all the tasks should be one of the first things to do on program entry. You cannot have code in between the CreateTask calls, as this will prevent the kernel from creating a circular linked list of task structures.
Failure to do this will result in undefined behaviour.

IMPORTANT!
The CreateTask function Must be called from within the programs 'main()' function.
The memory allocation algorithm for the tasks needs to know the depth of the call stack to successfully allocate memory.

Task creation is static. All tasks for the program are created at the beginning of the program and exist for the duration of the program.

The first argument to CreateTask is a function pointer to the task, which is simply the function name.
The second argument is the size of the stack. The size of the stack will depend on the number of local variables used by the task, the number and nesting of function calls it makes and the local variables used by the called functions. The Micros peripheral hardware interrupts also use the stack to save context. If nested interrupts are used, the stack can grow quite large.
The size here is the depth of the stack and not the number of bytes. These are 16-bit processors, so a stack depth of 40 takes up 80 bytes of memory. In this example we set the stack depth to 56. We can use the watch window to see how much stack is actually used during program execution.

The kernel manages the tasks state and stack internally. A task can be in one of four states,

READY, the task is ready to run.
BLOCKED, the task is waiting for access to a resource.
WAITING, the task is waiting for an event.
ASLEEP, the task has suspended itself from running.

As tasks are created, the kernel creates a circular linked list of task structures.

```
Struct task
  {
  Pointer to next task in list;
  Pointer to this tasks stack top;
  The task ID (Hi-Byte) and the taskstate (Lo-Byte);
  Semaphores pending;
  Event flags;
  Sleep count;

#Ifdef DEBUG
    Trace Port
    Trace Bit
  #endif

    Pointer to this task function;
    Hi word of function pointer;

    If Pic30 or Pic33
       Registers[58] ;
    Else
       Registers[36] ;

    CORCON;
    PSVPAG;
    }
```

For a Pic30 or Pic33, a task needs a stack of 80 Bytes to store its state.
A Pic24 needs 22 fewer bytes, as it does not have the DSP registers.

The next lines of code set up the stack trace function and the task trace functions. The task must have been created before calling these functions.

```
#ifdef __DEBUG
TraceStack(&UsedStack[0]);
TraceTask(0, &PORTB, 0);
TraceTask(1, &PORTB, 1);
TraceTask(2, &PORTB, 2);
TraceTask(3, &PORTB, 3);
TraceTask(4, &PORTB, 4);
#endif
```

The TraceStack function call with the address of the previously defined UsedStack[] array as an argument initialises the array for use by the kernel.
The TraceTask functions allow us to see the execution of the tasks using the 'Simulator Logic Analyzer' window. The scheduler will set an I/O port line high when the task is started and clear the line when the task is stopped.
The first argument is the task ID. ID's are assigned by the kernel in the order of task creation, beginning at 0.
The second argument is the address of the I/O port to use and the third argument is the Port I/O line.
Here we will use Port B and I/O lines RB0, RB1, RB2, RB3 and RB4 to trace task execution.
To enable the Logic analyser, from the Mplab menu select,
'Debugger / Settings / Osc / Trace'. And enable the check box 'Trace All'.
Set the Buffer Size to 1 M lines.
OK that, then go to 'View' and select 'Simulator Logic Analyzer'.
In the Logic Analyzer window, select 'Channels' and select RB0, RB1, RB2, RB3 and RB4.

If debugging hardware and the task trace functions are not required, delete or comment out these calls. The kernel only calls these functions if they have been defined.

Multitask(TMR_PERIOD);

This function call starts the multitasking kernel. The arguments are the previously defined TMR_PERIOD in ‘multitask.h’ that set the timing of the program.

return 0;
} /* Closing brace of main */

The remainder of the program is the coding for the tasks.
As this example is just to demonstrate the creation of tasks and task traces and the overall structure and layout of a program, the tasks do nothing but wait in an infinite loop and increment their global variables, task_1 goes to sleep for two turns to demonstrate the sleep function. This is useful if you have a low priority task that only needs to run at a lower frequency

The tasks call the re-entrant function Delay(delay); This function is safe to call as it uses no global or static data and calls no other functions.

At the end of the program we have a conditional compile of the Stack overflow/underflow trap and the illegal address access trap. If the stack size is too small and overflows, or the program attempts an illegal memory access, the program will end up here.

On the Mplab menu, select ‘View/Watch’ to view the watch window. Select ‘Add Symbol’ in the watch window and add the ‘UsedStack’ array. Variables can also be dragged and dropped from the source code into the watch window. Expand the ‘UsedStack’ variable and right click on a value. Select properties and change the display to decimal.
UsedStack[0] is the task id and the value is the maximum stack depth used by the task.

Select Watch 2 tab and add the var0 to var4 global variables to the watch window. These can be seen being updated as the program runs.

Build the program and run it. The watch window will be updated in real time. If you pause the program, the logic analyser will display the order that the tasks have run. Unfortunately, the Logic Analyzer does not update in real time while the program is running. One way around this is to place a breakpoint in a task. Now, running the program to the breakpoint, you can see the Logic Analyzer tracing the task switches.
You can use the ‘Zoom Axis’ and ‘Scroll Axis’ speed buttons in the Logic Analyzer to zoom into a task switch. Using the cursors, you can see the number of instruction cycles needed to affect a task switch.

To speed up single stepping in the debugger, uncheck 'Trace All' in the debugger settings.

The points to remember are that the list of tasks should be created with one unbroken sequence of ‘CreateTask’ function calls from within the ‘main’ function.
Multitasking will start with the first task in the list.

Example_2.

The second project illustrates the use of event flags. Open project ‘Example_2.mcp’. This code is essentially the same as example_1, but now simulates a dsPic33FJ256GP710.

Example_2 adds the following defines.

```
/*---- Define Event flags as single bits ---*/

#define EVENT_0 0x0001
#define EVENT_1 0x0002
#define EVENT_2 0x0004
#define T2_INTERRUPT 0x0008
```

Each task has a 16-bit integer to store event flags. Each bit can be an event flag. To use event flags, you would normally define some meaningful names to the bits. The event flags are common to all tasks. If two tasks are waiting for EVENT_1, both will resume running when the event is cleared.

```
WaitForEvent(EVENT_1);
```

The function call ‘WaitForEvent’ sets the event flag, changes the task state to WAITING and gives up the rest of its time slice.
As event flags are bits, event flags can be combined. For example, one could call,

```
WaitForEvent(EVENT_1 + EVENT_2) ;

TriggerEvent(EVENT_1);
```

Calling TriggerEvent(EVENT_1); clears the event flag and sets the tasks state waiting on the event to ready. The task will now resume running next time it’s time slice occurs.

On the Debugger settings menu, set the Trace buffer size to 2 M lines and check the 'Break on Trace Buffer Full' checkbox. Build and run the example, the program will break when the trace buffer is full and display the results in the logic analyzer window. This make take a few seconds to update depending on the speed of your system.

This example also demonstrates the use of a peripheral interrupt to clear an event flag. Timer 2 is initialised with its interrupt priority set to 3. All peripheral interrupts should have a priority of 2 or greater to prevent contention with the kernel.
The default priority level for the peripheral interrupts on processors reset is 4.
Timer 2 sets Port bit RC1 on entry and clears it on exit. Add channel RC1 to the Logic Analyzer window.
Timer 3 is initialised with its interrupt priority set to 5. This will result in peripheral nested interrupts to occur. The stack sizes of the tasks have been increased to 64 to insure this does not cause a stack overflow. Timer 3 interrupt toggles Port bit RC5. Use the cursors in the Logic Analyzer window to help see the sequence of events.

Example_3.mcp

This project demonstrates the use of Semaphores to control access to a shared resource, but now simulates a Pic24FJ128GA010. The stack structure is smaller as this micro doesn’t have the dsp registers of the Pic30/33 series.

Semaphores are implemented as Global unsigned integers that must be initialised to 0 before they are used. These variables must be accessible to all the functions that wish to use them. There is no limit to the number of semaphores used.

/*--- Define and initialise global Semaphores to 0 ---*/

```
uint16_t Semaphore_1 = 0;
uint16_t Semaphore_2 = 0;
```

The semaphore functions are always used together. The following code fragment shows one way to use the semaphore functions to control access to a function called 'Critical_function(uint8_t arg, uint8_t *var);'

The argument to SetSemaphore and ClearSemaphore is the address of the semaphore.

```
void Critical_function(uint8_t arg, uint8_t *var)
  {
  SetSemaphore(&Semaphore_1);

  /* functions code */

  ClearSemaphore(&Semaphore_1);
  }
```

To help visualise what is happening, the Critical function sets RC1, RC2 or RC3 high depending on which task has the semaphore, then low when the function exits.
This can be seen in the Logic Analyzer window.
In the debugger/Settings menu, the 'Break on Trace Buffer Full' checkbox should be selected and the Buffer size set to 2 M Lines. This will stop the program and display a full cycle in the Logic Analyzer window.

IMPORTANT!
The semaphore functions are for the use of the tasks and the kernel, they cannot be called from a peripheral interrupt. It would make no sense to do so. The peripheral interrupts always have higher priority than the tasks. If an interrupt called a semaphore function and was blocked on a semaphore, the multitask kernel would stop as the task timer interrupt has the lowest priority. If using nested interrupts, the peripheral interrupts should use the functions DisableInterrupts(); and EnableInterrupts(); to manage critical sections.

Example_4.mcp

This project demonstrates the message passing functions, but now simulates a Pic24HJ64GP206. This variant has a more complex oscillator configuration. For this demo, we will leave the Frequency of Oscillator setting at 40MHz.

/*--- Message passing functions ---*/

```
void CreateMessage(uint8_t MsgID, uint8_t message_size);
Bool MessageWrite(uint8_t MsgID, void *message);
Bool MessageWaiting(uint8_t MsgID);
Bool MessageRead(uint8_t MsgID, void *message);
Bool ByteWrite(uint8_t MsgID, uint8_t index, uint8_t byte);
uint8_t ByteRead(uint8_t MsgID, uint8_t index);
```

A message can be of any type, it could be a structure, an array, a value or a character string, so long as the task sending the message and the task receiving the message know what form the message takes. The size of the message can be up to 255 bytes.
An enumeration should be defined to give the messages meaningful names and also create the message ID's which should be zero based. IE: 0,1,2,3,4 etc.

/*--- Message names ---*/

```
enum{TXT_MSG = 0, STRUCT_MSG, ISR_MSG, BUFFER};
```

Creating messages.

```
CreateMessage(TXT_MSG, 12);
CreateMessage(STRUCT_MSG, sizeof(TESTSTRUCT));
CreateMessage(ISR_MSG, 24);
CreateMessage(BUFFER, BUFFERSIZE);
```

The first argument to the CreateMessage function is the message ID, the second argument is the message size in bytes. Memory for the messages is assigned from the top of the memory space growing downwards. The kernel sets the top of the system stack below the messages.

IMPORTANT!
The calls to CreateMessage should be after the calls to CreateTask and before the multitasking kernel is started. All calls to CreateMessage should be in one unbroken sequence.
Messagebox creation is static. All messages for the program are created at the beginning of the program and exist for the duration of the program.

The kernel manages the Messages internally and creates a linked list of message structures.

```
Struct message
   {
   Pointer to next message in list;
   Message ID;
   Message Size;
   MessageBuffer[Sizeof message];
   }
```

Message Functions.

```
Bool MessageWrite(uint8_t MsgID, void *message);
```

This call writes a message to the message box. The first argument is the message ID. The second argument is the address of the array or variable containing the message. The function returns True if the call was successful, and False if there was an error. The call will only fail if the MsgID is invalid.
After the message has been written, the function sets the internal message flag to indicate the message box is full.

```
Bool MessageWaiting(uint8_t MsgID);
```

This function returns the state of the messages internal flag. True if a message is waiting to be read and False if the message box is empty or has been read.

```
Bool MessageRead(uint8_t MsgID, void *message);
```

This call reads the message. The first argument is the message ID. The second argument is the address of the array or variable to receive the message.
The function returns True if the call was successful, and False if there was an error. The call will only fail if the MsgID is invalid.

After the message has been read, the function clears the internal message flag to indicate the message box has been read.
The message state flag just serves as an indicator. The flag being set does not prevent the message box from being written to. A call to MessageWrite with the flag being set will just over write the existing message.

A call to MessageRead reads the message but does not clear it. A call to MessageRead with the flag being clear will read the contents of the last written message.
The read and write message functions can be called from the processors peripheral interrupts. The event flags can be used to make a task suspend itself from running until a message is ready. The task sending the message can trigger the event flag after writing the message.

Build and run the example. Add the message read arrays and variables to the Watch window. Place breakpoints just after the MessageRead calls to see the results in the Watch window.

Example_5.mcp

This project demonstrates the Read/Write byte message functions, It simulates a Pic24FJ256DA210. This variant has DSRPAG and DSWPAG registers in place of the PSVPAG register.
Uncomment this line at the top of 'multitask.s' file to add support for this device.

```
.equiv NO_PSV_PAGE,1          ;Adds support for PIC24FJ256DA210 family.
```

Byte Read/Write Functions.

Bool ByteWrite(uint8_t MsgID, uint8_t index, uint8_t byte);

This function allows you to write a single byte to a message structure. The first argument is the message ID. The second argument is the index into the message buffer and the third argument is the byte value to write. The function returns true if the byte was written and False if there was an error. The call will fail if the index is out of bounds or the MsgID is invalid.

uint8_t ByteRead(uint8_t MsgID, uint8_t index);

This function allows you to read a single byte from a message structure. The first argument is the message ID. The second argument is the index into the message buffer. The function returns the value of the byte at the index in the message buffer
If the index is out of bounds or the MsgID is invalid, the function will return 0. Not a lot of help? Just ensure the index value and MsgID are valid before making the call!

These two functions can be used to create data structures such as stacks and circular buffers.

Example 5 implements a simple stack and a circular buffer. The circular buffer is filled from an interrupt and read by one of the tasks into the global variable data[8]; The data array is global so that it can be seen being updated in the watch window if real-time watch updates are enabled.

Conclusion.

The examples have demonstrated the use of the kernel using one of each variant in the Microchip 16-bit family.

Some simple rules need to be followed by the programmer to make the system work.

Define FREQ_OSC.
Define TIME_SLICE

The unbroken sequential creation of tasks from within main().
The unbroken sequential creation of messages.
Multitasking will start with the first task in the list.

Tasks are programmed as the task template, a never-ending loop that does not return.
Peripheral interrupts should have a minimum priority of 2.

The goal was to produce a lean, mean and fast multitasking system that is simple to use.
Just three basic control mechanisms of rtos systems have been implemented.
Nevertheless, complex systems can be built up from simple building blocks.

Appendix A.

System Timers.

Depending on the specific variant, the 16-bit device family offers several timers.
These timers are designated as Timer1, Timer2, Timer3, ..., etc.
Each timer module is a 16-bit timer/counter consisting of the following

readable/writable registers:

• TMRx: 16-Bit Timer Count register
• PRx: 16-Bit Timer Period register associated with the timer
• TxCON: 16-Bit Timer Control register associated with the timer

Each timer module also has the associated bits for interrupt control:

• Interrupt Enable Control bit (TxIE)
• Interrupt Flag Status bit (TxIF)
• Interrupt Priority Control bits (TxIP<2:0>)

With certain exceptions, all of the 16-bit timers have the same functional circuitry.
The 16-bit timers are classified into three types to account for their functional differences:

• Type A time base
• Type B time base
• Type C time base

Some 16-bit timers can be combined to form a 32-bit timer.
Some are dedicated timers that are associated with peripheral devices. For example, this includes the time base associated with the input capture or output compare modules. A timer can trigger an A/D conversion. Timer1 has support for implementing a real time clock.

Changing system Timers.

The timer selected for the task time slice switching, should be one that is not being used by the peripheral devices in the program.

All instructions operating on the chosen timer have been defined as macros in ‘multitask.s’

To change the timer, consult the data sheet for the selected processor and change the registers in the macro to suit the new timer.

Then change the interrupt vector for the selected timer.

In the following example, the macro has been edited to use Timer 4 as the task timer and the interrupt vector has been changed to the __T4Interrupt.

```
/*--- Macros for Timer interrupt ---*/

.macro StartTaskTimer          ;Initialise task timer
mov w0,PR4                     ; TMR_PERIOD  to timer period register
clr TMR4                       ;Clear timer
bset IPC6,#T4IP0               ;Set timer priority to 1
bclr IPC6,#T4IP1
bclr IPC6,#T4IP2
bclr IFS1,#T4IF                ;Clear interrupt flag
bset IEC1,#T4IE                ;Enable interrupt
mov #0x8010,w0
mov w0,T4CON                   ;Start Timer, Prescale 1:8
.endm

.macro ClearInterrupt
bclr IFS1,#T4IF                ;Clear interrupt flag
.endm

.macro StartTasking            ;Enable Tasking interrupt
bset IEC1,#T4IE                ;Set interrupt enable flag
.endm

.macro StopTasking             ;Disable Tasking interrupt
bclr IEC1,#T4IE                ;Clear interrupt enable flag
.endm


/*--- Global Task Functions ---*/

.global __T4Interrupt          ;Change the interrupt vector prototype

/*--- Task scheduler Timer interrupt ---*/

__T4Interrupt:                 ;Change the interrupt function vector
```

Rebuild the program. Timer 4 will now be used by the kernel for task switch timing.