Introduction

µTasker is an operating system designed especially for embedded applications where a tight control over resources is desired along with a high level of user comfort to produce efficient and highly deterministic code.

The operating system is integrated with TCP/IP stack and important embedded Internet services along side device drivers and system specific project resources.

µTasker and its environment are essentially not hardware specific and can thus be moved between processor platforms with great ease and efficiency.

However the µTasker project setups are very hardware specific since they offer an optimal pre-defined (or a choice of pre-defined) configurations, taking it out of the league of "board support packages (BSP)" to a complete "project support package (PSP)", a feature enabling projects to be greatly accelerated.

Files

The following files are used to define the operating system configuration and use in their order of respective absolute importance:

```
types.h                 type defines used by uTasker
uTasker.h               operating system defines and prototypes
config.h                system configuration(s)
driver.h                driver defines and prototypes
stack\tcpip.h           tcp/ip stack defines and prototypes
hardware\hardware.h     hardware interface prototypes

hardware\cpu\cpu.c      hardware specific code for initialisation,
                        interrupts and physical accesses
uTasker.c               operating system routines
watchdog.c              system watchdog task
uMalloc.c               memory management
driver.c                driver interface
tty_drv.c               serial interface driver
xxx_drv.c               various other driver interfaces
global_timer.c          optional global timer task
stack\xxx.c             various files for TCP/IP support routines
```

In addition to these files there are a number of application files which may be required. In a basic configuration there is usually one application task file, which we will call application.c.

Task concept and configuration

µTasker configures itself in a very dynamic fashion so that it uses only the necessary amount of space for its operation. The user should supply a configuration table containing the tasks which should run and the resources they need. It is recommended that the user performs this as follows:

```
int main(void)
{
  fnInitHW();                        // perform hardware initialisation if required

  fnInitialiseHeap(ctOurHeap);       // define dynamic heap size

  // start the operating system
  uTaskerStart((TTASKTABLEINIT *)ctTaskTable, ctNodes, PHYSICAL_QUEUES);*2

  while (1) {
    uTaskerSchedule();
  }
  return 0;                          // …in fact we never return
}
```

*1 See the section "Heap memory utilisation" for details about heap initialisation

*2 `TASK_LIMIT uTaskerStart(`

```
     const UTASKTABLEINIT    *ptATaskTable,
     const signed char       *a_node_descriptions,
     const PHYSICAL_Q_LIMIT  nr_physical_queues);
```

The user passes a list of all possible tasks which could be used in the system and a string defining which tasks should be entered into the process table on starting. This allows variable configurations to be selected at start.

Nr_physical_queues tells µTasker how many queues will be needed for physical interfaces (serial, IIC, LAN, SPI etc.)

The reference task table and the node description are defined in config.h. Here is an example:

```
const UTASKTABLEINIT ctTaskTable[] = {
{ "Wdog", fnTaskWatchdog, NO_QUE, 0,(unsigned short)( 0.2 * SEC ), RTMK_GO},
{ "Blink", fnTaskWatchdog, SMALL_QUE, (unsigned short)( 0.5 * SEC ),
(unsigned short)( 0.5 * SEC ), RTMK_GO},
{ 0 }                                                  // end of task list
};

NODE_LIMIT OurNodeNr = 1;
```

Using OurNodeNr it is possible to configure depending on a node address, for example read in from a DIP switch on power up in `fnInitHW()`. In singe node systems it can be a constant value.

```
const signed char ctNodes[] = {
  1,                // configuration number 1
  TASK_WATCHDOG,    // watchdog task
  TASK_BLINK,       // Blink task
  0,                // end of configuration 1

  2,                // configuration number 2
  TASK_WATCHDOG,    // watchdog task
```

```
  0,                // end of configuration 2

  3,                // configuration number 3
  TASK_BLINK,       // Blink task
  0,                // end of configuration 3

  0                 // end of configuration list
};
```

If the node 1 is started, the tasks WATCHDOG and BLINK will be configured. WATCHDOG has no input queue, no start delay and is started periodically every 200ms. BLINK has a small queue size reserved for receiving internal messages, is started the first time after a delay of 500ms and then periodically every 500ms.

If node 2 is configured, only the WATCHDOG task will be entered and scheduled. No resources will be allocated for use by BLINK.

If node 3 is configured, only the BLINK task will be entered and scheduled. No resources will be allocated for use by WATCHDOG.

Note that the task names are available as strings but all referencing is performed by comparing the initial letter of the task name. This is very efficient but care must be taken that each task has a unique letter at the start of its name. TASK_BLINK is in this case a simple define to the first letter #define TASK_BLINK ‘B’.

Tasker RAM use

µTasker uses the following variables of its own to operate:

```
UTASK_TICK uTaskerSystemTick  = 0; // system tick counter
static TTIMETABLE *tTimerList = 0; // pointer to timer list
static TTASKTABLE *tTaskTable = 0; // pointer to process table
```

The timer list and task table are created from heap depending on the number of tasks configured, their use of task timers and their use of queue space. That means that only space from heap is used where absolutely necessary as defined by the user.

Further variables are maintained by the queue driver and other interfaces defined in the system, which are detailed in the appropriate section.

Tasking

The routine uTaskerSchedule() should normally be called from a forever loop. It allows µTasker to schedule the tasks depending on their state. They may be running at every occasion, be suspended, or be scheduled when a message is in their input queue or when a timer or interrupt event wakes them.

The scheduling order is as defined in the configuration table beginning from the top of the list and ending at the bottom of the list before starting at the top again.

Task states

#define UTASKER_STOP 0x00
    // task stopped. Waiting for event to cause it to run again.

#define UTASKER_GO 0x01
    // task free to run. It will however only run once if it has a repetition timer set. Otherwise this is equivalent to polling mode

#define UTASKER_SUSPENDED 0x02
    // the task is suspended when its monostable timer has been stopped

#define UTASKER_ACTIVATE 0x04
    // the task is scheduled to run once

The initial task state is defined in the task table. When a task uses timers it is usual to set the initial state to UTASKER_STOP and the task will be woken by the timer event(s).

To allow a task to run immediately UTASKER_GO can be set. A task with a periodic timer defined will set itself to UTASKER_STOP mode after its first scheduling and will then be woken according to timer events. Others will continue to run in polling mode (once per scheduling cycle, meaning as fast as possible).

UTASKER_ACTIVATE will also cause the task to be scheduled immediately but it will always be set back to UTASKER_STOP mode after it has run once, irrespective of whether it is configured for periodic timer operation or not.

Tasks can modify their own states or the state of other tasks as desired. For example a task can set itself into polling mode until it has detected something it is scanning for (polling) and afterwards set itself back to an inactive state.

Another task can stop a periodic task from operating or start its periodic function again.

Some examples:

```
uTaskerStateChange(TASK_KEY_SCAN, UTASKER_SUSPENDED);
```

The task used to scan keys will be suspended. This means that its periodic key scanning timer will be stopped.

```
uTaskerStateChange(TASK_KEY_SCAN, UTASKER_STOP);
```

The task used to scan keys will be set to the stopped mode. This will restart its periodic timer and the key scanning will operate periodically again.

```
uTaskerStateChange(OWN_TASK_TIMER, UTASKER_GO);
```

This task switches its own operation to polling mode (it runs as often and as fast a possible). This assumes that it doesn't have a periodic timer configured.

```
uTaskerStateChange(OWN_TASK_TIMER, UTASKER_STOP);
```

This task switches itself to the stop state and it will only be woken again by specific events.

```
uTaskerStateChange( ANOTHER_TASK, UTASKER_ACTIVATE);
```

This simply causes another task to be scheduled.

TICK and Timers

Mono-Stable Timer

The µTasker supports several timer strategies, the simplest being the mono-stable timer associated to a task. Timer operation is based on a system TICK interrupt and a microprocessor timer is always reserved for this job.

The TICK resolution is defined in config.h.

#define TICK_RESOLUTION 50 // define a 50ms operating system TICK

The periodic timer interrupt is initialised in uTaskerStart(), when the system start.

Each task in the system can have one mono-stable timer. This mono-stable timer is automatically allocated when the task is defined to start delayed, or to run periodically, since the µTasker uses this mono-stable timer to achieve this task. Should the task code want to make use of a mono-stable timer during normal operation it needs to ensure that the task receives the timer resources by defining a start delay or else by setting the delay value NO_DELAY_RESERVE_MONO (maximum possible value), which will cause the resource to be allocated without actually performing a delay.

The timer can then be used as a mono stable timer, which is practical for most applications and especially for monitoring protocol timeouts. The timer can be started and restarted (retriggered) and stopped. Should it time out, it will cause the task to be woken with a timer event message to recognise the cause.

```
#define T_WAIT_BEFORE_FIRST_CHECK (7*SEC)
#define E_CHECK_MAIL              2
uTaskerMonoTimer( OWN_TASK, T_WAIT_BEFORE_FIRST_CHECK, E_CHECK_MAIL );
```

This example shows a timer being started (or retriggered) with a 7 second timeout. When the timer times out it will wake the task with the event E_CHECK_MAIL. *Warning: The event number 0 should never be used as timer event since it signals a non-event.*

Note that each task has only one mono-stable timer and starting another timer for this task – whatever it is another delay or another event - will take over a presently running mono-stable timer for the defined task. A second call of the same mono-stable timer function acts thus as a retrigger if the first timer had not yet timed out.

```
uTaskerStopTimer( OWN_TASK );
```

This call is used to stop a mono-stable timer before it has fired. Note that the task is always used as timer reference.

The mono-stable timer feature is adequate for many designs – indeed it is rarely necessary to need any more extended support. However there are additional timer features which can be configured, if required.

Global Timer Task

By setting `#define GLOBAL_TIMER_TASK` in the project config.h file, a dedicated timer task is included which allows a task or any number of tasks to define additional mono-stable type timers for their own use. The maximum number of mono-stable timers handled by the global timer task is defined by `#define TIMER_QUANTITY 5` (for example).

The use of the global timers is very similar to that of the tasks own mono-stable timer. The previous example can be modified to use global timers as follows

```
uTaskerGlobalMonoTimer( OWN_TASK, T_WAIT_BEFORE_FIRST_CHECK, E_CHECK_MAIL );
uTaskerGlobalStopTimer( OWN_TASK, E_CHECK_MAIL );
```

Note that in addition the timer event must be specified when stopping a global mono-stable timer since there may be a number of different events belonging to the calling task.

When a timer event occurs, it is handled identically, whether from the task's mono-stable timer or from the global timer – see examples of timer event handling in subsequent sections. Not too that the task must be defined with an input queue in order to be able to receive timer events.

Hardware Timer Queue

A third technique supported by the µTasker uses a second hardware timer to generate high resolution timeouts. Before discussing this further it is important to understand the limitations of the mono-stable timers so that the benefits of this possibility can be appreciated.

The mono-stable timers are built around the system TICK and so have a resolution determined by the repetition rate of the system TICK itself. The resolution can be increased by decreasing the TICK repetition period but this may unnecessarily increase the processing demand on the CPU as it has to handle the TICK interrupt at a faster rate. The maximum delay possible if defined by the storage type DELAY_LIMIT which is defined in `types.h` for the specific project. Generally an unsigned short is practical since a typical TICK values of 50ms gives a maximum timeout period of about 54 minutes (65'535 x 50ms), which is adequate for the majority of timeouts.[Note that it if of course quite easy to expand the timeout at the application layer when extremely long timeouts are necessary. See the DHCP code for an example since long DHCP timeouts are quite common and a maximum of around 100 years is supported].

The third thing to note is that the programmed delay is synchronous to the TICK itself. If a task is synchronous to the TICK (for example it was woken due to a timer event) a new delay will be very accurate since the delay's first delay unit starts at the beginning a delay period. If the task starting the delay is however not synchronous to the TICK, for example it starts a delay on receiving a message from the serial port which can arrive randomly, the first TICK unit has a resolution of 0..TICK_RESOLUTION. All such delays will thus have a resolution of the programmed delay +0/- TICK_RESOLUTION and this becomes more noticeable as the delay period decreases.

This may not be an issue in the majority of applications and the worst case can be quite well controlled by TICK_RESOLUTION. Hardware timer support does however offer an improvement when short delays must have a higher resolution than that practical using the mono-stable support – the down-side is that a free hardware timer is required, but, if this is possible, the µTasker support can be activated by setting `#define GLOBAL_HARDWARE_TIMER` in conjunction with `#define GLOBAL_TIMER_TASK`.

The use of the global timer with hardware support requires the following modification when starting the timer. Stopping a timer and the timeout itself is handled in an identical manor.

```
uTaskerGlobalMonoTimer((signed char)(OWN_TASK | HARDWARE_TIMER),// hardware timer
                                   (DELAY_LIMIT)(23/MSEC),      //delay in ms
                                   E_CHECK_MAIL );
```

The maximum possible delay depends on the configuration possibilities of the hardware timer itself. Generally a few hundred ms to a few seconds are possible – see the notes concerning the specific processor type support [eg. NE64 limit is 167ms at 1ms resolution]. There may however be some flexibility depending on the necessary resolution actually required. Obviously such high resolution timers are of primary use for quite short but accurate delays, whereas the standard mono-stable delays are usually adequate for general purposes. In any case, the user has quite a high degree of flexibility in order to get the best results for the job in hand.

Task structure

A task generally has the following structure

```
void fnApplication(TTASKTABLE *ptrTaskTable)
{
    static int iState = 0;

    QUEUE_HANDLE PortIDInternal = ptrTaskTable->TaskID;
    unsigned char ucInputMessage[MEDIUM_MESSAGE];

    if (!iState) {
        fnTaskInitialisation();
        iState = 1;
    }

    while (fnRead( PortIDInternal, ucInputMessage, HEADER_LENGTH)) {
      switch (ucInputMessage[MSG_SOURCE_TASK]) {
        case TIMER_EVENT:
           if (ucInputMessage[MSG_TIMER_EVENT] == E_WAIT_BEFORE_SENDING) {
             //
           }
           else if (ucInputMessage[MSG_TIMER_EVENT] == E_PROTOCOL_TIMEOUT) {
             //
           }
           break;

        case INTERRUPT_EVENT:
           if (ucInputMessage[MSG_INTERRUPT_EVENT] == E_PORT_CHANGE) {
             //
           }
           break;

       case TASK_TEMPERATURE:
          fnRead( PortIDInternal, ucInputMessage,
                                   ucInputMessage[MSG_CONTENT_LENGTH]);
           if (E_NEW_TEMPERATURE == ucInputMessage[0]) {
             //
           }
           break;
      }
    }
}
```

The first time a task is scheduled, it may need to perform an initialisation function, after which it sets its own state to a different value so that the initialisation is not performed again on subsequent executions (unless of course this were desired).

If a task has an input queue, defined in the task configuration, it checks and processes its input queue by reading from its queue reference, passed by the scheduler when it is called. All internal messages are built up with a defined message header, which allows for passing messages between tasks and also between nodes:

1. Destination Node
2. Source Node
3. Destination Task
4. Source Task
5. Message data content length or special event

Depending on the source task, a received message is recognised as either a special event (timer or interrupt) or a message from another task. In the case of special events, the header contains an event rather than details of additional content length and so the event can be quickly processed as it doesn't carry data.

Messages between tasks

Just as task mono-stable timers use timer messages to wake a task and inform of an event, tasks can wake other tasks and communicate by sending internal messages.

Here is an example of a routine used to send messages or maximum length MAX_LCD_MESSAGE from one task to a predefined task called TASK_LCD.

```
static void fnSendToLCD(unsigned char ucCommand, unsigned char *ucData, unsigned
char ucLength)
{
  unsigned char ucMessage[ HEADER_LENGTH + 1 + MAX_LCD_MESSAGE];

  ucMessage[ MSG_DESTINATION_NODE ]  = INTERNAL_ROUTE;
  ucMessage[ MSG_SOURCE_NODE ]       = INTERNAL_ROUTE;
  ucMessage[ MSG_DESTINATION_TASK ]  = TASK_LCD;
  ucMessage[ MSG_SOURCE_TASK ]       = OWN_TASK;
  ucMessage[ MSG_CONTENT_LENGTH ]    = ucLength+1;
  ucMessage[ MSG_CONTENT_COMMAND ]   = ucCommand;
  uMemcpy(&ucMessage[MSG_CONTENT_EVENT+1], ucData, ucLength);
                           // add payload

  fnWrite(INTERNAL_ROUTE, ucMessage, HEADER_LENGTH + ucLength + 1);
                           // pass the message to the destination task
}
```

First the message is built containing routing information (if no node routing is required, the destination and source node values should be set to INTERNAL_ROUTE). The task wants to send a command with a certain amount of data to the LCD task. The length field is set to include the data length plus a command byte, informing what the data signifies.

A write with a handle INTERNAL_ROUTE is interpreted as an internal message and it will be put in to the input queue of the receiving task if this can be found (write returns the number of bytes copied; this will be zero if the destination task were not to be found).

The receiving task will be woken and can read the messages as soon as it is scheduled again.

Since the data is copied to the internal queue of the receiver, it can immediately be destroyed if no longer required locally (as in the example it is built on the temporary stack, which is no longer valid after the subroutine is exited).

If large data blocks require to be passed from one task to another, which are static to the transmitting task, it is possible to transmit a pointer to the block rather than the complete block. The sending task must however ensure that the block remain valid until the receiver task has used it. This technique can however not be used across multiple physical nodes.

Another technique is offered for increased efficiency with respect to the use of stack in the message preparation routine.

By setting `#define SUPPORT_DOUBLE_QUEUE_WRITES` (in config.h) the following code can be used. The queue system also remains compatible with the other technique in this case, meaning that both methods can be used within the project:

```
// message sending example using SUPPORT_DOUBLE_QUEUE_WRITES

static void fnSendToLCD(unsigned char ucCommand, unsigned char *ucData, unsigned
char ucLength)
{
  unsigned char ucMessage[ HEADER_LENGTH + 1]; *1

  ucMessage[ MSG_DESTINATION_NODE ]  = INTERNAL_ROUTE;
  ucMessage[ MSG_SOURCE_NODE ]       = INTERNAL_ROUTE;
  ucMessage[ MSG_DESTINATION_TASK ]  = TASK_LCD;
```

```
  ucMessage[ MSG_SOURCE_TASK ]        = OWN_TASK;
  ucMessage[ MSG_CONTENT_LENGTH ]     = ucLength+1;
  ucMessage[ MSG_CONTENT_COMMAND ]    = ucCommand;

  fnWrite(INTERNAL_ROUTE, ucMessage, 0);*2
  fnWrite(INTERNAL_ROUTE, ucMessage, ucLength);*3
}
```

*1 Only the header plus command needs to be built on the local stack
*2 First a write of the header is performed with a length of zero (this is interpreted as a message header)
*3 Secondly the data content itself is sent

Events

There are three event types: TIMER_EVENT; INTERRUPT_EVENT and user definable events.

It has already been seen that when a task mono-stable timer fires, it wakes its associated task with an event and an event code. The timer event code was allocated when the specific timer delay was started and is used to interpret the exact timer event to process. A timer event is received as a message with only a header and no additional data.

An interrupt event is used in a very similar manor to a timer event since it is also a message with only a header. It carries an interrupt event code to distinguish its cause. Interrupt events are described in more detail after the user definable events have been introduced.

User definable events

A user definable event is essentially a message between tasks, as described in “Messages between tasks”. The message header contains the source, a data length and a command. The commands between tasks can also be interpreted as events between tasks (their definition is usually added at the end of config.h since the represent global defines used for the project) and enables simple state event machines to be realised by using the source and command to define a unique local event and the attached date to be passed to the state machine for specific processing.

Here is a possible implementation, although it can also be mixed with a more general handling as shown in the section “Messages between tasks”:

```
    unsigned short usEvent;
    unsigned char  ucLength;

    while (fnRead( PortIDInternal, ucInputMessage, HEADER_LENGTH)) {
      usEvent = ((ucInputMessage[MSG_SOURCE_TASK]<<8) |
      switch (ucInputMessage[MSG_SOURCE_TASK]) {
        case TIMER_EVENT:
        case INTERRUPT_EVENT:
           usEvent |= ucInputMessage[MSG_TIMER_INTERRUPT_EVENT]);
           fnHandleEvent(usEvent, 0, 0);
           break;

       default:
           ucLength = ucInputMessage[MSG_CONTENT_LENGTH];
          fnRead( PortIDInternal, ucInputMessage, ucLength);
                                  // read command and possible data
           usEvent |= ucInputMessage[0];
           fnHandleEvent(usEvent, &ucInputMessage[1], (ucLength-1));
           break;
      }
    }


static void fnHandleEvent(
                  unsigned short usEvent,
                  unsigned char *ptrData,
                  unsigned char ucDataLength)
{…};        // local state event machine treating unique events with possible data


// example event definitions
#define EVENT_TIMEOUT_SOURCE (TIMER_EVENT<<8 | SOURCE_TIMEOUT)
#define EVENT_INTERRUPT_OPEN (INTERRUPT_EVENT<<8 | OPEN_INTERRUPT)
#define EVENT_TEMPERATURE_CHANGE (TASK_TEMPERATURE<<8 | NEW_TEMPERATURE)
```

Interrupt events

Interrupt events are used mainly to signal to a task that an interrupt has been processed and there is possibly new information available based on this processing. This type of event is useful when there is data processing required after an interrupt has occurred which has a lower priority than the interrupts routine itself or is not suitable to be handled in the interrupt routine itself.

Here is an example of the processing of a frame of data received by an Ethernet controller. The controller signals the processor of the availability of the data via an interrupt. The interrupt routine performs some low level hardware processing which takes place as quickly as possible and starts a task using an interrupt message event which should then complete the processing – for example handle ARP/IP/TCP protocols.

```
static const unsigned char EMAC_RXA_int_message[ HEADER_LENGTH ] = {
    INTERNAL_ROUTE, INTERNAL_ROUTE , // internal message routine
    TASK_ETHERNET,         // handling task
    INTERRUPT_EVENT,       // interrupt event
    EMAC_RXA_INTERRUPT };  // interrupt event code

__interrupt void emac_rx_b_a_c_isr(void) // a receive buffer has a valid frame
{
  IMASK &= ~RXACIE;                 // disable interrupt but mask further
                                    // use until application has read buffer
  eth_rx_control->chars = RXAEFP;   // put the length in the buffer
  RXAEFP = 0;
  fnWrite(INTERNAL_ROUTE, (unsigned char*)EMAC_RXA_int_message, HEADER_LENGTH);
                                    // send an interrupt event to the handling task
}
```

This example shows an interrupt routine preparing information about the interrupt and waking the responsible handling task by sending it a predefined interrupt message signaling the specific event. The handling task can then process the message without the processing code having to be in the interrupt routine, which may otherwise cause other interrupts to be blocked for an unnecessary long period of time.

Heap memory utilisation

It has been seen that the µTasker itself requires only a small number of bytes of static RAM and then builds up the tasks, timers and queues as required in heap memory. It is often the case that this heap memory will be allocated during initialisation by µTasker and by certain application tasks for system operation and is never freed. In such a case only an extremely simple and efficient heap management is required.

On the other hand it is possible that application tasks and their routines will temporarily use memory from heap and free it after use – in this case a more sophisticated heap management is necessary. In many systems a mixture of the two types may also be appropriate.

The µTasker environment offers an extremely simple and efficient method for allocation of memory, which assumes that allocated data will not be freed. The restriction on fixed allocation is desirable because it 'forces' system construction for optimal deterministic operation. Should it be absolutely necessary to use a memory pool then the standard library delivered with the compiler will almost certainly include the standard malloc() and free() supprt which can be used in parallel or alternatively (simply define the routine uMalloc() to malloc() in config.h)

The µTasker implementation includes a feature which also allows dynamic sizing of the heap memory resource itself. This means that it doesn't rely on such setting having to be fixed in linker files or as defines in the heap management routines themselves and is very suited to configurable systems where the exact functionality is defined at run time and also the maximum heap size itself is variable.

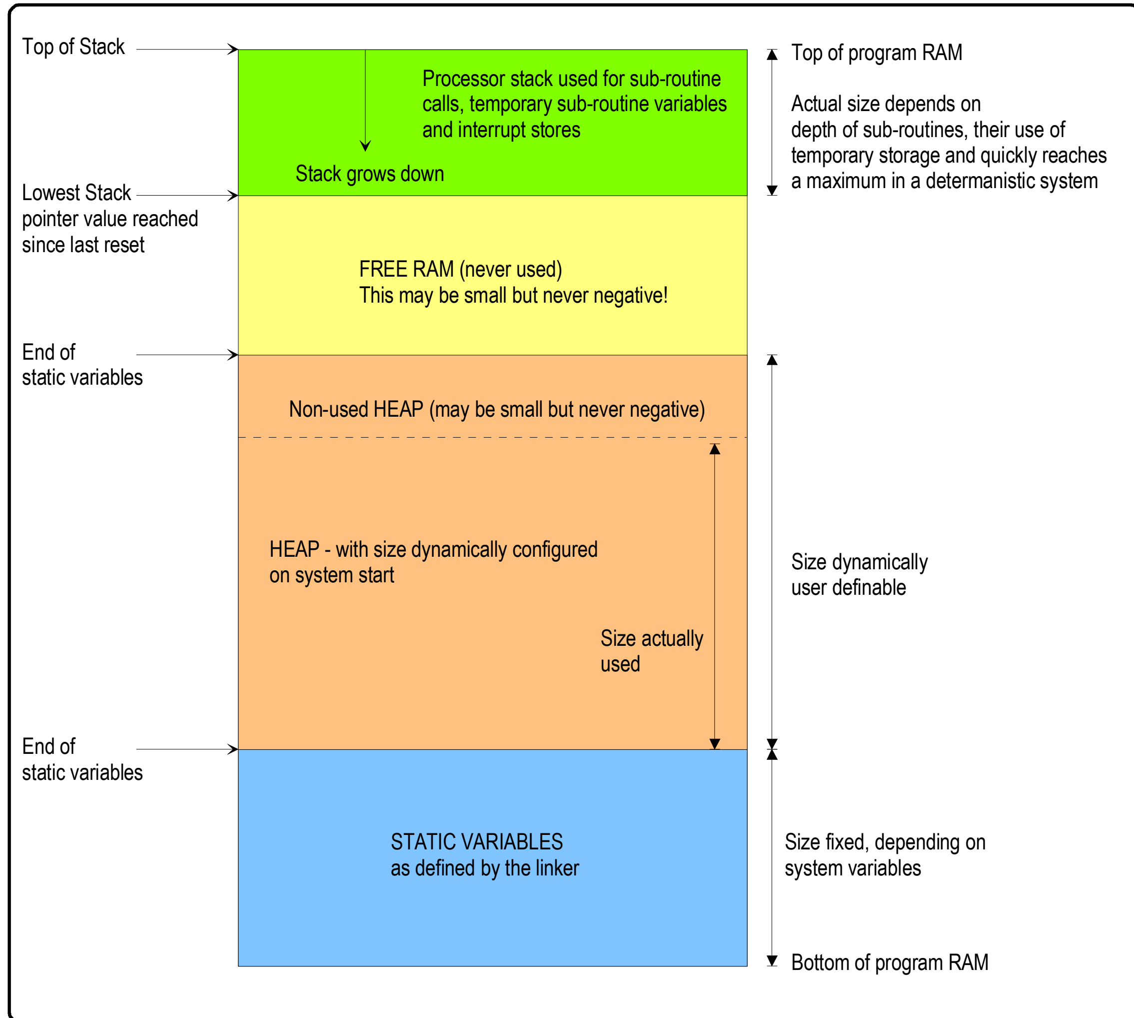
The heap size is configured at initialisation by using the call:

```
fnInitialiseHeap(ctOurHeap);

const HEAP_NEEDS ctOurHeap[] = {
    {1, 1000},              // node  1 requires this amount of heap space
    {2, 2000},              // node  2 requires this amount of heap space
    {3, 500},               // node  3 requires this amount of heap space
    {0}                     // end
};
```

This example shows a system with three possible nodes, each requiring different amounts of heap memory for their operation. In a system with only one fixed node, there only needs to be one single entry.

The result of this call is the configuration of the RAM memory as shown in the following diagram:



RAM memory map showing the dynamically sized heap block located after the static variables and before the system stack

The µTasker heap management software locates the top of the system static variables and adds a region for use by the heap management itself on top of the static variables. In most systems the processor stack is situated at the uppermost part of RAM and grows downwards.

This means that the size of the heap can be set to the size required for the present system and any remaining memory is available for the processor stack. The user can optimise the heap size so that it is adequate for the requirements, thus leaving the largest possible separation between the top of heap and the processor stack.

This free space between the two represents a security margin since a system will have problems if the stack grows down enough to corrupt data allocated on the heap.

The µTasker memory management includes routines which allow the user to check the margin and to define optimum heap size settings:

`STACK_REQUIREMENTS fnStackFree(void);` // returns the amount of non used memory between the top of heap and the lowest used processor stack location.

`HEAP_REQUIREMENTS fnHeapAvailable(void);` // returns the size of heap actually defined

`HEAP_REQUIREMENTS fnHeapFree(void);` // returns the amount of heap presently not allocated

It is to note that heap is allocated from bottom up and so any unused space at the top of heap increases the safety margin since stack is less likely to corrupt heap data at the upper most addresses.

When memory is allocated from heap, the heap management routine automatically clears the memory to all 0x00. This means that the calling functions can be assured that the contents are blank and do not have to reset them to this starting condition.

Library Routines

The µTasker operating system was developed with several goals in mind. There are:

- Simplicity but with user comfort
- tight control of used resources
- reliable and deterministic performance

Furthermore the processors which it supports tend to be single chip network enabled devices from different manufacturers and with different architectures. Moving between target processors and cross compilers should be made as transparent and efficient as possible

This leads to the last subject about the use of libraries. In order to achieve the various goals the µTasker avoids the use of standard library routines as such. The main reasons are for control purposes which makes also the support of various compiles (library manufactures) more transparent and aids in control of code execution and debugging when working in a simulation environment.

The development environment is based on a simulator making the most of the powerful features offered by VisualStudio. See the tutorial to your processor to see how you can take it for a test drive before hardware is available and how this approach can indeed save a great deal of time in comparison to conventional methods.

Here are a few examples:
Instead of `malloc()` the µTasker code uses `uMalloc()`
Instead of `memcpy()` the µTasker code uses `uMemcpy()`
Instead of `strlen()` the µTasker code uses `uStrlen()`

These are in fact local equivalents of the standard library routines and it is also very easy to use the standard library routine if it were necessary – just add a define in config.h as in the following example - `#define uMemcpy memcpy` and the standard routine will indeed be taken from the standard library. The disadvantage is however that it is not so easy to see exactly what is being linked in from the library and adding debug code in the standard library can be more complicated.

Then there are a number of specific routines in the µTasker code (see driver.h for a complete list) such `fnDecStrHex()` which converts an ASCII decimal input to its hexadecimal equivalent. The collection has resulted from several years of working with the µTasker on projects which represent its strongest deployment area. They have been optimised over the period to include the right mix of features and to achieve added efficiency in comparison to the standard `printf()` type support, which can be a major cause of unnecessarily large code sizes (although most compiler manufacturers allow also minimalist versions to be defined in the libraries).

Use the routines in the knowledge that they have been specially developed for the work which the µTasker was designed to do – especially embedded IP work. And there is still nothing to stop the use of the standard library code when developing your own application programs if this suits you better.

One last note – start up code and low level routines such as multiply, divide etc. still are delivered by the compiler's libraries to a high degree since there is no real point in the µTasker code taking over such things. However the µTasker routines sometimes use methods to avoid such 'slower' routines actually being called when it has been seen that there are methods of optimising standard routines to avoid such arithmetic. So make the most of the µTasker stuff which you like or use your own methods where appropriate – it is very flexible.

It is hoped that the µTasker operating system not only gives you a simple but comfortable environment to get your own projects up and running but also that you can benefit from its hidden power and flexibility on the way to realising highly reliable and deterministic solutions. Together with the µTasker TCP/IP stack and the powerful real-time simulation environment you should find your project times can be greatly reduced compared to using conventional methods.

Best of luck and have fun!!