RTOS Concepts

This is a brief introduction to basic Real Time Operating Systems concepts.

What is an RTOS

An RTOS is an operating system specialized for real time operations. In order to be classifiable as an RTOS an operating system must:

- Have response time **predictability**.
- Be **deterministic**.

Other qualities like speed, features set, small size etc, while important, are not what really characterize an RTOS.

Systems Classification

Any system can classified in one of the following categories.

Non Real Time systems

A non real time system is a system where there are no deadlines involved. Non-RT systems could be described as follow:

"*A non real time system is a system where the programmed reaction to a stimulus will certainly happen sometime in the future*".

Soft Real Time systems

A Soft real time system is a system where not meeting a deadline can have undesirable but not catastrophic effects, a performance degradation for example. SRTs could be described as follow:

"*A soft real time system is a system where the programmed reaction to a stimulus is almost always completed within a known finite time*".

Hard Real Time systems

An Hard Real Time (HRT) system is a system where not meeting a deadline can have catastrophic effects. HRT systems require a much more strict definition and could be described as follow:

"*An hard real time system is a system where the programmed reaction to a stimulus is guaranteed to be completed within a known finite time*".

Considerations

As you can see speed is not the main factor, predictability and determinism are. It is also important to understand that it is not the RTOS that makes a system SRT or HRT but the system design itself, the RTOS is just a tool that you can use in the right or wrong way.

Note that both SRT and HRT processes could coexist within the same system, even non critical processes without any RT constraints could be included in a design.

Scheduling, States and Priorities

Most RTOSs, including ChibiOS/RT, implement "fixed priority preemptive" scheduling algorithm. The strategy is very simple and can be described using few rules:

- Each thread has its own priority level, priorities are fixed and do not change unless the system is specifically designed to do so.
- Each *active* thread can be in one of the following states:
  - **Running**, currently being executed by a physical core.
  - **Ready**, ready to be executed when a physical core will become available.
  - **Waiting**, not ready for execution because waiting for an external event. Most RTOSs split this state in several sub-states but those are still waiting states.
- Each physical core in the system always executes the highest priority thread that is ready for execution.
- When a thread becomes ready and there is a lower priority thread being executed then preemption occurs and the higher priority thread is executed, the lower priority thread goes in the ready state.

If the system has N cores the above strategy ensures that the N highest priority threads are being executed in any moment. Small embedded systems usually have a single core so there is only one running thread in any moment.
An explanation of how priorities are organized in ChibiOS/RT can be found in the article "Priority Levels".

Interrupts handling

An important role of an *embedded* RTOS is handling of interrupts. Interrupts are an important events source to which a system designed around an RTOS is supposed to react. We can classify interrupt sources in two main classes:

- RTOS-related interrupt sources. This class of interrupts are required to interact with the RTOS in order to wakeup threads waiting for external events.
- Non RTOS-related interrupt sources. Interrupt sources that do not need to interact with the RTOS. This class of interrupts could also be able to preempt the kernel in those architectures supporting maskable priority levels (ARM Cortex-M3) or separate interrupt lines (ARM7).

A carefully designed RTOS should implement mechanisms efficiently handling the synchronization between threads and interrupt sources. Flexibility is important at this level, the capability to wake up single or multiple threads, synchronously or asynchronously is very valuable. On the other side threads should be able to wait for a single or multiple events.
Usually interrupt events are abstracted in a RTOS using mechanism like semaphores, event flags, queues or others, there is much variability in how this is implemented by the various RTOSs.

Do I need an RTOS?

It depends, you don't have to use an RTOS in order to design a predictable system but an RTOS offers you a methodology that can allow you to design a predictable system, without an RTOS you are basically on your own. Note that this methodology is not necessarily the "priority based multitasking" as implemented by ChibiOS/RT and many other RTOSs, this is just the most common scheme.

You may not need extreme predictability in your system but still want to use an RTOS simply because it can be convenient to use compared to a bare metal system. An RTOS, especially one designed for embedded applications, can also offer other services like, for example, a stable runtime environment, device drivers, file systems, networking and other useful subsystems.

What makes for a good RTOS?

Assuming that all the candidates can be classified as RTOSs having the mentioned minimum requirements, then are all the other features that make the difference. Usually some specific features or measurable parameters are highly regarded in RTOSs.

Response Time

An important parameter when evaluating an RTOS is its response time. An efficient RTOS only adds a small overhead to the system theoretical minimal response time. Typical parameters falling in this category are:

- **Interrupt latency**, the time from an interrupt request and the interrupt servicing. An RTOS can add some overhead in interrupt servicing. The overhead can be caused by extra code inserted by the RTOS into the interrupt handlers code paths or by RTOS-related critical zones.
- **Threads fly-back time**, the time from an hardware event, usually an interrupt, and the restart of the thread supposed to handle it.
- **Context switch time**, the time required to synchronously switch from the context of one thread to the context of another thread.

Of course an RTOS capable to react within 2µS is better than a system that reacts within 10µS. Note that what is really meaningful is the worst case value, if a system reacts in average in 5µS but, because jitter, can have spikes up to 20µS then the value to be considered is 20µS.

Jitter

A good RTOS is also characterized by low intrinsic jitter in response time. Intrinsic because jitter is also determined by the overall system design. Some factors that determine the system behavior regarding jitter are:

- Thread priorities assignment.
- Interrupt priorities assignment.
- Length and number of critical zones.
- Interactions between threads through shared resources protected by mutual exclusion.

- Use of priority inheritance or other jitter-reducing algorithms/strategies.

See the article "Response Time and Jitter".

Size

In an embedded system the RTOS is an important overhead in terms of occupied memory, a more compact RTOS is preferable being all the other parameters equal because memory cost.

Reliability

There are design choices that make some systems intrinsically more reliable that others. Dynamic allocation is a good example of a poor design choice because both unreliability and response time unpredictability of some allocation schemes. Fully static designs do not have those intrinsic limitations.

Synchronization Primitives

Variety in available primitives is also an important factor to be considered. Having the correct tool for the job can reduce development time and often also helps when integrating external code with the RTOS.

A good example is the lwIP TCP/IP stack, it assumes an RTOS offering semaphores with timeouts, if your RTOS does not support semaphores *and* timeouts then you have a problem and will have to find a workaround.

Designing an embedded application

ChibiOS/RT offers a variety of mechanisms and primitives, often it is better to focus on a single approach for the system design and use only part of the available subsystems. When designing your application you may choose among several design alternatives.

Single threaded superloop

Correct, single thread, it is not mandatory to use the multithreading features of the OS. You may choose to implements everything as a complex state machine handled in the main thread alone. In this scenario the OS still offers a variety of useful mechanisms:

- Interrupt handling.
- Virtual Timers, very useful in state machines in order to handle time triggered state transitions.
- Power management.
- Event Flags and/or Semaphores as communication mechanism between interrupt handlers and the main superloop.
- I/O queues.
- Memory allocation.
- System time.

In this configuration the kernel size is really minimal, everything else is disabled and takes no space. You always have the option to use more threads at a later time in order to perform separate tasks.

Message passing

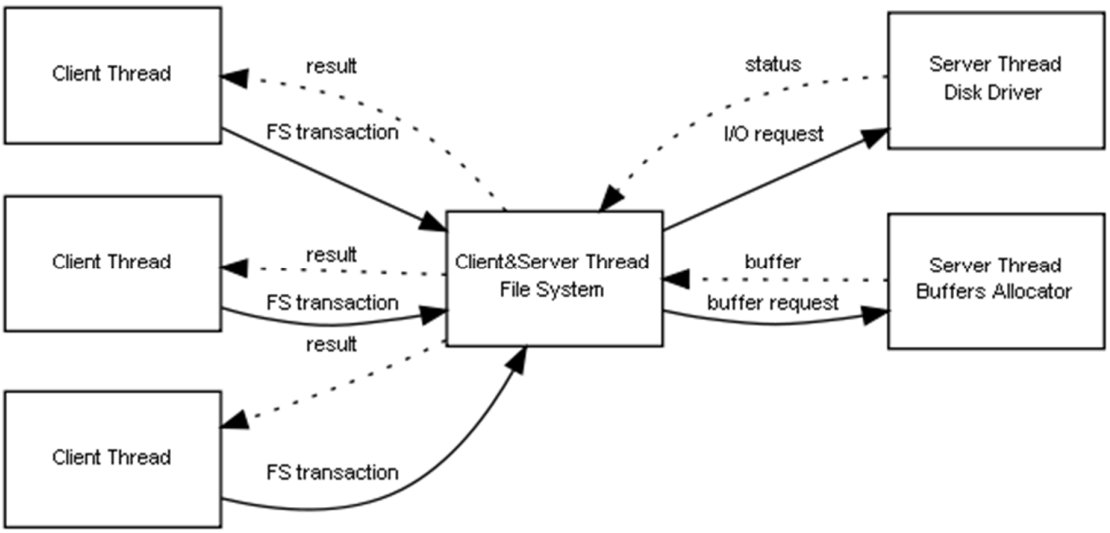
In this scenario there are multiple threads in the system that never share data, everything is done by exchanging messages. Each thread represents a service, the other threads can request the service by sending a message.
In this scenario the following subsystems can be used:

- Synchronous Messages.
- Mailboxes (asynchronous message queues).

The advantage of this approach is to not have to deal with mutual exclusion, each functionality is encapsulated into a server thread that sequentially serves all the requests. For example, you can have the following scenario:

- A buffers allocator server.
- A disk driver server.
- A file system server.
- One or more client threads.

Example:



Note that the threads should not exchange complex messages but just pointers to data structures in order to optimize the performance. Also note that a thread can be both client and server at the same time, the FS service in the previous scenario for example.

Threads sharing data

This is the most common scenario, several threads have access to both their private data and shared data. Synchronization happens with one of the mechanisms described in the "ChibiOS/RT mutual exclusion guide".

Mixed

All the above approaches can be freely mixed in a single application but usually it is preferred to choose a way and consistently design the system around it. The OS is a toolbox that offers a lot of tools but you don't have to use them all necessarily.

ChibiOS/RT Kernel Concepts

Naming Conventions

ChibiOS/RT APIs are all named following this convention: *ch<group><action><suffix>*(). The possible groups are: *Sys*, *Sch*, *Time*, *VT*, *Thd*, *Sem*, *Mtx*, *Cond*, *Evt*, *Msg*, *Reg*, *SequentialStream*, *IO*, *IQ*, *OQ*, *Dbg*, *Core*, *Heap*, *Pool*.

API Name Suffixes

The suffix can be one of the following:

- **None**, APIs without any suffix can be invoked only from the user code in the **Normal** state unless differently specified. See System States.
- **"I"**, I-Class APIs are invokable only from the **I-Locked** or **S-Locked** states. See System States.
- **"S"**, S-Class APIs are invokable only from the **S-Locked** state. See System States.

Interrupt Classes

In ChibiOS/RT there are three logical interrupt classes:

- **Regular Interrupts**. Maskable interrupt sources that cannot preempt (small parts of) the kernel code and are thus able to invoke operating system APIs from within their handlers. The interrupt handlers belonging to this class must be written following some rules. See the system APIs group and the web article How to write interrupt handlers.
- **Fast Interrupts**. Maskable interrupt sources with the ability to preempt the kernel code and thus have a lower latency and are less subject to jitter, see the web article Response Time and Jitter. Such sources are not supported on all the architectures. Fast interrupts are not allowed to invoke any operating system API from within their handlers. Fast interrupt sources may, however, pend a lower priority regular interrupt where access to the operating system is possible.
- **Non Maskable Interrupts**. Non maskable interrupt sources are totally out of the operating system control and have the lowest latency. Such sources are not supported on all the architectures.

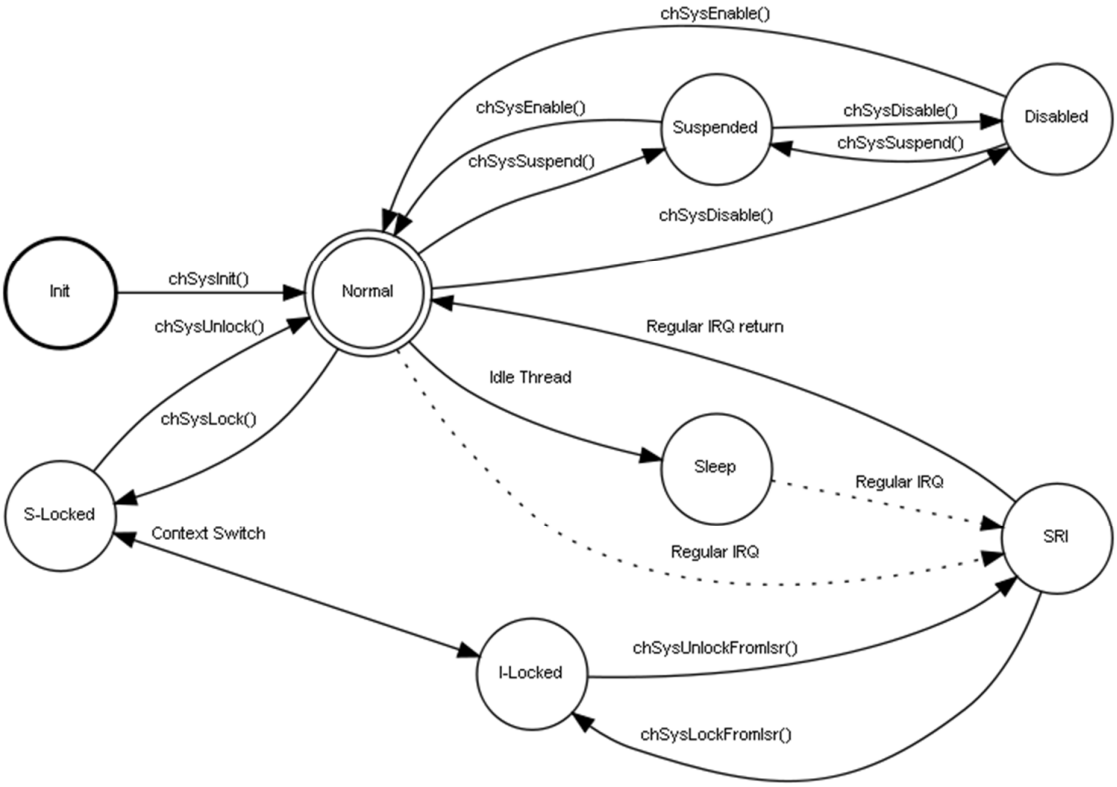
The mapping of the above logical classes into physical interrupts priorities is, of course, port dependent. See the documentation of the various ports for details.

System States

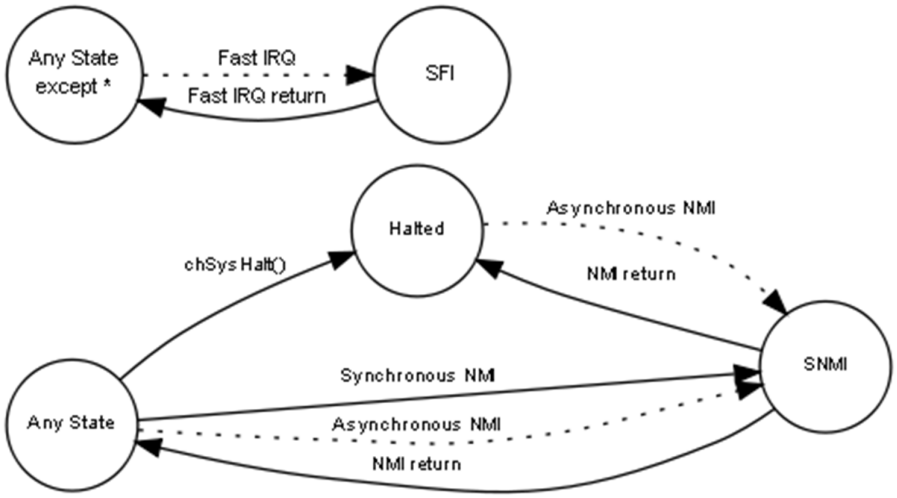
When using ChibiOS/RT the system can be in one of the following logical operating states:

- **Init**. When the system is in this state all the maskable interrupt sources are disabled. In this state it is not possible to use any system API except `chSysInit()`. This state is entered after a physical reset.
- **Normal**. All the interrupt sources are enabled and the system APIs are accessible, threads are running.
- **Suspended**. In this state the fast interrupt sources are enabled but the regular interrupt sources are not. In this state it is not possible to use any system API except `chSysDisable()` or `chSysEnable()` in order to change state.
- **Disabled**. When the system is in this state both the maskable regular and fast interrupt sources are disabled. In this state it is not possible to use any system API except `chSysSuspend()` or `chSysEnable()` in order to change state.
- **Sleep**. Architecture-dependent low power mode, the idle thread goes in this state and waits for interrupts, after servicing the interrupt the Normal state is restored and the scheduler has a chance to reschedule.
- **S-Locked**. Kernel locked and regular interrupt sources disabled. Fast interrupt sources are enabled. S-Class and I-Class APIs are invokable in this state.
- **I-Locked**. Kernel locked and regular interrupt sources disabled. I-Class APIs are invokable from this state.
- **Serving Regular Interrupt**. No system APIs are accessible but it is possible to switch to the I-Locked state using `chSysLockFromIsr()` and then invoke any I-Class API. Interrupt handlers can be preemptable on some architectures thus is important to switch to I-Locked state before invoking system APIs.
- **Serving Fast Interrupt**. System APIs are not accessible.
- **Serving Non-Maskable Interrupt**. System APIs are not accessible.
- **Halted**. All interrupt sources are disabled and system stopped into an infinite loop. This state can be reached if the debug mode is activated **and** an error is detected **or** after explicitly invoking `chSysHalt()`.

Note that the above states are just **Logical States** that may have no real associated machine state on some architectures. The following diagram shows the possible transitions between the states:



Note, the **SFI**, **Halted** and **SNMI** states were not shown because those are reachable from most states:

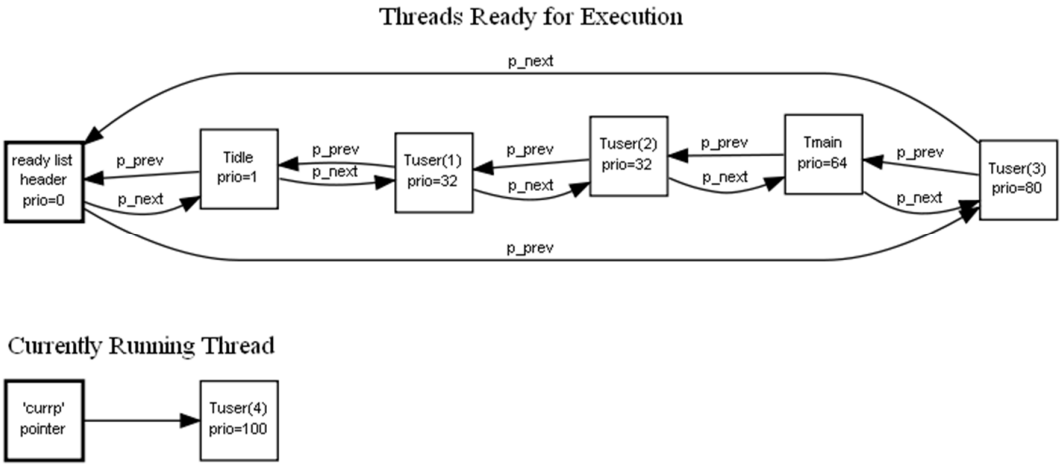


Attention:

* except: **Init**, **Halt**, **SNMI**, **Disabled**.

Scheduling

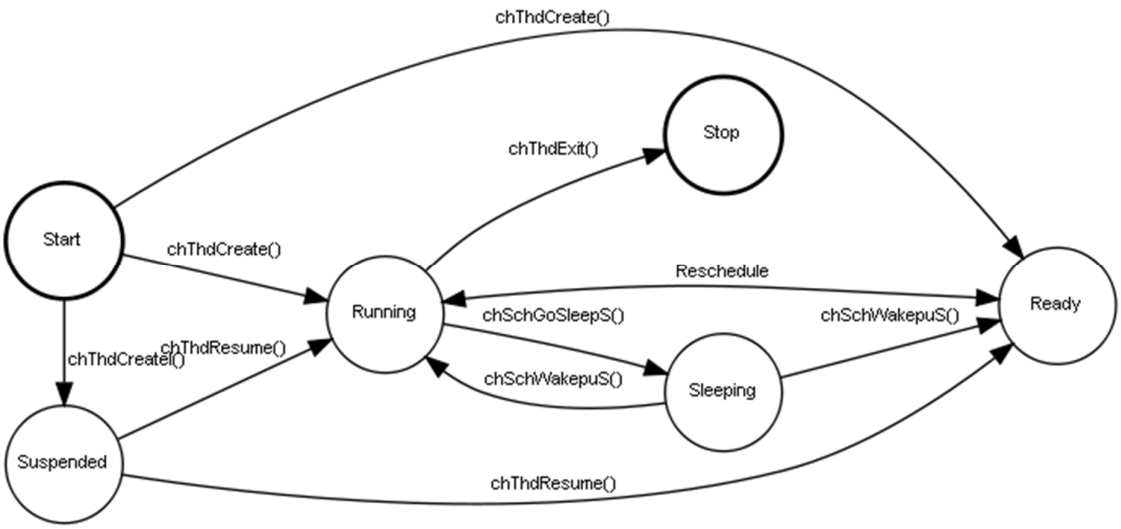
The strategy is very simple the currently ready thread with the highest priority is executed. If more than one thread with equal priority are eligible for execution then they are executed in a round-robin way, the CPU time slice constant is configurable. The ready list is a double linked list of threads ordered by priority.



Note that the currently running thread is not in the ready list, the list only contains the threads ready to be executed but still actually waiting.

Thread States

The image shows how threads can change their state in ChibiOS/RT.



Priority Levels

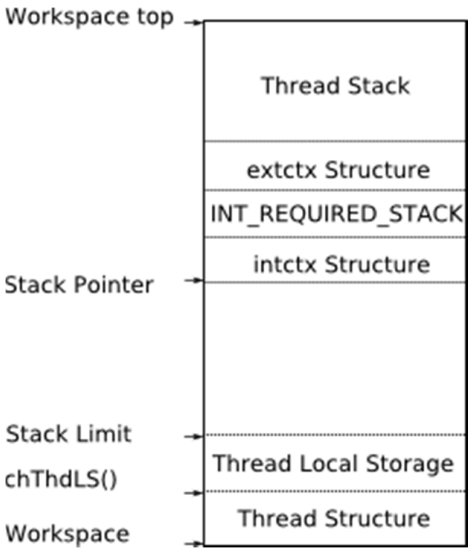
Priorities in ChibiOS/RT are a contiguous numerical range but the initial and final values are not enforced.
The following table describes the various priority boundaries (from lowest to highest):

- `IDLEPRIO`, this is the lowest priority level and is reserved for the idle thread, no other threads should share this priority level. This is the lowest numerical value of the priorities space.
- `LOWPRIO`, the lowest priority level that can be assigned to an user thread.

- NORMALPRIO, this is the central priority level for user threads. It is advisable to assign priorities to threads as values relative to NORMALPRIO, as example NORMALPRIO-1 or NORMALPRIO+4, this ensures the portability of code should the numerical range change in future implementations.
- HIGHPRIO, the highest priority level that can be assigned to an user thread.
- ABSPRO, absolute maximum software priority level, it can be higher than HIGHPRIO but the numerical values above HIGHPRIO up to ABSPRIO (inclusive) are reserved. This is the highest numerical value of the priorities space.

Thread Working Area

Each thread has its own stack, a Thread structure and some preemption areas. All the structures are allocated into a "Thread Working Area", a thread private heap, usually statically declared in your code. Threads do not use any memory outside the allocated working area except when accessing static shared data.



Note that the preemption area is only present when the thread is not running (switched out), the context switching is done by pushing the registers on the stack of the switched-out thread and popping the registers of the switched-in thread from its stack. The preemption area can be divided in up to three structures:

- External Context.
- Interrupt Stack.
- Internal Context.

See the port documentation for details, the area may change on the various ports and some structures may not be present (or be zero-sized).

How to create a thread

Creating a new thread is the most common development task when using an RTOS, this is how it is done in ChibiOS/RT.

Default Threads

After initializing ChibiOS/RT using `chSysInit()` two threads are spawned by default:

- **Idle thread**. This thread has the lowest priority in the system so it runs only when the other threads in the system are sleeping. This threads usually switches the system in a low power mode and does nothing else.
- **Main thread**. This thread executes your `main()` function at startup. The main thread is created at the `NORMALPRIO` level but it can change its own priority if required. It is from the main thread that the other threads are usually created.

Thread Classes

There are two classes of threads in ChibiOS/RT:

- **Static Threads**. This class of threads are statically allocated in memory at compile time.
- **Dynamic Threads**. Threads created by allocating memory at run time from a memory heap or a memory pool.

Creating a static thread

In order to create a static thread a working area must be declared using the macro `WORKING_AREA` as shown:

```
static WORKING_AREA(myThreadWorkingArea, 128);
```

This macro reserves 128 bytes of stack for the thread and space for all the required thread related structures. The total size and the alignment problems are handled inside the macro, you only need to specify the pure and simple desired stack size.

A static thread can be started by invoking `chThdCreateStatic()` as shown in this example:

```
  Thread *tp = chThdCreateStatic(myThreadWorkingArea,
                                 sizeof(myThreadWorkingArea),
                                 NORMALPRIO,     /* Initial priority.    */
                                 myThread,       /* Thread function.     */
                                 NULL);          /* Thread parameter.    */
```

The variable `tp` receives a pointer to the thread object, this pointer is often taken as parameter by other APIs. Now a complete example:

```
/*
 * My simple application.
 */

#include <ch.h>
```

```
/*
 * Working area for the LED flashing thread.
 */
static WORKING_AREA(myThreadWorkingArea, 128);

/*
 * LED flashing thread.
 */
static msg_t myThread(void *arg) {

  while (TRUE) {
    LED_ON();
    chThdSleepMilliseconds(500);
    LED_OFF();
    chThdSleepMilliseconds(500);
  }
}

int main(int argc, char *argv[]) {

  /* Starting the flashing LEDs thread.*/
  (void)chThdCreateStatic(myThreadWorkingArea,
sizeof(myThreadWorkingArea),
                          NORMALPRIO, myThread, NULL);
  .
  .
  .
}
```

Note that the memory allocated to `myThread()` is statically defined and cannot be reused. Static threads are ideal for safety applications because there is no risk of a memory allocation failure because progressive heap fragmentation.

Creating a dynamic thread using the heap allocator

Creating a thread from a memory heap is very easy:

```
  Thread *tp = chThdCreateFromHeap(NULL,            /* NULL = Default
heap. */
                                   THD_WA_SIZE(128),/* Stack size.
*/
                                   NORMALPRIO,      /* Initial priority.
*/
                                   myThread,        /* Thread function.
*/
                                   NULL);           /* Thread parameter.
*/
```

The memory is allocated from the specified heap and the thread is started. Note that the memory is not freed when the thread terminates but when the thread final status (its return value) is collected by the spawning thread. For example:

```
/*
 * My simple application.
 */

#include <ch.h>

/*
 * LED flashing thread.
```

```
 */
static msg_t myThread(void *arg) {

  unsigned i = 10;
  while (i > 0) {
    LED_ON();
    chThdSleepMilliseconds(500);
    LED_OFF();
    chThdSleepMilliseconds(500);
    i--;
  }
  return (msg_t)i;
}

int main(int argc, char *argv[]) {

  Thread *tp = chThdCreateFromHeap(NULL, THD_WA_SIZE(128), NORMALPRIO+1,
                                   myThread, NULL);
  if (tp == NULL)
    chSysHalt();    /* Memory exausted. */

  /* The main thread continues its normal execution.*/
  .
  .
  /*
   * Now waits for the spawned thread to terminate (if it has not
terminated
   * already) then gets the thread exit message (msg) and returns the
   * terminated thread memory to the heap (default system heap in this
   * example).
   */
  msg_t msg = chThdWait(tp);
  .
  .
}
```

Creating a dynamic thread using the memory pool allocator

A pool is a collection of equally sized memory blocks, creating a thread from a memory pool is very similar to the previous example but the memory of terminated threads is returned to the memory pool rather than to a heap:

```
/*
 * My simple application.
 */

#include <ch.h>

/*
 * LED flashing thread.
 */
static msg_t myThread(void *arg) {

  unsigned i = 10;
  while (i > 0) {
    LED_ON();
    chThdSleepMilliseconds(500);
    LED_OFF();
    chThdSleepMilliseconds(500);
    i--;
  }
  return (msg_t)i;
}
```

```
int main(int argc, char *argv[]) {

  Thread *tp = chThdCreateFromMemoryPool(myPool, NORMALPRIO+1, myThread,
NULL);
  if (tp == NULL)
    chSysHalt();    /* Pool empty. */

  /* The main thread continues its normal execution.*/
  .
  .
  /*
   * Now waits for the spawned thread to terminate (if it has not
terminated
   * already) then gets the thread exit message (msg) and returns the
   * terminated thread memory to the original memory pool.
   */
  msg_t msg = chThdWait(tp);
  .
  .
}
```