Embedded Systems
RTOS: The ChibiOS/RT Project

Martino Migliavacca
migliavacca@elet.polimi.it

November 18, 2011
Outline

1 Introduction
   - Introduction to ChibiOS/RT
   - Architecture
   - Source organization

2 ChibiOS/RT kernel
   - Introduction
   - System states
   - Base kernel services
   - Synchronization
   - Memory management
   - I/O support

3 Hardware Abstraction Layer
   - Introduction
   - ADC driver
   - Serial driver

4 Projects
   - Overview of available projects
**Introduction to ChibiOS/RT**

*ChibiOS/RT is a complete, portable, open source, compact and extremely fast RTOS.*

- **Complete**: threads, synchronization, memory management, hardware drivers
- **Portable**: layered architecture enables easy porting
- **Open source**: GPL3, GPL3 with linking exception, commercial license
- **Compact**: 5.5 Kb complete kernel size, 512 bytes RAM (on ARM Cortex-M3)
- **Extremely fast**: best in class context switch performance
ChibiOS/RT history

- ChibiOS/RT ancestor was an Operating System for Motorola 68000
- In 1989 it supported GCC, ran EMACS, was preemptive and realtime
- ...but in 1991 Linus Torvalds began the development of Linux
- The original full-featured OS turned in a minimalistic, efficient, RTOS: ChibiOS/RT father
- In 2007, after 15 years, it turned to ChibiOS/RT: an open source RTOS project targeted to embedded systems
- The project is lead and mainly developed by Giovanni Di Sirio
- ChibiOS/RT started growing in features, ports and users
- Every part of the project has been deeply documented (Doxygen)
ChibiOS/RT ohloh.net code analysis
ChibiOS/RT Features

- Not just a scheduler
- Support for startup and board initialization
- Many synchronization mechanism
- Flexible memory management
- I/O support tools
- HAL abstracting many common device drivers
- Complete test suite for kernel and HAL
- Debug checks
- Debug tools for eclipse
- PC simulator
- Integration with other open source projects
ChibiOS/RT architecture

- **Kernel**: the platform independent part of the OS kernel
- **Port layer**: the architecture/compiler dependent part of the OS kernel
  - System startup, interrupts abstraction, lock/unlock primitives, context switch related structures and code
  - Very little code, but the quality of the implementation can affect heavily the performance of the ported OS
  - Probably the most critical part of the whole OS

- **Hardware Abstraction Layer**: abstract device drivers
  - Common I/O API to the application
  - Totally portable across the various architectures and compilers

- **Platform layer**: device drivers implementations

- **Board Initialization**: files used to initialize the target board and the HAL before launching the application
Introduction

ChibiOS/RT kernel

Hardware Abstraction Layer

Projects

Components dependancies
Source organization: kernel file
Source organization: port layer
Source organization: HAL
Source organization: platform layer
Source organization: board initialization
Outline

1 Introduction
   • Introduction to ChibiOS/RT
   • Architecture
   • Source organization

2 ChibiOS/RT kernel
   • Introduction
   • System states
   • Base kernel services
   • Synchronization
   • Memory management
   • I/O support

3 Hardware Abstraction Layer
   • Introduction
   • ADC driver
   • Serial driver

4 Projects
   • Overview of available projects
ChibiOS/RT kernel

- **Static design**
  - Everything in the kernel is static, nowhere memory is allocated or freed
  - Allocator subsystems are optionally available, built as a layer on top of the fully static kernel

- **No error conditions**
  - System APIs have no error conditions
  - All the static core APIs always succeed if correct parameters are passed

- **Simple, fast and compact**
  - Each API function should have the parameters you would expect
  - Note: first “fast”, then “compact”

- **Portable**
  - Efficient layered architecture
  - CPU requirements: on 8-bit architecture with real stack pointer
  - Memory requirements: 1.2 Kb program memory, 256 bytes RAM
System states

- ChibiOS/RT kernel
- Hardware Abstraction Layer
- Projects

Diagram:
- Init
- Normal
- Suspended
- Disabled
- S-Locked
- I-Locked
- Sleep
- SRI

Transitions:
- chSysInit()
- chSysLock()
- chSysUnlock()
- chSysEnable()
- chSysDisable()
- chSysSuspend()
- Regular IRQ return
- Regular IRQ
- Context Switch
- chSysUnlockFromISR()
API name suffixes indicate from which system states a function can be invoked.

API 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
- **I**: I-Class APIs are invokable only from the I-Locked or S-Locked states
- **S**: S-Class APIs are invokable only from the S-Locked state

```c
void chMtxLock(Mutex *mp) {
    chSysLock();
    chMtxLockS(mp);
    chSysUnlock();
}
```
**Base kernel services**

- **System**: low level locks, initialization
- **Timers**: virtual timers and time APIs
- **Scheduler**: scheduler APIs, all the higher level synchronization mechanism are implemented through this subsystem, it is very flexible but **not recommended for direct use in application code**
- **Threads**: thread-related APIs
**Initialization, low level locks, idle thread**

- **chSysInit()**
  ChibiOS/RT initialization

- **chSysHalt()**
  Halts the system

- **chSysLock()**
  Enters the kernel lock mode

- **chSysUnlock()**
  Leaves the kernel lock mode

- **chSysLockFromIsr()**
  Enters the kernel lock mode from an ISR

- **chSysUnlockFromIsr()**
  Leaves the kernel lock mode from an ISR

- **_idle_thread()**
  Implements the idle thread infinite loop
ISRs abstraction macros

**CH_IRQ_PROLOGUE()**  IRQ handler enter code

**CH_IRQ_EPILOGUE()**  IRQ handler exit code

**CH_IRQ_HANDLER(id)**  Normal IRQ handler declaration

**CH_FAST_IRQ_HANDLER(id)**  Fast IRQ handler declaration

```c
CH_IRQ_HANDLER(myIRQ) {
    CH_IRQ_PROLOGUE();

    /* IRQ handling code, preemptable if the architecture supports it.*/
    chSysLockFromIsr();
    /* Invocation of some I-Class system APIs, never preemptable.*/
    chSysUnlockFromIsr();

    /* More IRQ handling code, again preemptable.*/
    CH_IRQ_EPILOGUE();
}
```
The strategy is very simple: the currently ready thread with the highest priority is executed.

The ready list is a double linked list of threads ordered by priority.
void chSchDoReschedule(void) {
    Thread *otp;

#if CH_TIME_QUANTUM > 0
    rlist.r_preempt = CH_TIME_QUANTUM;
#endif
    otp = currp;
    /* Picks the first thread from the ready queue and makes it current.*/
    setcurrp(fifo_remove(&rlist.r_queue));
    currp->p_state = THD_STATE_CURRENT;
    chSchReadyI(otp);
    chSysSwicth(currp, otp);
}

#define chSysSwitch(ntp, otp) {
    dbg_trace(otp);
    THREAD_CONTEXT_SWITCH_HOOK(ntp, otp);
    port_switch(ntp, otp);
}

void _port_switch(Thread *ntp, Thread *otp) {
    PUSH_CONTEXT();

    asm volatile("str sp,[%1,#12]"
                "ldr sp, [%0,#12]": "r"(ntp), "r"(otp));

    POP_CONTEXT();
}
- Threads priority levels are contiguous integers from LOWPRIO to HIGHPRIO
- **IDLEPRIO**: special priority level reserved for the idle thread
- **LOWPRIO**: lowest priority level usable by user threads
- **NORMALPRIO**: central priority level for user threads, as the main() thread
- **HIGHPRIO**: highest priority level usable by user threads
ChibiOS/RT supports multiple threads at the same priority level and schedules them using an aggressive round-robin strategy.

A round-robin rotation can happen if the currently executed thread:
- voluntarily invokes the `chThdYield()` API
- voluntarily goes into a `sleep` state
- is **preempted** by an higher priority thread
- if `CH_TIME_QUANTUM` constant is greater than zero and the specified **time quantum expired**

If `CH_TIME_QUANTUM` is equal to zero, the round robin becomes cooperative.
In ChibiOS/RT a thread is a C function owning a processor context, state informations and a dedicated stack area.

The working area is declared with a macro:
```c
static WORKING_AREA(waThread1, 128);
```

Threads are created with the chThdCreateStatic() API:
```c
Thread* chThdCreateStatic(void *wsp, size_t size, tprio_t prio, tfunc_t pf, void *arg)
```
- **wsp**: pointer to a working area dedicated to the thread stack
- **size**: size of the working area
- **prio**: the priority level for the new thread
- **pf**: the thread function
- **arg**: an argument passed to the thread function, it can be NULL
Threads API (part of)

- `chThdCreateStatic()`  
  Creates a new thread into a static memory area
- `chThdExit()`  
  Terminates the current thread
- `chThdWait()`  
  Blocks the execution of the invoking thread until the specified thread terminates, then the exit code is returned
- `chThdResume()`  
  Resumes a suspended thread
- `chThdTerminate()`  
  Requests a thread termination
- `chThdSleep()`  
  Suspends the invoking thread for the specified time
- `chThdYield()`  
  Yields the time slot
/*
 * Red LED blinker thread, times are in milliseconds.
 */

static WORKING_AREA(waThread1, 128);
static msg_t Thread1(void *arg) {

    (void)arg;
    while (TRUE) {
        palClearPad(IOPORT1, GPIOA_LED1);
        chThdSleepMilliseconds(500);
        palSetPad(IOPORT1, GPIOA_LED1);
        chThdSleepMilliseconds(500);
    }
}

int main(void) {
    /*
     * Creates the blinker thread.
     */
    chThdCreateStatic(waThread1, sizeof(waThread1), NORMALPRIO, Thread1, NULL);
Synchronization services

- **Semaphores**: counter and binary semaphores subsystem
- **Mutexes**: mutexes subsystem with support to the priority inheritance algorithm (fully implemented, any depth)
- **Condvars**: condition variables, together with mutexes the condition variables allow the implementation of monitor constructs
- **Events**: event sources and event flags with flexible support for and/or conditions and automatic dispatching to handler functions
- **Messages**: lightweight synchronous messages
- **Mailboxes**: asynchronous messages queues
Synchronization services diagram

- Binary Semaphores
- Mailboxes
- Condvars
- Mutexes
- Events
- Messages
- Semaphores
- Scheduler
- Port Layer
Events are an important feature in ChibiOS/RT, most device drivers generate events.

**EventSource**: a system object that can be broadcasted asynchronously in response of a system event.

**EventListener**: a system object that associates a Thread object to an event source.

**Registration**: the process of associating a Thread to an EventSource using an EventListener.

**Broadcast**: the operation performed on an event source in order to inform the registered threads that an event just occurred.

**Pending events**: event flags waiting to be served by a thread.

**Wait**: synchronous operation performed by a thread in order to wait for a specific combination of events.
EventSource and EventListener

- **EventSource**
  - `es_next`

- **Listeners List**
  - EventListener 1
    - `el_mask: 0x0004`
    - `el_listener`
    - `el_next`
  - EventListener 2
    - `el_mask: 0x0001`
    - `el_listener`
    - `el_next`
  - EventListener 3
    - `el_mask: 0x0C00`
    - `el_listener`
    - `el_next`
  - EventListener 4
    - `el_mask: 0x0001`
    - `el_listener`
    - `el_next`

- **Registered Threads**
  - Thread 1
    - `p_epending: 0x0000`
    - `p_ewmask: 0xFFFF`
  - Thread 2
    - `p_epending: 0x000F`
    - `p_ewmask: 0xOC01`
  - Thread 3
    - `p_epending: 0x0008`
    - `p_ewmask: 0x0001`
  - Thread 4
    - `p_epending: 0x0000`
    - `p_ewmask: 0xFFFF`
Event broadcasting

```c
void chEvtBroadcastFlagsI(EventSource *esp, eventmask_t mask) {
    EventListener *elp;

    chDbgCheckClassI();
    chDbgCheck(esp != NULL, "chEvtBroadcastMaskI");

    elp = esp->es_next;
    while (elp != (EventListener *)esp) {
        chEvtSignalFlagsI(elp->el_listener, elp->el_mask | mask);
        elp = elp->el_next;
    }
}

void chEvtSignalFlagsI(Thread *tp, eventmask_t mask) {

    chDbgCheckClassI();
    chDbgCheck(tp != NULL, "chEvtSignalI");

    tp->p_epending |= mask;
    /* Test on the AND/OR conditions wait states.*/
    if (((tp->p_state == THD_STATE_WTOREVT) &&
        ((tp->p_epending & tp->p_u.ewmask) != 0)) ||
        (tp->p_state == THD_STATE_WTANDEVT) &&
        ((tp->p_epending & tp->p_u.ewmask) == tp->p_u.ewmask)))
    chSchReadyI(tp)->p_u.rdymsg = RDY_OK;
}
Memory management
The core memory manager is a simplified allocator

It only allows to allocate memory blocks without the possibility to free them

This allocator is meant as a memory blocks provider for the other allocators

By having a centralized memory provider the various allocators can coexist and share the main memory

chCoreAlloc()  
Allocates a memory block

chCoreStatus()  
Reports the core memory status
Heap allocator

- The heap allocator implements a first-fit strategy
- It is subject to fragmentation
- Allocation time is not constant
- Its APIs are functionally equivalent to the usual malloc() and free()

- chHeapInit()
  Initializes a memory heap from a static memory area

- chHeapAlloc()
  Allocates a block of memory from the heap by using the first-fit algorithm

- chHeapFree()
  Frees a previously allocated memory block

- chHeapStatus()
  Reports the heap status
Memory pools

- Memory pools allow to allocate/free fixed size objects
- Allocation is done in **constant time** and reliably
- No memory fragmentation problems
- Memory pools can grow in size asking for memory to a provider
- More memory pools can be declared

**MEMORYPOOL_DECL**(name, size, provider)
Memory pool declaration macro

**chPoolInit()**
Initializes an empty memory pool

**chPoolAlloc()**
Allocates an object from a memory pool

**chPoolFree()**
Releases an object into a memory pool
# Memory allocators comparison

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Free capable</th>
<th>Constant time</th>
<th>Safe</th>
<th>From ISR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Objects</td>
<td>N/A</td>
<td>N/A</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Core Memory Manager</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Heap Allocator</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Memory Pools</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>C-Runtime</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
Dynamic threads

- If one of the memory allocators is available, threads can be declared dynamically.
- Thread working area can be allocated from both heaps and memory pools.

- `chThdCreateFromHeap()`
  Creates a new thread allocating the memory from the heap.

- `chThdCreateFromMemoryPool()`
  Creates a new thread allocating the memory from the specified memory pool.
Abstract Streams

- Abstract streams are abstract interface for generic data streams
- They make the access to streams independent from the implementation logic
- No code is present, streams are just abstract interfaces
- The stream interface can be used as base class for high level object types such as files, sockets, serial ports, pipes etc
- Methods are defined in BaseSequentialStreamVMT virtual methods table

```c
#define _base_sequential_stream_methods
   /* Stream write buffer method.*/
   size_t (*write)(void *instance, const uint8_t *bp, size_t n);
   /* Stream read buffer method.*/
   size_t (*read)(void *instance, uint8_t *bp, size_t n);

struct BaseSequentialStreamVMT {
   _base_sequential_stream_methods
};
#define chSequentialStreamWrite(ip, bp, n) ((ip)->vmt->write(ip, bp, n))
#define chSequentialStreamRead(ip, bp, n) ((ip)->vmt->read(ip, bp, n))
```
ChibiOS/RT queues are mostly used in serial-like device drivers

Input queues and output queues are defined

Bidirectional queue are implemented pairing an input and an output queue together

- `chIQInit()`
  - Initializes an input queue

- `chIQReset()`
  - Resets an input queue

- `chIQPutI()`
  - Input queue write

- `chIQGetTimeout()`
  - Input queue read with timeout

- `chOQPutTimeout()`
  - Output queue write with timeout

- `chOQGetI()`
  - Output queue read
Outline

1 Introduction
   - Introduction to ChibiOS/RT
   - Architecture
   - Source organization

2 ChibiOS/RT kernel
   - Introduction
   - System states
   - Base kernel services
   - Synchronization
   - Memory management
   - I/O support

3 Hardware Abstraction Layer
   - Introduction
   - ADC driver
   - Serial driver

4 Projects
   - Overview of available projects
The HAL is the abstract interface between ChibiOS/RT application and hardware.

A device driver is usually split in two layers:

- The high level device driver contains the definitions of the driver’s APIs and the platform independent part of the driver.
- The low level device driver (LLD) contains the platform dependent part of the driver.

The LLD may be not present in those drivers that do not access the hardware directly but through other device drivers (e.g., MMC_SPI).
HAL architecture

Application

High Level Driver

Low Level Driver

HAL shared low level code

Microcontroller Hardware
## Supported platforms

<table>
<thead>
<tr>
<th>Device Driver</th>
<th>ADC</th>
<th>CAN</th>
<th>EXTI</th>
<th>GPT</th>
<th>ICU</th>
<th>MAC</th>
<th>PAL</th>
<th>PWM</th>
<th>RTC</th>
<th>SDC</th>
<th>SER</th>
<th>SPI</th>
<th>UART</th>
<th>USB</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT91SAM7x</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AVR</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LPC11xx</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
</tr>
<tr>
<td>LPC13xx</td>
<td>No</td>
<td>–</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>LPC214x</td>
<td>No</td>
<td>–</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MSP430</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SPC563M64 / MPC563x</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>STM32F1xx</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>STM32L1xx</td>
<td>Yes</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>STM8L</td>
<td>No</td>
<td>–</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>STM8S</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>–</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
</tr>
</tbody>
</table>
ADC driver

- This driver defines a generic Analog to Digital Converter driver
- The driver supports several conversion modes:
  - **one shot**: the driver performs a single group conversion
  - **linear buffer**: the driver performs a series of group conversions then stops
  - **circular buffer**: when the buffer is filled, it is automatically restarted from the buffer base
- The driver is able to invoke callbacks during the conversion process
- It is not thread safe for performance reasons
- adcAcquireBus() and adcReleaseBus() to gain exclusive access
- Two main structures are used:
  - **ADCDriver**: the generic driver structure
  - **ADCCConfig**: the platform dependent configuration structure
ADC driver API

- `adcStart()`
  Configures and activates the ADC peripheral

- `adcStop()`
  Deactivates the ADC peripheral and enters low power mode

- `adcConvert()`
  Performs an ADC conversion

- `adcStartConversion()`
  Starts ADC conversion

- `adcStopConversion()`
  Stops ongoing conversion

- `adcAcquireBus()`
  Gains exclusive access to the ADC peripheral

- `adcReleaseBus()`
  Releases exclusive access to the ADC peripheral
ADC driver state machine

- ADC_STOP (Low Power)
  - adcInit()
  - adcStop()

- ADC_READY (Clock Enabled)
  - adcStart()
  - adcStopConversion()

- ADC_ACTIVE (Converting)
  - adcStartConversion() (async)
  - adcConvert() (sync)

- ADC_COMPLETE (Complete)
  - async callback (half buffer)
  - async callback (full buffer circular)

- ADC_ERROR (Error)
  - adcStartConversion() (async)
  - callback return

- adcStopConversion() (async)
  - >acg_endcb<

- >end_cb<
  - async callback (error)
  - >error_cb<

- callback return
This driver defines a generic full duplex serial driver

The driver implements a SerialDriver interface (VMT)

It uses I/O Queues for communication between the upper and the lower driver

Event flags are used to notify the application about incoming data, outgoing data and other I/O events

Two main structures are used:

- **SerialDriver**: the generic driver structure
- **SerialConfig**: the platform dependent configuration structure
Serial driver API

- **sdStart()**
  Configures and starts the driver

- **sdStop()**
  Stops the driver

- **sdWrite()**
  Direct blocking write to a SerialDriver

- **sdWriteTimeout()**
  Direct blocking write to a SerialDriver with timeout specification

- **sdAsynchronousWrite()**
  Direct non-blocking write to a SerialDriver

- **sdRead()**
  Direct blocking read from a SerialDriver

- **sdReadTimeout()**
  Direct blocking read from a SerialDriver with timeout specification

- **sdAsynchronousRead()**
  Direct non-blocking read from a SerialDriver
Serial driver state machine
Serial driver example: halconf.h

/**
 * @brief Default bit rate.
 * @details Configuration parameter, this is the baud rate selected for the default configuration.
 */
#if !defined(SERIAL_DEFAULT_BITRATE) || defined(__DOXYGEN__)
#define SERIAL_DEFAULT_BITRATE 38400
#endif

/**
 * @brief Serial buffers size.
 * @details Configuration parameter, you can change the depth of the queue buffers depending on the requirements of your application.
 * @note The default is 64 bytes for both the transmission and receive buffers.
 */
#if !defined(SERIAL_BUFFERS_SIZE) || defined(__DOXYGEN__)
#define SERIAL_BUFFERS_SIZE 16
#endif
Serial driver example: main.c

/*
 * Activates the serial driver 1 using the driver default configuration.
 */
sdStart(&SD1, NULL);

/*
 * Write "Hello world!" on the serial driver 1.
 */
sdWrite(&SD1, "Hello world!\n", strlen("Hello world!\n"));
Shell example: command parser

static void cmd_mem(BaseChannel *chp, int argc, char *argv[]) {
    size_t n, size;
    char buf[52];

    (void)argv;
    if (argc > 0) {
        shellPrintLine(chp, "Usage: mem");
        return;
    }
    n = chHeapStatus(NULL, &size);
    sprintf(buf, "core free memory : %u bytes", chCoreStatus());
    shellPrintLine(chp, buf);
    sprintf(buf, "heap fragments : %u", n);
    shellPrintLine(chp, buf);
    sprintf(buf, "heap free total : %u bytes", size);
    shellPrintLine(chp, buf);
}
Shell example: usage

```c
static const ShellCommand commands[] = {
    {"mem", cmd_mem},
    {"threads", cmd_threads},
    {"test", cmd_test},
    {"tree", cmd_tree},
    {NULL, NULL}
};

static const ShellConfig shell_cfg1 = {
    (BaseChannel *)&SD2,
    commands
};

while (TRUE) {
    if (!shellttp)
        shellttp = shellCreate(&shell_cfg1, SHELL_WA_SIZE, NORMALPRIO);
    else if (chThdTerminated(shellttp)) {
        chThdRelease(shellttp);  /* Recovers memory of the previous shell. */
        shellttp = NULL;          /* Triggers spawning of a new shell. */
    }
}
Projects: tiltOne firmware port

- Port the existing firmware code to ChibiOS/RT
- Exploit the command shell to receive commands from a PC
Write the firmware for an IMU exploiting ChibiOS/RT
Projects: ChibiOS/RT HAL port to Arduino

- Complete the port of ChibiOS/RT HAL to the Arduino platform
Projects: ChibiOS/RT HAL simulator

- Write an HAL simulator for ChibiOS/RT