

SimpleRTK: Minimal Real-Time Kernel for Time-Driven and Event-Driven Control

Diego García, Manel Velasco, and Pau Martí

¹ Automatic Control Department, Technical University of Catalonia
Barcelona, Spain

Research Report: ESAII-RR-10-01

January 2010

Abstract

This report presents the SimpleRTK, a minimal multi-tasking pre-emptive real-time kernel based on Microchip© dsPIC33FJ256GP710. It has been designed pursuing two goals. First, it aims at becoming a valuable software tool in practical assignments in undergraduate and graduate courses on networked and embedded control systems. Hence, it includes only basic services, avoiding excessive memory/time overhead and unnecessary complexity. Second, it offers simple but effective support to small control applications characterized by timing properties different than those exhibited by periodic controllers. In particular, it suits applications based on feedback scheduling approaches or on event-driven control methods, where tasks executions are non-periodic in the general case.

Keywords: Networked and embedded control systems, real-time kernel, time-driven and event-driven controllers

1 Introduction

Applications in networked and embedded control systems are severely constrained in terms of both available resource and performance requirements [1].

To suit these demands, the real-time and control communities have recently provided diverse theoretical results on both control and resource optimization for resource limited computing systems concurrently executing several controllers. Loosely speaking, most of these results, characterized by feedback scheduling approaches (e.g. [2, 3]) or event-driven control methods (e.g. [4, 5, 6, 7]), suggest to efficiently select the controllers' sampling periods. As a consequence, controllers' execution rates are different from those provided by the standard periodic sampling approach [8]. In addition, the implementation of controllers based on the referred theory in resource limited computing platforms is not straightforward, demanding specific kernel support, as stressed in [9].

This report describes in detail the SimpleRTK, a simple home-made multi-tasking pre-emptive real-time kernel targeting a low cost microprocessor, specifically designed i) to support non-conventional sampling mechanisms for control applications, and ii) to become a valuable software platform for the education of embedded systems engineers. It provides support for time-driven (multi-rate) and event-driven (or self-triggered) controllers. In short, it offers support for periodic hard real-time tasks under Earliest Deadline First (EDF) scheduling [10]. In addition, it supports event-driven control tasks by allowing re-setting task timing constraints at run-time. The kernel has an unlimited timing, that is, it has an infinite lifetime. As a synchronization and resource sharing mechanisms, semaphores have been adopted. Inter-task communication is achieved by Cyclical Asynchronous Buffers (CAB). A reduced set of primitives permit an easy task management as well as an intuitive usage of the implemented mechanisms.

The SimpleRTK can be tailored to obtain a computing platform with real-time kernel support for dynamic resource management (e.g., see [11], [12], or [13] for rate adaptation within available processor capacity, or [14] for an energy aware rate adaptation approach) as demanded by modern control applications. Note however that available real-time kernels or operating systems enhanced for example with rate adaptation are in general not suitable for simple microprocessor architectures due to resource limitations, such as real-time Linux [15]. Moreover, from an educational point of view, their internal structure is often too complex for those students with a low profile in (real-time) operating systems.

Hence, it looks more desirable to work with small real-time kernels tar-

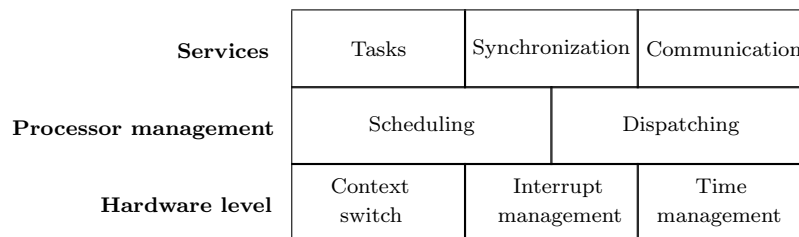


Figure 1: SimpleRTK structure.

getting small architectures (e.g., [16], [17] [18], [19], [20], [21], [22], [23], [24] [25] [26]). However, many of them do not provide support for task rate adaptation or their support for event-driven control applications has not been reported. In any case, they are complementary to SimpleRTK, and their internals are accessible, easy to understand and modify, in order to tailor them to the specific application needs. Many design principles of the SimpleRTK have been borrowed from the TinyRealTime kernel [21].

The kernel code is available under GPL (*General Public License*) license (version 3) at <http://code.google.com/p/simplertk/>. Its documentation is available in pdf in <http://simplertk.googlecode.com/files/refman.pdf> and in html in <http://simplertk.googlecode.com/files/html.zip>.

The simpleRTK has been designed using the Subversion system <http://subversion.tigris.org/> for the code development, Doxygen <http://www.stack.nl/~dimitri/doxygen/> as a tool for source code documentation generator, and Microchip MPLAB IDE <http://www.microchip.com/> for code development. The kernel prototype has been tested in the Full Flex board from Evidence <http://www.evidence.eu.com/>.

Figure 1 shows the hierarchy structure of the kernel, which can be divided into three layers:

- **Hardware layer.** This layer directly interacts with the dsPIC hardware. Mainly written in assembler, their functions and primitives relate to context switch, interrupt management and time management.
- **Processor management layer.** This layer primitives concern tasks scheduling and dispatching.
- **Services layer.** This layer contains the set of services provided by the kernel to the user. Basically, it provides primitives for task management, and task synchronization and communication.

The source code of the SimpleRTK is divided into 3 files. **simplertk.c** contains the primitives of the kernel management which basically refers to the kernel initialization, task management and semaphore management. **swctx.s** contains the context switch routine that is periodically generated by *Timer 1*. **cab.c** contains the primitives for the CAB management. Definitions are given in **simplertk.h** and **cab.h**.

The rest of this report is organized as follows. The target hardware and development software are described in Section 2. Section 3 presents the simpleRTK structure. Section 4 describes the diverse task management mechanisms. Section 5 describes synchronization and communication issues. Section 6 presents the memory organization and management. Section 7 and 8 describes the time management and scheduler, respectively. Section 9 describes the kernel operation. Section 10 describes two application examples using the SimpleRTK. Finally, Section 11 concludes the report. Appendix A lists all the kernel primitives, appendix B gives the kernel source code, and appendix C provides the basic instructions to compile the kernel.

2 Development framework

This section describes the hardware and software components used in the development of the SimpleRTK.

2.1 Hardware

The targeted micro-controller is the Microchip© dsPIC33FJ256GP710, which is the central unit of the Full Flex training board. The Flex board (in its full version) represents a good compromise between cost, processing power, and programming flexibility. It was produced as a development board for building and testing real-time applications using standard components and open source software. The board includes apart from the dsPIC, a socket for the 100 pin Plug-In Module (PIM), an ICD2 (in-circuit debugger) programmer connector, a USB (Universal Serial Bus) connector for direct programming, power supply connectors, a set of leds for monitoring the board, an on-board Microchip PIC18F2550 micro-controller for integrated programming, and a set of connectors for daughter boards piggybacking. Further information can be found in the Evidence webpage <http://www.evidence.eu.com>

The 16-bit dsPIC33FJ256GP710 main characteristics include 16 general purpose registers, Flash memory of 256 Kbytes, 30 Kbytes of SRAM memory (which includes 2 Kbytes of RAM DMA), up to 40 MIPS operation, several clock options, and 9 16-bit timers/counters with different pre-scalers. It also

contains a PWM module, as well as a Quadrature Encoder. See Microchip webpage <http://www.microchip.com> for further information.

2.2 Software

The software tools used throughout the source code development are the Subversion system, Doxygen, and MPLAB. The Subversion system is a source code control version software that helps to control changes in the source code. As a server, Google Code (<http://simplertk.googlecode.com/svn/trunk/>) has been used. Doxygen is a tool for automatic documentation generation. Finally, MPLAB Integrated Development Environment (IDE) is the source code development framework and cross-compilation platform, targeting the dsPIC, which includes a C editor. As a compiler, MPLAB C30 has been adopted, which is a port of the GCC compiler of the Free Software Foundation under the standard ANSI x3.159-1989. MPLAB also permits several debugging options. The programmer and debugger used is the ICD2 from Microchip, which permits to download the kernel code from MPLAB to the dsPIC.

3 Kernel structure and management

The kernel structure, shown in listing 1 contains the number of created tasks in `nbrOfTasks`, the current executing task identifier in `running`, a vector of task control structures named `tasks`, a semaphores vector named `semaphores`, the `memptr` pointer to the first free memory position, and the system ticks counter named `cycles`.

Listing 1: Kernel structure definition

```
struct kernel {
    unsigned char nbrOfTasks;    // number of tasks created
    unsigned char running;
    struct task tasks[MAXNBRTASKS+1]; // +1 for the idle task
    unsigned char semaphores[MAXNBRSEMAPHORES]; // counters for
                                                // semaphores
    unsigned int *memptr; // pointer to free memory
    unsigned long cycles; // number of ticks
} kernel;
```

Two primitives have been defined for kernel management. `srtInitKernel` initiates the oscillator, the information structures required for kernel and tasks management, and the real-time clock interrupts. The second primi-

tive, **srtCurrentTime** returns the current time in ticks from the **cycles** counter last overflow (see further details on time management in section 7).

Later in section 9 the kernel operation will be described in more detail.

4 Task management

Each task has associated a control structure of 11 bytes, as shown in listing 2.

Listing 2: Task structure definition.

```
struct task {
    unsigned int sp;           // Stack pointer
    unsigned long release;
    unsigned long deadline;
    unsigned char state;     // 0=terminated, 1=readyQ,
                            // 2=timeQ, 3=waiting for Sem1,
                            // 4=waiting for Sem2, etc.
};
```

The **sp** is the *Stack Pointer*, 16-bit pointer to address up to 65536 bytes (64 Kilobytes). The **release** indicates the relative instant, in *ticks*, when the task becomes ready for execution. In fact, it can be considered as the task period. The **deadline** indicates the relative deadline and **state** indicates the task state. States are defined later on in this section.

And example of task definition is given in listing 3. An infinite loop is defined, where the task actions must be coded. The task finishes with the **srtSleep** primitive, that sleeps the task.

Listing 3: Task definition.

```
void task(void *args){
    data=(TYPE*)args; //argument casting
    while(1){
        //Control operations
        ...
        //sleep task
        srtSleep(
            SECONDS2TICKS(task_release),
            SECONDS2TICKS(task_deadline));
    }
}
```

Task management is achieved using the following five primitives, which are listed in detail in Appendix A. **srtCreateTask** creates the task structure and its stack, with the option of passing an argument. The stack size

is user defined, and it is described in Section 6. **srtTerminate** specifies the task state to *terminated* and the task memory becomes garbage. **srtSleep** permits to specify periodic or aperiodic (event-driven) task executions by sleeping the task with a new relative *release* and *deadline*. **srtGetDeadline** returns the current task deadline and **srtGetRelease** returns the current task period.

The number of tasks is user defined in the constant `MAXNBRTASKS`. The maximum number of tasks is 255 due to the `char` type of the task identifier.

The tasks states and transition between states in the kernel are shown in Figure 2. Five states have been defined (see *simplertk.h*). A task becomes **RUNNING** when the *dispatcher* allocates the task to the processor. **READY** indicates that the task is ready to execute, but it can not be executed because the processor is currently serving another task. **SLEEP** indicates that the task is not active because is has been recently created or because is has been suspended for a given time interval. The task enters in **WAIT_OFFSET**+*i* when asks for using semaphore *i* (with $i > 0$) that was previously assigned to another task. Finally, **TERMINATED** indicates that the task has finished its execution.

As a design choice, there are no queues for ready or blocked tasks. This has been decided to reduce the kernel overhead. Note that usually queues are ordered by deadline (if EDF is used) or priorities (if Fixed Priority is used). However, since tasks may change their deadline at each task execution, the re-ordering would produce considerable overhead. Hence, the kernel will check both the task state and deadline in each task control block, and proceed with the EDF scheduling with those tasks being **READY**, and selecting that one with smallest deadline with respect to the current time.

5 Synchronization and communication

For resource sharing and synchronization, semaphores have been implemented. They can be managed by 3 primitives. **srtCreateSemaphore** creates a semaphore, indicating its identifier and the number of resources. **srtWait** acquires a resource by decrementing the number of available resources. If no resources are available, the task is blocked. **srtSignal** releases a resource. It must be noted that the implemented mechanisms may produce the priority inversion problem, which can be solved by implementing a priority inheritance protocol [27].

For inter-task communications, the technique of Cyclical Asynchronous Buffers (CAB) has been implemented. In general, many communication

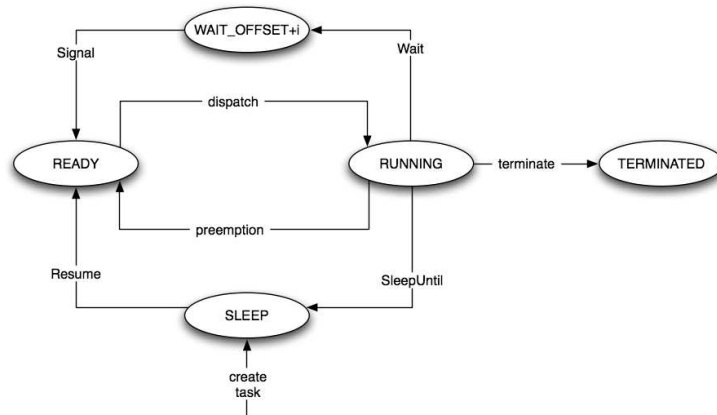


Figure 2: Task states diagram.

mechanisms block tasks, that is, the receiver task becomes blocked until the sender task sends the message. On the contrary, CABs offer non-blocking communication. Messages are not consumed for the receiver, they are maintained until they are over-written by another message. Note that if the receiver is faster than the sender, the same message can be read several times. And if the sender is faster than the receiver, messages are lost. This should not be a problem for example with a sampling task sending samples to a controller task. Controllers require using up-to-date information.

The CAB structure is shown in Figure 3. It consists of a control block, shown in Listing 4, with a `buffers` pointer that points to a vector of the CAB buffers, a `free` pointer that points to the free buffer, a `mrB` pointer that points to the most recent buffer, a `max_buf` variable that stores the number of buffers, and a `dim_buf` variable that stores the dimension of all buffers within the CAB.

Listing 4: CAB structure definition.

```

struct cab{
    buffer **buffers;    // pointers to buffers
    pointer free;       // free buffer
    pointer mrB;        // most recent buffer
    unsigned int max_buf; // maximum buffers
    unsigned int dim_buf; // buffers dimension
};
  
```

And each buffer has 3 entries, the `next` pointer that points to the next free buffer, the `use` entry that indicates the number of tasks that are accessing

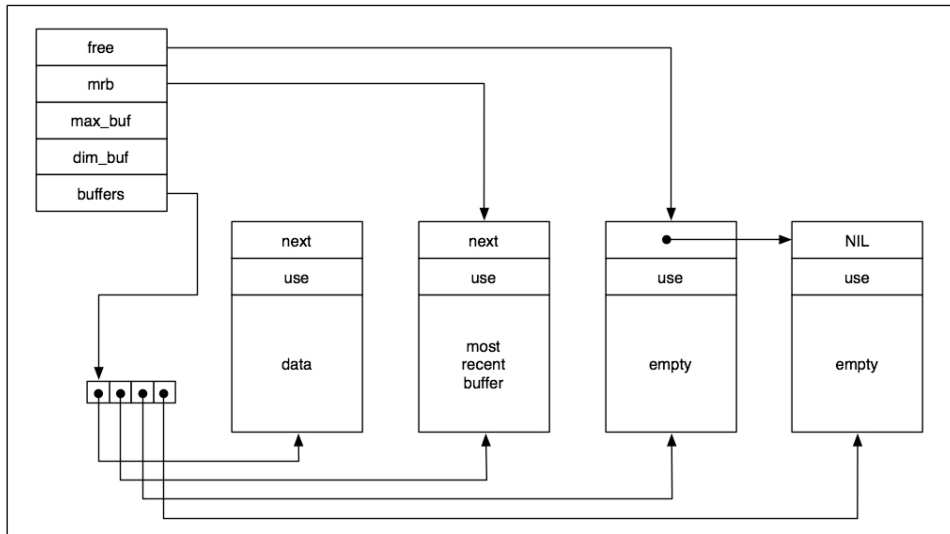


Figure 3: CAB data structure.

the buffer data, and the `data` entry that stores the message, as illustrated in Listing 5

Listing 5: Buffer structure definition.

```
struct buffer{
    pointer next;    // pointer to next buffer
    unsigned int use; // number of readers reading
                  // this buffer
    char *data;     // pointer to the data
};
```

The CAB management is achieved through 6 primitives. `srtOpencab` creates the CAB structure specifying the number of buffers and the data size for each buffer. `srtDeletecab` frees the CAB memory. `srtReserve` reserves a free buffer for writing. `srtPutmes` defines the current written buffer as the most recently used buffer. `srtGetmes` reserves the most recent buffer for reading. Finally, `srtUngget` frees the buffer after the reading. Hence, if a task wants to write a message, it reserves a buffer, writes the message, and marks the buffer as the most recent one. Then, if a task wants to read a message, it reserves the most recent buffer, reads the messages, and frees the buffer. In order to use CABs, specific memory must be used. This is further explained in next section.

6 Memory management

This section explains the stack structure associated to each task, the overall memory organization, and finally characterizes the kernel footprint.

6.1 Task stack

The task stack stores the local task variables and its context. The stack structure is shown in Figure 4. The stack starts being created in a descendent order from the inferior cell pointed by the `memptr` pointer, which points to the first free memory position.

Following a descending order, the stack registers are the following. `PSVPAG`, *Program Space Visibility Page*, its an 8-bit register that allows accessing to a specific region of the program space from the data space. This register is concatenated with the 16 bit EA, *Effective Address*¹, register in order to cover the address space size of the program space, 24bits. `CORCON`, *Core Control*, its a control register that allows configuring DSP features, activating (or not) the Program Space Visibility to map program space memory from the data space, and configuring the CPU priority level. `TBLPAG`, *Table Page*, its an 8 bit register that defines a 32K *words* region in the program space, allowing table operations. This register is concatenated with the EA register to cover the full addressing space. `RCOUNT`, *Repeat Loop Counter*, its a 14 bits register that contains the instruction `REPEAT` loop counter. `W0, W1, W2, . . . , W14`, *Working Registers*, are general purpose registers that can be used for data, addresses or register shifts. In the first task execution, the task argument address is stored in `W0`. `SR`, *Status Register*, its a 16 bit register that maintains the most recent executed instruction status. Additionally, it has 3 bits to determine the CPU priority level when an interrupt occurs. `SPLIM`, *Stack Pointer Limiter*, its a 16 bits register associated to the stack pointer. It is used to avoid the stack pointer overflow and an inferior memory access according the the stack memory reserved by the user. `SR<7:0> IPL3 PC<22:16>`, its the concatenation of the 0 to 7 bits of `SR` register, which indicate the flags status (negative, zero, overflow, etc.), the `IPL3` bit of `CORCON` register specifies that the interrupt priority is greater or lower than 7, and bits from 16 to 22 form the *Program Counter*. *task pointer* points to the task first instruction. *stack space reserved*, is the free memory space to store task temporal variables.

¹All effective addresses are of 16 bits and point to bytes inside the data space.

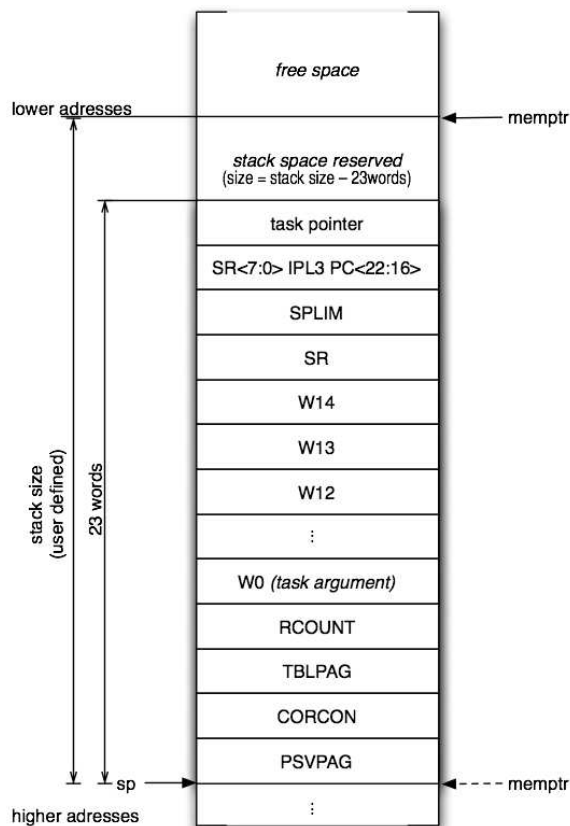


Figure 4: Task stack structure. *memptr* pointer (pointer to free memory), dotted line, indicates the memory position pointed before creating the stack. *sp* (*stack pointer*) and *memptr* pointers, solid lines, indicate the memory positions once the stack has been created.

6.2 Memory organization

The dsPIC memory organization is as follows. It contains 32768 data memory positions (32 Kilobytes). The first 2048 positions (addresses from 0x0000 to 0x07FF) are used for the SFR *Special Functions Registers* space, and the following 30720 positions (addresses from 0x0801 to 0x7FFF) represent the intern SRAM. In the SimpleRTK, the defined memory map is shown in Figure 5, where the kernel data structures and tasks structures and associated stacks can be identified. In addition, the *Heap* zone is dynamic memory

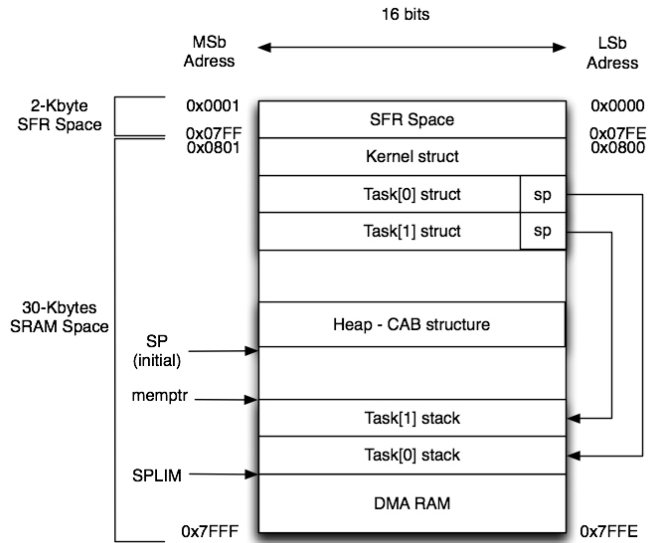


Figure 5: SimpleRTK data memory map.

reserved for optional CAB implementation. The task and kernel structures can be seen in Listings 2 and 1, respectively. The kernel data structure and task data structures have been assigned from the lower addresses. On the contrary, tasks stacks have been assigned from a high address given by the SPLIM register (its value is assigned in kernel linking time, and corresponds to the highest possible address of the non-used data memory). Note that the stack size is user defined, and therefore, its user responsibility to make a safe use of the memory.

The *Heap* memory is for optional CAB use. It must be explicitly reserved using dynamic memory functions such as *malloc* or *free*. When a CAB communication structure is used, it is stored in this memory area. Each CAB buffer occupies $6 + \text{dim_buf}$ bytes, where *dim_buf* is each buffer dimension. All buffers of a single CAB have the same size. Hence, the CAB *i* structure occupies $10 + 2 \times \text{max_buf} + \text{max_buf} \times (6 + \text{dim_buf})$, where *max_buf* is the CAB buffers number.

6.3 Kernel footprint

The minimum SimpleRTK size is 2007 bytes, which includes the semaphores mechanism and without using CABs.

As shown in the kernel structure, the tasks and semaphores structure

vectors have a size given by `MAXNBRTASKS` and `MAXNBRSEMAPHORES`, which are defined in *simplertk.h*. In particular, an extra task (+1) is added for the *idle* tasks, whose structure occupies the first vector position.

Each task control structure sizes 11 bytes. Hence, the size of the tasks control structure vector is $11 \times (\text{MAXNBRTASKS} + 1)$ bytes. On the other hand, each semaphore sizes 1 byte. Hence, the size of the semaphores vector is `MAXNBRSEMAPHORES` bytes. Summing up, the kernel structure sizes $8 + 11 \times (\text{MAXNBRTASKS} + 1) + \text{MAXNBRSEMAPHORES}$ bytes. In addition, each task has a stack associated, which stores the micro-controller context (46 bytes, as described in previous section 6) and the temporal variables, defined by the user and stored in `Space_reservedi`. Hence, the space required for `MAXNBRTASKS` stacks is $\text{MAXNBRTASKS} \times 46 + \sum_{i=0}^{i=\text{MAXNBRTASKS}} \text{Space_reserved}_i$ bytes, where $i = 0$ corresponds to the reserved space for the temporal variables of the *idle* task. Hence, the total kernel size is

$$19 + 57 \times \text{MAXNBRTASKS} + \text{MAXNBRSEMAPHORES} + \sum_{i=0}^{i=\text{MAXNBRTASKS}} \text{Space_reserved}_i$$

By adding m CAB structures, the total required memory is `Kernel` + $\sum_{j=1}^{j=m} \text{CAB}_j$ where `Kernel` size is given by the previous expression, and $\text{CAB}_j = 10 + 2 \times \text{max_buf} + \text{max_buf} \times (6 + \text{dim_buf})$ (given in the previous subsection).

7 Time management

The time management implemented in SimpleRTK is based on a counter that is incremented by the interrupt handling routine of a *Timer1*. The timer frequency depends on micro-controller operation frequency, which has been specified using the primary oscillator (XT, crystal) with PLL activated, and coded in the kernel initialization. In particular, the operation frequency is `FCY = 40 MHz`.

The system tick duration depends on the period value of the *timer* and `FCY`, and it is defined by the `TIMER_VALUE` constant in hexadecimal (`0x9C40`). With this configuration, the real-time clock resolution (seconds per instruction) is 25 ns. Hence, every 25ns, the timer increments by one the system time. After `0x9C40` (40.000) *Timer1* increments, that is, at each 1 ms, an interrupt is generated to increment the system tick counter, which is stored in *unsigned long cycles*.

The system time representation is linear (other standard representations include the cyclic representation [28], [29]). The maximum value for the system tick counter `cycles` is $2^{32} - 1$, which determines the *system life*

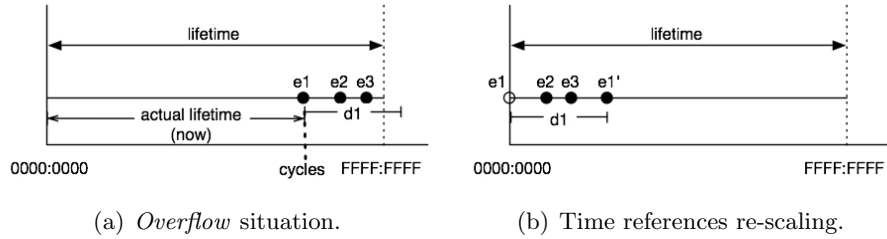


Figure 6: *Overflow* example of a time reference.

time. If this limit is not managed, the system has a finite time, and in particular, with the defined settings, this would be 49.71 days. Incrementing the prescaler will prolong the system life time. However, counter overflow will still occur.

In order to have an infinite system life time, counter overflow is detected and a time shifting technique applied. To detect the overflow, the system time is temporary stored in a variable, and the *Status register* overflow bit *SR* is checked whenever a time interval must be added to any time reference. If overflow exists, all time references are re-scaled using the `restartCycle()` function, which shifts all time references using the current time, and the `cycles` counter is reset to zero.

Figure 6 illustrates an example of time scaling when overflow occurs. Figure 6(a) shows 3 events e_1 , e_2 and e_3 which are near to the timing line limit (FFFF:FFFF). Being `cycles` the current time, if a quantity d_1 is added to the event e_1 such that the limit is reached, then a time shifting of `cycles` is forced to all time references. The result can be seen in Figure 6(b), where the same relative time distances between events is kept. And then the addition of d_1 on the e_1 event is performed resulting in a new e_1' event, without causing overflow.

This mechanism applies to the operations that increment time references, such as `cycles` increment and the task sleep operation when *release* and *deadline* are incremented. This mechanism does not take into account the time references defined by the user. Hence, it is the user responsibility to manage them properly.

8 Scheduler

The only scheduler implemented in the SimpleRTK is EDF, which schedules tasks dynamically according to their absolute deadline: the lower the

deadline, the higher the priority. In order to find the highest priority task, the scheduler performs a search through all the tasks to find the task with smallest deadline. Hence, it is worth noting that the number of created tasks (specified in `MAXNBRTASKS`) heavily influences the kernel time overhead in the sense that the scheduler must do a linear search on the tasks vector ($O(n)$, being n the number of tasks).

For periodic tasks, the EDF feasibility test is [10]

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

where C_i is the task i worst case execution time and T_i is the task i period. For aperiodic tasks, and in particular for those tasks that change their *release* and *deadline* using the `srtSleep(r,d)` primitive as in the case of self-triggered tasks, a feasibility test can be found in [30].

9 Kernel operation

The kernel internal operation consists in the initialization routine and the timer interrupt service routine (ISR). The first one configures the system time, initializes the kernel structure, the *idle* task structure, and configures the *Timer 1* interrupt.

The flux diagram of the timer ISR is shown in Figure 7. It performs the following steps. First, it saves the micro-controller context, saving all the register values into the running task (`RUNNING` state) stack. Then, it disables all interrupts, clears the interrupt flag, and increments the `cycles` tick counter. Then it stores the stack pointer (`W15` register) in the `SPTMP` global variable and the EDF scheduler is called. The scheduler checks if there is any `READY` task with higher priority than the `RUNNING` task. If there is such a task, it returns the task *id*. Otherwise, it returns `0xFF` (maximum value for `unsigned char`). If a context switch is required, the *dispatcher* is called to update the micro-processor stack pointer (`W15` register) to the stack of the task to be executed. Then, the context is loaded by assigning all the stack values to the micro-controller registers. Finally, the return instruction `RETFIE` is executed, which loads the `RCOUNT` register with the stack head 8 to 15 bits, restores the processor priority level by assigning the `CORCON` register `IPL3` bit to the stack head 7 bit, restores `PC Program Counter` register bits 16 to 22 to the head task 0 to 6 bits, decrements the stack head, and restores `PC Program Counter` register bits 0 to 15 with the 0 to 15 bits of the stack head. When the `PC` is loaded, the next instruction to be executed is defined.

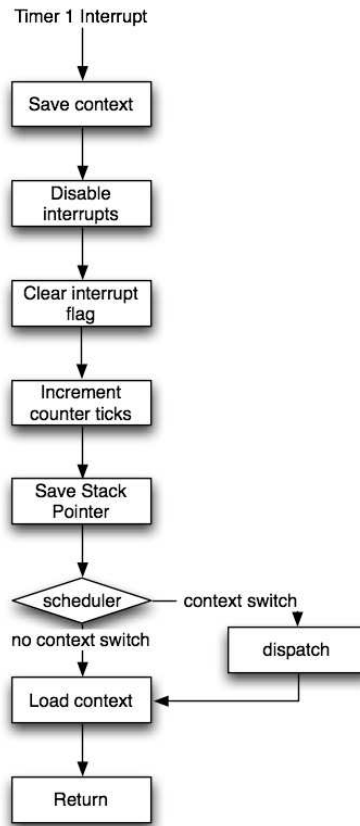


Figure 7: *Timer 1* interrupt service routine.

The number of instructions of this *Timer 1* routine is 133 when there is no task to manage, that is, $3.3\mu s$. For each task, 65 additional instructions are added ($1.6\mu s$ per task). This is due to the fact that the EDF scheduler search through all tasks. Hence, the SimpleRTK overhead with $MAXNBRTASKS$ tasks in the worst case is $1,6 \times MAXNBRTASKS + 3,3\mu s$.

It is worth noting that no specific interrupt management has been designed for the SimpleRTK. Different interrupt management techniques are discussed in [31]. The only interrupt management applied concerns to the fact that the CPU enters in priority level 7 (highest priority) when the Timer1 ISR executes. Otherwise, the CPU priority level is 1. If other interrupt handlers must be implemented, management of the CPU priority levels must be specified.

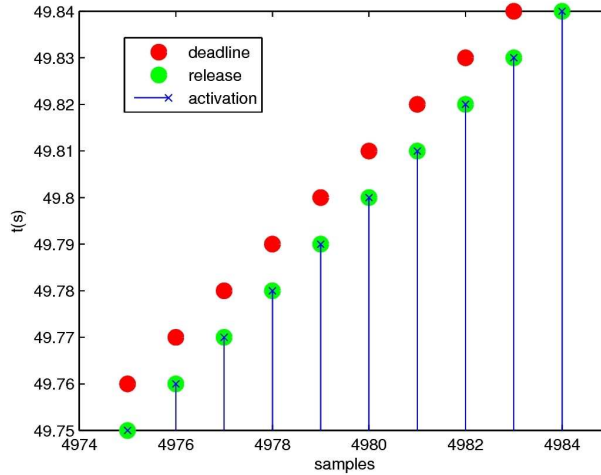


Figure 8: *Release, deadline* and activation time of a task for a CPU load of 99%.

10 Application examples

Two application examples have been developed to show the correct operation of the SimpleRTK. Their code can be found in <http://simplertk.googlecode.com/svn/trunk/tests/>. For the two applications, the Flex board has been used, and a periodic task using the RS232 communication link exports the relevant data.

10.1 Kernel stress

This application consists in incrementing the kernel load and to observe that a set of defined periodic tasks meet their temporal restrictions. To do so, 10 periodic tasks have been created. The first 9 are dummy periodic tasks that have a period of 10ms with a tunable execution time (using the `delay(τ)` instruction, where τ are nanoseconds) that permit to increment in a controlled manner the task set utilization. The 10th task is the debugging RS232 periodic task. Figure 8 shows the *release, deadline* and activation time of a task for a CPU load of 99% (when the dummy tasks have an execution time of 1.1 ms). As it can be seen, the expected timing is achieved. And the same occurs for lower processor utilization numbers. Note that the time spent for the monitoring task (6ns) with a period of 10ms, and the kernel (19 μ s) do not affect significantly the task set utilization (< 1%).

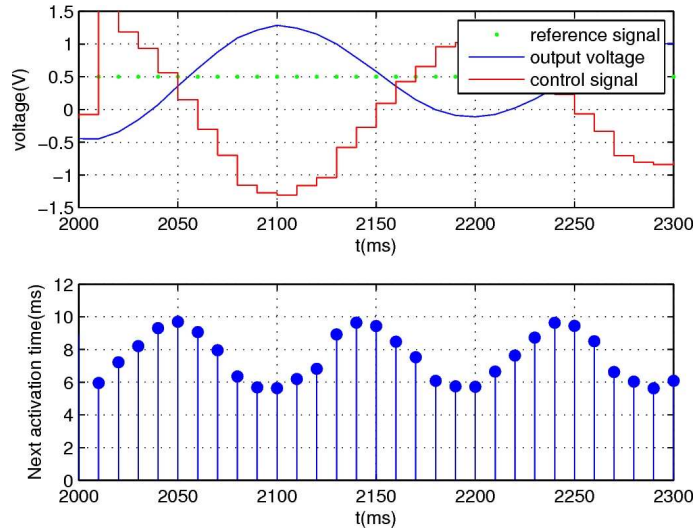


Figure 9: Plant response (top) and control task activation times (bottom).

10.2 Self-triggered control

In [32] it was shown a prototype implementation of an event-driven self-triggered controller, which is controller that at each job execution, apart from performing sampling, control algorithm computation and actuation, calculates the next job activation time as a function of the plant state. As a controlled plant, a double integrator electronic circuit was used.

This example mimics the application shown in [32]. Specifically, three tasks have been implemented. Two hard real-time periodic tasks, one for monitoring through RS232 with a period of $10ms$, and another one for changing the plant references (to emulate plant perturbations) with a period of $1s$. The third task is self-triggered, and this is achieved by specifying at the end of its execution a new release and deadline parameters using the `srtSleep` primitive.

Figure 9 shows the plant response and the activation times, which vary at each task execution. The interested reader is referred to [32] for further information.

11 Conclusions

This report has described the SimpleRTK, a minimal multi-tasking preemptive real-time kernel designed to support small control applications and

to become a valuable tool for embedded systems laboratory assignments.

The kernel has been evaluated from a real-time system point of view as well as in an embedded control application with two periodic tasks and one self-triggered control task in charge of controlling a double integrator electronic circuit.

References

- [1] G. Buttazzo, “Research Trends in Real-Time Computing for Embedded Systems,” *ACM SIGBED Review*, v. 3, n. 3, 2006.
- [2] P. Martí, C. Lin, S. Brandt, M. Velasco, and J.M. Fuertes, “Draco: Efficient Resource Management for Resource-Constrained Control Tasks,” *IEEE Transactions on Computers*, Jan. 2009.
- [3] M-M. Ben Gaid, A. Çela, and Y. Hamam, “Optimal Real-Time Scheduling of Control Tasks with State Feedback Resource Allocation,” *IEEE Transactions on Control Systems Technology*, Vol. 17, n. 2, pp. 309 – 326, March 2009.
- [4] P. Tabuada, “Event-triggered real-time scheduling of stabilizing control tasks,” *IEEE Transactions on Automatic Control*, vol. 52, n. 9, pp. 1680–1685, 2007.
- [5] W.P.M.H. Heemels, J.H. Sandee, and P.P.J. van den Bosch, “Analysis of event-driven controllers for linear systems;” *Int. Journal of Control*, 81(4), 571–590, 2008.
- [6] T. Henningsson, E. Johannesson, A. Cervin, “Sporadic Event-Based Control of First-Order Linear Stochastic Systems,” *Automatica*, vol. 44, n. 11, pp. 2890–2895, Nov. 2008
- [7] X. Wang, and M. Lemmon, “Self-triggered Feedback Control Systems with Finite-Gain L2 Stability” *IEEE Transactions on Automatic Control*, vol. 54, n. 3, pp. 452-467 , 2009.
- [8] K.J. Åström and B. Wittenmark, *Computer-Controlled Systems*, Third Edition, Prentice-Hall, 1997.
- [9] R. Marau, P. Leite, M. Velasco, P. Martí, L. Almeida, P. Pedreiras, and J.M. Fuertes, “Performing Flexible Control on Low Cost Microcontrollers using a Minimal Real-Time Kernel,” *IEEE Trans. Industrial Informatics*, Vol. 4, N. 2, pp. 125-133, May 2008.

- [10] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment," *Journal of the ACM*, vol. 20, n.1, pp. 40-61, 1973.
- [11] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, no. 3, pp. 289-302, Mar. 2002.
- [12] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *24th IEEE Real-Time Systems Symposium*, Dec. 2003, pp. 396-407.
- [13] S. Goddard and L. Xu, "A variable rate execution model," in *16th Euromicro Conference on Real-Time Systems*, July 2004, pp. 135-143.
- [14] M. Marinoni and G. Buttazzo, "Elastic dvs management in processors with discrete voltage/frequency modes," *IEEE Transactions on Industrial Informatics*, vol. 3, no. 1, pp. 51-62, 2007.
- [15] Real-time Linux Foundation, Inc., <http://www.realtimelinuxfoundation.org/>
- [16] J. A. Stankovic and K. Ramamritham, "The spring kernel: a new paradigm for real-time operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 3, pp. 54-71, 1989.
- [17] K. M. Zuberi, P. Pillai, and K. G. Shin, "Emeralds: a small-memory real-time microkernel," in *7th ACM symposium on Operating Systems Principles*, 1999, pp. 277-299.
- [18] P. Gai, G. Lipari, L. Abeni, M. di Natale, and E. Bini, "Architecture for a portable open source real-time kernel environment," in *Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, November 2000.
- [19] E. Mumolo, M. Nolich, and M. Noser, "A hard real-time kernel for motorola microcontrollers," in *23rd International Conference on Information Technology Interfaces*, June 2001.
- [20] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, "A new kernel approach for modular real-time systems development," in *13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

- [21] D. Henriksson and A. Cervin, “Multirate feedback control using the TinyRealTime kernel,” in *Proceedings of the 19th International Symposium on Computer and Information Sciences*, Antalya, Turkey, Oct. 2004.
- [22] uC/OS-II, <http://micrium.com/page/products/rtos/os-ii>.
- [23] J.J. Labrosse, *MicroC OS II: The Real Time Kernel*, 2002.
- [24] CMX TINY+ RTOS, <http://www.cmx.com/>
- [25] ERIKA, Evidence srl., <http://erika.tuxfamily.org/>
- [26] FreeRTOS, <http://www.freertos.org/>.
- [27] L. Sha, R. Rajkumar, and J.P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, *IEEE Transactions on Computers*, vol.39, n.9, pp. 1175–1185, 1990
- [28] A. Carlini and G. Butazzo, “An efficient time representation for real-time embedded systems, “, in *Proceedings of the 2003 ACM Symposium on Applied Computing*, pp. 705–712. ACM, New York, NY, USA, 2003.
- [29] G. Butazzo and P. Gai, “Efficient edf implementation for small embedded systems,”, in *Proc. of the 2nd Int. Workshop on Operating Systems Platforms for Embedded Real-Time applications*, Dresden, Germany, July 2006.
- [30] M. Velasco, P. Martí, and E. Bini, “Control-driven Tasks: Modeling and Analysis,” in *29th IEEE Real-Time Systems Symposium*, Barcelona, Spain 2008.
- [31] G. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.
- [32] E. Bini, M. Velasco, P. Martí, A. Camacho, “Demo Abstract: Implementation of Self-triggered Controllers”, in *Demo Session of 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, CA, USA, April, 2009.

A Primitives

This section lists the available primitives of the SimpleRTK to manage the kernel, tasks, semaphores, and CABS. For each primitive, a brief description is given, the syntax is defined, parameters are explained and a short example of usage is presented.

A.1 Kernel management

The primitives for kernel management are:

- kernel initialization.
- get elapsed time.

For using these primitives the definition file *simplertk.h* must be included.

A.1.1 Kernel initialization

- **Description**

It initializes the oscillator, the information structures required for kernel and task management, and the real-time clock interrupts.

- **Syntax**

```
void srtInitKernel ( int idlestack )
```

- **Parameters**

– *idlestack*: *idle* task stack size.

- **Example of use**

```
#include "simplertk.h"

int main(){
    srtInitKernel(80);
    ...
}
```

A.1.2 Get elapsed time

- **Description**

It returns the system current time in *ticks*. Reminding that the kernel as an infinite *timing*, it returns the system time elapsed from the last system *tick* counter overflow (not the time elapsed from the system initialization).

- **Syntax**

```
unsigned long srtCurrentTime ( void )
```

- **Example**

```
#include "simplertk.h"

void task(void *args){
    unsigned long systemTime;
    while(1){
        systemTime=srtCurrentTime();
        ...
        srtSleep(SECONDS2TICKS(0.05),
                SECONDS2TICKS(0.05));
    }
}
```

A.2 Task management

Task management primitives are:

- Task creation.
- Task termination.
- Get current (executing) task *deadline*.
- Get current (executing) task *release*.
- Sleep task.

For using these primitives the definition file *simplertk.h* must be included.

A.2.1 Task creation

- **Description**

It creates a task with an specific stack size, *release* and *deadline*, and optionally, with an argument direction. Both *release* and *deadline* must be in *tick* unit. To introduce these parameters in *second* units, the `SECONDS2TICKS(seconds)` function can be used, where `seconds` is the parameter in seconds units.

- **Syntax**

```
void srtCreateTask ( void(*) (void *) fun,
                    unsigned int stacksize,
                    unsigned long release,
                    unsigned long deadline,
                    void * args)
```

- **Parameters**

- *fun*: first task instruction address.
- *stacksize*: task stack size in *words*
- *release*: task *release* in *ticks*
- *deadline*: task *deadline* in *ticks*
- *args*: task argument address

- **Example**

```

#include "simplertk.h"

void myTask(void *args){
    while(1){
        //actions
        ...
        srtSleep(SECONDS2TICKS(0.06),
                SECONDS2TICKS(0.06));
    }
}

int main(){
    srtInitKernel(80); //initilize kernel
    srtCreateTask(
        myTask,    //task pointer
        100,      //task stack size (100 words)
        SECONDS2TICKS(0.06), //release
        SECONDS2TICKS(0.06), //deadline
        &null);   //task argument pointer

    ...
    while(1); //idle
}

```

A.2.2 Task termination

- **Description**

It terminates the current executing task.

- **Syntax**

```
void srtTerminate ( void )
```

- **Example**

```

#include "simplertk.h"

void myTask(void *args){
    while(1){
        //actions
        ...
        if(final_condition){
            srtTerminate();
        }
        srtSleep(SECONDS2TICKS(0.06),
                SECONDS2TICKS(0.06));
    }
}

```

A.2.3 Get current (executing) task *deadline*

- **Description**

It returns the current executing *deadline* in *ticks*.

- **Syntax**

```
unsigned long srtGetDeadline ( void )
```

- **Example**

```

#include "simplertk.h"

void myTask(void *args){
    unsigned long deadline;
    while(1){
        //actions
        ...
        deadline=srtGetDeadline();
        srtSleep(SECONDS2TICKS(0.06),
                SECONDS2TICKS(0.06));
    }
}

```

A.2.4 Get current (executing) task *release*

- **Description**

It returns the current executing task *release* in *ticks*.

- **Syntax**

```
unsigned long srtGetRelease ( void )
```

- **Example**

```
#include "simplertk.h"

void myTask(void *args){
    unsigned long release;
    while(1){
        //actions
        ...
        release=srtGetRelease();
        srtSleep(SECONDS2TICKS(0.06),
                SECONDS2TICKS(0.06));
    }
}
```

A.2.5 Sleep task

- **Description**

Moves the current executing task to the SLEEP state with a given *release* and *deadline* (both relative).

- **Syntax**

```
void srtSleep (unsigned long release,
               unsigned long deadline)
```

- **Parameters**

- *release*: new task *release*
- *deadline*: new task *deadline*

- **Example**

```

#include "simplertk.h"

void myTask(void *args){
    while(1){
        //actions
        ...
        srtSleep(SECONDS2TICKS(0.06),
                SECONDS2TICKS(0.06));
    }
}

```

A.3 Semaphore management

Semaphore management primitives are:

- Semaphore creation.
- Semaphore *Wait*.
- Semaphore *Signal*.

For using these primitives the definition file *simplertk.h* must be included.

A.3.1 Semaphore creation

- **Description**

It creates a semaphore which is identified by its position `semnbr` in the kernel semaphore vector, and with a free resource number `initVal`.

- **Syntax**

```

void srtCreateSemaphore ( unsigned char semnbr,
                        unsigned char initVal)

```

- **Parameters**

- *semnbr*: the semaphore identifier is its position in the semaphore vector.
- *initVal*: semaphore counter initial value.

- **Example**

```

#include "simplertk.h"

unsigned char semVar1=1;
unsigned char semVar2=2;

int main(){
    srtCreateSemaphore(semVar1, 1);
    srtCreateSemaphore(semVar2, 1);
}

```

A.3.2 Semaphore *Wait*

- **Description**

It acquires a resource by decrementing the number of the `semnbr` semaphore free resources. If there is no free resource, the task is suspended until it is release by the task that had de resource.

- **Syntax**

```
void srtWait ( unsigned char semnbr )
```

- **Parameters**

– *semnbr*: semaphore identifier.

- **Example**

```

#include "simplertk.h"

unsigned char semVar=1;

void task(void *args){
    while(1){
        srtWait(semVar);
        ...
        srtSleep(SECONDS2TICKS(0.05),
                SECONDS2TICKS(0.05));
    }
}

int main(){
    srtCreateSemaphore(semVar, 1);
}

```

A.3.3 Semaphore *Signal*

- **Description**

It releases the acquired resource by incrementing the number of the `semnbr` free resources. If there are tasks waiting for the released resource, the highest priority blocked task becomes `READY`.

- **Syntax**

```
void srtSignal ( unsigned char semnbr )
```

- **Parameters**

– `semnbr`: semaphore identifier.

- **Example**

```
#include "simplertk.h"

unsigned char semVar=1;

void task(void *args){

    while(1){
        srtWait(semVar);
        ...
        srtSignal(semVar);
        srtSleep(SECONDS2TICKS(0.05),
                SECONDS2TICKS(0.05));
    }
}

int main(){
    srtCreateSemaphore(semVar, 1);
}
```

A.4 CAB management

The CAB (*Cyclical Asynchronous Buffers*) structure management primitives are:

- CAB creation.

- CAB deletion.
- *Buffer* reservation for writing.
- Update the most recent *buffer*.
- *Buffer* reservation for reading.
- Reading *buffer* release.

For using these primitives the definition file *cab.h* must be included.

A.4.1 CAB creation

- **Description**

It creates a CAB structure for intertask communication.

- **Syntax**

```
cab* srtOpencab (unsigned int num_buffers,
                unsigned int dim_buffers)
```

- **Parameters**

- *num_buffers*: *buffers* number
- *dim_buffers*: each *buffer* dimension, in *bytes*

- **Example of use**

A CAB with 2 *buffers* is created, each *buffer* with 4 data bytes.

```
#include "../cab.h"
#include "../simplertk.h"

cab *MyCab;

int main(){
    MyCab=srtOpencab(2,4);
    ...
}
```

A.4.2 CAB deletion

- **Description**

It deletes the CAB structure, freeing the used memory.

- **Syntax**

```
void srtDeletecab (cab *c)
```

- **Parameters**

– *c*: CAB pointer to be deleted.

- **Example of use**

```
#include "../cab.h"
#include "../simplertk.h"

cab *MyCab;

int main(){
    MyCab=opencab(2,4);
    ...
    srtDeletecab(MyCab);
}
```

A.4.3 *Buffer* reservation for writing

- **Description**

It returns the pointer to a free *buffer* for writing data.

- **Syntax**

```
pointer srtReserve (cab *c)
```

- **Parameters**

– *c*: pointer to CAB

- **Example of use**

```

void taskWriter(void *args){
    pointer pw;

    while(1){
        pw=srtReserve(MyCab);
        *pw->data='c';
        *(pw->data+1)='a';
        ...
        srtSleep(SECONDS2TICKS(0.05),
                SECONDS2TICKS(0.05));
    }
}

```

A.4.4 Update the most recent *buffer*

- **Description**

Once the data has been written in the reserved *buffer*, this primitives marks this buffer as the most recent *buffer*.

- **Syntax**

```
void srtPutmes (cab *c, pointer p)
```

- **Parameters**

- *c*: pointer to CAB
- *p*: pointer to the most recent *buffer*

- **Example of use**

```

void taskWriter(void *args){
    pointer pw;

    while(1){
        LATBbits.LATB14 ^= 1;
        pw=srtReserve(MyCab);
        *pw->data='c';
        *(pw->data+1)='a';
        srtPutmes(MyCab,pw);
        srtSleep(SECONDS2TICKS(0.05),
                SECONDS2TICKS(0.05));
    }
}

```

A.4.5 *Buffer* reservation for reading

- **Description**

It returns the pointer to the most recent *buffer* to access the written data, and increments its usage counter.

- **Syntax**

```
pointer srtGetmes (cab *c)
```

- **Parameters**

– *c*: pointer to CAB

- **Example of use**

```
void taskReader(void *args){
    pointer pread;
    while(1){
        pread=srtGetmes(MyCab); // data in
                                // pread->data
        ...
        srtSleep(SECONDS2TICKS(0.05),
                 SECONDS2TICKS(0.05));
    }
}
```

A.4.6 Reading *buffer* release

- **Description**

It decrements the *buffer* usage counter. If the counter is 0 (no other task is accessing the buffer), and if it no longer the most recent *buffer*, it becomes a free *buffer*.

- **Syntax**

```
void srtUnget (cab *c, pointer p)
```

- **Parameters**

– *c*: pointer to CAB

– *p*: pointer to the *buffer* to be released

- **Example of use**

```
void taskReader(void *args){
    pointer pread;
    while(1){
        pread=srtGetmes(MyCab); // data in
                                // pread->data
        srtUnget(MyCab,pread);
        srtSleep(SECONDS2TICKS(0.05),
                SECONDS2TICKS(0.05));
    }
}
```

B Source code

The SimpleRTK can be found in the Google Code SVN server at <http://simplertk.googlecode.com/svn/trunk/>.

B.1 simplertk.c and .h source code

The following listings show the **simplertk.c** and the **simplertk.h** code. They mainly include kernel, tasks and semaphores primitives.

Listing 6: simplertk.c code.

```
#include "simplertk.h"

_FOSCSEL(FNOSC_PRIPLL); // Primary (XT, HS, EC) Oscillator with PLL
_FOSC(OSCIOFNC_ON & POSCMD_XT); // OSC2 Pin Function: OSC2 is Clock Output - Primary Oscillator
_FWDT(FWDTEN.OFF); // Watchdog Timer Enabled/disabled by user software
_FGS(GCP.OFF); // Disable Code Protection

/***** KERNEL DATA STRUCTURES *****/
struct task {
    unsigned int *sp; // Stack pointer
    unsigned long release;
    unsigned long deadline;
    unsigned char state; // 0=terminated, 1=READY, 2=SLEEP
                        // 3=waiting for Sem1, 4=waiting for Sem2, etc.
};

struct kernel {
    unsigned char nbrOfTasks; // number of tasks created so far
    unsigned char running;
    struct task tasks[MAXNBRTASKS+1]; // +1 for the idle task
```

```

        unsigned char semaphores[MAXNBRSEMAPHORES]; // counters for semaphores
        unsigned int *memptr; // pointer to free memory
        unsigned long cycles; // number of major cycles since system start

        //unsigned long nextHit; // next kernel wake-up time
    } volatile kernel;

    static int current_cpu_ipl;

    void EnableInterrupts(){
        RESTORE_CPU_IPL(current_cpu_ipl);
        __asm__ volatile("disi #0x0000"); /* enable interrupts */
    }

    void DisableInterrupts(){
        SET_AND_SAVE_CPU_IPL(current_cpu_ipl, 7); /* disable level 7 interrupts */
        __asm__ volatile("disi #0x3FFF"); /* disable interrupts */
    }

    /*****
    static void restartCycle(void){
        unsigned char i;

        DisableInterrupts(); // turn off interrupts
        //tasks time shift
        for (i=1; i <= kernel.nbrOfTasks; i++) {
            if (kernel.tasks[i].state!=TERMINATED){
                kernel.tasks[i].release-=kernel.cycles;
                kernel.tasks[i].deadline-=kernel.cycles;
            }
        }
        kernel.cycles=0; // reset system clock
        EnableInterrupts(); // turn on interrupts
    }

    volatile unsigned int sptemp;

    char scheduler(){
        unsigned char running, oldrunning;
        struct task *t;
        unsigned char i;
        unsigned long now;
        unsigned long nextHit;

        //add cycle if timer value
        if ((IFS0bits.T11IF == 1) && (PRI==TIMER_VALUE)){
            if (kernel.cycles==0xFFFFFFFF) restartCycle();
            kernel.cycles++;
        }

        nextHit = 0x7FFFFFFF;
        oldrunning = kernel.running;

```

```

running = 0;

//Read clock
now=kernel.cycles;

//Release tasks from SLEEP and determine new running task

for (i=1; i <= kernel.nbrOfTasks; i++) {
    t = &kernel.tasks[i];
    if (t->state == SLEEP) {
        if (t->release <= now) {
            t->state = READY;
        } else if (t->release < nextHit) {
            nextHit = t->release;
        }
    }
    if (t->state == READY) {
        if (t->deadline < kernel.tasks[running].deadline) {
            running = i;
        }
    }
}

if (running != oldrunning) { // perform context switch?
    //store old context
    t = &kernel.tasks[oldrunning];
    t->sp=sptemp;

    kernel.running = running;
} else {
    running=NO_CONTEXT_SWITCH; // no context switch
}

return running;
}

void dispatch(void){
    struct task *t;
    t = &kernel.tasks[kernel.running];
    //task switch
    sptemp=t->sp;
}

```

/***/

```

void srtInitKernel(int idlestack){
    // Clock setup for 40MIPS
    CLKDIVbits.DOZEN = 0;
    CLKDIVbits.PLLPRE = 0;
    CLKDIVbits.PLLPOST = 0;
    PLLFBDbits.PLLDIV = 78;
}

```

```

    /* Wait for PLL to lock */
    while (OSCCONbits.LOCK!=1);

    kernel.memptr= (void*)(SPLIM - idlestack );
    kernel.nbrOfTasks= 0;
    kernel.running = 0;
    kernel.cycles = 0x00000000;

    //kernel.nextHit = 0x7FFFFFFF;

    kernel.tasks[0].sp=kernel.memptr;
    kernel.tasks[0].deadline = 0x7FFFFFFF;
    kernel.tasks[0].release = 0x00000000;

    /* Enable Timer1 Interrupt and Priority to "7" */
    ConfigIntTimer1(T1_INT_PRIOR_7 & T1_INT_ON);
    WriteTimer1(0);

    unsigned int timer_prescaler1;
    switch (PRESCALER){
    case 1:
        timer_prescaler1=T1_PS_1_1;
        break;
    case 8:
        timer_prescaler1=T1_PS_1_8;
        break;
    case 64:
        timer_prescaler1=T1_PS_1_64;
        break;
    case 256:
        timer_prescaler1=T1_PS_1_256;
    }
    OpenTimer1(T1_ON & T1_GATE_OFF & T1_IDLE_STOP &
    timer_prescaler1 & T1_SYNC_EXT_OFF &
    T1_SOURCE_INT, TIMER_VALUE);

    /* set pin (AN10/RB10)-->(CON6/Pin28) drive state low */
    LATBbits.LATB10 = 0;
    /* set pin (AN10/RB10)-->(CON6/Pin28) as output */
    TRISBbits.TRISB10 = 0;
}

extern void _T1Interrupt(void);

void srtCreateTask(void (*fun)(void*), unsigned int stacksize, unsigned long release, unsigned
    unsigned int *sp;
    struct task *t;
    int i;

    DisableInterrupts();// turn off interrupts

    ++kernel.nbrOfTasks;

```

```

kernel.memptr -= (stacksize+23); // decrease free mem ptr
sp = kernel.memptr;

*sp++ = fun; // store PC
*sp++ = 0x0000; // SR<7:0> IPL3,CORCON<3> PC<22:16>

*sp++=SPLIM; //SPLIM
*sp++=0x0000; //SR

*sp++ = args; // wo
*sp++ = 0x0000; // w1
// initialize stack
for (i=0;i<12;i++)
*sp++ = 0x0000; // w13-w2
*sp++ = kernel.memptr+24; // w14

*sp++=0x0000; //RCOUNT
*sp++=0x0000; //TBLPAG
*sp++=0x0024; //CORCON
*sp++=0x0000; //PSVPAG

t = &kernel.tasks[kernel.nbrOfTasks];
t->sp=sp; // store stack pointer
t->release = release;
t->deadline = deadline;
t->state = SLEEP;

EnableInterrupts();
_T1Interrupt(); // call interrupt handler to schedule
}

void srtCreateSemaphore(unsigned char semnbr, unsigned char initVal) {
    DisableInterrupts(); // turn off interrupts
    kernel.semaphores[semnbr-1] = initVal;
    EnableInterrupts(); // set enabled interrupts;
}

void srtWait(unsigned char semnbr) {
    struct task *t;
    unsigned char *s;

    t = &kernel.tasks[kernel.running];

```

```

    DisableInterrupts (); // disable interrupts

    s = &kernel.semaphores[semnbr-1];
    if ((*s) > 0) {
        (*s)--;
    } else {
        t->state = semnbr + WAIT_OFFSET; // waiting for Sem#semnbr
        _T1Interrupt (); // call interrupt handler to schedule
    }

    EnableInterrupts (); // reenable interrupts
}

void srtSignal(unsigned char semnbr) {

    unsigned char i;
    struct task *t;
    unsigned long minDeadline = 0xFFFFFFFF;
    unsigned char taskToReadyQ = 0;

    DisableInterrupts (); // disable interrupts

    for (i=1; i <= kernel.nbrOfTasks; i++) {
        t = &kernel.tasks[i];
        if (t->state == (semnbr + WAIT_OFFSET)) {
            if (t->deadline <= minDeadline) {
                taskToReadyQ = i;
                minDeadline = t->deadline;
            }
        }
    }

    if (taskToReadyQ == 0) {
        kernel.semaphores[semnbr-1]++;
    } else {
        kernel.tasks[taskToReadyQ].state = READY; // make task ready
        _T1Interrupt (); // call interrupt handler to schedule
    }

    EnableInterrupts (); // reenable interrupts
}

unsigned long srtCurrentTime(void) {
    return kernel.cycles;
}

void srtSleep(unsigned long release, unsigned long deadline) {

    struct task *t;
    unsigned long temp_res;
    t = &kernel.tasks[kernel.running];

    DisableInterrupts (); // turn off interrupts

```

```

        t->state = SLEEP;

        temp_res=t->deadline+deadline; //worst overflow case test
        if (SR&0x0004) restartCycle(); //status register overflow bit -> restart life counters
        t->deadline+= deadline;
        t->release+= release;

        EnableInterrupts ();
        _T1Interrupt (); // call interrupt handler to schedule
    }

    unsigned long srtGetRelease(void) {
        return kernel.tasks[kernel.running].release;
    }

    unsigned long srtGetDeadline(void) {
        return kernel.tasks[kernel.running].deadline;
    }

    void srtTerminate(void) {
        DisableInterrupts ();

        kernel.tasks[kernel.running].state = TERMINATED;
        EnableInterrupts ();
        _T1Interrupt (); // call interrupt handler to schedule
    }

```

Listing 7: simplertk.h code.

```

/*! \file simplertk.h
    \brief This file includes the Simple Real Time Kernel functions declarations.
    */

#ifndef _SIMPLERTK_
#define _SIMPLERTK_

#include <p33fj256mc710.h>
#include <timer.h>

/*!
    \def MAXNBRTASKS
    Max number of tasks
    */
#define MAXNBRTASKS 0
/*!
    \def MAXNRSEMAPHORES
    Max number of semaphores
    */

```

```

#define MAXNBRSEMAPHORES 0

/*
PRESCALER      PERIOD      TIMER VALUE
      1          1ms        ->    0x9C40
      8          10ms       ->    0xC350
      256        100ms      ->    0x3D09
      256        10ms       ->    0x061A
*/
#define PRESCALER 1
#define TIMER_VALUE 0x9C40
#define TICKSPERSECOND (40E6 / PRESCALER)/TIMER_VALUE

#define TERMINATED 0
#define READY 1
#define SLEEP 2
#define WAIT.OFFSET 2

/*!
\brief Change seconds to ticks
\param T seconds
*/
#define SECONDS2TICKS(T) ((unsigned long)((T)*TICKSPERSECOND))

#define RAMEND 0x7FFF

#define NO_CONTEXT_SWITCH 0x7F //No context switch
/***** KERNEL DATA STRUCTURES *****/
struct task;

struct kernel;

// Disable all interrupts
void DisableInterrupts();
// Enable all interrupts
void EnableInterrupts();

char scheduler();
/***** CLOCK INTERRUPT HANDLER *****/
void __attribute__((__interrupt__, __auto_psv__)) _T1Interrupt(void);

/***** API *****/
/*!
\brief Initialize the kernel
\param idlestack size of the idle stack
*/
void srtInitKernel(int idlestack);

/*!
\brief Create a task
\param fun task source code adress
\param stacksize task stack size for local variables (in words, word = 2 bytes)

```

```

    \param release task release (ticks)
    \param deadline task deadline (ticks)
    \param args task arguments adress
    */
    void srtCreateTask(void (*fun)(void*), unsigned int stacksize, unsigned long release, unsigned long deadline);

    /*!
    \brief Create a semaphore
    \param semnbr semaphore identifier
    \param initVal initial value
    */
    void srtCreateSemaphore(unsigned char semnbr, unsigned char initVal);

    /*!
    \brief Wait on a semaphore
    \param semnbr semaphore identifier
    */
    void srtWait(unsigned char semnbr);

    /*!
    \brief Signal a semaphore
    \param semnbr semaphore identifier
    */
    void srtSignal(unsigned char semnbr);

    /** Get the current system time (ticks) */
    unsigned long srtCurrentTime(void);

    /*!
    \brief Put a task to sleep until a certain time
    \param release the release time of the task
    \param deadline the deadline time of the task
    */
    void srtSleep(unsigned long release, unsigned long deadline);

    /** Get the release time (ticks) of the running task */
    unsigned long srtGetRelease(void);

    /** Get the deadline (ticks) of the running task */
    unsigned long srtGetDeadline(void);

    /** Terminate the execution of the current task */
    void srtTerminate(void);

#endif

```

B.2 swctx.s source code

The following listing shows the **swctx.s** source code. It contains the interrupt service routine that is periodically generated by timer 1 and gives the kernel timing.

Listing 8: swctx.s code.

```
.extern _sptemp
.text
    .global __T1Interrupt
__T1Interrupt:
    ; bset LATB,#10          ; debug timing
    ;; save context

    push    _SPLIM
    push    _SR

    push.d  W0
    push.d  W2
    push.d  W4
    push.d  W6
    push.d  W8
    push.d  W10
    push.d  W12
    push    W14

    push    _RCOUNT
    push    _TBLPAG
    push    _CORCON
    push    _PSVPAG

    disi    #2
    disi    #0x3FFF          ; disable interrupts

    ;; services timer 1 interrupts
    mov W15, _sptemp
    call _scheduler
    bclr IFS0,#3            ; clear the interrupt flag
    mov #127,W1
    cp w0,w1

    bra Z, __noswctx
    call _dispatch
    mov _sptemp,W15

__noswctx:
    bclr IFS0,#3            ; clear the interrupt flag

    ;; load context
```

```

pop     _PSVPAG
pop     _CORCON
pop     _TBLPAG
pop     _RCOUNT

pop     W14
pop.d   W12
pop.d   W10
pop.d   W8
pop.d   W6
pop.d   W4
pop.d   W2
pop.d   W0

pop     _SR
pop     _SPLIM

disi    #2
disi    #2

; bclr  LATB,#10                ; debug timing

retfie                ; and return from interrupt
.end

```

B.3 cab.c and .h source code

The following listings show the **cab.c** and the **cab.h** source code. It contains the primitives related to the CAB management.

Listing 9: cab.c code.

```

#include "cab.h"
#include "simplertk.h"
#include <stdlib.h>

cab *srtOpencab(unsigned int num_buffers, unsigned int dim_buffers){
    cab *c=malloc(sizeof(cab));
    c->buffers=malloc(sizeof(buffer *)*num_buffers);

    unsigned int i;
    for(i=0;i<num_buffers;i++){
        c->buffers[i]=malloc(sizeof(buffer));
        c->buffers[i]->data=malloc(dim_buffers);
        c->buffers[i]->use=0;
        if(i<num_buffers-1) c->buffers[i]->next=c->buffers[i+1];
    }
}

```

```

        else c->buffers [i]->next=NIL;
    }

    c->free=c->buffers [0];
    c->mrbs=c->buffers [ num_buffers -1];
    c->max_buf=num_buffers;

    return c;
}

void srtDeletcab (cab *c){
    unsigned int i;
    // release memory of the cab
    for (i=0;i<c->max_buf; i++){
        free (c->buffers [i]->data);
        free (c->buffers [i]);
    }
    free (c);
}

pointer srtReserve (cab *c){
    pointer p;
    DisableInterrupts ();
    p=c->free; //returns the free buffer
    EnableInterrupts ();
    return p;
}

void srtPutmes (cab *c, pointer p){
    DisableInterrupts ();
    //update the free buffer
    if (c->mrbs->use == 0){
        c->mrbs->next = c->free;
        c->free = c->mrbs;
    }
    //update de mrbs buffer
    c->mrbs=p;
    EnableInterrupts ();
}

pointer srtGetmes (cab *c){
    pointer p;
    DisableInterrupts ();
    //return the mrbs buffer for use it
    p=c->mrbs;
    p->use++;
    EnableInterrupts ();
    return p;
}

void srtUnget (cab *c, pointer p){
    DisableInterrupts ();
    //release the buffer
    p->use--;
    //if it is the last using the mrbs buffer and
    // there is new one -> updates the free buffer

```

```

        if ((p->use == 0) && (p!= c->mrbs)){
            p->next=c->free;
            c->free= p;
        }
    EnableInterrupts ();
}

```

Listing 10: cab.h code.

```

/*! \file cab.h
   \brief This file includes the Simple Real Time Kernel CAB communication mechanism.
   */

#ifndef _CAB_
#define _CAB_

#define NIL 0

typedef struct buffer buffer;
typedef struct cab cab;
typedef struct buffer *pointer;

/*!
   \struct buffer
   \brief Buffer of data
   */
struct buffer{
    pointer next; /**< pointer to next buffer */
    unsigned int use; /**< number of readers reading this buffer */
    char *data; /**< pointer to the data */
};

/*!
   \struct cab
   \brief CAB Structure (Cyclical Asynchronous Buffers)
   */
struct cab{
    buffer **buffers; /**< pointers to buffers */
    pointer free; /**< free buffer */
    pointer mrbs; /**< most recent buffer */
    unsigned int max_buf; /**< maximum buffers */
    unsigned int dim_buf; /**< buffers dimension */
};

/*!
   \brief Creates a CAB structure.
   \param num_buffers number of buffers
   \param dim_buffers dimension of each data buffer
   \return Pointer to the new cab
   */
cab *srtOpencab(unsigned int num_buffers, unsigned int dim_buffers);

```

```

/*!
\brief Delete a CAB
\param c CAB pointer to be deleted
*/
void srtDeletcab(cab *c);

/*!
\brief Reserve a free buffer to write data
\param c CAB pointer to reserve the buffer
\return Pointer to the buffer reserved
*/
pointer srtReserve(cab *c);

/*!
\brief Update the most recent buffer
\param c CAB pointer destiny
\param p Pointer to the new most recent buffer
*/
void srtPutmes(cab *c, pointer p);

/*!
\brief Reserve the most recent buffer to read data
\param c CAB pointer
\return Pointer to the most recent buffer
*/
pointer srtGetmes(cab *c);

/*!
\brief Release the readed buffer
\param c CAB pointer
\param p Pointer to the buffer to be released
*/
void srtUnget(cab *c, pointer p);

#endif

```

C Kernel compilation basic instructions

This section describes how to compile the SimpleRTK together with a simple application. The code of the application **myappl.c** is shown in listing 11. It consists of a *main* that initializes the orange led of the Flex board, the kernel, and creates the periodic task *mytask*. Task *mytask* turns ON and OFF the orange led every second. Hence, one single periodic task has been created. In addition, in **simplertk.h**, the number of tasks must be specified, that is `#define MAXNBRTASKS 1`.

Listing 11: myapp.c code.

```

#include "simplertk.h"

unsigned int led_status = 0;

void mytask(void *args){
    while(1){
        if (led_status==0) {
            /* if the led is on, turn it off */
            LATBbits.LATB14=1;
            led_status = 1;
        } else {
            /* if the led is off, turn it on */
            LATBbits.LATB14=0;
            led_status = 0;
        }
        srtSleep(
            SECONDS2TICKS(1), //new release
            SECONDS2TICKS(1)); //new deadline
    }
}

int main(void){
    LATBbits.LATB14=0; /* Configure Flex orange led */
    TRISBbits.TRISB14=0;

    srtInitKernel(80);
    /* periodic task with period = 1s */
    srtCreateTask(
        mytask, //task pointer
        100, //task local data size
        SECONDS2TICKS(1), //release
        SECONDS2TICKS(1), //deadline
        0);

    //idle loop
    while(1);
}

```

Using MPLAB as a cross-compilation platform (and ICD2 as a in-circuit debugger/programmer), the following steps must be followed. A project must be created, e.g., *myproject*, including the required kernel files (**simplertk.c**, **simplertk.h**, and **swctx.s**) and the application **myappl.c**, as shown in Figure 10. Files **cab.c** and **cab.h** have not been included because CABs are not used.

If CABs were used for intertask communication, the linker should be configured to specify the *Heap* memory, which is a memory space reserved

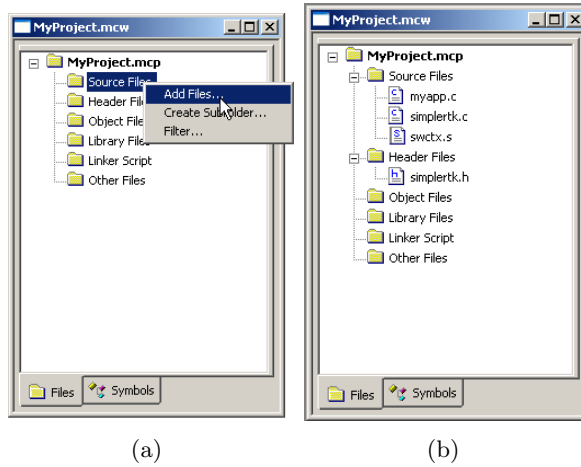


Figure 10: MPLAB *Project*.

for dynamic memory. As outlined in Section 6, a CAB size is given by $10 + 2 \times \text{max_buf} + \text{max_buf} \times (6 + \text{dim_buf})$. To reserve this memory size, the parameter `--heap=mem` must be added to the *linker*, where `mem` is the required number of bytes. In MPLAB, this can be specified in the project build options (`Project`→`Build options`→`Project`), in the MPLAB LINK30 entry, where the number of bytes must be specified, as shown in Figure 11.

With or without CABs, the project must be compiled. Figure 12 shows the compilation mode (release or debugger)² for compiling. The *release* mode is for the final application.

Once the project has been compiled, the output window should state BUILD SUCCEEDED. Then, the project can be downloaded to the dsPIC and executed. The Flex orange led should go ON and OFF every second.

²If *release* is chosen, the ICD2 must be selected as a programmer. If *debugger* is chosen, the ICD2 must be selected as a debugger.

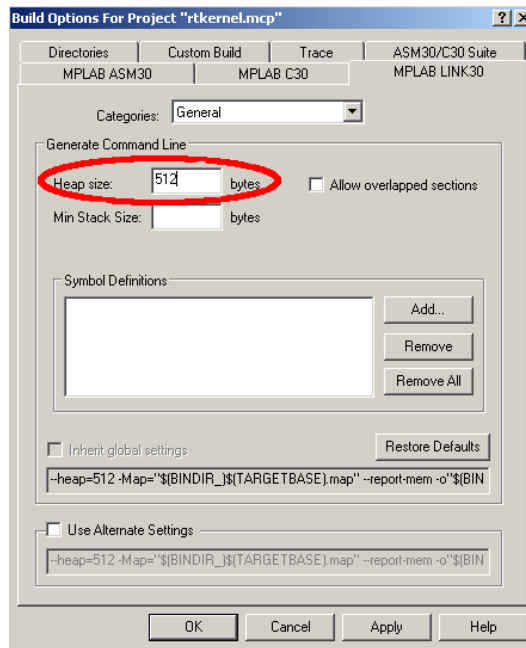


Figure 11: *Heap* memory reservation.

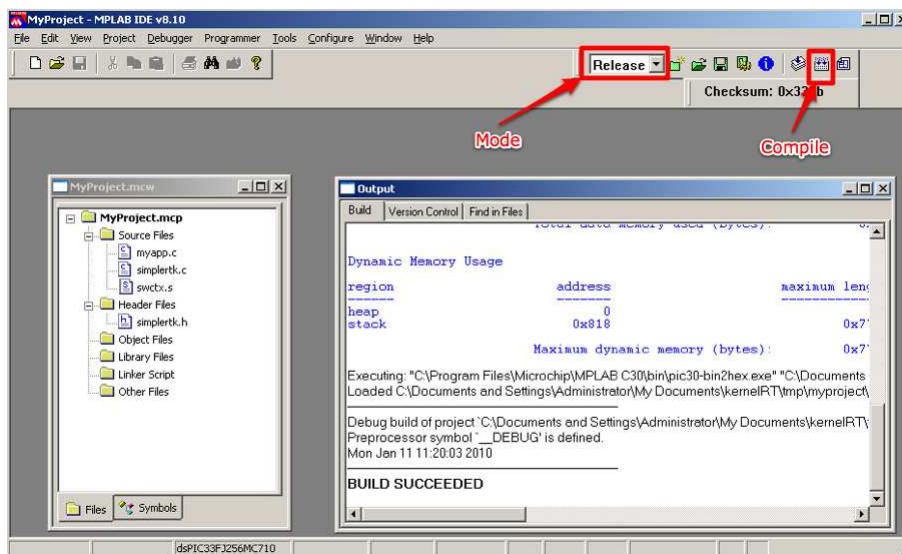


Figure 12: Project compilation.