Programming Real-Time Embedded systems: baremetal on GR740 boards

Frank Singhoff

Bureau C-203

University of Brest, France

Lab-STICC UMR CNRS 6285

singhoff@univ-brest.fr

Summary

Introduction

- **Properties/constraints of embedded critical real-time systems:**

1. As any real-time systems: functions and timing behavior must be predictable.
2. Extra requirements or constraints:
   - Limited resources: memory footprint, power, ...
   - Reduced accessibility for programmers.
   - High level of autonomy (predictability).
   - Interact with their environment, with sensors/actuators (predictability).

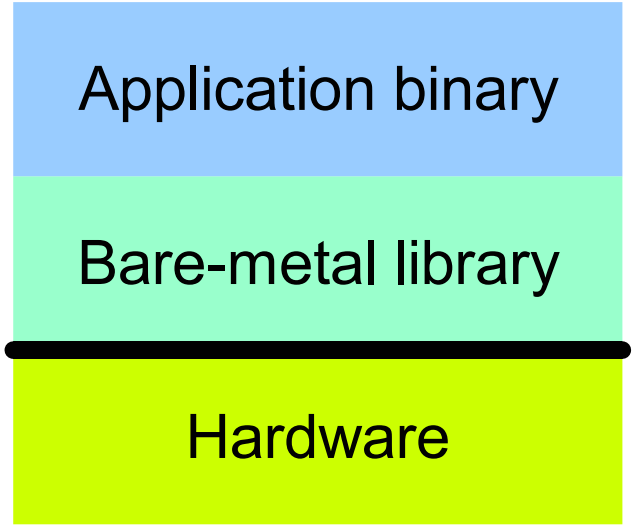
- **Various kinds of execution platforms.**

Introduction

- **Trends in embedded critical real-time systems:**
  - Reduce costs.
  - Use Commercial-off-the-Shelf components (COTS) : hardware (multicore, not radiation hardened processors), operating systems (e.g. Linux RT).
  - Embed complex payloads (e.g. image processing, Artificial Intelligence algorithms).
  - Optimize SWAP.
  - Rethink/review software isolation. Mixed criticality.
  - But keep predictability for critical software.
- **Strong changes on execution platforms.**

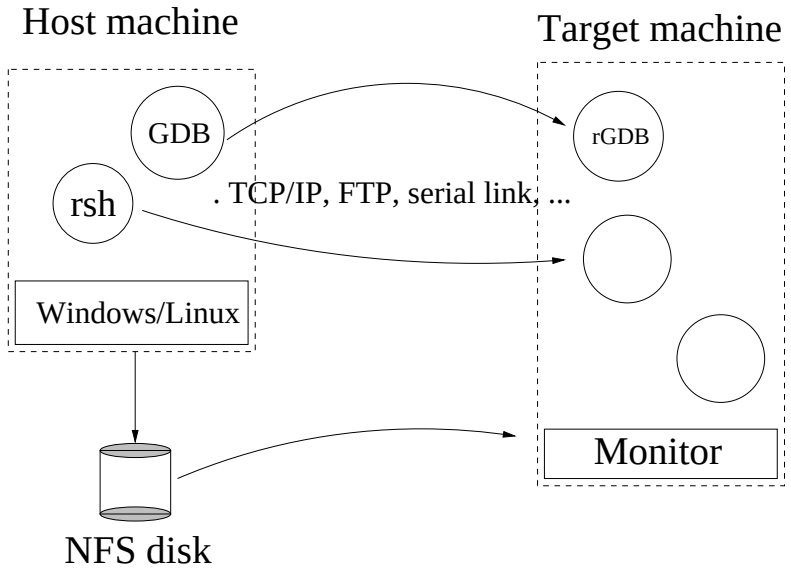
Summary

A baremetal platform (1)



- **Bare-metal**
  - Highest level of predictability. Full access to hardware.
  - No operating system.
  - No memory protection : system and application are linked together
  - High development cost if system re-design.
  - Think about a library linked with your application.

A baremetal platform (2)



- **Cross-compiling**: because targets have a limited amount of resource (configurability) and are composed of specific hardware/software (timing behavior).
- **Host:** where we compile the program.
- **Target:** where we run the program.

A baremetal platform (3)

- **BCC (Baremetal cross-compiler):**
  - BCC is a cross-compiler for LEON2, LEON3 and LEON4 processors. C and C++ applications.
  - No operating system: and then no usual system concepts: no task, no semaphore nor mutex, no memory protection, no scheduling, ...
  - Basic AMP support (e.g. processor atomic statements). SMP is not supported.
  - A library: *libbcc*
  - ...

A baremetal platform (4)

- **BCC contains:**
  - C standard library with full math support.
  - Timer management, console management.
  - Drivers.
  - About 50 functions for direct (not through system abstraction) control hardware resources: cache units, interrupts, processor, fpu, trap, memory, DMA, bus, ...
  - Several BSP for Sparc processors.
  - Hooks called at starting time to do initilizations: e.g. __bcc_init_170

A baremetal platform (5)

$bcc\_timer\_tick\_init\_period$	Install a clock for the system clock
$bcc\_timer\_get\_us$	Read system clock
$usec\_per\_tick$	Read clock period

A baremetal platform (6)

bcc_flush_cache	Flush L1 icache and dcache
bcc_flush_dcache	Flush L1 dcache
bcc_flush_icache	Flush L1 icache
bcc_get_ccr	Read cache control register

Summary

1. Introduction
2. A baremetal platform
3. Example
4. Offline scheduling
5. Summary

BCC example (1)

```
#include <bcc/bcc.h>
#include <bcc/bcc_param.h>
#include <stdio.h>
#include <stdlib.h>

void __bcc_init70(void) {
  int ret;
  /* 1 tick every 100 us */
  ret = bcc_timer_tick_init_period(100);
  if (BCC_OK != ret) {
    exit(EXIT_FAILURE);
  }
}

int main(void) {
  printf("Hello\n");
  return EXIT_SUCCESS;
}
```

BCC example (2)

- $main$: single entry point. SINGLE FLOW of CONTROL.
- $\_\_bcc\_init70$: do initializations before starting of the program. Initially, there is a limited clock service.
- $return()$: stops the application. We can switch off the board!

BCC example (3)

- **Cross compiling**

1. Compile on Linux and generate a SPARC binary:

```
#make
sparc-gaisler-elf-gcc  -qbsp=leon3 -O2 -g -o hello.c
sparc-gaisler-elf-gcc  -qbsp=leon3 hello.c -o init.exe
sparc-gaisler-elf-size init.exe
   text    data     bss filename
  27120    1176     744 init.exe
```

BCC example (4)

- **Cross-compiling (cont)**

2. Send the binary to the Board/Leon processor (TCP/IP, serial link, ...).
3. Run the program on the board/Leon processor. Software emulator `tsim` (Leon 3 processor simulator).

```
#tsim init.exe
 TSIM/LEON3 SPARC simulator, version 2.0.15 (evaluation version)
allocated 4096 K RAM memory, in 1 bank(s)
allocated 32 M SDRAM memory, in 1 bank
allocated 2048 K ROM memory
read 2257 symbols
tsim> run
resuming at 0x40000000
Hello
Program exited normally on  CPU 0.
tsim> quit
```

BCC example (5)

- **Compare the same program with RTEMS and BCC:**

```
sparc-gaisler-elf-gcc  -qbsp=leon3 -O2 -g -o hello.c
sparc-gaisler-elf-gcc  -qbsp=leon3 hello.c -o init.exe
sparc-gaisler-elf-size init.exe
   text    data     bss filename
  27120    1176     744 init.exe

sparc-gaisler-elf-size hello.o
   text    data     bss  filename
     34       0       0  hello.o

sparc-gaisler-rtems5-gcc  -qbsp=leon3 -O2 -g -o hello.c
sparc-gaisler-rtems5-gcc  -qbsp=leon3 hello.c -o init.exe
sparc-gaisler-rtems5-size init.exe
   text    data     bss  filename
 139376    4832   21920  init.exe
```

BCC example (6)

- **Development process**

1. Write code and test on Linux/Windows (functionnal part).
2. Write code and simulate its behavior on a real-time simulator (e.g. tsim simulator)
3. Verify behavior/performances on real boards/real-time boards (grmon tool, e.g. GR712RC, GR740).
4. Flash binaries into the board (mkprom tool).

Summary

1. Introduction
2. A baremetal platform
3. Example
4. Offline scheduling
5. Summary

Offline scheduling (1)

- **Online vs offline:**
  - Online scheduling: order of task execution is deciding during runtime. Adapted to unplanned event. Higher schedulablity but lower predictability.
  - Offline scheduling: order ot task execution is fully knwon prior execution. Cannot adapt scheduling to unexpected event. Lower schedulablity by high predictability.

Offline scheduling (2)

- **Preemptive vs nonpreemptive:**
  - Non preemptive: cannot stop a task during its execution. No need to task synchronization tools (semaphore), ease communication. Safer from a concurrency point of view.
  - Preemptive: allow to stop a tasj during its execution. Lower latency, i.e. better adapted to urgent event support. Better schedulability. Make programming concurrent program difficult. Overhead due to preemptivity.

Offline scheduling (3)

- **Dispatcher is a program that drives an offline scheduling:**
  - Schedule of the functions of the system is stored in a table
  - Dispatcher reads the table sequentially and actives functions as expressed in the table
  - How to build the table ? i.e. called scheduling table.
  - Fully predictable: no uncertaincy due to the scheduler.
  - Lack of flexibility: 1) cannot adapt behavior in case of unexpected event 2) need a re-design of the scheduling if features are added/removed/changed.
  - Ease testing/analysis/certification: 1) no need to certify the OS 2) the less complex the execution platform is, the easier the analysis is 3) no task means no race condition.

Offline scheduling (4)

- Interface example of a dispatcher:

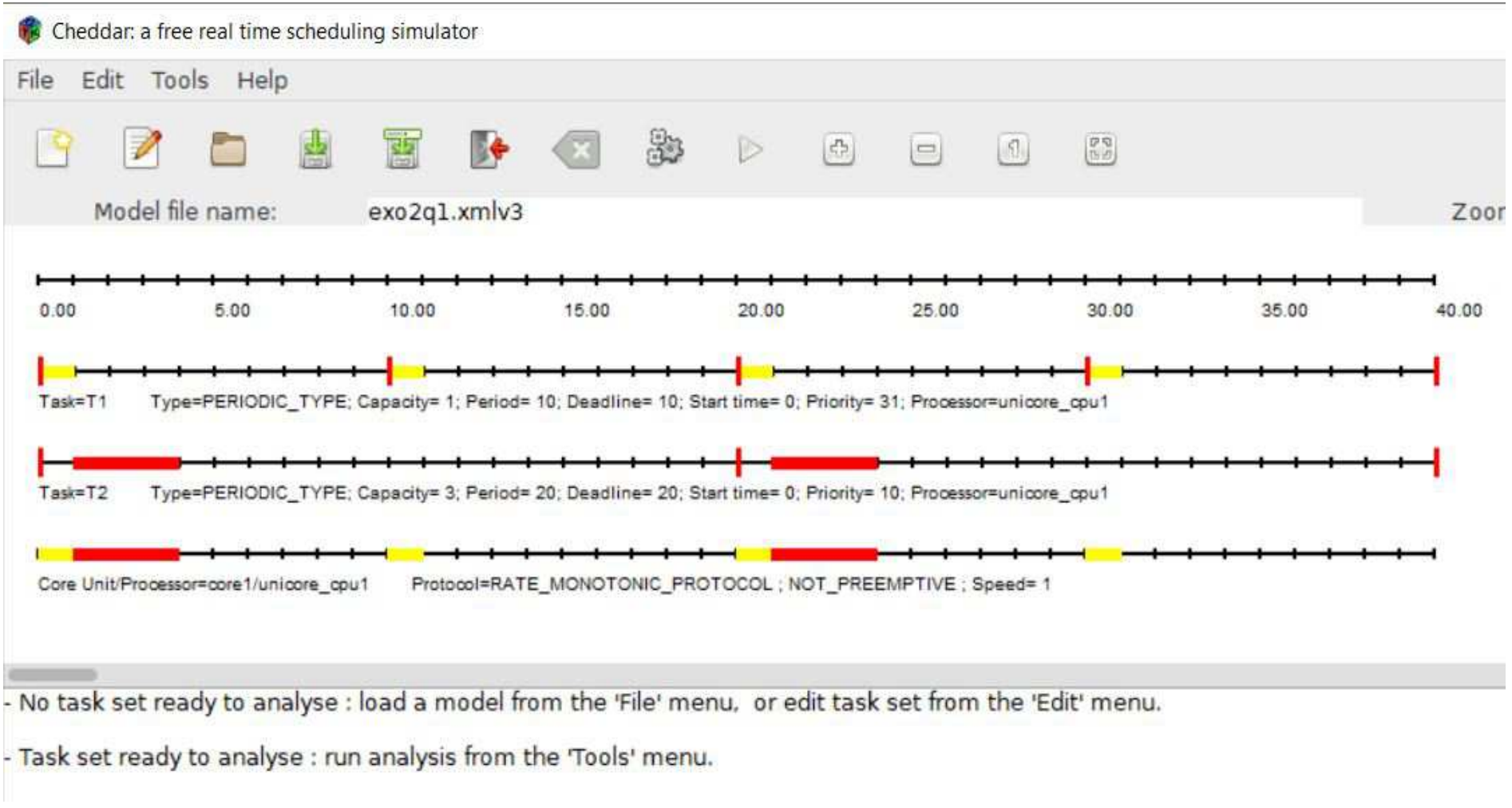
```
typedef void* (*function_type) (void*);
typedef struct nonpreemptive_entry_t {
  uint32_t duration;
  function_type entrypoint;
  void* arg;
} nonpreemptive_entry_t;

void nonpreemptive_active_wait(uint32_t duration);
void nonpreemptive_init(void);
void nonpreemptive_add_function(nonpreemptive_entry_t entry);
void nonpreemptive_add_array_function(
    nonpreemptive_entry_t entries [], int size);
void nonpreemptive_start(int iteration);
void nonpreemptive_stop(void);
```

Offline scheduling (5)

- $nonpreemptive\_init$: perform all initializations needed by the dispatcher. Must be called before any interaction with the dispatcher.
- $nonpreemptive\_active\_wait$: produce an active wait for the current function of at least $duration$ microseconds.
- $nonpreemptive\_add\_function$: add one function in the scheduling table.
- $nonpreemptive\_start$: start to run functions declared in the scheduling table. Functions are run according to their order in the scheduling table.

Offline scheduling (6)

- Example of use:



Offline scheduling (7)

- Example of use:

```
void* T1(void* arg) ...
void* T2(void* arg) ...

int main(void) {

  nonpreemptive_entry_t entries [3]
        = { {1000,T1,NULL}, {9000,T2,NULL}, {10000,T1,NULL} };

  nonpreemptive_init();
  nonpreemptive_add_array_function(entries, 3);
  nonpreemptive_start(2);

  return EXIT_SUCCESS;
}
```

Summary

1. Introduction
2. A baremetal platform
3. Example
4. Offline scheduling
5. Summary

Summary

- **Baremetal means:**

1. Low memory footprint and high predictability
2. Low complexity => ease static analysis/test/certification
3. Adapted to high criticality systems
4. High cost of development.