ChibiOS-EmbrIO

Embryo Virtual Machines running on Chibi RTOS

Caccia Claudio
Matr. 751302, (claudiogiovanni.caccia@mail.polimi.it)

Report for the master course of Embedded Systems
Reviser: ing. Martino Migliavacca (martino.migliavacca@gmail.com)

Received: September, 16 2012

Abstract

The report describes the implementation of Embryo virtual machines on a Cortex™-M3 ARM processor. Embryo is a library designed to interpret a subset of programs coded in a C-like syntax language known as *Small*. The library is tiny enough to be used on embedded systems with a reduced amount of memory. The idea behind this work is to make it possible to load, execute, unload and exchange different programs on an embedded system in a way that can be roughly compared to the one that we normally use on personal computers, laptops or in complex embedded systems like smartphones. A feature like this can make a small system highly flexible, allowing the possibility to change its behavior at runtime, without the need to load the entire code and reboot.
In order to achieve this goal, the Embryo library has been modified to be used on the embedded system that we used for our tests, and it has then been integrated with a very performing RTOS like ChibiOS/RT. Some programs have then been compiled, loaded and executed on a demo-board.

1 Introduction

When we think about Embedded Systems we can divide them into two main categories: on one side we have complex systems, like latest generation smartphones, where we can have a huge amount of memory, multi-core processors, multiple interfaces and where we can download and use a lot of different applications and then cancel them without the need of switching the system off. On the other side we have small systems, like micro-controllers, which perform just one or few tasks for their entire life. We can just change some parameters only if we have implemented some callbacks on the system in order to update the values at runtime by using some communication interface (e.g. Serial, Fieldbus...). We cannot change the overall behavior of the system. For example, if we have implemented a PID controller, we can modify its gains, but if we need to change the type of controller we need to code a new one, compile and then load it on the processor (e.g. by means of a flash programmer or by using a bootloader).

In some environments it can be particularly interesting and time saving to change the behavior of the system at runtime, e.g. in robotics, in particular in cooperative robotics, it could be possible to change the way a single component acts without the need to keep all the possible configurations on board, or in artificial vision, it could be possible to change classifiers or algorithms, test and use them in a simple way.

So we investigated the possibility to implement a system based on virtual machines, so that one could load, execute and unload different components by selecting one or more program and launch it on a Virtual Machine managed by an underlying Operative System.

We needed a virtual machine small enough to be executed on an embedded system and the choice fell on *Embryo*. At the same time we needed all the features that a RTOS can provide (e.g. task scheduling, IPC, memory management, HAL...) and we selected *ChibiOS/RT*. Finally we needed an environment to implement and test the first basic features that we coded: for this purpose we selected a demo-board based on *ARM Cortex™-M3* processor.
In the following sections we describe in further detail these components.

1.1 Embryo

Embryo implements a scripting language used by parts of the Enlightenment Foundation Libraries (EFL) [1]. EFL is a set of open source graphical software libraries derived from the Enlightenment window manager project [2]. The libraries are meant to be portable and optimized to be functional even on devices like PDAs, and are used in many different applications.
The EFL structure is widely described in [3], while **Figure 1** illustrates all of the components and how they are

related.
Embryo [4] is a tiny library designed to interpret limited *Small* programs compiled with *embryo_cc*, the included compiler.

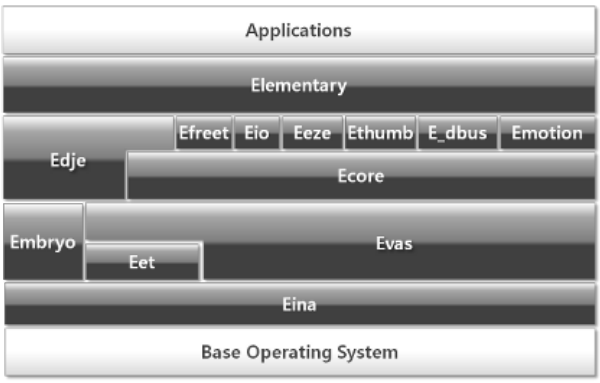

Figure 1: EFL structure.

The *Small* language, renamed Pawn [5], is an open source scripting language intended to be embeddable. It is a simple, typeless, 32-bit extension language with a C-like syntax. A source program is compiled to a binary file for optimal execution speed. The compiler outputs a *P-code* (or bytecode) that subsequently runs on an abstract machine.
Embryo implements a subset of the *Pawn/Small* features as described in [6]. In its most recent version (1.7.0), Embryo depends on another EFL library, named Eina [7]. Eina is a library that implements APIs for data types and provides some tools like opening shared libraries, errors management, type conversion and memory management. Although some of these features can be useful, we preferred to rely on an older version of Embryo (1.1.0) which is not depending on any other library, neither for the compiler, nor for the abstract machine.

1.2 ChibiOS/RT

ChibiOS/RT is a RTOS designed for embedded applications on 8, 16 and 32 bits micro-controllers, focused on size and execution efficiency [8]. The kernel size can range from a minimum of 1.2Kbytes up to a maximum of 5.5Kbytes on a STM32 Cortex^{TM}-M3 processor. The kernel is able to perform a Context Switch in 1.2 microseconds on a 72 MHz STM32.
ChibiOS/RT has also a lot features [9] that simplify application development on embedded systems:

- Efficient and portable preemptive kernel,
- Best in class context switch performance.
- Static architecture, everything is statically allocated at compile time.
- Dynamic extensions, dynamic objects are supported by an optional layer built on top of the static core.
- Rich set of primitives: threads, virtual timers, semaphores, mutexes, condition variables, messages, mailboxes, event flags, queues.
- HAL component supporting a variety of abstract device drivers

The current stable release of ChibiOS/RT is 2.4.2 and we worked on this one.

1.3 Demoboard

We decided to work with an ARM Cortex^{TM}-M3 processor [10], for its performance and for its wide range of interfaces and because it is one of the most used in industry applications. We selected a demoboard (see **Figure 2**) produced by Olimex [11], based on a *STM32F107* processor [12]. Among other features, we mainly used the following components:

- 256 Kbyte Flash, 64 Kbyte RAM
- 10 timers
- 2 ADCs
- JTAG connector with ARM 2x10 pin layout for programming/debugging
- RS232
- 2 user buttons
- 2 status led

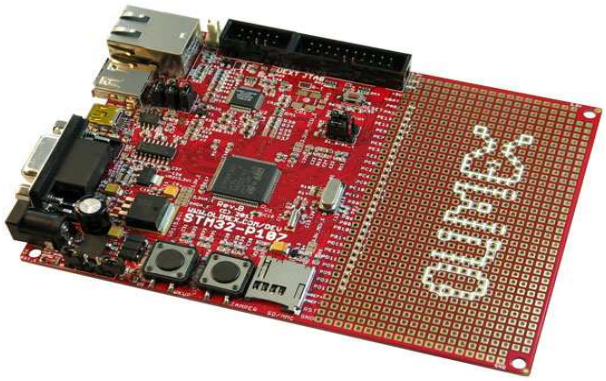

Figure 2: Olimex STM32P107 demoboard.

1.4 Development and testing equipment

The code has been written using the *Eclipse CDT* IDE [13], and has been compiled using the CodeSourcery ARM EABI compiler, maintained by Mentor Graphics [14].
In order to run and debug the program, we used the JTAG interface produced by Olimex (see **Figure 3**), which allows debug operations via ARM JTAG interface and communication via an RS232 Serial interface.

Figure 3: Olimex JTAG interface.

The setup operations followed instructions found here [15] and here [16].

2 Development

In this section we describe the main phases of the development of the project.

2.1 Project Structure

The project structure is the following:

```
/ ..... git root
├── ChibiOS-EmbrIO
├── embryo
│   ├── embryo-cc ..... compiler
│   ├── embryo-lib ..... abstract machine
│   └── embryo-sim ..... simulator
├── report
└── ChibiOS/RT
```

The project is managed by using a Git revision control system and it is hosted on `http://code.google.com/p/chibios-embrio/`. ChibiOS/RT is mainly untouched and is out of revision control. The compiler and the abstract machine derive from the Enlightenment code, while the simulator is a really small tool developed in order to test the basic functionalities of the abstract machine. The original Embryo project is managed by the *GNU Automake* tool. In order to use different compilation parameters (the abstract machine can be compiled for ARM but also for a host pc), a new set of `Makefiles` has been generated.

2.2 Embryo Compiler

The compiler underwent the least modifications. None of the components that build the bytecode has been touched. The original code is very Linux-oriented, looking for default paths like `/usr/bin` or into the `/proc` filesystem.

As some of the development has been carried on in a Windows environment, the whole setup of the paths has been simplified. The Embryo Abstract Machine allows the user to define his own *native functions*, which extend the basic functionalities of a program and allow to reduce the code size. The Embryo compiler just needs to know the signature of the native functions; the user can include any `*.inc` files with the functions signatures. The user needs only to indicate the directories where the compiler must search for the files. The source code is very similar to C:

```
/* files with native functions to include */
#include <printXXX>
#include <embrio01>
```

Listing 1: include directives in Embryo

We encountered some difficulties when looking for `*.inc` files and so we slightly modified the way Embryo compiler looks into the directories where the files are.
Finally we changed the command line arguments used to call `embryo_cc`, using the conventions of `getoptlong.h`.

```
Usage:   embryo_cc -c <filename> [options]

Options:
        -i (--include) <name> path of include
            files
        -o (--output) <name> base name of
            output file
        -S (--size) <num> stack/heap size in
            cells (default=4096, min=65)

        -h (--help) Display usage information
```

Listing 2: Command line arguments

2.3 Embryo Abstract Machine

The Embryo Abstract Machine needed modifications in two areas: definition of *native functions* and *memory management*. Embryo gives the possibility to define one's own functions and so we used it to call ChibiOS/RT services within the Embryo environment. As we wanted to test the Abstract Machine also on a pc, we used the `#define` directive to select the compilation environment.

2.3.1 Native Functions

```
Embryo_Cell
_embryo_sleep(Embryo_Program *ep, Embryo_Cell *
    params)
{
#ifdef _CHIBIOS_VM_
  chThdSleepMilliseconds(1000);
#else
  sleep(1);
#endif

  return 0;
}
```

Listing 3: Native function *sleep*

The code above shows that under _CHIBIOS_VM_ we can call ChibiOS/RT functions.
The following code shows the usage of the board serial port (called SD3 in our environment), so that a string inside an Embryo VM can be directly displayed on a log terminal. While testing the functionalities, we simply print on stdout.

```
Embryo_Cell
_embryo_print(Embryo_Program *ep, Embryo_Cell *
    params)
{
  char *s;

  /* params[1] = str */
  if (params[0] != (1 * sizeof(Embryo_Cell))){
    return -1;
  }

  STRGET(ep, s, params[1]);
#ifdef _CHIBIOS_VM_
  chprintf((BaseChannel*)&SD3, "%s",s);
#else
  printf("Stringa: %s\n", s);
#endif
  return 0;
}
```

Listing 4: Native function *print*

2.3.2 Memory Management

ChibiOS/RT has a static kernel, but offers different ways to manage memory for the applications [17]:

- Core Memory Manager
- Heap Allocator
- Memory Pools

The *core memory manager* is a simplified allocator that only allows to allocate memory blocks without the possibility to free them, it is meant to be a memory blocks provider for the other allocators.
The *heap allocator* implements a
first-fit strategy to allocate chunks of memory. It is subject to fragmentation, and the allocation time is not constant. Its APIs are functionally equivalent to the usual C malloc() and free().
Memory pools allow to allocate and free fixed size objects, in constant time and reliably (no fragmentation problems, thread safe...). Memory pools can grow in size if a suitable memory provider has been defined. As we want the possibility to load, unload and change Virtual Machines at execution time, flexibility plays a key role in memory management. We needed to analyze how and where the Embryo Abstract Machine allocates and releases memory and translate the operations into the best suitable ChibiOS/RT APIs.
We defined:

- an EmbrIO Memory Heap
- a set of Memory Pools

We used the Heap for all the elements that are of unknown size at compile time (e.g. bytecode) while we defined different Memory Pools for every set of objects that can be allocated and freed. We defined the maximum number of Virtual Machines that can be run at the same time and allocated the corresponding pools (e.g. the Embryo_Program structure, the Thread running the VM, the VM itself).

```
// define the max number of virtual machines and
    programs
#define MAX_EMBRIO_VM_NUM 4
...
{
  /* init MP for thread used for VMs */
  chPoolInit( &THD_mp, THD_WA_SIZE(
      THD_STACK_SIZE), NULL);
...
  /* preallocate */
  for (i = 0; i < MAX_EMBRIO_VM_NUM; ++i) {
    chPoolFree(&THD_mp, thd_vm_wa_p[i]);
  }
...
  /* allocate a MP element to a Thread */
  thdp = chThdCreateFromMemoryPool( &THD_mp,
      prio, vm_thread, (void *)vm);

}
```

Listing 5: Memory Pools

2.4 Embryo Simulator

A new component, although very simple, has been added to the original structure of the Embryo compiler and abstract machine, in order to test on a normal computer the basic functionalities. The abstract machine can be compiled to become a static library that can be linked to a testing program. The basic operations needed to use an Embryo program are described in [4].

```
int main(int argc,char *argv[]) {
  Embryo_Program* ep;
  Embryo_Status es;

  /* init Embryo */
  embryo_init();
  /* load program */
  ep = embryo_program_load(argv[1]);
  /* Starts a new VM session */
  embryo_program_vm_push(ep);
  /* run program from main function */
  es = embryo_program_run(ep,
      EMBRYO_FUNCTION_MAIN);
  /* at the end stop VM and free memory */
  embryo_program_free(ep);
  /* close Embryo */
  embryo_shutdown();
  return 0;
}
```

Listing 6: Embryo basic simulator

2.5 ChibiOS-EmbrIO

ChibiOS-EmbrIO is the core part of the program. It derived from demo code written in ChibiOS-2.4.2 and adapted to include all the functions and APIs to enable ChibiOS/RT subsystems and Embryo Abstract Machine.

2.5.1 Configuration files

ChibiOS/RT has a specific set of files used to define all the architecture (see **Figure 4**) [18]:

- `chconf.h` defines Kernel parameters and options, such as system tick frequency, or IPC elements to be loaded.
- `halconf.h` to define the high level Hardware Abstraction Layer (HAL) components. It allows to enable or disable the various device drivers for the application. The low level HAL is hardware specific and is part of the ChibiOS/RT source tree.
- `mcuconf.h` to define driver configurations and it is hardware specific. In our case *STM32F1xx* definitions were used.

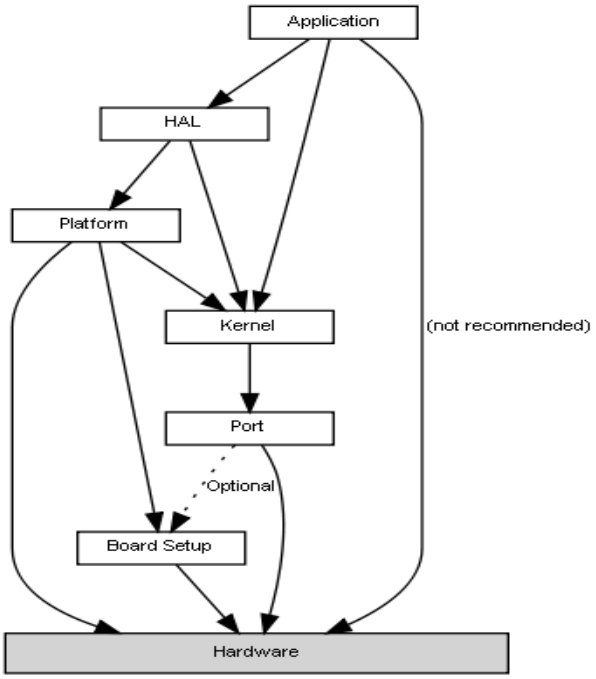

Figure 4: ChibiOS/RT architecture.

As our demoboard is already part of the ones defined in ChibiOS/RT, it was quite simple to reuse and adapt all the configuration files to our needs.

2.5.2 Application code

The application code takes care of setting up the system and then running the Embryo Virtual Machines. The setup phase initializes all the ChibiOS/RT Kernel and HAL components, and then initializes all the Embryo components that need some ChibiOS/RT API, e.g. Memory Management (see 2.3.2).
ChibiOS-EmbrIO is organized with and *Virtual Machine Manager* and multiple *Virtual Machines*, from zero up to a maximum number defined at compile time. Each Virtual Machine is represented by a C `struct` with the following fields:

```
struct EmbrioVM {
  /* state of the VM (running, etc.)*/
  EmbrioVMstate_t state;
  /* thread running the VM */
  Thread      *tp;
  /* program to be executed */
  Embryo_Program  *ep;
  /* function pointer to callback */
  Embrio_hook    hook;
  /* callback parameters */
  int         hook_params;
  /* next VM in linked list of VMs */
  EmbrioVM     *vm_next;
};
```

Listing 7: EmbrIO VM

- *state* represents the current condition of the machine: running, stopped, waiting, etc.
- *tp* is a pointer to the thread that runs the VM on ChibiOS/RT
- *ep* is a pointer to the Embryo program to be executed
- *hook* is a function pointer to a callback that can be defined at runtime
- *hook_params* is a parameter that can be passed to the callback
- *vm_next* is a pointer to the next VM in the chain of VM created.

The Virtual Machine Manager, at the present state of development, is a simple `struct` that maintains a status, the number of VMs and a pointer to the first of them.

```
typedef struct {
  EmbrioVMMstate_t  state;
  uint8_t         vm_count;
  EmbrioVM        *vm_first;
} EmbrioVMManager;
```

Listing 8: EmbrIO VM

After initializing ChibiOS/RT, two threads are spawned by default [19]:

- **Idle thread**. This thread has the lowest priority in the system so it runs only when the other threads in the system are sleeping. This thread switches the system in a low power mode and does nothing else.
- **Main thread**. This thread executes the `main()` function at startup. It is from the main thread that the other threads running the VM are created.

In ChibiOS-EmbrIO the main thread initializes all the needed timers (GPT: General Purpose Timer), used either by different drivers (e.g. SPI) or to call cyclic routines and configures all the external interrupts (EXT: generic EXTernal driver) and subsequent callbacks (e.g. what to do when a button is pressed), and then spawns a thread for each Embryo Virtual Machine.

```
/* allocate Memory Pool element for first VM */
vm[0] = (EmbrioVM*)chPoolAlloc(&EVM_mp);

/* Pool allocated */
if(vm[0] != NULL){
  /* insert VM in linked list of VMs */
  embrioVMMinsert(vm_man, vm[0]);
  /* load Embryo Program */
  vm[0]->ep = embryo_program_load_local(...);
}

/* program loaded correctly */
if(vm[0]->ep != NULL){
  /* set initial callback */
  vm[0]->hook = NULL;
  /* spawn and start VM thread */
  vm[0]->tp = vmStart(vm[0], NORMALPRIO, "VM0");
}
```

Listing 9: EmbrIO VM start

After initialization, every Virtual Machine is running, and the execution depends on the following elements:

- *Embryo bytecode*: the operations that are coded in each Embryo Program,
- *Native Functions*: these are the connection points between Embryo and ChibiOS/RT, their implementation allow to use ChibiOS/RT services inside an Embryo VM,
- *Drivers*: which drivers that are enabled and what functions are called as a callback,
- *Callbacks*: which Embryo functions are invoked by the callback.

As the overall behavior strongly depends on the elements above, for that reason some tests have been conducted to implement few simple functionalities in order to analyze the system performances. In the following section we describe the setup and the tests that we carried on.

3 Application Code

3.1 Embryo Code

We decided to implement two Virtual Machines with similar source code. As described in [20], Pawn and also Embryo allow an *event-driven* programming model. The main advantage of this model consists in building reactive programs, which respond to multiple events. The program has more than one entry point: while the `main` function runs immediately after you start the script and it runs only once, every function whose name begins with @ runs every time it is called. We coded two similar programs to be executed on two different VMs. The following code is an example:

```
#include <printXXX>
#include <embrio01>
/** global state variable */
new state = 0

/** event associated to first timer:
 *  VM toggles first led of demoboard if UP*/
@eventGPT1()
{
  if(state == 1){
    toggleLED1()
  }
}

/** event associated to second timer:
 *  VM reads samples from ADC if UP */
@eventGPT2()
{
  if(state == 1){
    readADC1()
  }
}

/** event associated to external interrupt
 *  the VM toggles state (UP or DOWN) and prints
 *  the status on serial port */
@eventEXTI1()
{
  if(state == 0){
    state = 1
    printXXX("\r\nVM 0 UP\r\n")
  }else{
    state = 0
    printXXX("\r\nVM 0 DOWN\r\n")
  }
}
/** application entry point:
 *  sets a 'state' variable to 1 (UP)
 * and prints on the serial port */
main()
{
  state = 1
    printXXX("\r\nVM 0 UP\r\n")
}
```

Listing 10: EmbrIO first VM

3.2 Embryo Compile and Load

The code has then been compiled with the following command string:

```
embryo-cc -c vm0.p -S 128
```

Listing 11: compile command

The execution outputs a `vm0.eaf` file (Embryo Assembly File). It is important to note that, in order to reduce stack size and overall program size, it is mandatory to use the `-S` option using the least feasible stack/heap size, otherwise `Embryo_Program` requires too many Kbytes of memory. Further development in ChibiOS-EmbrIO will allow

assembly code to be loaded into memory by using some communication interface (e.g. Serial RS232, CAN, etc...). At present state, no particular interface is fully functional and so we decided to load the assembly code together with the rest, linking it at compile time. We first used the following command:

```
arm-none-eabi-objcopy -I binary -O elf32-
    littlearm -B arm vm0.eaf vm0_eaf.o
```

Listing 12: object code command

By linking the `vm0_eaf.o` together with the rest of object files, we can use the following `extern` variables to access the code, which will be loaded in RAM.

```
/* VM0 code references */
extern unsigned char _binary_vm0_eaf_start;
extern unsigned char _binary_vm0_eaf_end;
extern unsigned char _binary_vm0_eaf_size;
```

Listing 13: code location variables

When loading a new program, we refer to the variables above to access the data.

```
/* VM0 load from memory */
vm[0]->ep = embryo_program_load_local(
        &_binary_vm0_eaf_start,
        &_binary_vm0_eaf_end,
        &_binary_vm0_eaf_size,
        /* log options */
        (BaseChannel*)&SD3, TRUE);
```

Listing 14: Program load

Upon successful load, the program can be executed by an Embryo Virtual Machine.

3.3 Drivers and Callbacks

We decided to implement the following drivers and callbacks:

- *TIM1*: timer 1 gives the *heartbeat* of the Virtual Machine, if it is in UP state makes the led blink. Each Virtual Machine makes a different led blink. The function name is the same (*@eventGPT1*) for every VM, but the code is different.
- *TIM2*: timer 2 triggers an ADC conversion: the first VM has a function named (*@eventGPT2*) that is used to read the ADC samples and to print data on serial port. The function operates only when the Virtual Machine is UP.
- *TIM3*: timer 3 triggers a SPI read: the second VM has a function named (*@eventGPT3*) that is used to read the SPI data from a sensor connected to the board and to print the data on serial port. The function operates only when the Virtual Machine is UP.
- `EXTI1`: the external interrupt driver can be used to connect different interrupt sources to different callbacks. In our case we connected a callback to the first user button and another callback to the second user button. The first VM defines a function named `@eventEXTI1` that toggles its state and the second VM has a similar function named `@eventEXTI2`.

3.4 Dispatch of Callbacks

In this first implementation of Embryo Virtual Machines on ChibiOS/RT, we decided not to implement *argument passing* to and from Embryo functions, leaving the implementations of *hook* and *hook_params* to further development.
Each callback simply needs to call an event function in a program without arguments, and accomplishes this task by looking for the function in each machine. If the function is part of the code (i.e. it can be run on the VM), the Embryo function `embryo_function_find` does the work for us: it returns a specific value (`EMBRYO_FUNCTION_NONE`) if there is no corresponding function or a number that can be used to identify and execute the function itself.
The following code illustrates the operation described here:

```
/* TIM1 callback */
void gpt1cb(GPTDriver *gptp) {

  EmbrioVM *vm_tmp;
  Embryo_Function ef;

  /* the first VM is pointed by VM Manager */
  vm_tmp = vm_man->vm_first;

  /* we look into all VMs */
  while(vm_tmp != NULL){

    /* function that tells if a function is in
       the VM */
    ef = embryo_program_function_find(vm_tmp->ep
        , "@eventGPT1");

    /* if available send a message to thread */
    if(ef != EMBRYO_FUNCTION_NONE){
      chMsgSend(vm_tmp->tp,(msg_t)ef);
    }

    /* we go to next VM */
    vm_tmp = vm_tmp->vm_next;
  }
}
```

Listing 15: Callback looking for a function in VMs

In the example above, a timer event triggers the callback function (named `gpt1cb`), which looks for a specific function (here `@eventGPT1`) and, if it is found, sends a message to the corresponding thread. Note that different VMs can implement the same function in different ways, allowing some flexibility: in our case, with the same callback, one VM makes the green led blink and the other VM

makes the yellow led blink.
The *EmbrIO_VM* `struct` carries all the necessary information (i.e. pointer to the program, pointer to the thread, ...) to perform the operation.
When a Virtual Machine thread is spawned, it loads the code, executes the main function and finally enters an infinite loop waiting for messages. Note that a callback sends a different message to each thread that is involved, and the message carries the value of the function to be executed, coded as a `Embryo_Function` parameter. When the message arrives, the thread wakes up and executes the corresponding function. The following code illustrates the operations described above:

```
{
  /* load Embryo Program */
  embryo_program_vm_push(vmp->ep);

  /* run main function */
  es = embryo_program_run(vmp,
       EMBRYO_FUNCTION_MAIN);

  /* enter infinite loop */
  while(TRUE){
    /* wait for message */
    tp = chMsgWait();

    /* get message = function to be executed */
    ef = (Embryo_Function)chMsgGet(tp);

    /* acknowledge sending thread */
    chMsgRelease(tp, RDY_OK);

    /* execute function */
    es = embryo_program_run(vmp, ef);
  }
}
```

Listing 16: Callback looking for a function in VMs

The code above is the same for all the threads that are spawned. The only difference consists in the Embryo code that is loaded when the Virtual Machine is created.

4 Tests

We tested the application in two different phases. At first we analyzed the behavior of the system itself (see setup in Figure 5):

- 2 Virtual Machines
- 3 GPT callbacks
- 2 EXT interrupt callbacks connected to buttons
- mock native functions for ADC and SPI callbacks (substituted with serial port log)

Within this environment we verified that the system worked as expected: with the two buttons enabling and disabling the VMs.
We tried changing the timers frequency and verified that the system continued working. In a subsequent phase we started configuring the *SPI* and *ADC* HAL of ChibiOS/RT, to be able to read two different temperature sensors:

- a 5kΩ NTC thermistor [21] with a simple conditioning circuit [22].
- a DS18B20 SPI temperature sensor with a connection scheme similar to the one used for an Arduino platform and described in [23].

We used port C pins 10 to 12 to connect SPI sensor.

5 Conclusions

The main target of the project was to make Embryo and ChibiOS/RT work together, allowing Embryo programs run on Abstract Machines in a ChibiOS/RT environment. We saw that this kind of setup is feasible and promising in a lot of applications.
Further development is still needed to complete some features and to extend the functionalities of ChibiOS-EmbrIO. In particular we cite:

- Load Embryo programs at runtime, using some communication interface (e.g. CAN).
- Implement more native functions in order to use as many ChibiOS/RT services as possible, making ChibiOS/RT and Embryo more deeply connected. As a result, much more degrees of freedom will be available when writing Embryo programs.
- Allow the use of *argument passing* when calling Embryo function or use return values.
- Complete the code of the Virtual Machine Manager, for example making it the central component that manages all the Virtual Machines, for example by using the *queue* ChibiOS/RT service from external interrupts to the manager and messages from the manager to the VMs.
- extend some Embryo functionalities (e.g. *string* usage or *Float* operations) if interesting for some applications.

Further analysis of the performances is also needed. In particular we cite:

- Analyze the optimal size of Embryo Programs, limiting stack/heap size without worsening Virtual Machine performances.
- Measure execution time, in order to determine how fast we can execute VMs and so update data (this can be really interesting if applied to controllers). On the other hand, it can be interesting to determine if the message passing environment reaches a critical level for performances.

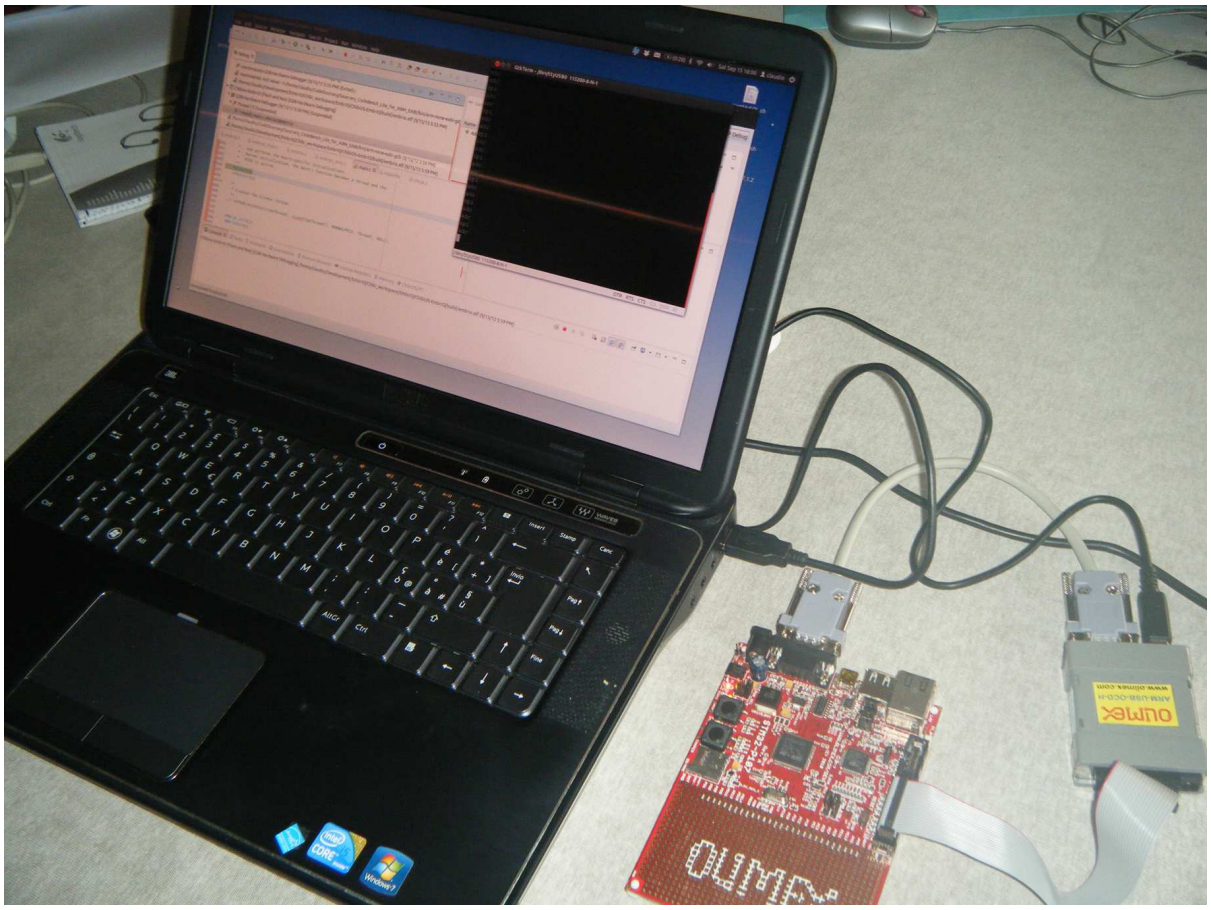

Figure 5: Application and Hardware setup.

- Evaluate if it is possible to optimize memory usage: for example when loading an Embryo program.

As a final remark we can state that ChibiOS-EmbrIO can bring to a very interesting further development.

References

[1] Wikipedia: Enlightenment foundation library. http://en.wikipedia.org/wiki/Enlightenment_Foundation_Libraries

[2] Wikipedia: Enlightenment window manager. http://en.wikipedia.org/wiki/Enlightenment_(window_manager)

[3] Enlightenment: homepage. http://www.enlightenment.org/

[4] Enlightenment: Embryo. http://docs.enlightenment.org/auto/embryo/

[5] Compuphase: Pawn language. http://www.compuphase.com/pawn/pawn.htm

[6] Enlightenment: Embryo language. http://docs.enlightenment.org/auto/embryo/Small_Page.html

[7] Enlightenment: Eina library. http://docs.enlightenment.org/auto/eina/

[8] Wikipedia: Chibios/rt. http://en.wikipedia.org/wiki/ChibiOS/RT

[9] Di Sirio, G.: Chibios/rt homepage. http://www.chibios.org/dokuwiki/doku.php?id=start

[10] ARM.com: Arm cortex-m3 processor. http://www.arm.com/products/processors/cortex-m/cortex-m3.php

[11] olimex.com: Olimex demoboard. https://www.olimex.com/dev/stm32-p107.html

[12] STMicroelectronics: Stm32 f 107 processor. http://www.st.com/internet/mcu/product/221020.jsp

[13] Eclipse.org: Eclipse (c/c++ development). http://www.eclipse.org/cdt/

[14] Mentor: Sourcery codebench. http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/

[15] Di Sirio, G.: Using an eclipse-based ide. http://www.chibios.org/dokuwiki/doku.php?id=chibios:guides:eclipse2

[16] Migliavacca, M.: Unimib informatica industriale course website. http://irawiki.disco.unimib.it/irawiki/index.php/Informatica_Industriale_/_Informatics_for_Industrial_Applications_2011/12 (2012)

[17] Di Sirio, G.: Chibios/rt memory management. http://www.chibios.org/dokuwiki/doku.php?id=chibios:howtos:manage_memory

[18] Di Sirio, G.: Chibios/rt general architecture. http://www.chibios.org/dokuwiki/doku.php?id=chibios:documents:architecture

[19] Di Sirio, G.: Chibios/rt threads. http://www.chibios.org/dokuwiki/doku.php?id=chibios:howtos:createthread

[20] Compuphase: Pawn language guide. http://www.compuphase.com/pawn/Pawn_Language_Guide.pdf

[21] Wikipedia: Thermistor. http://en.wikipedia.org/wiki/Thermistor

[22] embedded.com: Thermistor circuit. http://embedded.com/design/embedded/4023910/Temperature-Measurement-Technique

[23] bildr.org: Ds18b20 and arduino. http://bildr.org/2011/07/ds18b20-arduino/