IN323: Bootstrap project

Author : C. Garion <garion@isae.fr> and J. Hugues <jerome.hugues@isae.fr>
Audience : IN
Date :

Introduction

This mini-project will start the ISAE Computer Science curriculum. The goal of the project is to use an small Arduino board [1] to emit Morse coded sentences using its LED (cf. figure 1). The base code will be developed first on a classical desktop and then cross-compiled and tested on a real board.

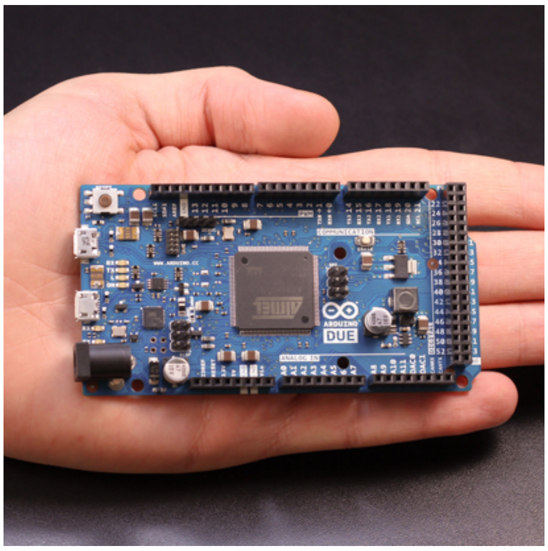

Figure 1: An Arduino Due board (photo by the Arduino team)

Morse code [2] is a particular encoding of the basic Latin alphabet with only two characters, "`-`" (dash) and "`.`" (dot). For instance, the Morse encoded string `-- --- .-. ... .  -.-. --- -.. .` represents the ASCII encoded string `MORSE CODE`. Notice that:

- consecutive letters are separated by a space in the Morse encoded string
- spaces between consecutive words are considered as letters and are represented by a space

To simplify the project, we will only be interested in uppercase letters (no numbers, lowercase letters or punctuation signs). We will also suppose that Morse sentences length will not exceed 200 characters.

Objectives

The main objective of this mini-project is to test and eventually complete students knowledge on some prerequisites on which many courses of the CS curriculum depend. Each prerequisite will be tackled both on theoretical and practical point of view.
The following notions will be presented:

- Software Configuration Management (SCM)
- C programming language basics and advanced concepts
- build process
- cross-compilation and target integration
- Test-Driven Development (TDD) and mock objects

The programming language used in this project is the C programming language. Some useful references are [3, 4, 5, 6, 7]. Do not forget that most C functions are documented on UNIX platforms via man pages (e.g. `man printf` will give you the manual page of the `printf` function). A small reference card will also be provided. Please avoid "Le site du Zéro" or similar websites and use good references...
Moreover, you have to write code conforming to the JPL Coding Standard for the C programming language [8] during this project.

Schedule

The schedule of the mini-project is presented on the table 1 (each session is a 2h30 session). For each lab session, you will have to commit your work on a Subversion repository before the due date.

Session	Session day	Concepts	Due date
S1	09/03/14 14:00-16:45	SCM and C basics	09/08/14 12:15
S2	09/08/14 09:30-12:15	C language: advanced concepts	09/08/14 12:15
S3	09/09/14 09:30-12:15	Build process & Cross compilation	09/09/14 12:15
S4	09/22/14 14:00-16:45	Test-Driven Development	09/22/14 23:00

Table 1: Bootstrap mini-project schedule

Sessions 1-2: SCM and C

1 Software Configuration Management

In this section, you will checkout your personal Subversion repository and prepare it for the mini-project. A good reference on Subversion is [9].
The URL for your repository is `https://eduforge.isae.fr/repos/IN323/login` where `login` is your login on ISAE computers. Use your regular password when asked.

1. checkout the main branch of your repository which is classically `trunk`. You will obtain a directory with a `compile-file.sh` script, an empty `src` repository, a `include` directory containing some header files for the C basics session and a `ext` directory containing helper files.

2. create a `dev` branch to work on the mini-project. This branch is for development purposes only, your work will be evaluated on the main branch. The branch will be located into the `branches` directory of your repository. Switch to the `dev` branch.

Do not forget to merge your `dev` branch into your main branch when you have finished the session exercises. Beware, there may be some conflicts (introduced by your beloved professors) to solve...

2 C programming language

In this section, you will implement in C a basic translator from characters to Morse code and from Morse code to characters. A basic data structure that can be used to implement such a translator is an *associative array* [10, 11]. An associative array is a "generalization" of an array in which the index of the array are not only integers but may be any type. The indexes are called *keys* and the corresponding values are called *values*. An efficient way to implement associative arrays is to use *hash tables* in which the values are stored in a classical array and a *hash function* compute the index of the value corresponding to a particular key. Notice that collisions may occur when the hash function computes the same value for two different keys and should be treated properly, but we will not address this problem.
The prototypes of the functions you will have to implement are defined in header files located in the `include` directory of your repository. You will also find in the `ext` directory:

- a file `morse.txt` containing the Morse code for the ASCII characters A to Z in this order
- a file `sentence.txt` containing the Morse sentence to translate for question 4

Finally, the `compile-file.sh` script can be used to compile a C file containing a `main` function into an executable. Study the script to understand how it works.

Please document your C source files using Doxygen [12], a javadoc-like documentation system supporting C (and other languages). The Makefile in your repo includes a rule to generate documentation with Doxygen.

You will test your implementation using a `test.c` file containing only a `main` function.

1. we will interest first in the ASCII to Morse conversion
   (a) how to implement a hash table for the ASCII to Morse conversion? Remember that characters in the C programming language are *integer* types. You can use the ASCII table [13] to find which integer is associated to a particular character.
   (b) document the `hash_ascii` and `convert_ascii_to_morse` functions.
   (c) implement the `hash_ascii` and `convert_ascii_to_morse` functions and test them.

You can use the `string` library to manipulate strings (e.g. use the `strcmp` function to compare strings). However, it would be more useful for you to implement those two functions only with C basic constructs (particularly pointers).

When implementing a program in C, you may have memory management problems. You can use the Valgrind tool [14] to detect memory problems.

2. find a "natural" hash function for the Morse alphabet and implement it via the `hash_morse` function. Is this function suitable for our needs?

3. as a hash table cannot be used, we must find a new data structure to handle the Morse to ASCII translation problem. In the spirit of Huffman coding used for data compression [10, 15], we can use the binary tree represented on figure 2.

   When wanting to translate a Morse letter, start from the tree root and analyze each symbol in the Morse letter: when encountering a "`.`" go left, when encountering a "`-`" go right. For instance, when translating `.-.` go left-right-left in the tree and find `R`.

   In order to simplify the implementation, space symbol has now a Morse code which is `----`.

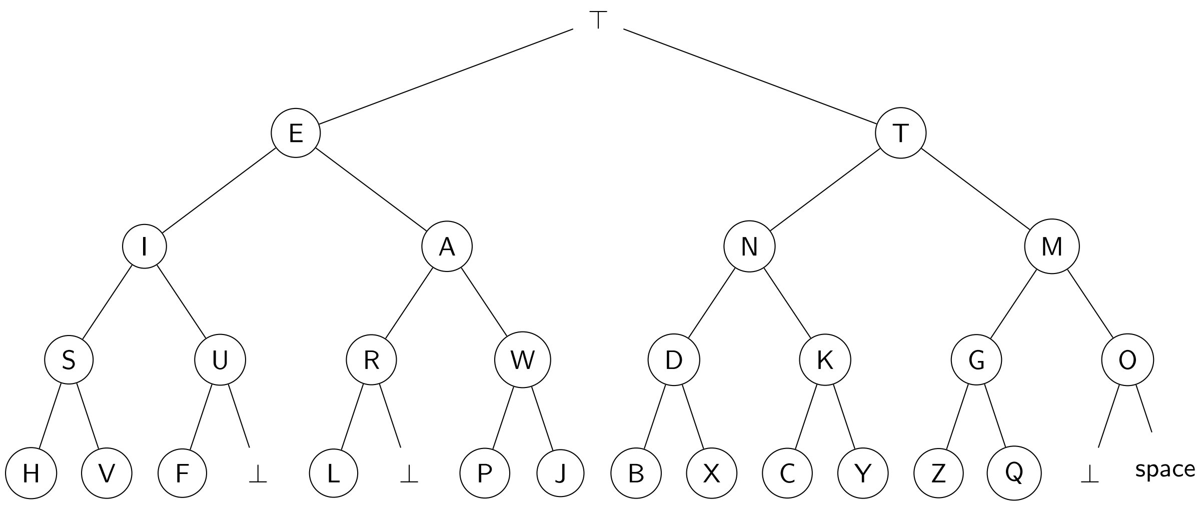

Figure 2: A binary tree for Morse to ASCII translation

   (a) define a data structure to handle the previous binary tree.
   (b) implement the `convert_morse_char_to_ascii` function.
   (c) implement the `convert_morse_to_ascii` function.

You can use the `strtok` function of the `string` library to analyze the Morse encoded sentence, it will be easier.

4. translate the following MORSE sentence:

```
.. ---- .-.. --- ...- . ---- -.-. ---- .--. .-. --- --. .-. .- -- -- .. -. --.
```

5. (bonus) using the idea behind the previous binary tree structure and a matrix of characters, find a more efficient implementation of `convert_morse_char_to_ascii` using `hash_morse`.

Session 3: Build process & Cross compilation

For this session, we will work on the `prise.isae.fr` workstation. You can connect to this station using 'ssh -Y student@prise.isae.fr', and work in a directory you'll create there.

In this session, we will now port the project to the final target. We will consider a simplified setup made of an ATMega328p MCU and a ramp of several leds.

The ATMega 328p is an AVR-family micro-controller. From its datasheet at Atmel Ltd.[1]:

"The high-performance Atmel 8-bit AVR RISC-based microcontroller combines 32 KB ISP flash memory with read-while-write capabilities, 1 KB EEPROM, 2 KB SRAM, 23 general purpose I/O lines, 32 general purpose working registers, three flexible timer/counters with compare modes, internal and external interrupts,serial programmable USART, a byte-oriented 2-wire serial interface, SPI serial port, 6-channel 10-bit A/D converter (8-channels in TQFP and QFN/MLF packages), programmable watchdog timer with internal oscillator, and five software selectable power saving modes. The device operates between 1.8-5.5 volts. By executing powerful instructions in a single clock cycle, the device achieves throughputs approaching 1 MIPS per MHz, balancing power consumption and processing speed."

In the following, we will not use a real board, but instead a simulator: `simavr`[2]. This tool provides an accurate support for AVR MCUs, while providing a confortable setup for debugging.
Code for `simavr` is organised as a two sets of C units:

- `<main>.c` configures the MCU and loads the code for the AVR target;
- `<mcu_name>_<main>.c` is the main entrypoint for the AVR code.

The following source code is the main entrypoint of a function that simply blinks some leds from right to left. We use two libraries: `leds.h` for blinking some leds, and `delay.h` for implementing a small delay. Note we redefine `F_CPU` to match the actual MCU core frequency.

Listing 7: `atmega328p_ledramp.c`

```c
/** AVR default include files **************************************************/
#ifdef F_CPU
#undef F_CPU
#endif

#define F_CPU 16000000L  /* Set MCU frequency to 16 MHz */
#include <util/delay.h>

#include "leds.h" /* Project specific library for managing leds */

/** SIMAVR specific macros *****************************************************/
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega328p");  /* Emulate an ATMega328p chipset */

/** Main entrypoint ************************************************************/
int main(int argc, char **argv) {
  uint8_t mask = 0;

  led_init();

```

[1] http://www.atmel.com/devices/atmega328p.aspx
[2] https://github.com/buserror-uk/simavr

```
  for (;;) {
    mask <<= 1;
    if (!mask)
      mask = 1;
    else
      leds_on (mask);
    _delay_ms (250);
  }
}
```

It relies on the `leds.h` library that supports basic function to light a set of leds defined by a bitmask.

Listing 8: `leds.h`

```
/* ATMega328p library for managing leds */

#ifndef __LEDS__
#define __LEDS__
void led_init (void);
/* Configure ports to light leds */

void leds_on (unsigned char mask);
/* Switch on/off leds */
#endif
```

To compile this example, you'll have to type 'make -f Makefile.simavr' in your terminal. This makefile sets the compilation chain that is specific to the simavr target.
Note you may need to edit it to add your own files.

1. test the main function provided as an example;
2. implement the `send_to_arduino.[c|h]` and `arduino.[c|h]` APIs defined in the previous session. Write a small example to demonstrate separately each function. You may propose a scheme to map morse characters '-' and '.' to a combination of leds
3. implement a main function that will repeat periodically the string "all work and no play makes jack a dull boy".
4. ATmega328p has a limited memory of 32KiB for code, 2KiB for variables. Does your implementation fits?

Session 4: Test-Driven Development

3 Test-Driven Development

In this section, you will use the Unity [16] or the CppUTest [17] unit testing framework to test your implementation of the Morse to ASCII and ASCII to Morse translators.

1. complete your Makefile to use Unity or CppUTest. Your test files should be in a `tests` directory.

> At ISAE, the header files for Unity are located at `/usr/local/unity/include` and the libraries at `/usr/local/lib`. The header files for CppUTest are located at `/usr/local/cpputest/include` and the librairies at `/usr/local/cpputest`.
> Do not forget to use `-I` and `-L` switches of `gcc` to use directories you need.

> When using CppUTest, remember that you have to manage two types of files:
>
> - your application files, written in C
> - your test files, "written" in C++
>
> You should thus write two targets for compilation, one for C files and one for C++ files (whose suffix is `.cpp`). You should also include the `CppUTest` and `CppUTestExt` librairies for linking unit tests files.

2. using your "test" file developed in section 2, write unit tests for the functions defined in `ascii_to_morse.h` and `morse_to_ascii.h`.

> Beware with CppUTest, when using `malloc` in a C++ file, you have to explicitly cast the result to the desired pointer type.

3. we want now to write the `send_morse_letter` function that use the Arduino LED to visually communicate. The prototype of the function is defined in `send_to_arduino.h` (cf. listing 9). It uses the `light_led` function defined in `arduino.h` (cf. listing 10)[3].

Listing 9: send_to_arduino.h

```c
#ifndef SEND_TO_ARDUINO_H
#define SEND_TO_ARDUINO_H

static const int MINUS_TIME=3000;
static const int DOT_TIME=1000;
static const int RELAX_TIME=1500;

int send_morse_letter(char *morse_letter);

#endif
```

Listing 10: arduino.h

```c
#ifndef ARDUINO_H
#define ARDUINO_H

```

[3]The `light_led` function is not real and is only use for this exercise.

```
static const int HIGH=255;
static const int LOW=0;
static const int LED1=0;
static const int LED2=1;
static const int LED3=2;

/**
 * light a led
 *
 * @param[in] number the led number
 * @param[in] value the value to set the led
 * @param[in] time the time to send the signal
 */
void light_led(int number, int value, int time);

#endif
```

(a) write the `send_morse_letter` function.

(b) using either Unity and CMock [19] or CppUTest and its extensions, write a single test verifying that `send_morse_letter` calls correctly `light_led` with `-.-.` as an argument.

References

[1] *Arduino*. 2013. URL: http://www.arduino.cc.

[2] Wikipedia Contributors. *Morse code*. 2013. URL: http://en.wikipedia.org/wiki/Morse_code.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd edition. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[4] Wikibooks Contributors. *C Programming*. 2013. URL: http://en.wikibooks.org/wiki/C_programming.

[5] Zed A. Shaw. *Learn C the hard way*. 2013. URL: http://c.learncodethehardway.org/book/.

[6] Steve Summit. *comp.lang.c Frequently Asked Questions*. 2013. URL: http://c-faq.com/.

[7] Brian W. Kernighan. *Programming in C: a tutorial*. 2013. URL: http://www.lysator.liu.se/c/bwk-tutor.html.

[8] Brian Kernighan et al. *JPL Institutional Coding Standard for the C Programming Language*. California Insititute of Technology. Mar. 3, 2009. URL: http://lars-lab.jpl.nasa.gov/.

[9] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version control with Subversion*. 2013. URL: http://svnbook.red-bean.com/.

[10] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd edition. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[11] Wikipedia Contributors. *Associate array*. 2013. URL: http://en.wikipedia.org/wiki/Associative_array.

[12] Dimitri van Heesch. *Doxygen*. 2013. URL: http://www.stack.nl/~dimitri/doxygen/index.html.

[13] Wikipedia Contributors. *ASCII*. 2013. URL: http://en.wikipedia.org/wiki/ASCII.

[14] Valgrind Developers. *Valgrind*. 2013. URL: http://www.valgrind.org.

[15] Wikipedia Contributors. *Huffman coding*. 2013. URL: http://en.wikipedia.org/wiki/Huffman_coding.

[16] Mike Karlesky, Mark Vandervoord, and Greg Williams. *Unity*. 2013. URL: http://throwtheswitch.org/white-papers/unity-intro.html.

[17] CppUTest contributors. *CppUTest*. 2013. URL: http://cpputest.github.io/.

[18] Andreas Schneider. *cmocka – a unit testing framework for C with mock objects*. 2013. URL: http://www.cmocka.org/.

[19] Mike Karlesky, Mark Vandervoord, and Greg Williams. *CMock*. 2013. URL: http://throwtheswitch.org/white-papers/cmock-intro.html.