Introduction to 8-bit microcontrollers

J.-M Friedt

26 août 2024

Questions :

1. What is the benefit of using the `make` make tool and its `Makefile` configuration?
2. What is the benefit of defining the program name as a variable in a `Makefile`?
3. What is the benefit of separating files containing functions related to a given feature of a microcontroller?
4. What option of `gcc` allows for linking with a static library `libtoto.a`?
5. How is the path to the directory including the `libtoto.a` library defined during compilation?
6. What is the name of the virtual serial port interface created by Linux when communicating with a microcontroller running the LUFA library?

The purpose of this lab work session is to :

1. better understand the functioning of the `gcc` compiler, especially its optimization options
2. organize code into separately compiled modules, ideally reusable (function libraries)
3. automate the compilation sequence of separate programs using the `make` tool
4. apply the acquired knowledge, especially on `make`, to use USB communication by linking with the LUFA library.

1 Optimization options

A C code is not bijective with an assembly code : different compilers will generate different implementations of the same C code, and the same compiler can be configured to optimize the code according to various criteria (size, speed). These optimizations can however have undesirable effects if not mastered.

Listing 1 – Blinking LED with manual delay

```
1  #include <avr/io.h> //E/S ex PORTB
2  #define F_CPU 16000000UL
3
4  void my_delay(long duree)
5  {long k=0;
6   for (k=0;k<duree;k++) {};
7  }
8
9  int main(void){
10   DDRB |=1<<PORTB5;
11   DDRE |=1<<PORTE6;
12   PORTB |= 1<<PORTB5;
13   PORTE &= ~(1<<PORTE6);
14
15   while (1){
16     PORTB^=1<<PORTB5;PORTE^=1<<PORTE6;
17     my_delay(0xffff);
18   }
19   return 0;
20 }
```

The example provided above follows the naïve approach of implementing a delay as an empty loop.

The code is compiled with `avr-gcc -mmcu=atmega32u4 -Wall -o blink.out blink.c` and the resulting assembly language is displayed with `avr-objdump -dS blink.out`

1. **compile with the `-O0` option and observe the resulting `my_delay()` function.**

2. **compile with the `-O2` option and observe the resulting `my_delay()` function.**
3. **compile with the `-O1` option and observe the resulting `my_delay()` function.**
4. **compile with the `-O2` option after prefixing the `k` variable definition with the `volatile` keyword which prevents the compiler from optimizing code related to a given variable, and observe the resulting `my_delay()` function.**
5. **compile with the `-g -O2` options and observe the resulting `my_delay()` function.**

2 Modular programming – separate compilation

A program is efficiently separated into independent modules, or libraries. Separate compilation involves compiling each program into independent objects, which are then linked into a single executable.

In this example, we want to be able to reuse the waiting function by compiling as an object that can be called by various programs. However, we must declare the availability of the function in the object : this is the role of the header file `.h`.

⚠Warning : a header file *never* contains C code.

Listing 2 – Main file `separe_blink.c`

```
#include <avr/io.h> //E/S ex PORTB

#define F_CPU 16000000UL

#include "separe_delay.h"

int main(void){
  DDRB |=1<<PORTB5;
  DDRE |=1<<PORTE6;
  PORTB |= 1<<PORTB5;
  PORTE &= ~(1<<PORTE6);

  while (1){
    PORTB^=1<<PORTB5;PORTE^=1<<PORTE6;
    mon_delai(0xffff);
  }
  return 0;
}
```

Listing 3 – Delay function : `separe_delay.c`

```
void mon_delai(long duree)
{long k=0;
 for (k=0;k<duree;k++) {};
}
```

Listing 4 – Fichier d'entête de définition des fonctions : `separe_delay.h`

```
void mon_delai(long);
```

These files are compiled, in order to first generate two objects which are then linked into a binary executable, with

```
avr-gcc -mmcu=atmega32u4 -Os -Wall -c separe_delay.c
avr-gcc -mmcu=atmega32u4 -Os -Wall -c separe_blink.c
avr-gcc -mmcu=atmega32u4 -o blink_separe.out separe_blink.o separe_delay.o
```

A dedicated tool allows for taking dependencies into account : `make`, which uses a configuration file named by default `Makefile`.

Furthermore, separate compilation requires that the functions or variables defined in another object are declared. Header files (*headers*) group function definitions, while the definition of a variable prefixed by `extern` announces a definition in another object (with associated memory allocation).

Listing 5 – Makefile example

```
all:  separe_blink.out


separe_blink.out: separe_delay.o separe_blink.o
   avr-gcc -mmcu=atmega32u4 -Os -Wall -o separe_blink.out separe_blink.o separe_delay.o
```

```
6
7  separe_delay.o: separe_delay.c separe_delay.h
8     avr-gcc -mmcu=atmega32u4 -Os -Wall -c separe_delay.c
9
10 separe_blink.o: separe_blink.c
11    avr-gcc -mmcu=atmega32u4 -Os -Wall -c separe_blink.c
12
13 clean:
14    rm *.o separe_blink.out
15
16 flash: separe_blink.out
17    avrdude -c avr109 -b57600 -D -p atmega32u4 -P /dev/ttyACM0 -e -U flash:w:separe_blink.out
```

Two methods are traditionally defined in the `Makefile` : `all` indicates the final target to reach, and `clean` provides the commands to clean the directory. Each line contains the target, the dependencies and then the method to implement if the dependencies are resolved.

⚠Attention : each method must start with a tabulation and *not* with spaces.

3 Complements about `Makefile`

In order to avoid modifying multiple locations in the `Makefile` that will be used throughout this lab session, we introduce a new concept : the variable.

A `Makefile` variable is defined by VARIABLE=value and will be called by `$(VARIABLE)` (note – the convention is not the same as for the *shell*[1] which separates the variable name from the $ with braces instead of parentheses.

Listing 6 – Exemple de `Makefile` avec variable.

```
1  MCU=atmega32u4
2  REP=$(HOME)/enseignement/sqnx/platforms/Atmega32/
3  #TARGET=gpio_int
4  TARGET=gpio_int_edge
5  #TARGET=timer_ovfirq
6  #TARGET=timer_irq
7  #TARGET=wdt
8  #TARGET=wdt_rs232
9  #TARGET=input_capture
10 #TARGET=input_capture_sendai
11 #TARGET=PRN_villedavray_header
12 #TARGET=switch_sendai
13 #TARGET=tmp
14 #TARGET=usart_int
15 #TARGET=gpr
16
17 all:  $(TARGET).hex
18
19 $(TARGET).hex: $(TARGET).out
20    avr-objcopy -Oihex $(TARGET).out $(TARGET).hex  # for MS-Windows (winavr requires .hex)
21    avr-objdump -dSt $(TARGET).out > $(TARGET).lst
22
23 $(TARGET).out: $(TARGET).o
24    avr-gcc -L$(REP)/VirtualSerial/ -mmcu=$(MCU) -o $(TARGET).out $(TARGET).o -lVirtualSerial
25
26 $(TARGET).o: $(TARGET).c
27    avr-gcc -Wall -g -mmcu=$(MCU) -I$(REP)/VirtualSerial/ -I$(REP)/lufa-LUFA-140928/ -DF_USB=16000000UL \
28    -std=gnu99 -Os -c $(TARGET).c
29    avr-gcc -Wall -g -mmcu=$(MCU) -I$(REP)/VirtualSerial/ -I$(REP)/lufa-LUFA-140928/ -DF_USB=16000000UL \
30    -std=gnu99 -Os -S $(TARGET).c
31
32 flash: $(TARGET).out
33    avrdude -c avr109 -b57600 -D -p $(MCU) -P /dev/ttyACM0 -e -U flash:w:$(TARGET).out
34
35 clean:
36    rm *.o $(TARGET).out
```

1. C. Newham, *Learning the bash Shell : Unix Shell Programming (In a Nutshell)*, Ed. O'Reilly (2005)

4 Communication – calling a library

LUFA[2] is a rich library for managing the USB protocol, which is complex to implement [1] given the variety of conditions of use and transfers carried out during the initialization of transactions with the PC. These initializations are specifically intended to inform the PC operating system of the capabilities of the connection (number of endpoints, datarate, which driver can handle these transactions on the PC).

A library acquired as source code must be compiled. Search for the compressed archive of the library at `http://jmfriedt.free.fr/LUFA_light_EEA.tar.gz`. Uncompress and unpack the contents of the file with `tar zxvf LUFA_light_EEA.tar.gz`. The archive is decompressed and unpacked into the subdirectory `lufa`, where we will work. Enter the `VirtualSerial_lib` directory and compile the library with `make lib` (note the presence of a file named `Makefile` in this directory). During compilation, the headers are located in `lufa-LUFA-140928` and the complete archive of the functions provided in the library in `libVirtualSerial.a`. We will only refer to this last file when using the communication features on the USB bus. As a convention of `gcc`, linking the library named `libtoto` provided as a static archive `libtoto.a` is achieved with the option `-ltoto` at the end of the `gcc` command sequence. It might be desirable to inform in which directories the libraries are located with the `-L directory` option.

We provide a simplified version of the library and a simple example of transactions on a virtual serial port, named under GNU/Linux `/dev/ttyACM0`. To compile a program linked to this library :

1. go into the directory `VirtualSerial_example`, modify the `Makefile` to point the `-L` option to the directory containing `libVirtualSerial.a` (presumably `../VirtualSerial_lib`),
2. compile the example with `make`
3. optionally compile and transfer the program to the microcontroller on the Olinuxino32U4 board with `make flash_109`,
4. verify the software is functioning properly with `cat < /dev/ttyACM0`
5. `cat` will wait before displaying received messages on the screen, whether it be a line feed `\n`, or after filling a memory buffer. It is sometimes easier to display immediately all characters received on the serial port, for example with `minicom -D /dev/ttyACM0` where the `-D` option defines the communication port. We quit `minicom`[3] with "CTRL-a" followed by "q".

Thus, we specify in the `Makefile` the path to this library at the time of linking by `-L./lib` (relative path from the current directory, which could be an absolute path). Similarly, each object needs the definitions of the functions provided by the library as mentioned in the header files whose location is defined by `-I./include`.

Note the need to include the header file `VirtualSerial.h` provided in `VirtualSerial_example` which defines the functions necessary for the operation of USB, as well as the following data structures

```
extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
extern FILE USBSerialStream;
```

Listing 7 – Communication through a (asynchronous) virtual serial port on a USB bus

```
1  #include <avr/io.h>
2  #include <avr/wdt.h>
3  #include <avr/power.h>
4  #include <avr/interrupt.h>
5  #include <string.h>
6  #include <stdio.h>
7
8  #include "VirtualSerial.h"
9  #include <util/delay.h>
10
11 extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
12 extern FILE USBSerialStream;
13
14 int main(void)
15 {
16  char ReportString[20] = "World\r\n\0";
17
18  SetupHardware();
19 /* Create a regular character stream for the interface so that it can be used with the stdio.h functions */
20   CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
```

2. *Lightweight USB Framework for AVRs*, `www.fourwalledcubicle.com/LUFA.php`
3. in case of failure, make sure Modem Manager is not being executed, other remove the package with `apt remove -purge modemmanager`

```
21   GlobalInterruptEnable();
22
23   for (;;)
24     {
25      fprintf(&USBSerialStream,"Hello␣");
26      fputs(ReportString, &USBSerialStream);
27
28 //   les 3 lignes ci-dessous pour accepter les signaux venant du PC
29      CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
30
31      CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
32      USB_USBTask();
33      _delay_ms(500);
34     }
35 }
```

When compiling, `make` runs the following sequence :

```
avr-gcc -mmcu=atmega32u4 -Os -Wall -c separe_delay.c
avr-gcc -c -mmcu=atmega32u4 -Wall -I. -I../lufa-LUFA-140928/ -DF_USB=16000000UL \
        -DF_CPU=16000000UL -Os -std=gnu99  VirtualSerial.c
avr-gcc -mmcu=atmega32u4 -L. separe_delay.o VirtualSerial.o -o VirtualSerial.elf -lVirtualSerial
```

The USB interface is initialized with `CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);` while the watchdog and the clock speed are configured with `SetupHardware()` (read the source code at `VirtualSerial_lib/Virtu`
Finally, the three lines

```
CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
USB_USBTask();
```

in the infinite loop handle the messages coming from the PC to the microcontroller. Removing these lines will not prevent the communication from microcontroller to PC, but too many characters coming from the PC will fill the USB input buffer and crash the firwmare.
Exercise : complement the message displayed by default with the size of the three integer types handled by the C language.

Références

[1] J.-M Friedt, S. Guinot, *Programmation et interfaçage d'un microcontroleur par USB sous linux : le 68HC908JB8*, GNU/Linux Magazine France, Hors Série **23** (November/Décembre 2005)