Morse-based Twitter
An example of real time application for Embedded Systems

The "Morse-based twitter" is an example of a real time application for Embedded Systems. The behaviour of this application is implemented using TrampolineOS, an open-source implementation of the OSEK/VDX real time operating system. TrampolineOS supports different kind of real hardware solutions and, for this situation, I choose to develop, run and test it on an Arduino Nano (that has on board the same microcontroller of the Arduino UNO, the one officially supported by this implementation of the RTOS).

Because of this choice (regarding the work on a real hardware and not only via software), the source code contains some elements that are hardware-dependent. I had, in fact, to manage LEDs and there are some serial output strings for debugging purpose.

An overview of the implemented architecture

In order to satisfy correctly the requested behaviour of the application, the first thing to do is to organize a suitable architecture that can describe it correctly.

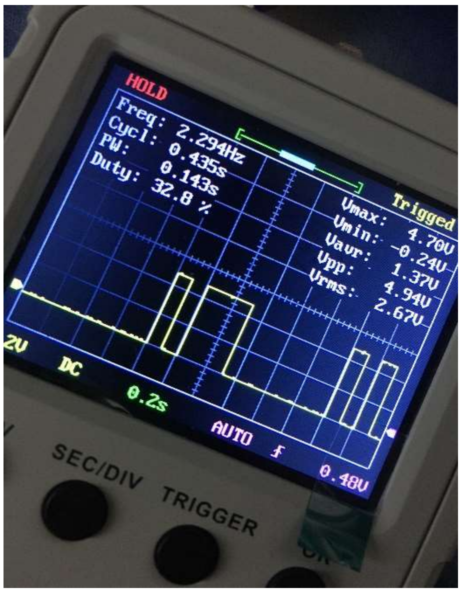
Thanks to the OIL file (that in an OSEK/VDX RTOS it's used as preliminary configuration of the Operating System), I can setup the general architecture of the implementation. The most important elements specified in this file are one Task, two Alarms and an Event.

The single Task is:

- TaskTwitterer: it is an endless-loop tasks that manages inter-messages pause and the right message to display. For each message, it calls a function that manages the selected message such as conversion to morse code, display timing, bit timing, inter-bit pauses and so on and so forth.
  It continuously checks if the end flag is set or if it reached the end of the message. If it is, the control is returned to TaskTwitterer.

The two Alarms are:

- StopDisplay: after 180 seconds since its activation, a callback is called whose action is to properly set the end flag variable.
- GeneralOneShot: It's used both for bit timing (rising an event each 100 ms) and for inter-message pause long 0.5 seconds. It fires the event EVTGeneralOneShot.



i) How the led output looks like at the oscilloscope. In the figure we can see the beginning of the first sentence: the letter 'A' followed by the beginning of 'F'.

The source code is divided into different parts:

- code.cpp: contains routines for setting up the system like serial port baud rate, the pin to use in order to drive the led and so on and so forth. Declares the array of messages to display and other useful variables.
- task.cpp: contains the main task routine. Here we have the core of the application.
- callback.cpp: meant to store callbacks' code, in this project we have only one callback and its code is described here.
- comm_fnct.h: header file that contains all the main macros, type definitions, constant definitions and useful function declarations.
- comm_fnct.cpp: source code of other functions (different from tasks and callbacks code). Here we found the definition of the function for displaying the message, single bit displaying and the Morse code conversion utility.

A few details about Alarms and Counters

Regarding the management of the StopDisplay alarm, TrampolineOS for Arduino already contains a definition of a counter called SystemCounter, without the need to specify it in the OIL file. The problem with this thing is that the default maximum value of the counter is the maximum value representable on 16 bits.

In order to have a count of 180.000 ticks more or less (please refer directly to the source code in order to know exactly how much ticks are needed to count for 180 seconds), we need more than 16 bits of storage for the counter. A re-defined SystemCounter is specified with a maximum value on 4 bytes.

Another important aspect to take into account is the duration of a tick. In fact, according to TrampolineOS documentation, for an Alarm based on SystemCounter **a duration of 1000 ticks is equal to 1024 ms**. For this way, a properly manual conversion needs to be made in order to set correctly the various constant values described in the source code.

This is due to the use of Timer0_OVF interrupt, generated by the main Atmega 328p's timer. By default, it's set with a prescaler of 64. The internal clock is 16 Mhz and Timer0 is based on a register of 8 bits. This means that we count up from 0 to 255 with a frequency of 250 Khz. Doing a few calculations, we discover that the frequency of the interrupt is about 976.6 Hz, means that every 1024 us we have a Timer0_OVF interrupt. It's better to not change this configuration because it's the same timer used by other predefined Arduino function like millis() and micros().

Task synchronization and events timing

The task synchronization and all the application timing are accomplished through the use of Events or Call-backs set by Alarms. In order to have a correct timing and avoid time drifts, all the Alarms that manage those Events are restarted before taking important actions. Using micros() to measure delays with a precision of 4 us, is possible to compute **relative errors in respect of the specifications** of the application. In the 100 ms of bit timing, there is a mean value error of 0.35%. In the 500 ms pause, there is an error of 0.06%. In the 180 s pause, there is an error of 0.0175%.

The serial port

The serial port is configured, so it's possible to use it in order to give a look into the actions that the program is doing. This is mainly used for debug purpose and to check if the system is working correctly. To enable It, please add the option "*CFLAGS = "-D DEBUG"*" in the OIL file. Please note that with this option enabled, the overall size of the program will be larger.

Memory occupation

The compiled program is not too much big. From a static analysis**, the code is large about 7.59 KB and data** (that is loaded into RAM during the boot) **is large about 0.89 KB** (this can be different depends on the compiler used or if there is a change on the source code but approximately this is the correct value). The code is stored in the Arduino UNO's 32 KB flash memory.

To be more precise, 7124 bytes are occupied by the .text segment, 652 bytes are occupied by the .data segment, and 213 bytes by the .bss segments. The Program size is computed as .text + .data, while the Data size is computed as .data + .bss.

A particular attention was demanded to RAM occupation (only 2 KB of SRAM are available in the Arduino UNO) during the write of the source code. In fact, all the variables used are of the minimum needed size (this is accomplished **using the stdint.h library** which defines the exact byte size of each variable type).

A look-up table is used to encode a character into Morse code. **All the character translations are stored as bits** (0b10 for a dot, 0b1110 for a dash) with a NRZ encoding. They are stored as stream of bits in a uint16_t variable for each one, and they are terminated by a 1. The biggest encoded character is 15-bit long +1 as termination.