

Embedded electronics exam

J.-M Friedt, March 18, 2022

1 Problem (2 points/answer)

Correlations, the mathematical operation of searching for a pattern x in a noisy dataset y , are ubiquitous in most daily activities, whether for decoding CDMA messages as transmitted by GPS satellites or when watching television requiring synchronizing with the preamble of the DVB-T protocol. Hence, our interest here is to investigate correlations implemented on a microcontroller. x and y can be either real or complex (e.g. output of an IQ detector) time series.

A cross-correlation $xcorr(x, y)$ between two datasets x and y is computed, thanks to the convolution theorem and the close relationship between convolution and correlation, as

$$xcorr(x, y)_n = \underbrace{\sum_{k=0}^{N-1} x_k \cdot y_{k+n}^*}_{\text{time domain}} = iFT(\underbrace{FT(x) \cdot (FT(y))}_{\text{frequency domain}})$$

with FT the Fourier transform and iFT the inverse Fourier transform. The cross-correlation is a matched filter used to find the pattern x (e.g. a transmitted signal in a RADAR system) in the measured sequence y (e.g. the observed signal including time-delayed target reflections). The objective is to demonstrate cross-correlation calculation using FreeRTOS. A sample program is provided at https://github.com/jmfriedt/tp_freertos in the `8_xcorr_fft` sub-directory.

1. Fetch the archive from https://github.com/jmfriedt/tp_freertos so that your local copy holds the whole history of the repository, allowing in the future to submit corrections if errors are found in the project. How did you fetch the files to achieve such a result?
2. Compile the project which links against the stable version of FreeRTOS found at <https://www.freertos.org/a00104.html> and execute in the **official** qemu emulation for ARM processors named `qemu-system-arm` (found in `/usr/bin` to be sure the right emulator is used). A `qemu` method is provided in the Makefile to make this task easier. Demonstrate that the project is compiled and executes properly.
3. Investigate the software and describe what mechanism is used to schedule the Fourier transform, multiply the pattern with the noisy signal, and compute the inverse Fourier transform of the result. How did we make sure that the inverse Fourier transform is only computed after the multiplication was completed, which itself is only computed after completing the Fourier transforms?
4. The output of the display is hardly usable as is. Explain why, and implement the mechanism that allows for making sure that the displayed results are not mixed up.
5. Two codes `c1` and `c2` are considered, one of which has been transmitted and received in the noisy message `meas`. Although all these data provided in `data.h` are real valued, an interleaved real part/imaginary part is needed to run the FFT provided by CMSIS, the Common Microcontroller Software Interface Standard providing several Digital Signal Processing libraries. By plotting the output of both correlations $xcorr(c1, meas)$ and $xcorr(c2, meas)$, which code was most probably transmitted and received in `meas`?
6. How much memory is available on the used processor? How much is actually allocated to FreeRTOS? Justify both answers.
7. Add the function for displaying the amount of memory used by each task. Is the stack allocated to each task wisely selected? How much memory is left, if any, to each task?
8. The Fourier transform implemented in the CMSIS library can be either used with floating point number, fixed point Q31 or fixed point Q15 (where Q31 or Q15 is equivalent to Q1.31 or Q1.15 fixed point notation). Considering two variables `a` and `b` are encoded as two 32-bit integers, how do you implement the fixed point multiplication in Q31 representation? Is this operation properly implemented in the proposed software?
9. Why is Q15 or Q31 favored over floating point arithmetic on low power microcontrollers, even when running FreeRTOS? Is this argument valid on the platform used in this demonstration?
10. The proposed example implements sur cross-correlation in the Fourier domain benefiting from the Fast Fourier Transform (FFT), but the mathematical operation can also be implemented in the time domain according to its definition given above. Implement such a function, making sure to use ring buffers to represent x and y provided as arguments to a function `xcorr_time(int32_t *x, int32_t *y, int n)` with `n` the number of elements. This ring buffer implementation means that when `k+n` exceeds the length of the array, the sum continues with the first elements considering that each dataset repeats periodically. Compare your implementation in the time domain with the calculation resulting from the frequency domain analysis. Notice that the cross-correlation of a real valued signal is a real function, so that the interleaved real/imaginary representation needed to run a FFT might not be best suited for the time-domain implementation.

2 Questions (1 point/answer)

11. What microprocessor peripheral might prevent reaching a register through the address provided in the datasheet?
12. What functions, in user and/or kernel space, circumvent this issue: what are the input and output arguments of these functions?
13. Where is the compiler targeting the embedded board located in the Buildroot tree structure?
14. How do you add this entry to the system's variable defining all the locations where executables might be located, assuming the `buildroot` install is located at the top level of the `myuser` user home directory?
15. Name a few benefits of running an operating system such as GNU/Linux on an embedded board, and a few drawbacks. Classify clearly the list between positive and negative aspects.

Negative grades if copies from Google/web search irrelevant to the questions are provided as answers.

Answers

1. `git clone --recursive https://github.com/jmfriedt/tp_freertos` as provided in the README to remind of the `--recursive`, followed by tuning the paths to the FreeRTOS source code and `libopenm3` to resolve header and libraries.
2. `make qemu` must display something like

```
0x00693A5A 0x00467499
0xFFEB18DE 0x001F4682
0xFFD7E17E 0x0077C425
0xFFDD2614 0x0005E9FB
0x003E1663 0xFF5ACC13
0x000296disp=0x00000001
0x00000000 0x00000000
0xFFFFF966 0xFFFFEC14
0x00008356 0xFFFFF081
0x00001F80 0x0000117A
0x00013C7F 0xFFFFE284
0x0000208B 0x00000627
...
0x00000C94 0xFFFFEEC5
0x00000D26 0x000007EA
52 0x00663981
0x0000919C 0xFFFFB439
0x000006A9 0x0000043D
0x00006C5C 0x0000045F
```

with most of the solution displayed as two columns of hexadecimal represented values.

3. The correct sequence is provided by communication through **queues**, since the complex conjugate multiplication can only take place after receiving the three FFTs, and the inverse FFTs can only be computed when the multiplication was completed and its results communicated through another queue.
4. The dump of the second question was focused on two display abnormalities, before the “...” hiding some lengthy display. Indeed the `do_display()` is very long to execute and chances are that the scheduler will preempt another task while displaying, mixing the output of various tasks. The mechanism we considered for making sure only a single task accesses the communication resource is a **single** mutex shared by all display functions. This way, only a single function can access the communication peripheral until it completes its task and lets another instance of `do_display()` run

```
void do_display(int32_t *i, int n)
{
    int k;
    char c[25];
    uart_puts("disp=\0"); hex(i[2*N], c); uart_puts(c); uart_puts("\n\0");
    xSemaphoreTake( xMutex, portMAX_DELAY );
    for (k=0; k<n; k+=2)
        { cpl(i[k], i[k+1], c); // {hex(i[k], &c[2]);
          uart_puts(c); uart_puts("\n\0");
        }
    xSemaphoreGive( xMutex );
}
```

5. computing the cross-correlation between the measurement `meas` and each code `c1` and `c2` provided either in `data.h` or on the github repository, for example using Octave’s `xcorr` function, exhibits a sharp correlation peak with `c1` and noise with `c2` (see picture on the github project). Hence, `c1` was the message transmitted in `meas`.

6. The two answers are one the one hand found in the linker script `ld/stellaris.ld` as provided as option of the `-T` flag of `arm-none-eabi-gcc` in the Makefile to see that ram (rwx) : `ORIGIN = 0x20000000, LENGTH = 64K`, and then in `src/FreeRTOSConfig.h` we find

```
#define configTOTAL_HEAP_SIZE ( ( size_t ) ( 49000 ) )
```

7. the `vTaskList()` call lists all executed function and remaining memory. Its output looks like

```
ps          X          1          908         6
IDLE       R          0          54          7
fftm       B          2          978         3
fft1       B          1          978         1
mul        B          1          970         4
ifft       B          1          970         5
fft2       B          3          978         2
```

with a bit less than 1 KB left for each task out of the initial 4 KB. Executing `vTaskList` requires enabling

```
#define configUSE_TRACE_FACILITY 1
#define configUSE_STATS_FORMATTING_FUNCTIONS 1
```

in `src/FreeRTOSConfig.h` and making sure to `make clean` to recompile FreeRTOS with the new options before `make`

- The fixed point multiplication **requires** casting to 64-bit values the intermediate operation of multiplying two integers before shifting the result by 31 positions to the right to align properly the fixed point. This is properly implemented as

```
int64_t tmp1, tmp2;
tmp1=(int64_t)c1[k]*(int64_t)meas[k]+(int64_t)c1[k+1]*(int64_t)meas[k+1]; // x.*conj(c1)
tmp2=-(int64_t)c1[k+1]*(int64_t)meas[k]+(int64_t)c1[k]*(int64_t)meas[k+1]; // x.*conj(c1)
c1[k]=(tmp1>>31);
c1[k+1]=(tmp2>>31);
```

- Fixed point arithmetic is favored on embedded systems for their efficiency over floating point operations offer emulated with software. In the case of this Stellaris processor, the use of floating point numbers could be justified since a floating point unit (FPU) is available, but somehow the QEMU emulator did not allow executing FPU access from FreeRTOS so we still ended up using integer arithmetic.
- The time domain cross-correlation using a ring buffer is implemented as

```
void xcorr(int32_t *x, int32_t *y, int n)
{
    int ki, ko;
    int32_t *xc;
    int64_t sum;
    xc=(int32_t*)malloc(n*sizeof(int32_t));
    for (ko=0;ko<n;ko++)
        {sum=0;
        for (ki=0;ki<n;ki++)
            {sum+=(int64_t)x[ki]*(int64_t)y[(ki+ko)%n];}
        xc[ko]=(sum>>31);
        }
    for (ko=0;ko<n;ko++) x[ko]=xc[ko];
}
```

with two loops, the inner one for the integral and the outer one for the time offsets.

- MMU or Memory Management Unit
- `mmap` in user space or `ioremap` in kernel space
- `output/host/usr/bin` with `output` where all Buildroot outputs are located, `host` because the cross-compiler is executed on the host, and `bin` where the executables are located.
- `PATH=$PATH:/my/buildroot/output/host/usr/bin`
- Benefits: filesystem, networking stack, memory management and scheduling, consistent design with applications running on PC (code re-use) at the expense of additional maintenance burden over the long term, additional resources and a new API to learn.