Wireless Networking

FFT operation with an intermittent power-source

Thijmen Ketel - 4623258
y.m.t.ketel@student.tudelft.nl

Carlo Delle Donne - 4624718
c.delledonne@student.tudelft.nl

Dimitris Patoukas - 4625943
d.patoukas@student.tudelft.nl

March 30th, 2017

Contents

1 Introduction

The advancements in technology call for devices to be smaller, faster and less power-hungry. In this sense, devices are being developed that do not even need charging in a classical way. These devices can harvest energy from the environment by, for example, a small solar-panel or RF electromagnetic waves. Some of these are already being used today in for example NFC-chips in bank cards.

The technological challenge in this can be made more complex with the addition of larger and more extensive calculations. Most of these devices need to work with power-sources that are not stable. These intermittent sources cause devices to operate in a different way. The uncertainty of operation that is introduced with a fluctuating power-source needs to be resolved by a different way of coding. Challenges range from data-consistency to speed to energy consumption.

In this project a program will be developed that is able to perform a *Fast Fourier Transform* (FFT), or a slimmed down form of FFT to reduce calculations and complexity, while dealing with an intermittent power-source. This has to be implemented on a MSP430 platform from Texas Instruments and has to be written in the C programming language so it can be easily ported to different platforms.

The group consists of three students from the MSc Embedded Systems, from three different countries (Netherlands, Italy and Greece). This mixture of backgrounds and educational levels can provide different views on how to approach the problem at hand.

2 First steps

In this chapter the first steps that are taken will be laid out. This includes the plans that are made, the approach that is taken by the team and the first implementations on real-world hardware.

2.1 Hardware: Texas Instruments MSP430

The very first step that was taken was the selection of the hardware that would be used, namely a micro-controller from Texas Instruments: the MSP430FR5994 [1]. This platform, as is clear in the name, is also used in the prototypes that are made at the Embedded Software (ES) department at TU Delft. This means that the code for the development kit can easily be ported to the platform used at ES-department.

2.2 Organisation

To create a correct and clear organisation for the project, multiple services are used:

- **GitHub**
  For code organisation and version control, GitHub will be used. A public repository is created hold the code and review additions from each of the team members. Data-sheets will also be kept on the repository to keep them available for reference.
- **WhatsApp & Slack**
  For communication, platforms as WhatsApp and Slack will be used. WhatsApp mainly for communication between the team members and Slack is meant for communication that is public for the rest of the class.
- **ShareLaTeX**
  The report of the project needs to be written with the help of LaTeX for concurrent and neat formatting. To keep this easy to use for every team member an online editer is used: ShareLaTeX. This editor is able to be used by multiple members at the same time and packages for formatting do not have to be downloaded.

Furthermore, meetings between the team members will be documented in a GitHub Markdown file for easy and simple formatting. The important parts will be transferred to the LaTeX document.

2.3 Discrete Fourier Transform

The next step to take is to implement a common *Discrete Fourier Transform* (DFT) on the development kit to get a feeling of the process [3]. The DFT is meant as a simple (in terms of math) operation so that the essence of the program is executed and the program can be reworked to an intermittent redundant system.

2.3.1 Signal creation

To implement the DFT, the first thing needed is a representation of an input signal, which we chose to be a combination of multiple sine waves. Because it is the discrete transform, the signal is represented by an *array* which holds a value for every time-step. The first problem that is encountered is the lack of a floating point unit (FPU) in the MSP430FR5994. This means either that floating point operations need to be handled by software (with libraries), resulting in a larger overhead, or that fixed point numbers should be used.

Fortunately, the MSP430FR5994 does have a *Low-Energy Accelerator* (LEA) which can be used for many mathematical operations and DSP-like applications. This LEA allows faster multiplications and even includes some *Fast Fourier Transform* utilities.

The way fixed points are handled on the MSP430FR5994 is with the help of two libraries that takes care of the representations and calculations of fixed point variables: the *IQmathLib* library and the *DSPLib* library. The way this is done can be configured by specifying the amount of bits that are be used for the decimals of the fixed point variable. In this way magnitude of variables can be sacrificed to gain precision in calculations [4].

2.3.2 DFT

Now that the input signal has been created, the implementation of the DFT can start. The code found in [3] is simple C-code but because it uses multiple floating point operations is has to be adapted to be compatible with the MSP430FR5994.

To do this, library documentation needs to be read and some simple functions to port the code will be used. The following decisions are made concerning the DFT:

- To conserve accuracy the decision is made to set the "fractional part" bits to 15. This means that numbers between -1 and 1 are allowed with an accuracy of 0.00003051758 (this can of course be changed later on);
- To calculate the magnitude of the Fourier coefficients the multiplications can cause an overflow. To counteract this problem the signal is scaled, and it is converted back after such operations;
- To decrease overhead as much as possible, the code will use many API function to perform standard mathematical operations. These functions would otherwise introduce large amounts of overhead due to the floating point operations.

When the code is ported and tested, the created input signal can be run through the DFT code to generate a spectrum. The code for this will be available on the repository. To keep the computation time relatively short, the essential runs are done with the following settings:

- The amount of samples (time-steps) that are given to the DFT is 32.
- The sample frequency is 32 *Hz*.
- Due to the chosen sample frequency, only input frequencies lower than 16 *Hz* are allowed for accurate processing. In this case frequencies of 2 *Hz* and 10 *Hz* are chosen.
- To keep the samples within the allowed range they are scaled down $\frac{1}{32}$ times their original value. This can be corrected after processing due to the linearity property of the Fourier transform.

Even though the DFT is applied on a relatively small sample size, low signal frequency and with use of the LEA, the resulted execution time is almost 5 seconds. To decrease this time the clock-speed of the micro-controller might be increased later on, for now this will be sufficient. The resulted discrete Fourier coefficients are stored in an array which the Texas Instruments Code Composer Studio IDE can form into a graph, which is shown in figure 1.

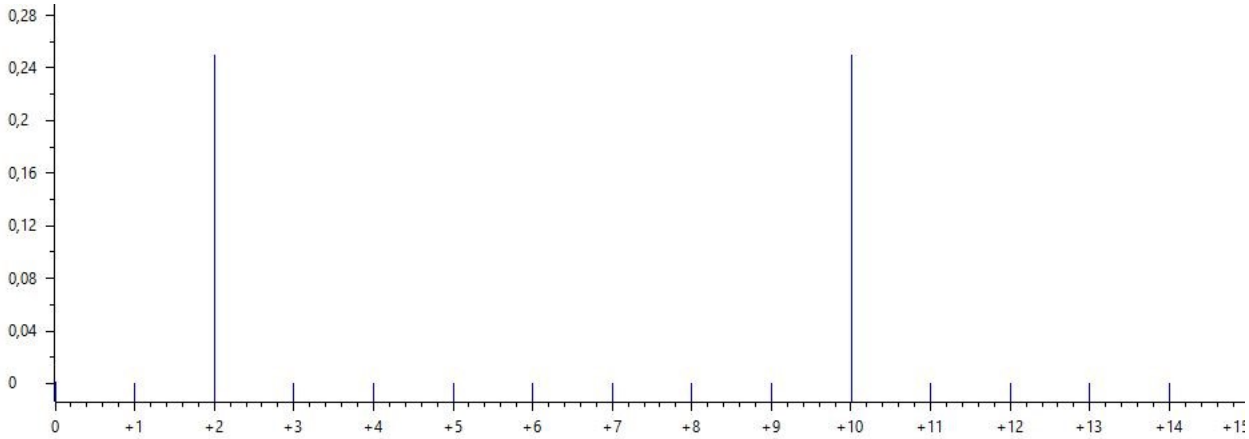

Figure 1: Discrete-Fourier Transform from the generated signal in section 2.3.1. The signal was created with two sinusoidal waves, one with a frequency of `2 Hz` and one with a frequency of `10 Hz`. This is visible from the spikes displayed in the graph (this picture is a screen-grab from the Code Composer Studio IDE)

These first steps are completed with the successful execution of the Discrete Fourier Transform (with maybe some room for optimisation). The next steps will be creating a first solution for operation under an intermittent energy-source condition. After which the MSP430FR5994 will be powered through another embedded platform which can simulate a randomly intermittent energy-source to test the proposed solution. In the next section the approach to create a solution will be laid out together with explanations of systems from which inspirations might be drawn.

3 Intermittent operation

The next step is to find a way to make the system redundant for intermittent operation. This means that the system can ensure data integrity even though it is rebooted during run-time. The system will need to be able to save its state and after a reboot check if a state has been saved and then reload that specific state. This will require an advanced way of writing and reading the current system state.

3.1 Mementos

One way to handle operations under an intermittent energy-source is to apply the *Mementos* pattern to the code [5] (which of course is a small nod to Christopher Nolan's *Memento*). This software design pattern is developed to perform a dynamic check-pointing operation on the program during run-time. The system is able to sense the available energy that is left in the system and based on that, decide to checkpoint the program or continue computation. In case of a checkpoint the systems state (or only the essential data) will be stored in non-volatile memory. After a reboot the program can load the data from memory and continue its operation.

The benefits of a dynamic system like Mementos is that state saves and read/write overhead is reduced to a minimum due to the evaluation of available energy. Checkpoints can be introduced on only the most important parts of the program and less energy will be wasted by unnecessary read/write operations.

The drawbacks however do have a larger implication. Hardware support is needed to keep track of the available energy in the system, which will create extra overhead to deal with. This makes implementation of Mementos harder on the large scale as many micro-controllers and applications do not always have the hardware support needed or are able deal with the overhead that is added.

Due to the fact that Mementos is one of the first methods that offer a redundant way of executing code with an intermittent energy-source, it is not suited for this project.

3.2 DINO (Death Is Not an Option)

Another advanced method that can deal with operation under an intermittent energy-source is *DINO* (Death Is Not an Option) [2]. This is a coding style and compiler addition that uses, unlike Mementos, a static check-pointing system. The systems is a completely software-based solution, removing the need for hardware support. This makes it able to be implemented on almost any kind of platform.

With DINO, the programmer sets *task boundaries* to split the main program into sub-sections. *Tasks* are operations between two task boundaries, and they are dynamically defined by the execution flow. At a task boundary the program state is guaranteed to be consistent with the code executed thus far. Failures can occur, but in such cases the program restarts from the last consistent task boundary, making DINO a resilient and reliable approach.

3.3 Chain

The third and final method that is discussed here, is *Chain*. This method is the most advanced and most recent method for correct intermittent behaviour. [6] The basic idea of the Chain method involves the creation of *tasks* by the user that compute data and then exchange this through *channels*. A task is restartable and never sees inconsistent state, because its input and output channels are separated.

The program is written as a collection of tasks and each task is free to define task-local volatile variables and access peripherals. The task construction prescribes that each task has at least one *predecessor* task and at least one *successor* task and one task marked as the *origin* task (task that executes when the device powers up for the first time).

While the tasks in the system do not have direct access to the non-volatile memory, the channels between the tasks do. This means that the channels are the only way task receive and transmit their data. A channel between two tasks consists of a single write and a single read. This means that data is consistent between tasks and that the tasks can compute as much as the programmer wants without data corruption in case of a reboot.

3.4 Proposed approach

Before any implementation is done on the MSP430FR5994, a short layout of the plan will be done (the implementation will be discussed in section 4). Because the *Chain* method is the most recent one, the approach will be largely based on that concept.

A *library* will be constructed that has methods and functions to define tasks within the program. Instead of the channels that are discussed in the paper, the library will just use so called *fields*. These fields can hold variables (integers, arrays, etc.) that are stored in non-volatile memory. Each pair of tasks (current and successor, or current and predecessor) has a field between them. This field can only be written by one task and only be read by the other. This means that if the system suffers a reboot while executing a task, no non-volatile data becomes corrupt.

The library can then be used to divide the main program into smaller tasks with fields that are consistent in between them. This leaves the challenge for the programmer to correctly choose the task borders for the application. Tasks can be made a certain size depending on the amount of instructions can be done in the average run-time.

4 Intermittent library

In this section, the efforts in creating a *library* that handles the task division and transfer of data in the system is described. This section is divided into three different part: *task-definition*, *field-definition* and *consistent-operation test*. Each section will cover a part of the development (and in the case of the last section, testing).

4.1 Task-definition

The main part of the method is the division of the program into separate tasks. This means in a practical sense that the normal sequence of code in the `main`-program needs to be divided into separate functions. These functions are then attached to a task the in the intermittent program. In code-listing 1 the task assignment macro is shown.

Listing 1: Macro to create a new task.

```
#define NewTask(NAME, FN, HAS_SELF_CHANNEL)                    \
        task NAME = {                                          \
            .task_function = FN,                               \
            .has_self_channel = HAS_SELF_CHANNEL,              \
            .dirty_in = 0                                      \
        };
```

This macro has three parameters:

- **NAME**: Parameter to give the task a name.
- **FN**: This parameter holds the function that is to be attached to the task.
- **HAS_SELF_CHANNEL**: This parameter can be set if the task requires a channel to itself (used for looping).

This macro mostly speaks for itself, except for the `HAS_SELF_CHANNEL` parameter. This parameter can be set if the task is not followed by a different task, but by itself. This will create a special field which will handle the data writes and reads correctly, more on that in section 4.2.

After tasks are created, a task has to be selected that will be the starting task for when the program is executed for the first time. This can be done by another macro (see listing 2).

Listing 2: Macro to select the initial task.

```
#define InitialTask(TASK)                                      \
        static __program_state __prog_state = {                \
            .curr_task = &TASK,                                \
        };
```

This macro holds a parameter `TASK` for a particular task, the memory-address of that task will be given to the `.curr_task` variable. This variable will keep track of the most current task that has to be executed in the system.

This leaves one macro that is related to task definition in the library: task switching, this can be seen in listing 3.

Listing 3: Macro to switch to another task.

```
#define StartTask(TASK)                                        \
        start_task(&TASK, &__prog_state);
```

This macro mostly speaks for itself, it can be called by a task to issue the activation of the next task in line. This is done by calling the `start_task`-function. This function can be seen in listing 4.

Listing 4: Function to start a task.

```
void start_task(__task *t, __program_state *ps)
{
    t->dirty_in &= 0xFF;
    ps->curr_task = t;

    t->task_function();
}
```

This function handles the correct switching to a different task form another task. Some important and essential operations happen in this function:

- `t->dirty_in &= 0xFF`: This action is meant to "clean" a read/write check byte, more on this in section 4.2.
- `ps->curr_task = t`: This sets the current task of the program to the task that is to be executed.
- `t->task_function()`: This executes the function that is bound to the task that is to be executed.

The issue is of course that the program might be interrupted between the task switching, this means that the program needs to resume from the place where data was last consistent. This is handled by the `resume_program()`-function (see listing 5).

Listing 5: Function to resume the program from where the data was last consistent.

```
void resume_program(__program_state *ps)
{
    if (ps->curr_task->has_self_channel) {
        uint16_t d_h = (ps->curr_task->dirty_in >> 8);
        uint16_t d = (d_h << 8) + d_h;
        ps->curr_task->dirty_in ^= d;
    }

    ps->curr_task->task_function();
}
```

This function is responsible for one of the most essential parts of the library: resuming the program after reboot. This function consists of two parts: the `if`-statement and the `ps->curr_task->task_function()` operation. The `if`-statement concerns itself with the correct handling of so called "self-channels", more in this in section 4.2. The second part of this function makes sure that the program resumes from the last data-consistent state by executing `curr_task->task_function()` parameter of the `ps` (program state) structure. As was explained earlier, the `start_task()` function can change this `ps` structure to switch to a next task. Because this structure is saved in the micro-controllers' non-volatile memory, the program will always continue from the last saved, data-consistent state. How the data-consistency is achieved will be more clearly explained in section 4.2.

4.2 Field-definition

Data that needs to be preserved through reboots need to be stored in non-volatile memory. To make sure this happens in the correct way and not in the middle of a task, the library uses persistent *fields*. These field act as channels between the different tasks, of from a task to itself. A field can only be written by one task and only be read by another. This way the library can guarantee data-consistency in between different tasks. In the case of a so called *self-field* the library defines actually two fields which alter between a being written and being read. To elaborate on this process, some of the macro's and functions will be discussed.

The first macro to be discussed is related to the creation of a *persistent data field* between successive tasks: `NewField` (shown in listing 6).

Listing 6: Macro to create a persistent field between two successive tasks.

```
#define NewField(SRC, DST, NAME, TYPE, LEN)                    \
        TYPE __##SRC##DST##NAME[LEN] = {0};                    \
        __field __##SRC##DST##NAME##__ = {                     \
            .length = LEN,                                     \
            .base_addr = &__##SRC##DST##NAME                   \
        };
```

This macro holds the following parameters:

- **SRC**: This parameter holds the name of the source task. This task only has *write* permissions to the field.
- **DST**: This parameter holds the name of the destination task. This task only has *read* permissions to the field.
- **NAME**: This parameter holds the *name* that the field will be given.
- **TYPE**: This parameter holds the type of *variable* that will be stored in the field.
- **LEN**: This parameter holds the length of the field. If `LEN` is larger than one, the field will be an *array*.

The body of the macro might seem complicated on first glance, but the idea is quite simple. A variable will be created with the type that is given as a parameter, the name is a concatenation of the `SRC`, `DST` and the `NAME` parameters. As a standard the variable is defined as an *array* with length `LEN`, the compiler will define this as a single variable if the length is *one*. After that, a new structure is created to hold the created variable: `__field`. This structure will be given the same name as the field but with extra underscores to sign that it is only for internal use in the library. The length will be given as well as the memory-address of the field variable.

4.3 Consistent-operation testing

To test the constructed library for its data-consistent functionality, an intermittent energy-source for the MSP430FR5994 will be simulated. This will be done by using another micro-controller (NXP Mbed LPC1768 [7]) and a very slow PWM (Pulse Width Modulation) signal (period of about one second). The duty-cycle of the PWM signal will start at about 0.4% and will be varied by adding or subtracting a random percentage between a certain range from the duty-cycle. This way the long intervals between operation and short period of execution time is simulated.

5 Intermittent DFT implementation

Now that the library is created to support intermittent operation, the DFT program can be ported to be data persistent. This will be done in different steps: first the DFT program will be divided into different task on an abstract level, after that the creation of the program will be explained and lastly the intermittent DFT implementation will be tested for different levels of accuracy and execution-time.

5.1 Abstract task division

Coming soon...

5.2 Creating the program

Coming soon..

5.3 Testing the intermittent DFT implementation

Coming soon...

References

[1] MSP430FR5994 LaunchPad Development Kit. [Online]. Available: http://www.ti.com/tool/msp-exp430fr5994

[2] Brandon Lucia, Benjamin Ransford, *A simpler, safer programming and execution model for intermittent systems*, Newsletter ACM SIGPLAN Notices - PLDI '15, Volume 50, Issue 6, June 2015, Pages 575-585

[3] Simple DFT in C.[Online]. Available: https://batchloaf.wordpress.com/2013/12/07/simple-dft-in-c/

[4] Texas Instruments, MSP430 IQmathLib Users Guide version 01.10.00.05. January 19 2015

[5] Benjamin Ransford, Jacob Sorber, Kevin Fu, *Mementos: system support for long-running computation on RFID-scale devices*, Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, Pages 159-170

[6] Alexei Colin, Brandon Lucia, *Chain: tasks and channels for reliable intermittent programs*, OOPSLA 2016 Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Pages 514-530

[7] NXP mbed LPC1768 prototyping development board. [Online]. Available: https://developer.mbed.org/platforms/mbed-LPC1768/