POLITECNICO DI TORINO

Master degree course in Mechatronic Engineering

Final project work

Performance analysis of an embedded system

Supervisor:
Prof. Massimo Violante

Candidate:
Luigi Vicari

July 2018

To my family,
without them none of this
would have been possible

Abstract

The main purpose of this thesis is to perform the analysis of the performance of an embedded system. In the first part are discussed some general concepts regarding the debugging and tracing operations and why these are fundamental in real-time applications, followed by a brief overview of the hardware software tools that have been used for this thesis. In the second part will be presented how it is possible to perform basic debugging and tracing with the Lauterbach μTrace in Trace32 environment.
The major purpose of this thesis will be discussed on the third part, where have been developed two services that are able to set-up a complete debugging and tracing environment for a bare metal application and a μClinux application running on the TWR-K70F120M development board. Furthermore, in the last part of this thesis will be presented how these services works on a signal processing application, where different analysis will be performed.

Contents

Nomenclature

AHB Advanced High-performance Bus

CPU Central Processing Unit

DDR Double Data Rate

DWT Data Watchpoint and Trace

ECU Electronic Control Unit

ETB Embedded Trace Buffer

ETM Embedded Trace Macrocell

FFT Fast Fourier Transform

FIR Finite Impulse Response

GCC GNU Compiler Collection

GDB GNU Debugger

GUI Graphical Unit Interface

IEEE Institute of Electrical and Electronics Engineers

ISR Interrupt Service Routine

ITM Instrumentation Trace Macrocell

JTAG Joint Test Action Group

KDS Kinetis Design Studio

MAPBGA Molded Array Process Ball Grid Array

MCU MicroController Unit

MMU Memory Management Unit

MTB Micro Trace Buffer

OS Operating System

PC Program Counter

PC Program Counter

R/W Read/Write

RTT Real-Time Trace

SDRAM Synchronous Dynamic Random Access Memory

SLC Single Level Cell

SWD Serial Wire Debug

SWO Serial Wire Output

SWV Serial Wire Viewer

TPIU Trace Port Interface Unit

1

General concepts

In this chapter are presented basic concepts of software debugging and tracing, and a general overview of real-time applications. After standard ISO 26262 will be introduced. This standard is the main motivation for the development of this thesis because, since this standard prescribes to perform different tests while facing with the design of an automotive electronic component that as an impact for the safety of the driver an people around the driver, e.g., an automatic breaking system. In particular, ISO 26262 prescribes the usage of *Resource usage test* when talking about software unit testing. From this kind of requirements it is needed to make use of the technology that will be described in the following chapters.

1.1 Debugging and tracing

Debugging and tracing software applications is one of the final steps in a development project, and if the code reaches an high level of complexity, debugging can have an huge impact on the time to market.
There are two main categories of tools that allow to debug the application

- Hardware based debuggers
- Software based debuggers

Hardware based debuggers: makes use of dedicated hardware to access the target memory and processor. Since the debugging operation is not handled by software, it is possible to debug the application even at the bootstrap

or when the system crashes (*post-mortem debugging*). Breakpoints are handled by hardware, i.e., when a breakpoint is reached the whole system stops the execution and neither the kernel, neither other applications runs on the target. The debugger is able to access physically the memory over the complete address range.
This presents some disadvantages like the lose of synchronization and communication with the peripherals.

Software based debuggers: makes use of a standard interface to the target such as Ethernet or a serial line. In this case there is not the possibility to perform bootstrap debugging, neither post-mortem debugging.

1.2 Real time applications

A large number of applications, from biomedical to automotive, have real-time constraints.
Let's take as example the ECU (Electronic Control Unit) responsible for the engine management of a vehicle. Based on information acquired through sensors such as airflow meter, throttle sensor and so on, it is responsible to calculate different engine parameters (e.g., mass of fuel to be injected, injection time) at each engine cycle. Such a process is classified as *hard* real-time, since it requires that data must be computed before a precise deadline. As it is possible to deduce from this example, real-time systems must satisfy the following requirement: respond exactly to external events within a finite time, called **deadline**.
Real-time applications are classified in:

- **Hard real time:** missing a deadline of such applications can have catastrophic effects on the controlled environment.
- **Firm real time:** missing a deadline makes the result useless, but it does not cause serious damages to the controlled environment.
- **Soft real time:** missing a deadline deteriorates the performance of the system, but does not cause problems to it.

Real-times systems can be in some cases classified as **safety critical**, if they are involved in safety relevant applications, e.g. autonomous driving

applications, or **mission critical**, if the fault of the system does not lead to loss of human lives, but a huge waste of money, e.g. systems controlling the engine of a satellite.

1.3 ISO 26262

ISO 26262 is an international standard that addresses specifically the automotive industry. It must be strictly respected when addressing the design of electrical and electronic components that are safety-critical. In particular, ISO 26262 provides requirements that must be respected both for the hardware and software design in the whole development process. ISO 26262 focuses on the **functional safety** concept, i.e., assure the correct functionality of the item not only when systems operates correctly, but even managing situation where hardware failures or operator errors occurs. When addressing hazard analysis and risk assessment of electronic items, the standard cares only to physical injuries of people, i.e., driver and passenger of the car, and people that surrounds the vehicle, e.g., pedestrian, cyclists, and so on. The development process of the item can be seen as a V-shaped model, starting from the top level (system design), going down to the software implementation, and coming back to the top, through a chain of testing operations.

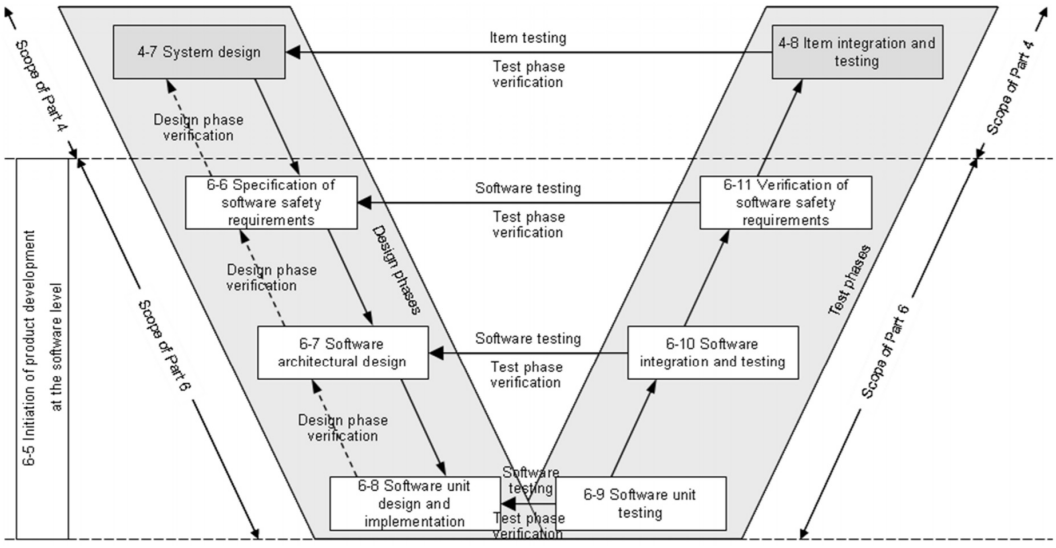

Figure 1.1: V-shaped model

1.3.1 ASIL determination

One of the first concept of ISO 26262 is to have different parameters for the hardware and software design of the item depending on the *safety level* that it is assigned. For such a reason, ISO 26262 establishes four different safety critical ASIL (Automotive Safety Integrity Level), from ASIL D (the most severe) to ASIL A (the less severe). To these is added the QM for which the item is classified as non safety-relevant.
To assign the ASIL to an item it is performed the *Hazard analysis and risk assessment*. This consist on the identification of the all possible relevant operational situation in which the vehicle can be involved, and evaluating in these situation the impact of possible hazards. For each combination of operational situation and hazard, the following characteristic are evaluated, based on the possible injury of people (harm):

- Controllability: possibility to avoid the harm by timely reaction of the user
- Exposure: probability to be in a certain operational situation
- Severity: measure of the harm that can be caused to people

By the combination of this elements, it is possible to assign the ASIL to the item.

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Figure 1.2: ASIL determination

1.3.2 Item development

During the item development, the standard prescribes which kind of test must be performed for each ASIL level.
Against hardware faults the following software checks are prescribed from the standard.

Table 4 — Mechanisms for error detection at the software architectural level

Methods		ASIL			
		A	B	C	D
1a	Range checks of input and output data	++	++	++	++
1b	Plausibility check[a]	+	+	+	++
1c	Detection of data errors[b]	+	+	+	+
1d	External monitoring facility[c]	o	+	+	++
1e	Control flow monitoring	o	+	++	++
1f	Diverse software design	o	o	+	++

[a] Plausibility checks can include using a reference model of the desired behaviour, assertion checks, or comparing signals from different sources.

[b] Types of methods that may be used to detect data errors include error detecting codes and multiple data storage.

[c] An external monitoring facility can be for example an ASIC or another software element performing a watchdog function.

Figure 1.3: Hardware fault tests

For unit testing the following tests are imposed by the standard.

Table 10 — Methods for software unit testing

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	+	++
1d	Resource usage test[c]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[d]	+	+	++	++

[a] The software requirements at the unit level are the basis for this requirements-based test.

[b] This includes injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

[c] Some aspects of the resource usage test can only be evaluated properly when the software unit tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[d] This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

Figure 1.4: Unit tests

It is possible to notice that for the unit testing it is recommended to perform the resource usage test for ASILs A, B and C, while it is mandatory for ASIL D applications. Resource usage test must be performed to check if the unit has requirements, in terms of computing capability, that is compatible with the hardware platform used. With this test must be checked as the usage of the memory and the execution time of the software. This type of test must be performed with powerful tools that are able to accurate measure execution times, and track the evolution of the software in time. This kind of operation is exactly what it is possible to do with the technology that is used for this thesis, and practical examples of measurement of execution times of functions will be discussed in the following chapters.

2

Hardware and software tools

This section presents an overview of the different technologies used to develop this thesis.
Hardware devices are the *Lauterbach µTrace*, a powerful device used for tracing the application, and the *TWR K70F120M* development board equipping a Cortex M4 µC.
The main software devices are: *Trace32*, as debug and tracing environment, and *Kinetis Design Studio* to develop bare metal applications.

2.1 Lauterbach µTrace

This device is a lower cost solution developed by *Lauterbach* that specifically targets the *Cortex-M* family.

Figure 2.1: Lauterbach µTrace

2.1.1 µTrace features and characteristics

Lauterbach µTrace is equipped with:

- USB 3.0 interface to the host computer
- 256MB trace memory
- 10, 20 or 34 pin half-size connector for target hardware

Furthermore, this device supports standard JTAG, SWD (Serial Wire Debug) and cJTAG (IEEE 1149.7 [1]), and it works in the voltage range 0.3V to 3.3V, but it is also tolerant to 5V inputs.
It provides different features for debugging and tracing the software running on the target hardware. It can be employed for C/C++ debugging, supports simple and complex breakpoints and memory read/write operations during the program execution. For trace operations, the device interacts with *CoreSight* components:

- ETM and ITM data over 4-bit TPIU in Continuous mode
- ITM over SWO (Serial Wire Output)

Lauterbach µTrace supports ETB (Embedded Trace Buffer) and MTB (Micro Trace Buffer). Combining ITM and ETM data allows the integration between R/W accesses and instruction flow informations.
It is possible to perform OS-aware tracing, code coverage analysis and energy measurements (using Trace32 Analog Probe).

2.1.2 Cortex-M CoreSight components

Here is presented an overview of the CoreSight components that implements trace support for Cortex-M chips. The architecture and the interaction with the Lauterbach µTrace device is reported in the figure below.
It is possible to notice from the figure that ITM is a regular memory-mapped peripheral for the CPU that is accessible through the AHB (Advanced High-performance Bus).
DWT (Data Watchpoint and Trace) unit and SWV are features that are implemented in Cortex-M3, Cortex-M4 and Cortex-M7.
For in depth informations about CoreSight components it is possible to consult the CoreSight reference manual [2].

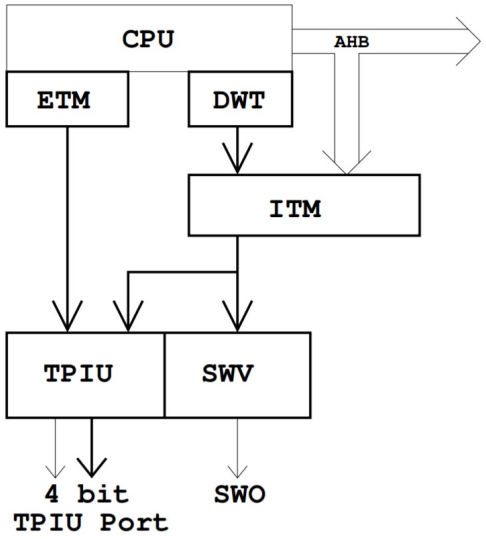

Figure 2.2: CoreSight Components

Embedded Trace Macrocell: ETM is an optional, simple component that can be connected to Cortex-M3, Cortex-M4 and Cortex-M7. It can only generate basic informations about the execution flow. In particular, Cortex-M ETM does not provide any support for data tracing and does not contain comparators to filter informations. Furthermore, it does not support cycle accurate tracing or ContextID tracing.

Data Watchpoint and Trace: DWT is an optional, more complex component that can be connected to Cortex-M3, Cortex-M4 and Cortex-M7. This component is able to monitor data accesses and the PC (Progam Counter) of the CPU. It contains a certain amount of comparators, that can be used to trigger different actions when a match occurs, it can send periodic informations about the PC, the ISR (Interrupt Service Routine) and data access. Furthermore, it is able to halt the CPU, trigger the ETM and count specific types of CPU cycles.

Instrumentation Trace Macrocell ITM is a component used by the DWT to send data to an external debug or trace tool. The ITM is seen by the CPU as a memory-mapped peripheral that contains a certain amount of

addresses. Through write operations in these addresses, the software is able to send data to the external debug and trace tool. To use ITM it is needed to modify the source code of the application.
ITM has 32 channels organized as shown in the following figure.

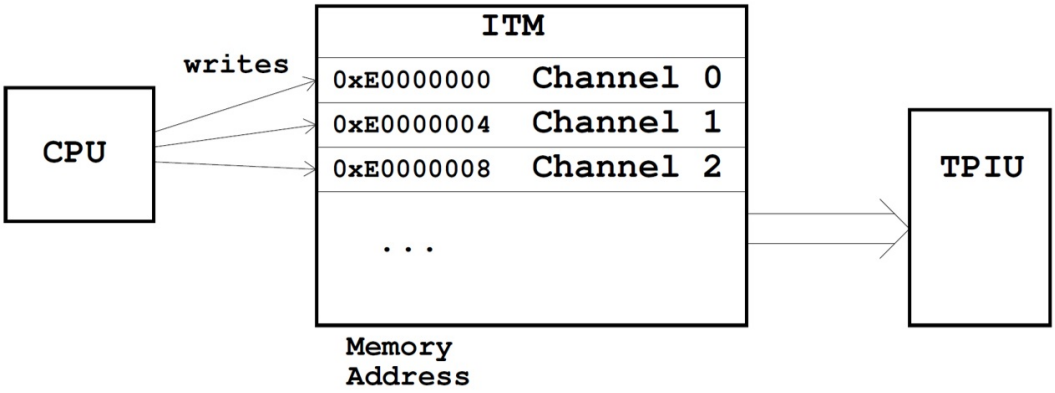

Figure 2.3: ITM Channels

Trace Port Interface Unit: TPIU is the item responsible to emit data collected from both ITM and ETM through pins of the chip. It is needed to select in which mode the TPIU must work between:

- **Trace Port mode**: TPIU uses up to 4 data pins and a clock to send synchronously data. ETM and ITM data are merged by means of the *Formatter protocol* and exported as a single stream of bytes.
- **SWV mode**: TPIU uses only one pin to export data through a single serial signal. The transmission is asynchronous. In this operational mode, only ITM data can be sent.

For this thesis, in order to have a more performing tracing, the first option for the TPIU has been used.

Embedded Trace Buffer: ETB is used to route exported data coming from ITM and ETM directly to the ETB instead of pins off chip. By means of debug connection, Trace32 is able to read ETB content.

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	TMS|TMSC|SWDIO
GND	3	4	TCK|TCKC|SWCLK
GND	5	6	TDO|-|SWO
GND (KEY)	-	8	TDI
GND	9	10	RESET-
GND	11	12	TRC CLK
GND	13	14	TRC DATA[0]
GND	15	16	TRC DATA[1]
GND	17	18	TRC DATA[2]
GND	19	20	TRC DATA[3]

Figure 2.4: 20 pin connector

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	TMS|TMSC|SWDIO
GND	3	4	TCK|TCKC|SWCLK
GND	5	6	TDO|-|SWO
GND (KEY)	-	8	TDI
GND	9	10	RESET-
GND	11	12	RTCK
GND	13	14	TRST- PULLDOWN
GND	15	16	TRST-
GND	17	18	DBGRQ (EMU0)
GND	19	20	DBGACK (EMU1)
GND	21	22	TRC CLK
GND	23	24	TRC DATA[0]
GND	25	26	TRC DATA[1]
GND	27	28	TRC DATA[2]
GND	29	30	TRC DATA[3]
GND	31	32	TRC EXT
GND	33	34	VREF-TRACE

Figure 2.5: 34 pin connector

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	TMS|TMSC|SWDIO
GND	3	4	TCK|TCKC|SWCLK
GND	5	6	TDO|- |SWO
GND (KEY)	-	8	TDI
GND	9	10	RESET-

Figure 2.6: 10 pin connector

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	TMS|TMSC|SWDIO
GND	3	4	TCK|TCKC|SWCLK
GND	5	6	TDOI-ISWO
GND (KEY)	-	8	TDI
GND	9	10	RESET-
GND	11	12	RTCK
GND	13	14	TRST- PULLDOWN
GND	15	16	TRST-
GND	17	18	DBGRQ (EMU0)
GND	19	20	DBGACK (EMU1)

Figure 2.7: 20 pin connector (SWO configuration)

Connectors: In the previous figures are reported connector types and configurations that are supported by Lauterbach µTrace.

2.2 TWR-K70F120M development board

TWR-K70F120M is a development board of NXP equipping a 32 bit ARM Cortex-M4 MCU.

Figure 2.8: TWR-K70F120M

The board presents the following features:

- MK70FN1M0VMJ12 core: 256 MAPBGA (Molded Array Process Ball Grid Array) at 120MHz
- On-board JTAG debug circuit (OSJTAG) with virtual serial port
- 1GB DDR2 SDRAM
- 2GB SLC NAND flash memory
- MMA8451Q 3-axis accelerometer
- 4 LEDs
- 4 Capacitive toch pads
- 2 Push button switches
- Potentiometer
- Battery holder for 20mm lithium battery
- Micro-SD card slot

The board is able to support μClinux kernel to build a Linux-based application and is equipped with a 20 pin JTAG interface that is used to connect the target hardware to the Lauterbach μTrace.
This is the main feature that led to the decision of using this development board.

Important note: By default, TWR-K70F120M is set to use the JTAG in SWV mode, but for this thesis the Trace Port mode is used. Therefore the following hardware modifications has to be done to the board:

- **Remove** R138
- **Remove** R11
- **Populate** R137 (0Ω)

If this hardware modifications are not performed, the scripts developed in this thesis will not work, because by default TRACE_CLKOUT signal is **not** connected to the debug connector (as it is reported in TWR-K70F120M User's manual [3]).

2.3 Kinetis Design Studio

KDS is an integrated development environment for Kinetis MCUs. It is based on open-source software: Eclipse, GCC (GNU Compiler Collection), GDB (GNU Debugger) and others.

2.3.1 Create and deploy bare metal project to target

A bare metal project is an application that runs on the target without the support of an OS. This kind of projects allows to not go through different layers of software.

Requirements: To be able to successfully deploy the application to TWR-K70F120M hardware, the latest version of PEMicro driver must be installed into the host.

Create the project: To create the project browse the *File* menu, and from *New*, select *Processor Expert Project*. *New Kinetis Project* window will appear. Choose a project name and select *Next*. Now from *Processor* select *Kinetis K*, *MK70*, *MK70F(120MHz,150MHz)*, and finally *MK70FN1M0xxx12*. Select *Finish*, and KDS will initialize the project.
In the *Sources* folder it is possible to find *main.c* where it is possible to write the user application.

Deploy the application: To deploy the application to TWR-K70F120M, it is possible to right-click the name of the project, and select *Debug As* and *Debug Configurations...*
The *Debug Configuration* window will appear. Expand *GDB PEMicro Interface Debugging* and select *<project_name>_Debug_PNE*.
By selecting *Debug*, KDS will deploy the application to the hardware, and at the end of the process it will open the new perspective from which it is possible to perform a software debugging of the application.

Micrium µC/OS: Micrium µC/OS-III operating system for TWR-K70F120M available in Micrium website [4] does not support KDS environment.

2.4 Trace32

Trace32 is the debugging environment developed by Lauterbach used for the development of this thesis. It provides all the standard debug features and gives access to advanced on-chip debug features. [5]

Debug features:

- JTAG, cJTAG, SWD debug interfaces
- Run control
- Flash programming
- Multi-core debugging
- OS support with task analysis
- HLL debugging

Trace features:

- Serial and parallel off-chip trace
- Non-intrusive flow trace
- Time-correlated multi-core trace
- Run-time analysis and statistics
- Long-time trace
- Code coverage

The system includes a logic analyser module that provides additional features as the protocol analyzer (CAN, I2C, etc..) or the energy profiling.

Important note: Lauterbach μTrace must be connected to the host before launching the application software. Alternatively it is possible to run the application in simulation mode. To do this, it is necessary to modify the content of *config.t32* in *T32* folder as follows:

```
PBI=SIM
```

2.5 μClinux

μClinux is an OS that includes 2.0, 2.4 and 2.6 Linux kernel releases. This release is intended for micro-controllers without MMUs (Memory Management Unit) [6].

2.5.1 μClinux application

From emcraft website [7] it is possible to download all the material required to build a μClinux application for the TWR-K70F120M development board. Here it is presented how to create and deploy a custom μClinux application [8].

GNU cross-build tools: After having unpacked the archive containing the software distribution, it is needed to install the GNU cross-build-tools. It is recommended to install these development tools into the `tools/` folder to avoid manual configuration of the `PATH`.

Build the application: To build a custom application it is possible to modify the existing *developer* project.
Firstable it is needed to perform the activation by going to the top of the Linux Cortex-M folder and running the following command

```
. ACTIVATE.sh
```

It is recommended to work on a copy of the *developer* project. To do this, move to the `projects/developer/` folder and then clone the project

```
make clone new=my_developer
```

Now in the `projects/` folder it is possible to find the project *my_developer*, where it is possible to write the custom application.
To perform the complete trace of the application, it is needed to build both the kernel and the application with debug symbols. It is possible to kernel symbols through the configuration menu of the kernel

```
make kmenuconfig
```

From *Kernel Hacking* it is possible to enable the setting *Compile the kernel with debug info*.
To compile the application with debug symbols, it is needed to add the `-g` flag

both to `CFLAGS` and `LDFLAGS` in the makefile located in `my_developer/app/` folder. Now it is possible to compile the whole project

```
make
```

Deploy the application: It is now possible to deploy the application to the board through the available virtual serial port by means of the kermit protocol. With the following script it is possible to load the *my_developer.uImage* into the target RAM.

```
#!/usr/local/bin/kermit

set port /dev/ttyACM0
set speed 115200
set carrier-watch off
set flow-control none
set prefixing all

echo {loading uImage}
PAUSE 1

OUTPUT loadb ${loadaddr} 115200\{13}
send my_developr.uImage
INPUT 180 {\{13}\{10}STM32F429-DISCO> }

IF FAIL STOP 1 INPUT timeout

echo {running kernel}
PAUSE 1
OUTPUT run addip; bootm\{13}
```

This script writes to the `loadaddr` of the device `ttyACM0` the `my_developer.uImage` binary file. After the file transfer, the script sends the uboot command `bootm` to start the kernel.

3

Trace32 debug and trace

In this chapter is presented how it is possible to debug an application with Trace32, without taking care of how to set-up the debug environment. [9]
Trace32 is a powerful environment that gives the possibility to debug the application running on a target by means of specific commands and/or by means of a GUI (Graphical User Interface). Furthermore, it is possible to completely automate the procedure, by creating a script containing multiple Trace32 commands.
Trace32 is equipped with a wide and detailed documentation that is possible to browse every time just by pressing F1 in Trace32 environment.

3.1 Trace32 commands

As shown in the figure, Trace32 environment is equipped with a command line where it is possible to insert commands to debug *on-the-fly* the user application. Below the command line it is possible to find the so called *soft-keys* that helps to enter a specific command step by step. Further informations, i.e. the target status and the debugging mode, are shown in the right part.

Figure 3.1: Trace32 Command Line

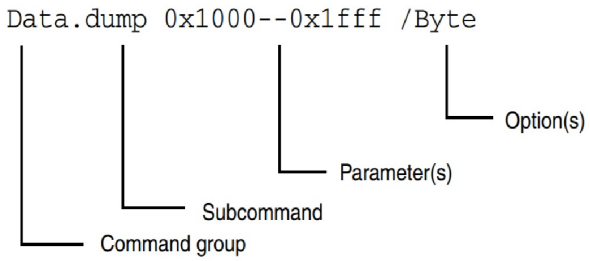

Figure 3.2: Trace32 command structure

A Trace32 command has a quite complex structure, in particular it is composed by the following elements:

- Command group
- Subcommand
- Parameter(s)
- Option(s)

The structure of a generic Trace32 command is shown in the picture.

Trace32 is not case-sensitive, but capital letters that are displayed in the softkeys are meaningful because in Trace32 there is the possibility to abbreviate the commands by means of the relevant letters, that are always written in upper case. For example, the following commands are equivalent

```
Data.List          d.l
Register.view      r
SYStem.Up          sys
```

By writing the command adding a blank and pressing F1, the documentation of the command will open.

Scripts: It is possible to assembly multiple Trace32 commands into a single .cmm file, executing the whole set of instructions by means of the command

```
DO <script_name> [parameter_list]
```

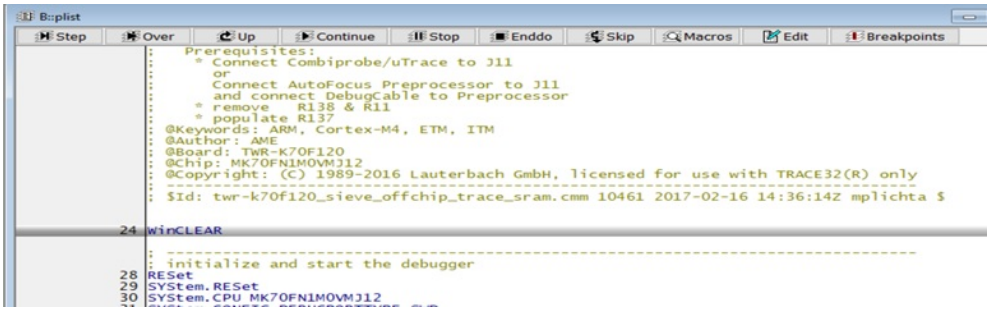

Figure 3.3: Single step script

If the script is not on the working directory, the path to the script (absolute or relative) must be inserted instead od the script name. To check the current working directory it is possible to run the command

```
pwd
```

It is possible to execute the script step by step by means of the commands

```
pstep                ;Enable the script single
                     ;step execution mode
DO <script_name>     ;Load the script that has
                     ;to be executed
plist                ;Show the script window
```

3.2 Source code

For the development of this section, three different applications have been used. The first one is a simple application that performs an infinite loop. Each step of the loop, the application waits for a certain amount of time, and immediately after waits for the double of the same time. After this, if a char variable named *debug_char* is stored the value *z*, the content is set to *a*, otherwise, the ASCII value is incremented. After, the variable *runtime_start* is set to 1, and an array of size *ARRAY_SIZE* is created. Later this array is duplicated, and two different sorting algorithms are performed (selection sort algorithm, with complexity $O(n^2)$, and the quick sort algorithm, with worst case complexity $O(n^2)$, but average complexity $O(n \log_2 n)$). Here follows the application code.

```
/*
 * Copyright (c) 2015, Freescale Semiconductor, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms,
 * with or without modification, are permitted provided
 * that the following conditions are met:
 *
 * o Redistributions of source code must retain the
 * above copyright notice, this list of conditions
 * and the following disclaimer.
 *
 * o Redistributions in binary form must reproduce
 * the above copyright notice, this list of conditions
 * and the following disclaimer in the documentation
 * and/or other materials provided with the
 * distribution.
 *
 * o Neither the name of Freescale Semiconductor, Inc.
 * nor the names of its contributors may be used to
 * endorse or promote products derived from this
 * software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
 * AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
 * SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
```

```
 * DAMAGE.
 */

#include "MK70F12.h"
#include <math.h>

#define  WAIT_CONSTANT 10000
#define  ARRAY_SIZE    2000
#define  MAX_NUMBER    10000
#define  PI            3.141592653589793

void   create_array(int array[], int dim, int num);
void   selection_sort(int array[], int dim);
void   quickSort(int array[], int begin, int end);
void   plots(float* sine, int* square);

int main(void)
{

        int    debug_int = 10;
        float  debug_float = 1.0;
        char   debug_char = 'a';
        int    i;
        int    array[ARRAY_SIZE];
        int    array_copy[ARRAY_SIZE];
        float  sine_wave;
        int    square_wave;
        int    runtime_start;
        int    runtime_stop;

        //INFINITE LOOP
    for (;;) {

        //Wait
        for(i = 0; i < WAIT_CONSTANT; i++){
                ;    //NOP
        }
```

```
        //Wait the double of time
        for(i = 0; i < 2*WAIT_CONSTANT; i++){
            ;   //NOP
        }

        if(debug_char == 'z'){
                debug_char = 'a';
        }
        else{
                debug_char++;
        }

        runtime_start = 1;
        create_array(array, ARRAY_SIZE,
                            (int)debug_char);

        for(i = 0; i < ARRAY_SIZE; i++){
                array_copy[i] = array[i];
        }

        selection_sort(array, ARRAY_SIZE);
        quickSort(array_copy, 0, ARRAY_SIZE-1);
        runtime_stop = 1;

        plots(&sine_wave, &square_wave);

    }

    return 0;
}

void   create_array(int array[], int dim, int num){
        int i;
        int tmp;

        for(i = 0; i < ARRAY_SIZE; i++){
                array[i] = (i*num) % MAX_NUMBER;
        }
```

```
        num = num+1;
        while(num < ARRAY_SIZE){
                tmp = array[num];
                array[num] = array[num-1];
                array[num-1] = tmp;
                num += 10;
        }

        return;
}

void    selection_sort(int array[], int dim){
        int tmp;
        int i;
        int j;
        int ind_min;
        int min;

        for(i = 0; i < dim; i++){
                min = array[i];
                ind_min = i;

                for(j = i; j < ARRAY_SIZE; j++){
                        if(array[j] < min){
                                min = array[j];
                                ind_min = j;
                        }
                }

                tmp = array[i];
                array[i] = array[ind_min];
                array[ind_min] = tmp;
        }

        return;
}
```

```
void   quickSort(int array[], int begin, int end){
        int pivot, l, r;
        int tmp;

        if (end > begin) {
           pivot = array[begin];
           l = begin + 1;
           r = end+1;
           while(l < r)
                  if (array[l] < pivot)
                         l++;
                  else {
                         r--;
                         tmp = array[l];
                         array[l] = array[r];
                         array[r] = tmp;
                  }
           l--;
           tmp = array[begin];
           array[begin] = array[l];
           array[l] = tmp;
           quickSort(array, begin, l);
           quickSort(array, r, end);
        }

        return;
}

void   plots(float* sine, int* square){
        float t;

        *square = 0;
        for(t = 0; t < 10*PI; t += 0.001){
                *sine = sin(t);
                if(*sine >= 0){
                        *square = 1;
                }
                else{
```

```
                        *square = 0;
                }
        }
        *square = 0;

        return;
}
```

The second application has been developed to give a quick example on the usage of ITM as debugging technique, and here follows the application source code. Even in this case an infinite loop is performed, where in each step, the value of three different counters i, j and k is incremented. At each step the code is instrumented to send in the channels 0, 1 and 2 the values of the counters, furthermore in channel 3 value 1 is set when the i counter is re-setted, while, the value 2 is set when the i counter reaches the value 10001.

```
/*
 * Copyright (c) 2015, Freescale Semiconductor, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms,
 * with or without modification, are permitted provided
 * that the following conditions are met:
 *
 * o Redistributions of source code must retain the
 * above copyright notice, this list of conditions
 * and the following disclaimer.
 *
 * o Redistributions in binary form must reproduce
 * the above copyright notice, this list of conditions
 * and the following disclaimer in the documentation
 * and/or other materials provided with the
 * distribution.
 *
 * o Neither the name of Freescale Semiconductor, Inc.
 * nor the names of its contributors may be used to
 * endorse or promote products derived from this
 * software without specific prior written permission.
 *
```

```
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
 * AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
 * SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 */

#include "MK70F12.h"

static volatile unsigned int *ITM_BASE_CH = (volatile
                              unsigned int*) 0xE0000000;

#define ITM_TRACE_D8(_channel_, _data_) {\
        volatile unsigned int *_ch_=ITM_BASE_CH+
                                        (_channel_); \
        while(*_ch_ == 0); \
        (*((volatile unsigned char *)(_ch_))) =
                                            (_data_); \
}

#define ITM_TRACE_D16(_channel_, _data_) {\
        volatile unsigned int *_ch_=ITM_BASE_CH+
                                        (_channel_); \
        while(*_ch_ == 0); \
        (*((volatile unsigned short *)(_ch_))) =
                                            (_data_); \
}
```

```
#define ITM_TRACE_D32(_channel_, _data_) {\
        volatile unsigned int *_ch_=ITM_BASE_CH+
                                        (_channel_); \
        while(*_ch_ == 0); \
        (*((volatile unsigned int *)(_ch_)))=
                                        (_data_); \
}

int main(void)
{
        short i = 0;
        int j = 0;
        char k = 0;

    /* Write your code here */

    /* This for loop should be replaced.
       By default this loop allows a single stepping.*/
    for (;;) {
        ITM_TRACE_D16(0,i);
        ITM_TRACE_D32(1,j);
        ITM_TRACE_D8(2,k);

        if(i==0){
                ITM_TRACE_D8(3,1);
        }

        j++;
        i++;
        k++;
        if(i>10000){
                i=0;
                ITM_TRACE_D8(3,2);
        }
        if(j>500000){
                j=0;
        }
```

```
        if(k>=255){
                k=0;
        }
    }
    /* Never leave main */

    return 0;
}
```

The third application is built upon a μClinux OS. It is just a modification of the original *developer* project provided as example. An infinite loop is performed, where at each step a sample device is read, echoing the content.

```
/*
 * This is a user-space application that reads
 * /dev/sample
 * and prints the read characters to stdout
 */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
   char * app_name = argv[0];
   char * dev_name = "/dev/sample";
   int ret = -1;
   int fd = -1;
   int c, x;
   while(1){

      /*
       * Open the sample device RD | WR
      */
```

```
        if ((fd = open(dev_name, O_RDWR)) < 0) {
            fprintf(stderr, "%s: unable to open %s:
             %s\n", app_name, dev_name, strerror(errno));
            goto Done;
        }

        /*
         * Read the sample device byte-by-byte
        */
        while (1) {
            if ((x = read(fd, &c, 1)) < 0) {
                fprintf(stderr, "%s: unable to read %s:
                      %s\n", app_name, dev_name,
                       strerror(errno));
                goto Done;
        }
        if (! x) break;

            /*
            * Print the read character to stdout
            */
            fprintf(stdout, "%c", c);
        }

        /*
        * If we are here, we have been successful
        */
        ret = 0;

        Done:
        if (fd >= 0) {
            close(fd);
        }
    }

    return ret;
}
```

3.3 Application debug

Here are presented some basic instructions needed to perform a simple debug of the application software, from displaying the source code, to the handling of breakpoints, and so on. For detailed information about commands it is possible to refer to the different general references reported in the *References* section.

3.3.1 Display source code

To display the source code of the application it is possible to use the *Data.List* command

```
Data.List <function_name> [parameters]
```

By using this command, a new window displaying the source code starting from the first line of the function will appear. The row is highlighted in grey if it is the current value of the PC. There are two different possibilities to display the source code in Trace32, i.e., displaying the high level language, or the corresponding Asm instructions. In Trace32, it is the possibility to choose between these modes by using the *Mode* command

```
Mode. <option>
```

It is possible to choose between the following options

- Hll: Displays only high level language instructions
- Asm: Displays only the Asm instructions
- Mix: Displays both high level language and corresponding Asm instructions

Furthermore, there is the possibility, to display the source code from the current instruction of the PC, by using the following command

```
Data.list
```

In the window are present buttons to perform the basic operations such as *Step*,*Over*,*Diverge*,*Return*,*Up*,*Go*, and *Break*. In the following page is reported a picture of how this window looks like.

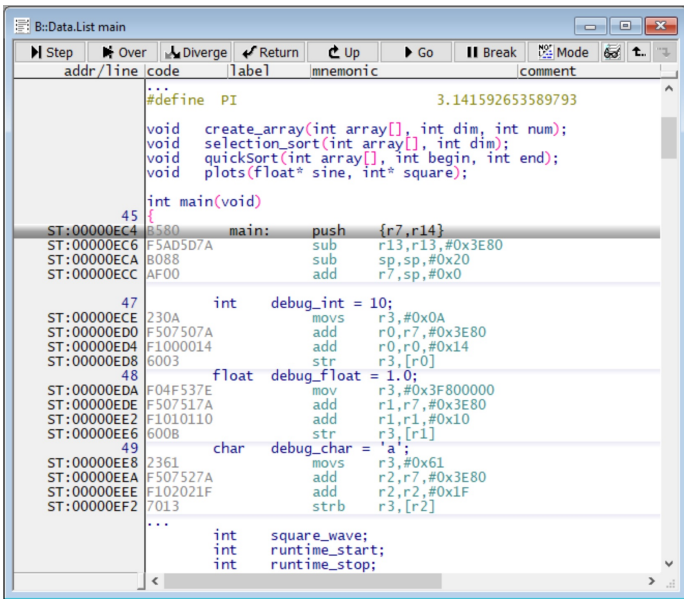

Figure 3.4: Source code in Trace32 (Mix mode)

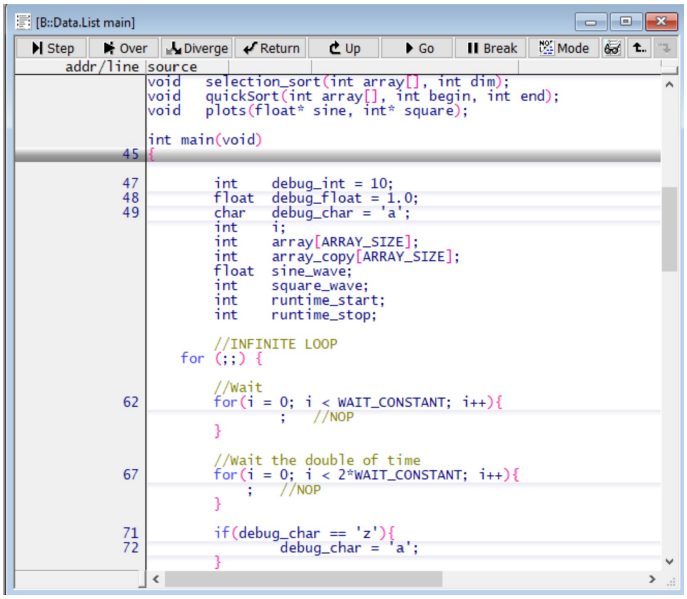

Figure 3.5: Source code in Trace32 (Hll mode)

3.3.2 Data, registers and peripherals

In Trace32 is possible to monitor and set at any time of data, CPU registers and peripherals. In particular it is possible to set data at run-time without generating extra load for the CPU if one of the following option is enabled in the system menu:

- CPU
- Nexus
- DAP

To access the system menu it is possible to run the command

```
SYStem
```

If more than one window is trying to access the memory, it is possible to enable the run-time memory access for all of them by enabling from the system menu the *DUALPORT* option, or running the following command

```
SYStem.Option DUALPORT ON
```

In Trace32 there is also the possibility to perform an intrusive access memory by selecting *CpuAccess Enable* mode from the system menu, or running the following command

```
SYStem.CpuAccess Enable
```

In this case, the real-time behaviour of the target is compromised because the debugger has access to the CPU resources.
To display the contents of the CPU registers it is possible to use the *Register* command, in particular the *view* subcommand

```
Register.view [/<options>]
```

Among all the possible options, the */SpotLight* options gives the capability of highlight the registers, which values have been modified during the last operations of the CPU, in particular, darker is the color, later in time happened the modification.
It is possible to modify the value of a register, by using the *Set* subcommand

```
Register.Set <register> <value>
```

Peripherals are really similar to registers, and in the same way it is possible to display and set the content of the registers by means of the *PER* command

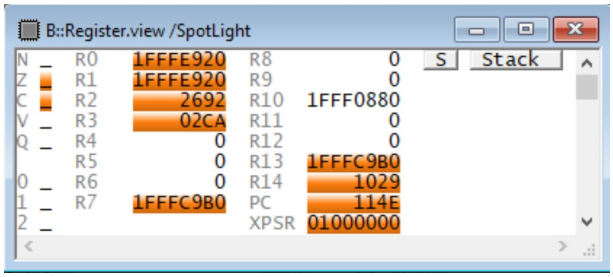

Figure 3.6: Register.view

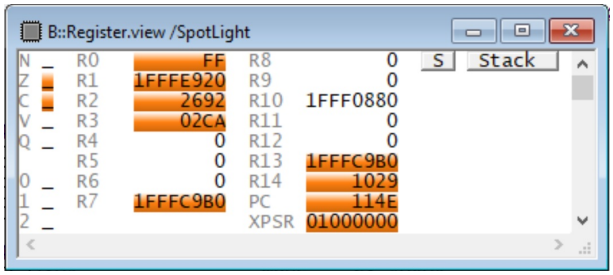

Figure 3.7: Register.Set R0 0xFF

```
PER.view [options]
PER.Set.simple <address>|<range> [%<format>] <value>
```

Similarly, it is possible to display and modify memory data by means of the *Data* command

```
Data.dump <address>|<range> [/<option>]
Var.watch [%<format>][<variable>]          ;alternative
Data.Set <address>|<range> [%format] <value>
         [/<option>]
```

In Trace32 there are three different ways to identify address ranges

- <start_address> −− <end_address>
- <start_address> .. <end_address>
- <start_address> ++ <offset_in_byte>

In the images it is shown how these commands looks like in Trace32 environment, applied for the application shown in the previous paragraph.

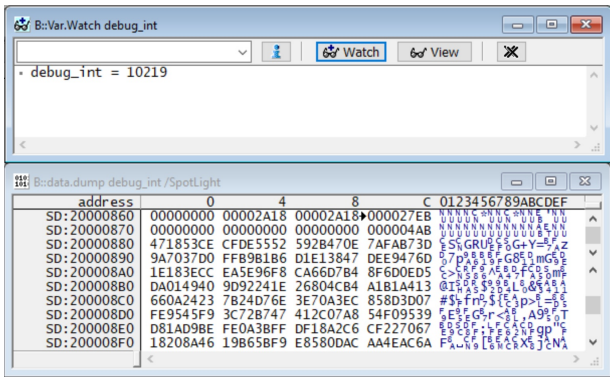

Figure 3.8: Var.Watch debug_int / Data.Dump debug_int /SpotLight

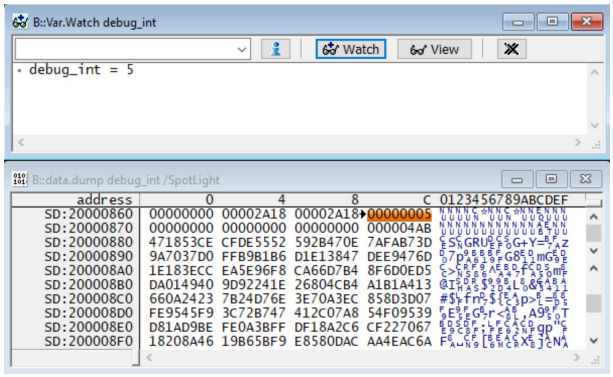

Figure 3.9: Data.Set debug_int %Long 0x05

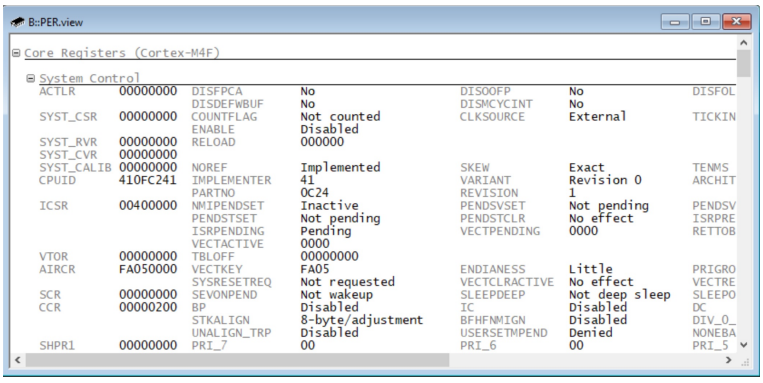

Figure 3.10: PER.view

3.3.3 Breakpoints

In Trace32 there is the possibility to handle two different types of breakpoints:

- Software breakpoints
- Onchip breakpoints

The number of software breakpoints that can be used while debugging the application is unlimited. The main problem of this kind of breakpoints is that they are implemented as intrusive breakpoints, altering real-time behaviour of the system when used. The architecture of the system could support the use of onchip breakpoints (TWR-K70F120M is compatible with onchip breakpoints). These breakpoints are limited in number, but they have the property to be non-intrusive, thus they do not affect the real-time behaviour of the system.
In Trace32 there is the possibility to set breakpoints to:

- Program: the breakpoint is set to a code address. In this case the breakpoint is triggered when the PC reaches the value of the address code where the breakpoint has been set.
- Read / Write / ReadWrite: the breakpoint is set to a data address. In this case the breakpoint is triggered when the specified operation (read, write or both of them) has been executed on the desired variable.

It is possible to set breakpoints in Trace32 via GUI, by double click on the line of code where the program breakpoints wants to be set, or selecting the variable name, and from the right-click menu, selecting *Breakpoint*. Furthermore, it is possible to set and list breakpoints by using the *Break* Trace32 command

```
Break.List
Break.Set [<address>|<range>] [/<breaktype>...]]
          [/<impl>]
```

When a certain breakpoint cause the system to stop its execution, it is highlighted in the list window. From the same window it is possible to delete the desired breakpoint by right-click on it and selecting *Delete*.
In Trace32 there is the possibility to perform different actions when a breakpoint is triggered:

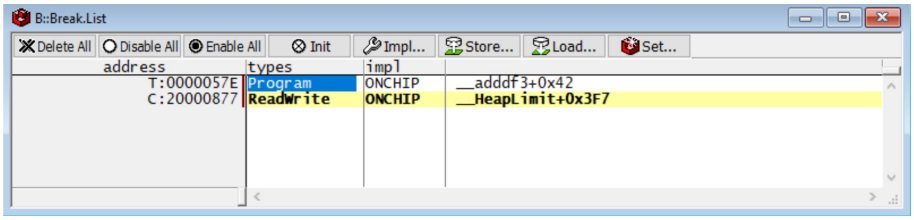

Figure 3.11: System halted at debug_char Read/Write breakpoint

- Stop: the entire system is halted when this breakpoint is triggered
- Spot: the system is halted just for the time needed to update the screen
- TraceEnable: this breakpoint will not halt the system, but will generate useful information used for trace analysis.

There are different options that can be used for breakpoints:

- Temporary OFF: the breakpoint is permanent
- Temporary ON: breakpoint is deleted next time the core stops the program execution
- DISable ON: breakpoint is disabled
- DISable OFF: breakpoint is enabled
- DISableHIT ON: breakpoint is disabled after this has been hit.

In Trace32 it is possible to set conditions on data breakpoints, e.g., the breakpoint can be triggered when a data reaches a certain value, or is greater than a certain value, and so on. Furthermore, there is the possibility to use advanced features for the breakpoint by setting different fields:

- COUNT: triggers the action of the breakpoint at the *n-th* hit of the breakpoint
- CONdition: triggers the action of the breakpoint only when the specified condition is true
- CMD: used to specify an action, or a set of actions, to be performed when the breakpoint has been triggered

3.4 Application trace

Here it is presented how to manage the CoreSight components, discussed in the previous chapter, and how to perform simple real-time analysis of the user application.

3.4.1 Manage CoreSight components

It is possible to manage all the parameters of the CoreSight components by accessing the different system menus, through the commands

```
ETM      ;opens the ETM configuration menu
TPIU     ;opens the TPIU configuration menu
ITM      ;opens the ITM configuration menu
```

To output PC data at regular intervals, the field *PCSampler* must be set. This interval is expressed in number of clock cycles, and this field must be set to a value, if it is required to perform an analysis on data. It is possible to set this parameter from the ITM menu, or using the command

```
ITM.PCSampler <value>
```

The value is expressed as a fraction of a power of two number, starting from 1/64 to 1/32768.

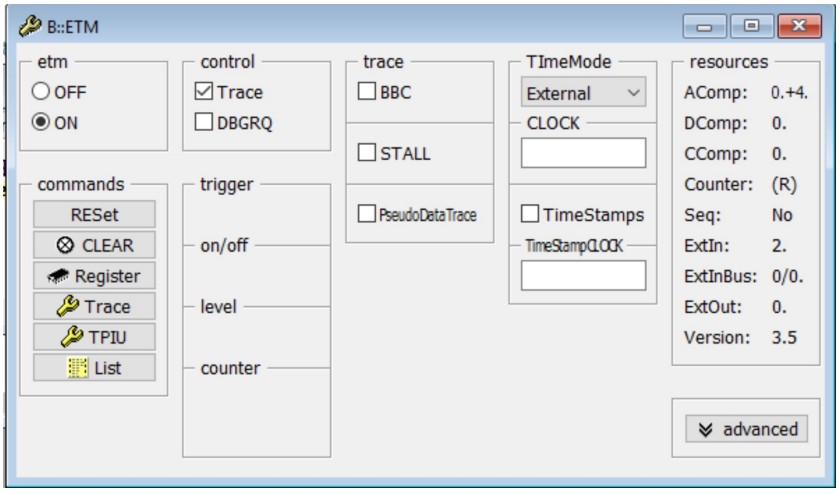

Figure 3.12: ETM menu

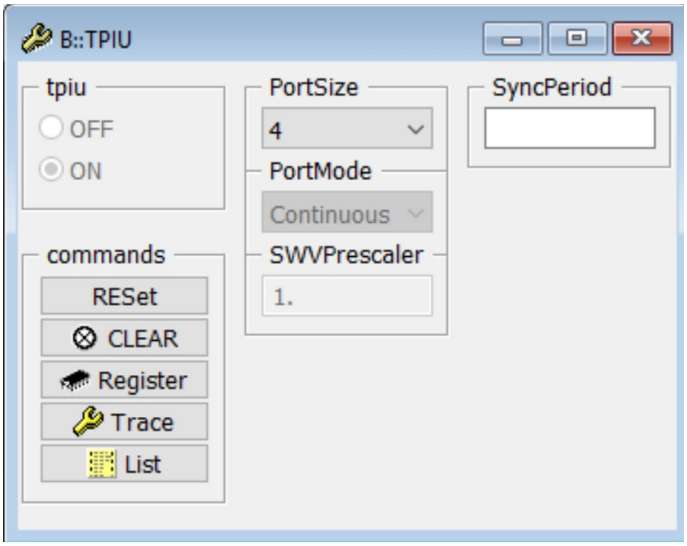

Figure 3.13: TPIU menu

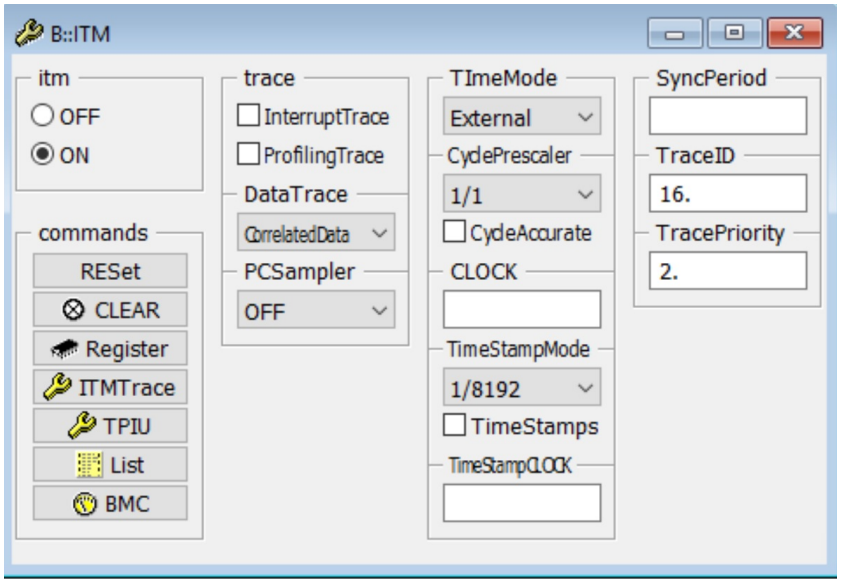

Figure 3.14: ITM menu

To list all the collected ITM and ETM trace data, it is possible to use the *List* subcommand

```
ITMTrace.List
ETMTrace.List
```

The result will be an huge set of data, as shown in figures, from which is possible to get useful results barely. For this, Trace32 gives the possibility to perform automatic analysis on the data.

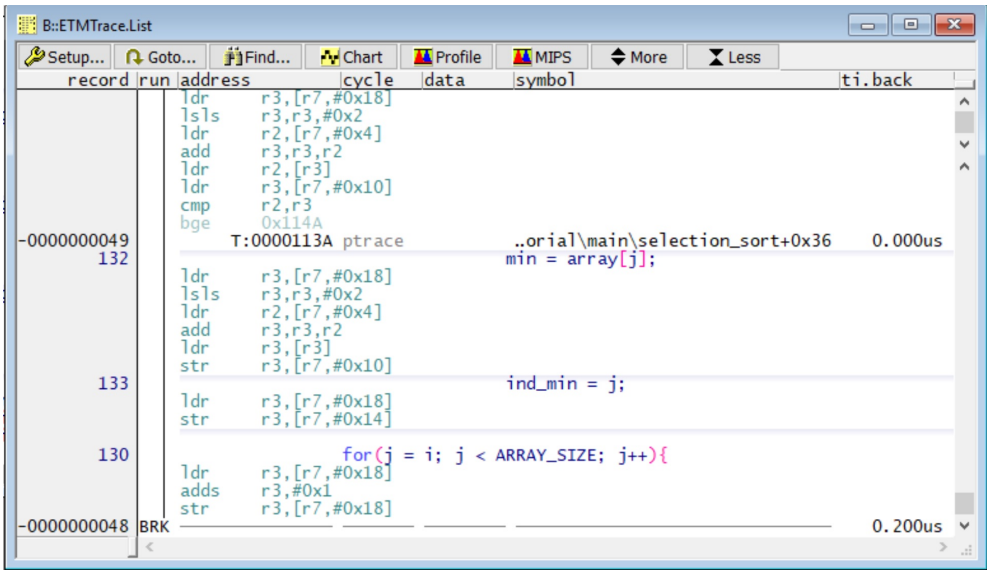

Figure 3.15: ETM data

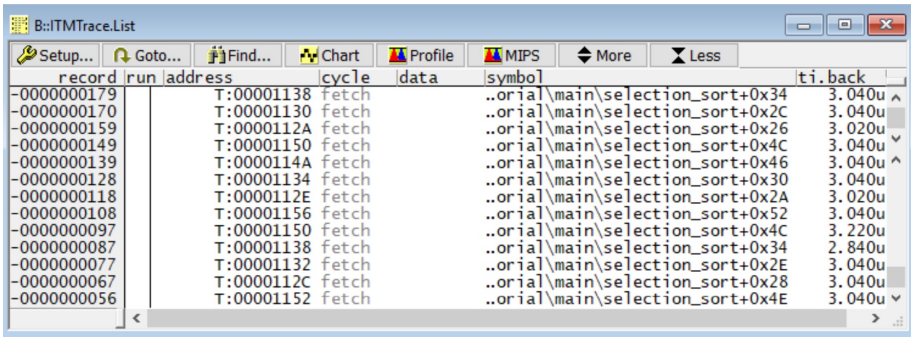

Figure 3.16: ITM data

3.4.2 Statistics on trace data

It is possible to list all the monolithic functions percentage of time spent running on the CPU, the worst case execution time and the average execution time, with a bar chart ordered by non increasing of ratio, by using the command

```
Trace.STATistic MAX AVeRage Ratio BAR /Sort Ratio
```

As it is possible to see from the figure, in the results are reported not only the functions of the main application such as *selection_sort* or *quickSort*, but are reported all the internal function called by the systems, e.g. all the routines for operations with double numbers and so on.

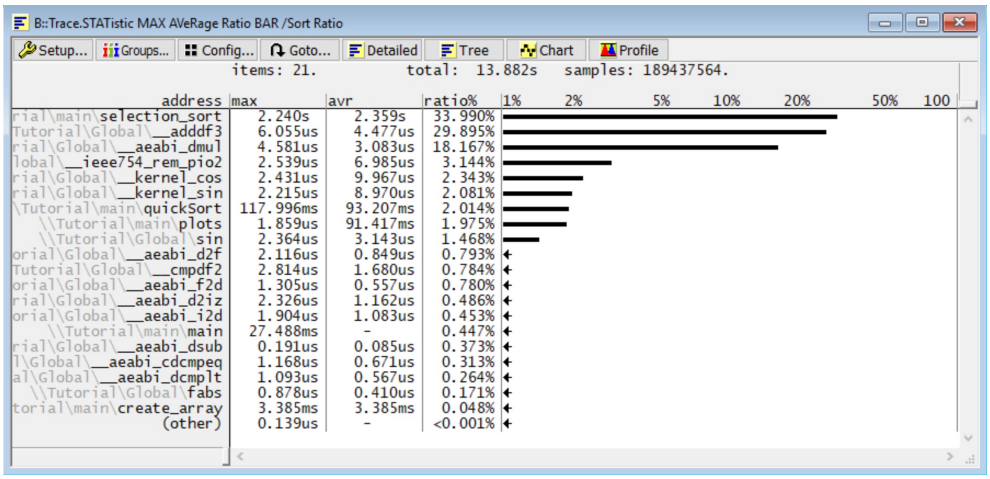

Figure 3.17: Ratio and execution times analysis

In Trace32 there is the possibility to show the time behaviour of the system, with informations on which function is running on the CPU for each instant of time. To display this kind of result, the command must be used

```
Trace.Chart.sYmbol
```

From these kind of information it is possible to investigate on which function is the bottleneck of the application, to give an idea on which segment of code should be optimized to get the greatest advantages for the overall system.
It is interesting to notice that the application used as reference for this chapter performs the sorting of the same vector of two thousand elements with two different algorithms: a selection sort and a quick sort. It is noticeable

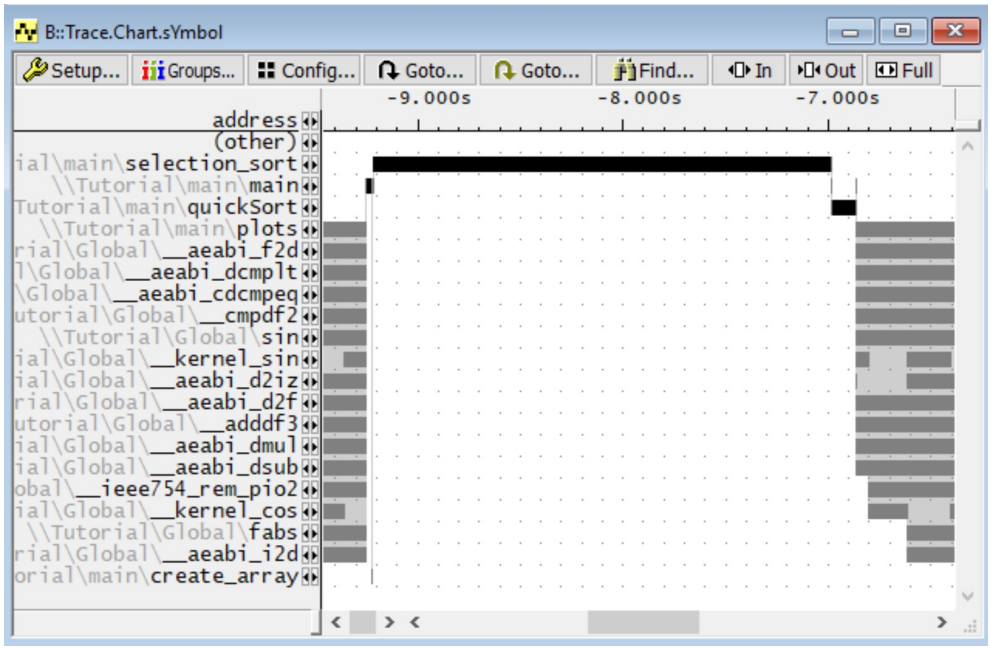

Figure 3.18: Time analysis

the huge difference in execution time between the two algorithms, with the selection sort algorithm that has an average execution time of 2.359s and a worst case execution time of 2.240s, while the quick sort average execution time is 93.207ms and worst case execution time 117.996ms (more than one order of magnitude).
Furthermore, Trace32 gives the possibility to draw the evolution in time of a variable by means of a 2D plot. To achieve such a result it is needed to have a breakpoint set to the desired variable that triggers the data trace for such variable, and after it is possible to run the trace statistics. Using the following commands

```
Break.Set square_wave /Onchip /Write /TraceData
Break.Set sine_wave /Onchip /Write /TraceData
Trace.DRAW.Var %DEFault square_wave
Trace.DRAW.Var %DEFault sine_wave
```

It is possible to obtain the results shown in the following figures. It is necessary to notice, that the breakpoint is mandatory. If it is not set, the result will not be shown.

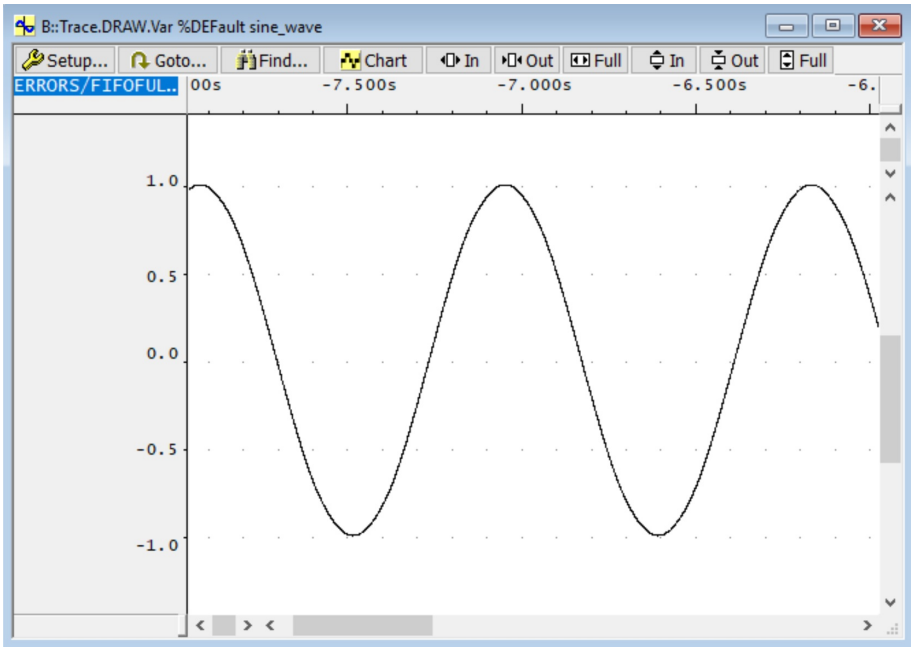

Figure 3.19: sine_wave time evolution

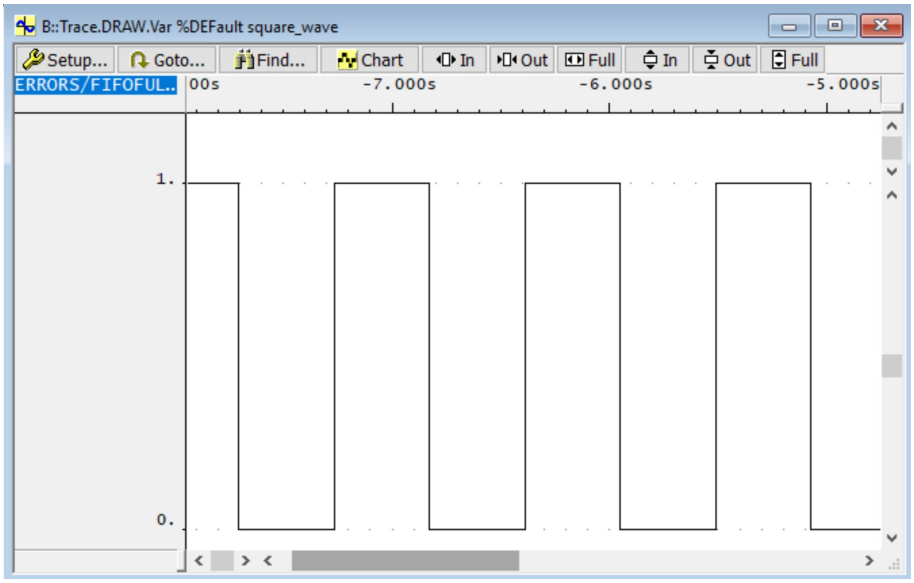

Figure 3.20: square_wave time evolution

To measure execution time between a starting and a final point, in Trace32 it is possible to make use of the *Runtime* feature. This feature is affected by some inaccuracy due to the JTAG communication time. To successfully use this feature it is needed to:

- Setting breakpoints for the starting point and the end point of measurement
- Initialize the runtime, and the area in which the results will be displayed
- Execute several times the code, printing the execution time at each step
- Display the results
- Remove the used breakpoints

For the example application, e.g., it is possible to set-up the following script that performs all the described steps to measure the time elapsed from the creation of the two arrays to the end of the sorting operation

```
;Set the breakpoints
Break.Set runtime_start /Write
Break.Set runtime_stop /Write

GO
WAIT !STATE.RUN()

;Initialize runtime and area
RunTime.Init

AREA.Create OUT
AREA.Select OUT
AREA.CLEAR OUT

;Perform 6 measurements of the execution time
RePeat 6.
(
  GO
  WAIT !STATE.RUN()
```

```
  &time=RunTime.LASTRUN()
  PRINT "Execution time: &time"

  GO
  WAIT !STATE.RUN()
)

;Display the final results
AREA.view OUT

;Remove breakpoints
Break.Delete runtime_start /Write
Break.Delete runtime_stop /Write
```

By running this script once the debug and trace session has been established, the obtained result is reported in the following figure. It is interesting from the figure the inaccuracy of 584.260μs. This means that this procedure can be used only for long time measurements, otherwise, the inaccuracy would be too high with respect to the measurement. Then, to perform smaller time segments measurements, other techniques must be exploited. One of this is to use the Instrumentation Trace Macrocell to perform the measurement. This technique gives more accurate results, but an instrumentation of the code is requested, for which several iteration of instrumentation of the code and building the application may be requested (that can be high time consuming operations depending on the size of the project).

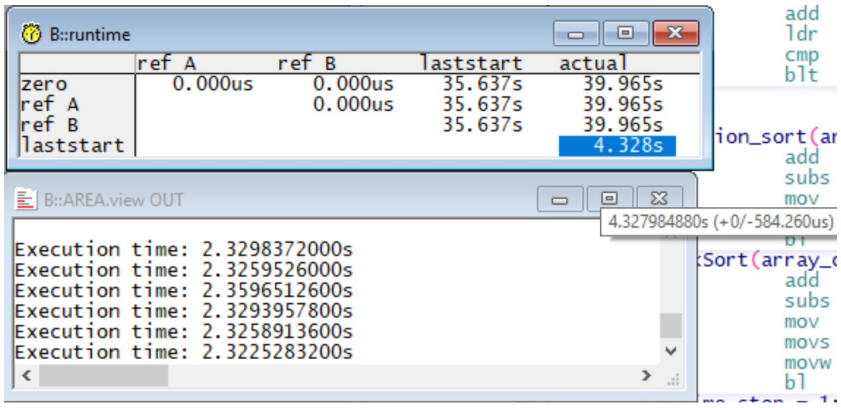

Figure 3.21: runtime statistics

ITM is used to instrument the software to send debug informations. It has been already discussed how ITM works in the previous chapter. Basing on how ITM is structured, it is possible to conclude that it is needed a set of software instructions to be able to write and read data from ITM channels. For this purpose, it is possible to add these following set of defines that provides an API to write 8, 16 or 32 bit data in a specified ITM channel.

```
static volatile unsigned int *ITM_BASE_CH = (volatile
      unsigned int*) 0xE0000000;

#define ITM_TRACE_D8(_channel_, _data_) {\
        volatile unsigned int *_ch_=ITM_BASE_CH+
                                        (_channel_); \
        while(*_ch_ == 0); \
        (*((volatile unsigned char *)(_ch_))) =
                                          (_data_); \
}

#define ITM_TRACE_D16(_channel_, _data_) {\
        volatile unsigned int *_ch_=ITM_BASE_CH+
                                        (_channel_); \
        while(*_ch_ == 0); \
        (*((volatile unsigned short *)(_ch_))) =
                                          (_data_); \
}

#define ITM_TRACE_D32(_channel_, _data_) {\
        volatile unsigned int *_ch_=ITM_BASE_CH+
                                        (_channel_); \
        while(*_ch_ == 0); \
        (*((volatile unsigned int *)(_ch_)))=
                                          (_data_); \
}
```

Once these macros have been defined it is possible to use simple functions to write data to the different channels, such as

```
ITM_TRACE_D8(3,1);
ITM_TRACE_D8(3,2);
```

3.4.3 Operating system aware tracing

Trace32 gives the possibility to set-up an environment that is able to be aware of operating system operations. The most important feature is that Trace32 is able to track task switches, giving the possibility to perform different statistic on task execution that are fundamental for safety-critical applications. For example, it is possible to find the worst case execution time of a task running on the target, and evaluate if the considered task, with the current scheduling algorithm, and the current load, is able to meet the deadline for data delivery.
To track context switches between the different tasks, it is possible to use the command

```
TASK.CONFIG(magic)
```

That returns as content the address of the variable that owns the information of witch task is currently running in the target. Than, it is possible to notice, that it is possible to use the return value of this function to set up a breakpoint, in order to be able to track each context switch that happens in the target. Than, to perform some analysis based on tasks, it is required to set the breakpoint with the following command

```
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

Once the breakpoint has set, it is possible to perform similar analysis that have been discussed for functions. In particular, it is possible to analyse the execution time of each task running on the target, such as the worst case execution time and the average execution time, with the total amount of time spent executing in the CPU (percentage value) shown as a bar diagram, with the following command

```
ITMCanalyzer.STATistic.TASK
```

Similarly, it is possible to have a chart showing in each instant of time which was the task currently running on the target with the following command

```
ITMCanalyzer.Chart.TASK
```

The results that are obtained from the μClinux application shown in the previous sections are shown in the following figures. It is interesting to notice, since this application continuously read data from a device, the *swapper* process, i.e. the process that takes care for I/O operations, is the process that keep the CPU busy for the greatest amount of time.

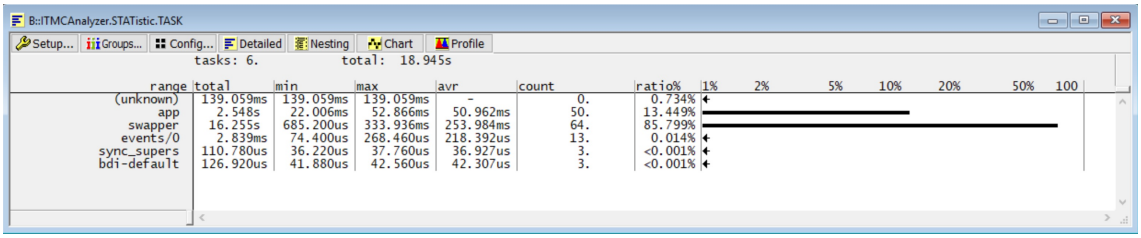

Figure 3.22: Task execution time statistics

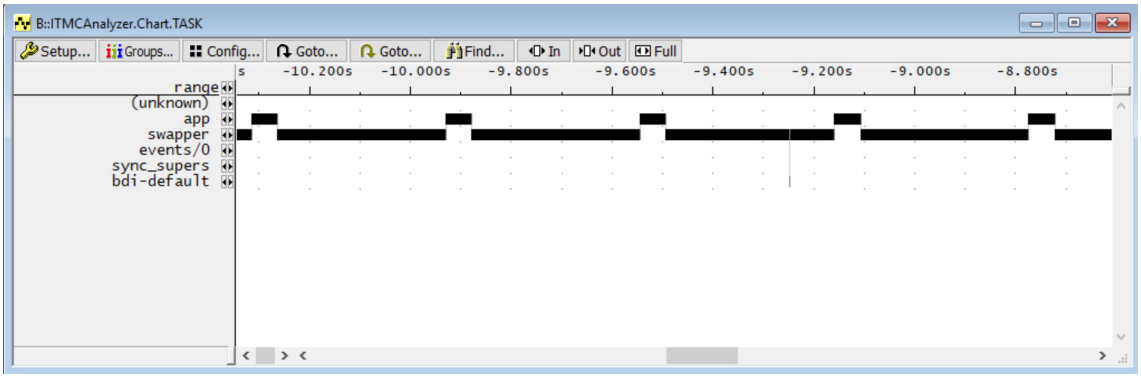

Figure 3.23: Task time evolution statistics

4

Trace32 set-up script for bare metal application

In this chapter it is shown how, starting, from an example script provided by Lauterbach, a service that is able to provide a working debugging and tracing environment for a bare metal application has been set-up.

4.1 Script requirements and parameters

To be able to successfully run the script the following requirement is needed:

- Place the script in the same folder of the .elf file
- The executable .elf file must be built with debugging informations (otherwise will not be possible to make use of symbols, i.e., the source code will not be visible in Hll mode, and so on)

This script has in input one parameter:

- Name of the .elf file to debug

To run the script is then possible to use the following command

```
DO <path>/twr-k70f120_configuration.cmm <name>.elf
```

Once the script has been processed from Trace32, the debugging and tracing environment will be ready for the use.

4.2 Script

Here follows the *twr-k70f120_configuration.cmm* code

```
; --------------------------------------------------------------------------
; @Title: Demo script for MK70FN1M0VMJ12 on TWR-K70F120
; with Offchip-Trace (RAM)
; @Description:
;   Loads the sieve demo application into RAM and sets
; up a demo debug
;   scenario.
;   The Offchip Trace using a Combiprobe/uTrace or
;   PowerTrace is set up.
;   Use this script for testing the Offchip-Trace.
;   Prerequisites:
;    * Connect Combiprobe/uTrace to J11
;      or
;      Connect AutoFocus Preprocessor to J11
;      and connect DebugCable to Preprocessor
;    * remove   R138 & R11
;    * populate R137
; @Keywords: ARM, Cortex-M4, ETM, ITM
; @Author: AME
; @Board: TWR-K70F120
; @Chip: MK70FN1M0VMJ12
; @Copyright: (C) 1989-2016 Lauterbach GmbH, licensed
; for use with TRACE32(R) only
; --------------------------------------------------------------------------
; $Id: twr-k70f120_sieve_offchip_trace_sram.cmm 10461
;  2017-02-16 14:36:14Z mplichta $

ENTRY &elfname

WinCLEAR

; --------------------------------------------------------------------------
; initialize and start the debugger
RESet
```

```
SYStem.RESet
SYStem.CPU MK70FN1M0VMJ12
SYStem.CONFIG.DEBUGPORTTYPE SWD
IF hardware.COMBIPROBE()||hardware.UTRACE()
(
  SYStem.CONFIG.CONNECTOR MIPI20T
)
SYStem.Option DUALPORT ON
SYStem.MemAccess DAP
SYStem.JtagClock CTCK 10MHz
SYStem.Up

GOSUB DisableWatchdog

; ------------------------------------------------------------------
; initialize offchip-trace (ETM ON, ITM ON)
IF hardware.COMBIPROBE()||hardware.UTRACE()||Analyzer()
(
  ; set PinMux and enable Clocks
  Data.Set SD:0x40048038 %Long 0x0000FFFF // PORTA_CLK
  Data.Set SD:0x40048004 %Long 0x00001000 // TRACE_CLK
  Data.Set SD:0x40049018 %Long 0x00000740 // TRACECLK
  Data.Set SD:0x4004901C %Long 0x00000740 // TRACED3
  Data.Set SD:0x40049020 %Long 0x00000740 // TRACED2
  Data.Set SD:0x40049024 %Long 0x00000740 // TRACED1
  Data.Set SD:0x40049028 %Long 0x00000740 // TRACED0
  Data.Set SD:0x40048068 %Long 0x00000000
                                          // Trace Clkdiv 0

  ; optional: setup the DCO here
  IF FALSE()
  (
    ; let the DCO run ~80-90MHz
    Data.Set SD:0x40064003 0x75
    ; NFC/32 + Trace/1
    Data.Set SD:0x40048068 %Long 0xF8000000
  )
```

```
  TPIU.PortSize 4
  TPIU.PortMode Continuous
  ITM.DataTrace CorrelatedData
  ITM.ON
  ETM.Trace ON
  ETM.ON
)
IF hardware.COMBIPROBE()||hardware.UTRACE()
(
  Trace.METHOD CAnalyzer
  Trace.AutoInit ON
  IF VERSION.BUILD.BASE()>=74752.
  (
    CAnalyzer.AutoFocus
  )
  ELSE
  (
    ; for uTrace & Combiprobe use manual calibration
    ; CAnalyzer.ClockDELAY Large
  )
)
IF Analyzer()
(
  Trace.METHOD Analyzer
  Trace.AutoInit ON
  Trace.AutoFocus
)

; ------------------------------------------------------------------------------
; Flash programming

; prepare flash programming (declarations)
DO ~~/demo/arm/flash/mk70.cmm PREPAREONLY

; ReProgram Flash
FLASH.ReProgram ALL
Data.LOAD.Elf "~~~~/&elfname"
FLASH.ReProgram OFF
```

```
; ------------------------------------------------------------------------
; start program execution
Go.direct main
WAIT !STATE.RUN()

Data.List main

ITM.PCSampler 1/64

ENDDO

DisableWatchdog:
(
  ; disable the Watchdog
  LOCAL &tmp1 &tmp2
  &tmp1=Data.Long(ST:0x20000000)
  &tmp2=Data.Long(ST:0x20000004)
  Register.SWAP

  ; The watchdog has a restrictive timing. It has to be
  ; configured and unlocked within a peripod
  ; of 20+256 cycles. Therefor the unlock sequence need
  ; to be done by a small target program.

  Data.Assemble ST:0x20000000  strh r1,[r0]
                      ;SD:0x4005200E = 0xC520   (Key 1)
  Data.Assemble ,              strh r2,[r0]
                      ;SD:0x4005200E = 0xD928   (Key 2)
  Data.Assemble ,              strh r4,[r3]
            ;SD:0x40052000 = 0x0000   (Config register)
  Data.Assemble ,              bkpt #0
  Register.Set PC 0x20000000
  Register.Set R0 0x4005200E
  Register.Set R1 0xC520
  Register.Set R2 0xD928
  Register.Set R3 0x40052000
  Register.Set R4 0x0
```

```
  Go.direct
  WAIT !RUN()

  Register.SWAP
  Data.Set ST:0x20000000 %Long &tmp1
  Data.Set ST:0x20000004 %Long &tmp2

  RETURN
)
```

This script is responsible for resetting the target as first operation. After, it sets all the required system configurations such as:

- CPU model
- Debug port type
- Dualport option
- Memory access
- JTAG clock

Furthermore, if it is being using the compiprobe cable or the μTrace the config connector is set to *MIPI20T*. Once these basic options have been set, JTAG pins are configured in order to use the 20-pin with 4 bit TPIU configuration. Immediately after, the CoreSight components are configured, in particular the TPIU port size is set to 4 in continuous mode, and ITM is instructed to send correlated data (i.e., merging ETM and ITM data togheter). Once these instructions have been executed, the canalyzer auto focus procedure will start, finding the best frequency at which operate. At the end of all these preliminary steps and configurations, the flash memory of the target is programmed by loading the user application, which name has been specified as parameter for the script. When the executable file has been loaded in flash memory, the execution is started, bypassing all bootstrap operations until the target reaches the *main* function where the system is halted. The script ends, setting a value of the PCSampler, in order to be able to perform data tracing, and displaying the source code of the *main* function.

5

Trace32 set-up script for μClinux applications

Starting from the script described in the previous chapter, has been realized a script able to set-up a debugging and tracing environment for an application running on a μClinux operating system. This environment is provided with μClinux awareness.

5.1 Script requirements and parameters

To be able to successfully run the script the following requirement is needed:

- Have u-boot already installed in the flash memory of the target
- The script must be placed in a folder that contains the vmlinux and uImage files and a single folder named "linux_ker" containing the file system of the linux application containing all the source files.
- Have vmlinux compiled with debugging informations
- Have the μClinux application compiled with debugging informations

The script takes as input the following parameters:

- The name of the μClinux application
- The name of the uImage that has to be loaded

- The com port through with the target is connected to the host

To run the script is then possible to use the following command

```
DO <path>/Linux_configuration.cmm <app\_name>
                    <image\_name>.uImage com<number>
```

Once the script has been processed from Trace32, the debugging and tracing environment will be ready for the use when the application will start on the target.

5.2 Script

Here follows the *Linux_configuration.cmm* code

```
; --------------------------------------------------------------------------------
; @Title: Demo script for MK70FN1M0VMJ12 on TWR-K70F120
; with Offchip-Trace (RAM)
; @Description:
;   Loads the sieve demo application into RAM and sets
; up a demo debug
;   scenario.
;   The Offchip Trace using a Combiprobe/uTrace or
;   PowerTrace is set up.
;   Use this script for testing the Offchip-Trace.
;   Prerequisites:
;    * Connect Combiprobe/uTrace to J11
;      or
;      Connect AutoFocus Preprocessor to J11
;      and connect DebugCable to Preprocessor
;    * remove   R138 & R11
;    * populate R137
; @Keywords: ARM, Cortex-M4, ETM, ITM
; @Author: AME
; @Board: TWR-K70F120
; @Chip: MK70FN1M0VMJ12
; @Copyright: (C) 1989-2016 Lauterbach GmbH, licensed
; for use with TRACE32(R) only
; --------------------------------------------------------------------------------
```

```
; $Id: twr-k70f120_sieve_offchip_trace_sram.cmm 10461
;  2017-02-16 14:36:14Z mplichta $

ENTRY &app_name &uimage &com

WinCLEAR

; --------------------------------------------------------------------
; initialize and start the debugger
RESet
SYStem.RESet
SYStem.CPU MK70FN1M0VMJ12
SYStem.CONFIG.DEBUGPORTTYPE SWD
IF hardware.COMBIPROBE()||hardware.UTRACE()
(
  SYStem.CONFIG.CONNECTOR MIPI20T
)
SYStem.Option DUALPORT ON
SYStem.MemAccess DAP
SYStem.JtagClock CTCK 10MHz
SYStem.Up

GOSUB DisableWatchdog

; --------------------------------------------------------------------
; initialize offchip-trace (ETM ON, ITM ON)
IF hardware.COMBIPROBE()||hardware.UTRACE()||Analyzer()
(
  ; set PinMux and enable Clocks
  Data.Set SD:0x40048038 %Long 0x0000FFFF // PORTA_CLK
  Data.Set SD:0x40048004 %Long 0x00001000 // TRACE_CLK
  Data.Set SD:0x40049018 %Long 0x00000740 // TRACECLK
  Data.Set SD:0x4004901C %Long 0x00000740 // TRACED3
  Data.Set SD:0x40049020 %Long 0x00000740 // TRACED2
  Data.Set SD:0x40049024 %Long 0x00000740 // TRACED1
  Data.Set SD:0x40049028 %Long 0x00000740 // TRACED0
  Data.Set SD:0x40048068 %Long 0x00000000
                                  // Trace Clkdiv 0
```

```
  ; optional: setup the DCO here
  IF FALSE()
  (
    ; let the DCO run ~80-90MHz
    Data.Set SD:0x40064003 0x75
    ; NFC/32 + Trace/1
    Data.Set SD:0x40048068 %Long 0xF8000000
  )

  TPIU.PortSize 4
  TPIU.PortMode Continuous
  ;ITM.DataTrace CorrelatedData

)
IF hardware.COMBIPROBE()||hardware.UTRACE()
(
  Trace.METHOD CAnalyzer
  Trace.AutoInit ON
  IF VERSION.BUILD.BASE()>=74752.
  (
    CAnalyzer.AutoFocus
  )
  ELSE
  (
    ; for uTrace & Combiprobe use manual calibration
    ; CAnalyzer.ClockDELAY Large
  )
)
IF Analyzer()
(
  Trace.METHOD Analyzer
  Trace.AutoInit ON
  Trace.AutoFocus
)

do ~~\demo\etc\terminal\serial\term.cmm &com
```

```
GO
Wait 5.s
Break

;Load the kernel
Data.LOAD.Binary &uimage A:0x08007FC0
Data.LOAD.Elf vmlinux /GNU /NoCODE /STRIPPART 4.
sYmbol.SourcePATH ./linux_ker\

GO
ITM.ON
ETM.Trace ON
ETM.ON
Trace.Arm
ITM.PCSampler 1/256

;Booting
TERM.Out "bootm" 0xA

;Loading uClinux awareness
Wait 1.s
TASK.CONFIG
    /T32\demo\arm\kernel\uclinux\linux-3.x\uclinux3.t32
MENU.ReProgram
    /T32/demo\arm\kernel\uclinux\linux-3.x\uclinux.men

;Setting up the autoloader
TASK.sYmbol.Option AutoLoad Process

Wait 50.ms
do /T32\demo\arm\kernel\uclinux\linux-3.x\app_debug.cmm
                                              &app_name

ENDDO

DisableWatchdog:
(
  ; disable the Watchdog
```

```
  LOCAL &tmp1 &tmp2
  &tmp1=Data.Long(ST:0x20000000)
  &tmp2=Data.Long(ST:0x20000004)
  Register.SWAP

  ; The watchdog has a restrictive timing.
  ;It has to be configured and unlocked within a period
  ; of 20+256 cycles. Therefor the unlock sequence need
  ; to be done by a small target program.
  Data.Assemble ST:0x20000000  strh r1,[r0]
       ;SD:0x4005200E = 0xC520   (Key 1)
  Data.Assemble ,              strh r2,[r0]
       ;SD:0x4005200E = 0xD928   (Key 2)
  Data.Assemble ,              strh r4,[r3]
       ;SD:0x40052000 = 0x0000   (Config register)
  Data.Assemble ,              bkpt #0
  Register.Set PC 0x20000000
  Register.Set R0 0x4005200E
  Register.Set R1 0xC520
  Register.Set R2 0xD928
  Register.Set R3 0x40052000
  Register.Set R4 0x0
  Go.direct
  WAIT !RUN()

  Register.SWAP
  Data.Set ST:0x20000000 %Long &tmp1
  Data.Set ST:0x20000004 %Long &tmp2

  RETURN
)
```

This script performs the same configuration of CoreSight components that has been used for the bare metal application. In addition, this script, provides a full working environment for the μClinux application. In particular, a terminal window to interact with the target hardware is displayed (serial communication between the host and the virtual serial port of the target). This terminal window will use the COM port specified as script parameter,

with a baud rate of 115200. It is possible to interact with the target through the terminal only when the target is running. The script performs an automatic procedure to load in the target RAM memory the kernel image and perform the boot operation with the following steps:

- Loads the .uImage file to the target RAM, which name has been specified as script parameter. For *TWRK70F120M* address is 0x08007FC0 chosen
- Through the terminal is sent the *bootm* command to the target (running u-boot), this command will start the boot sequence starting from the load address defined as an environment variable (this address is 0x08007FC0 by default for *TWRK70F120M* development board)
- File *vmlinux*, compiled with debug symbol informations, is used to load kernel symbols, through which will be possible to debug and trace kernel operations
- Path of symbols is set, in order to instruct Trace32, that all the source files can be found starting from the *linux_ker* folder

After the target completed the boot operations, the μClinux awareness is set-up through the following steps:

- Configuration of the μClinux 3.0 awareness through the *uclinux3.t32* file, provided by Lauterbach
- Set-up a menu for Trace32 environment from which it is able to manage μClinux debugging and tracing, through the *uclinux.men* file provided by Lauterbach

After these steps, it is set-up the symbol autoloader for μClinux applications, to automatically detect application symbols, through the command

```
TASK.sYmbol.Option AutoLoad Process
```

At this point all the requirements needed to run the script provided by Lauterbach *app_debug.cmm* are satisfied. Thus, this script starts the execution, and will wait until the desired application (which name is set as parameter) to debug starts its execution.

6

Performance analysis of FIR and FFT

In this chapter is shown how it is possible to use this technology to perform analysis on a signal processing application. In particular the performance of two different FIR filter implementations (with different complexities), and an FFT algorithm (with different number of points for which is computed the transform) will be analysed.

6.1 Application description

This application is responsible to filter a signal input, and perform the FFT after that a certain amount of samples have been obtained.

6.1.1 Input signal

The used input signal is a sum of sinusoidal functions at different frequencies, one at low frequency, the other two at high frequency (that are the components that it is needed to filter out by means of the low pass FIR filter). The input signal can be expressed by means of the following equation

$$u(t) = A_1 sin(2\pi f_1 t) + A_2 sin(2\pi f_2 t) + A_3 sin(2\pi f_3 t) \tag{6.1}$$

Where the following parameters have been chosen:

- $A_1 = 1$

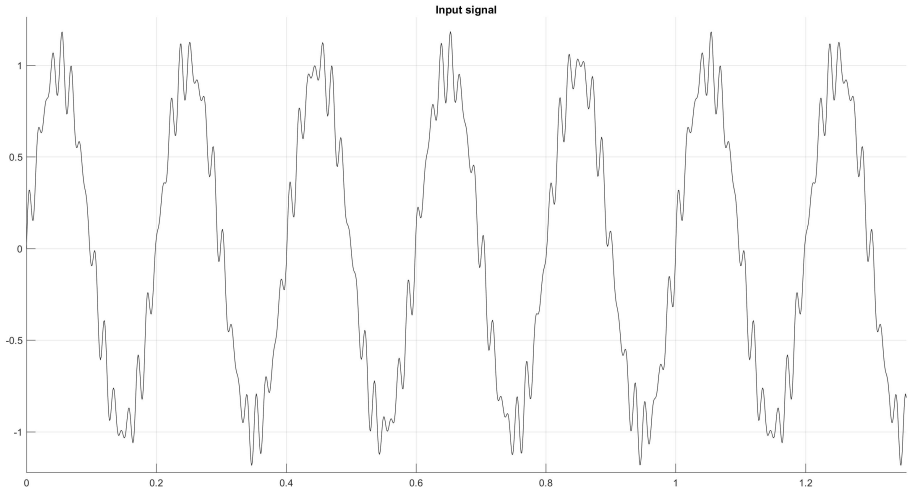

Figure 6.1: Input signal

- $f_1 = 5Hz$
- $A_2 = 0.1$
- $f_2 = 60Hz$
- $A_3 = 0.1$
- $f_3 = 77Hz$

The time behaviour of the input signal is reported in the figure.

6.1.2 FIR filter and Matlab filter designer

For this application has been used a low pass FIR digital filter. The design objective of this filter is to cut off the two high frequency components of the input signal. A FIR filter is characterized by the following discrete transfer function

$$H(z) = \sum_{k=0}^{N} b[k] z^{-k} \tag{6.2}$$

Where N is the order of the filter.
For this application four different filters have been designed, all of them

using the same technique, i.e. FIR low pass filter with Hamming window. The only design parameter that varies is the filter order. In particular have been designed filters with order:

- $N = 8$
- $N = 16$
- $N = 32$
- $N = 64$

To design the filter, Matlab Filter Designer tool has been employed. This tool provides the capability to choose the design parameters of the filter, and starting from these, it computes the b coefficients for the filter. The following design parameters have been used for the design:

- Response type: lowpass
- FIR: window
- Window: Hamming
- Sampling frequency Fs: $1000Hz$
- Cut frequency Fc: $20Hz$

The following values of the b polynomial have been obtained, listed from b_0 to b_N

```
N = 8
0.017556    0.048011    0.12235     0.1976      0.22897
0.1976      0.12235     0.048011    0.017556

N = 16
0.0079256   0.011884    0.022997    0.040163    0.061049
0.082476    0.101       0.11353     0.11796     0.11353
0.101       0.082476    0.061049    0.040163    0.022997
0.011884    0.0079256

N = 32
0.0023095   0.0028755   0.0041196   0.0061736   0.0091169
```

```
0.012967    0.017671    0.023109    0.029094    0.035386
0.041704    0.047743    0.053199    0.057786    0.061255
0.063416    0.064151    0.063416    0.061255    0.057786
0.053199    0.047743    0.041704    0.035386    0.029094
0.023109    0.017671    0.012967    0.0091169   0.0061736
0.0041196   0.0028755   0.0023095

N = 64
-0.00065146   -0.00061399   -0.00058867   -0.00056074
-0.0005114    -0.0004184    -0.00025681   3.0552e-19
0.00037924    0.00090754    0.0016096     0.0025072
0.0036176     0.0049527     0.0065184     0.008313
0.010328      0.012546      0.014943      0.017487
0.02014       0.022857      0.025589      0.028283
0.030885      0.033339      0.035594      0.037598
0.039306      0.040677      0.041681      0.042293
0.042499      0.042293      0.041681      0.040677
0.039306      0.037598      0.035594      0.033339
0.030885      0.028283      0.025589      0.022857
0.02014       0.017487      0.014943      0.012546
0.010328      0.008313      0.0065184     0.0049527
0.0036176     0.0025072     0.0016096     0.00090754
0.00037924    3.0552e-19    -0.00025681   -0.0004184
-0.0005114    -0.00056074   -0.00058867   -0.00061399
-0.00065146
```

In figure are reported the bode diagram of the different FIR filters, while in the table are reported the values of attenuation at the frequency of interest for the application ($60Hz$ and $77Hz$). It is interesting to notice how the simplest filter is not able to cut the high frequency components of the original signal.

Filter order	Att.@$60Hz(dB)$	Att.@$77Hz(dB)$
8	-1.746	-2.902
16	-6.735	-11.445
32	-26.078	-70.479
64	-76.285	-70.140

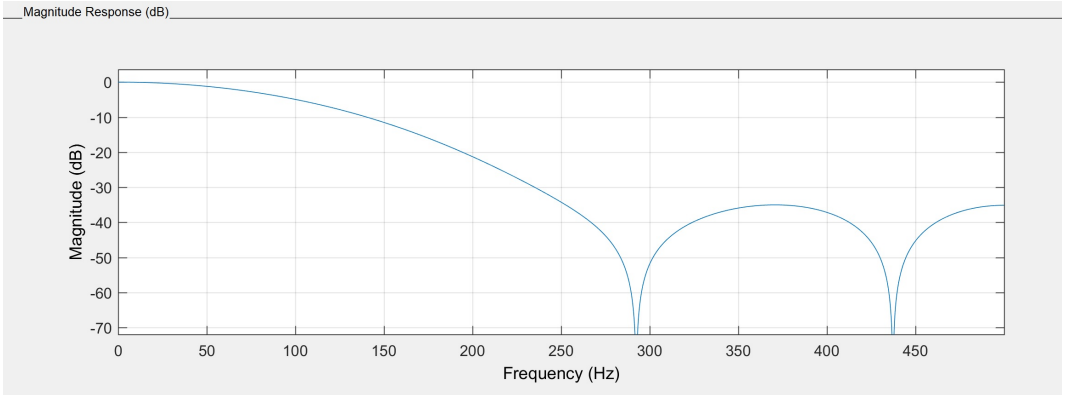

Figure 6.2: Bode diagram: $N = 8$

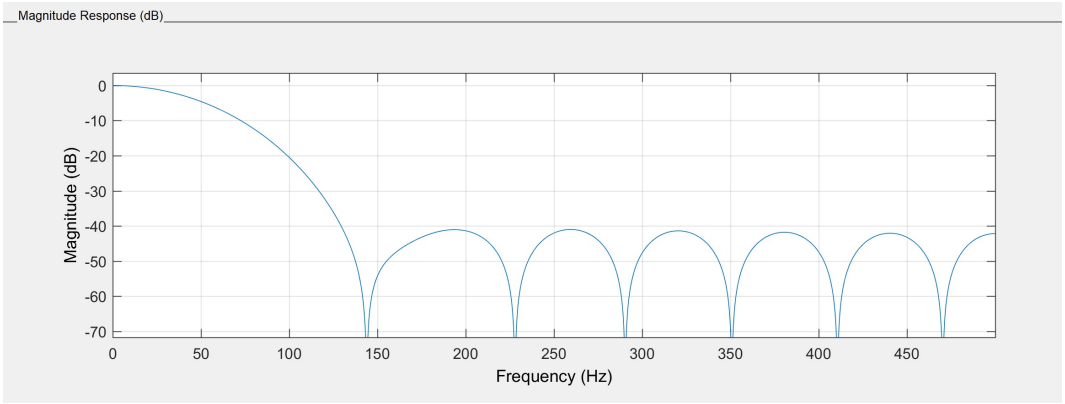

Figure 6.3: Bode diagram: $N = 16$

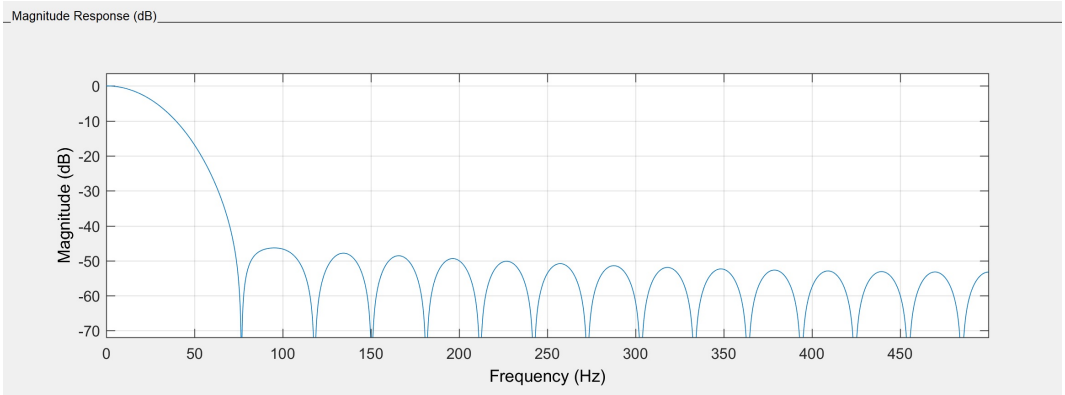

Figure 6.4: Bode diagram: $N = 32$

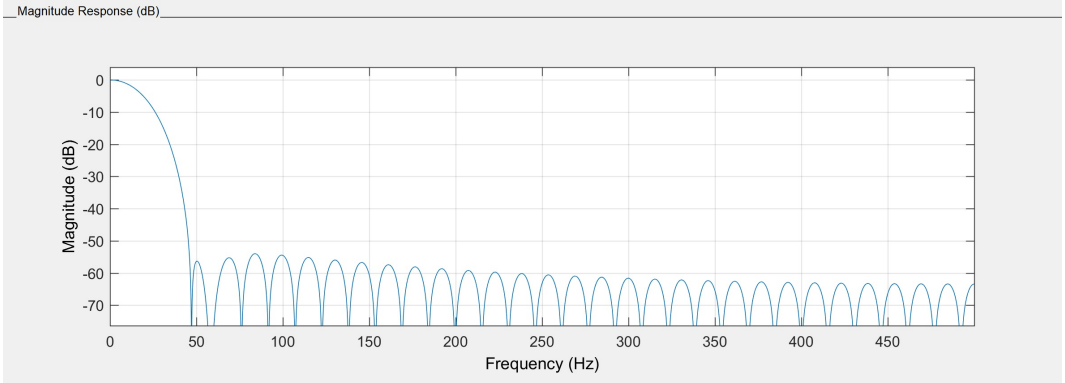

Figure 6.5: Bode diagram: $N = 64$

6.1.3 Fast Fourier Transform

To use faster algorithms, the FFT is computer over a number M of point, where M is a power of two number. By doing this, it is possible to exploit faster decimation in time or decimation in frequency algorithms. For this application different values of M are used, in particular:

- $M = 256$
- $M = 512$
- $M = 1024$

6.2 Code generation

Here it is presented how the source code has been obtained both for the FIR filters, and the FFT. In particular two different implementations for the FIR filter have been exploited.

6.2.1 FIR filter code

To produce a function that given a sample of the input signal is able to filter it and return the filtered signal as result, the following theoretical result has been exploited: write the transfer function $H(z)$ in the time domain, by using

the inverse *Z-transform*, obtaining the following difference equation

$$y[n] = \sum_{k=0}^{N-1} b[k]u[n-k] \tag{6.3}$$

Than it is simple to notice that the filtered value is a linear combination of the current and past values of the input signal. Then, a buffer storing $N+1$ values is needed. To build this buffer the structure shown in figure has been used. At each step, before the sampling, the current value of the input signal is inserted in the first position of the array and, after that the sample has been filtered, the buffer is updated, by shifting all the elements of the buffer to the right of one position, discarding the last one.

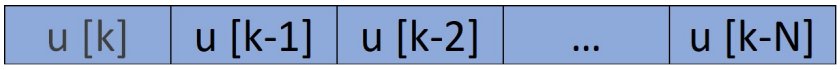

Figure 6.6: Buffer contents

With these consideration, the following implementation of the FIR filter has been obtained

```
float slow_filter(float sample, float* filt_b,
                                    float* buffer){
    float filtered = 0.0;
    int i;

    buffer[0] = sample;
    for(i=0; i<FILT_ORDER+1; i++){
        filtered += filt_b[i]*buffer[i];
    }
    for(i=FILT_ORDER; i>0; i--){
        buffer[i] = buffer[i-1];
    }

    return filtered;
}
```

By looking at this code, it is easy to see that there is a time consuming over-head in this code due to the update of the buffer, i.e., bigger is the filter order, bigger is the number of memory read/write operations that have

to be performed at each step to shift the buffer. To improve the computing performance of the filter, the following consideration has been done to achieve code optimization: do not shift at each step the overall content of the buffer, but place the value of the signal to be filtered in the position of the input signal to be discarded. For such an implementation, it is needed a comparison and an additional counter. The idea of such a buffer is represented in figure.

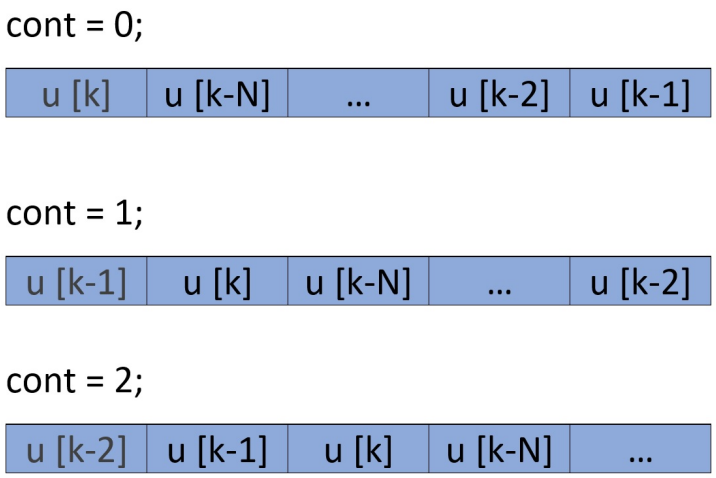

Figure 6.7: Buffer contents

This results into the following implementation of the FIR filter

```
float fast_filter(float sample, float* filt_b,
                  float *buffer, int *cont){
    float filtered = 0.0;
    int i, j;

    buffer[*cont] = sample;
    j = *cont;
    for(i=0; i<FILT_ORDER+1; i++){
        if(j>FILT_ORDER){
            j=0;
        }
        filtered += filt_b[i]*buffer[j];
        j++;
```

```
    }

    *cont = *cont - 1;
    if(*cont<0){
        *cont = FILT_ORDER;
    }

    return filtered;
}
```

6.2.2 FFT code and Matlab coder

To obtain FFT code a fast-prototyping approach has been used. In particular has been used the automatic code-generation process feature of Matlab. This feature is implemented into one Matlab app called Matlab coder, that gives the possibility to generate the code of a matlab function. To generate a code that is compatible with the target hardware (*TWRK70F120M* development board), it is needed to install an additional support package for ARM code generation, and selecting this tool chain during the code generation process. The result of such operation is the following set of source files:

- MW_target_hardware_resources.h
- my_fft.c
- my_fft.h
- my_fft_initialize.c
- my_fft_initialize.h
- my_fft_terminate.c
- my_fft_terminate.h
- my_fft_types.h
- rt_noninfinite.c
- rt_noninfinite.h

- rtGetInf.c
- rtGetInf.h
- rtGetNan.c
- rtGetNan.h
- rtwtypes.h

The API that provides the capability to perform the FFT of a generic signal can be found in the file *my_fft.h* that has the following content

```
/*
 * Academic License - for use in teaching,
 * academic research, and meeting
 * course requirements at degree granting
 * institutions only.
 * Not for
 * government, commercial, or other organizational use.
 * File: my_fft.h
 *
 * MATLAB Coder version            : 3.2
 * C/C++ source code generated on  : 05-Jul-2018
 *                                   22:54:44
 */

#ifndef MY_FFT_H
#define MY_FFT_H

/* Include Files */
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include "rtwtypes.h"
#include "my_fft_types.h"

/* Function Declarations */
extern void my_fft(const double u[256],
                   creal_T y[256]);
```

```
#endif

/*
 * File trailer for my_fft.h
 *
 * [EOF]
 */
```

As it is possible to notice from the API, the listed code is able to perform a 256 point FFT. The API is similar for 512 point FFT and 1024 point FFT. As it is possible to notice it gets as inputs an array of points (the filtered signal to which the FFT is computed) and the array of the results that are complex numbers (*creal_T*). The definition of this type and other types can be found in *rtwtypes.h* file that contains the following definitions

```
/*
 * Academic License - for use in teaching, academic
 * research, and meeting
 * course requirements at degree granting institutions
 * only.  Not for
 * government, commercial, or other organizational use.
 * File: rtwtypes.h
 *
 * MATLAB Coder version            : 3.2
 * C/C++ source code generated on  : 01-Jul-2018
 *                                         17:28:28
 */

#ifndef RTWTYPES_H
#define RTWTYPES_H
#ifndef __TMWTYPES__
#define __TMWTYPES__

 /* ===================================================
 * Target hardware information
 *   Device type: ARM Compatible->ARM Cortex
 *   Number of bits:     char:   8    short:   16
 *                       int:  32
```

```
 *                        long:  32
 *                        native word size:  32
 *   Byte ordering: LittleEndian
 *   Signed integer division rounds to: Zero
 *   Shift right on a signed integer as
 *                      arithmetic shift: on
 * ========================================================== */

 /* ========================================================== *
 * Fixed width word size data types:                          *
 *   int8_T, int16_T, int32_T
 *           - signed 8, 16, or 32 bit integers               *
 *   uint8_T, uint16_T, uint32_T
 *           - unsigned 8, 16, or 32 bit integers             *
 *   real32_T, real64_T
 *           - 32 and 64 bit floating point numbers           *
 * ========================================================== */
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef int int32_T;
typedef unsigned int uint32_T;
typedef float real32_T;
typedef double real64_T;

/* ========================================================== *
 * Generic type definitions: real_T, time_T,
 *                boolean_T, int_T, uint_T,
 *                ulong_T, char_T and byte_T.
 * ========================================================== */
typedef double real_T;
typedef double time_T;
typedef unsigned char boolean_T;
typedef int int_T;
typedef unsigned int uint_T;
typedef unsigned long ulong_T;
typedef char char_T;
```

```
typedef char_T byte_T;

/* ========================================================
 * Complex number type definitions                        *
 * ======================================================== */
#define CREAL_T

typedef struct {
  real32_T re;
  real32_T im;
} creal32_T;

typedef struct {
  real64_T re;
  real64_T im;
} creal64_T;

typedef struct {
  real_T re;
  real_T im;
} creal_T;

typedef struct {
  int8_T re;
  int8_T im;
} cint8_T;

typedef struct {
  uint8_T re;
  uint8_T im;
} cuint8_T;

typedef struct {
  int16_T re;
  int16_T im;
} cint16_T;

typedef struct {
```

```
  uint16_T re;
  uint16_T im;
} cuint16_T;

typedef struct {
  int32_T re;
  int32_T im;
} cint32_T;

typedef struct {
  uint32_T re;
  uint32_T im;
} cuint32_T;

/* ============================================================
 * Min and Max:                                               *
 *   int8_T, int16_T, int32_T                                 *
 *           - signed 8, 16, or 32 bit integers               *
 *   uint8_T, uint16_T, uint32_T                              *
 *           - unsigned 8, 16, or 32 bit integers             *
 * ============================================================ */
#define MAX_int8_T          ((int8_T)(127))
#define MIN_int8_T          ((int8_T)(-128))
#define MAX_uint8_T         ((uint8_T)(255))
#define MIN_uint8_T         ((uint8_T)(0))
#define MAX_int16_T         ((int16_T)(32767))
#define MIN_int16_T         ((int16_T)(-32768))
#define MAX_uint16_T        ((uint16_T)(65535))
#define MIN_uint16_T        ((uint16_T)(0))
#define MAX_int32_T         ((int32_T)(2147483647))
#define MIN_int32_T         ((int32_T)(-2147483647-1))
#define MAX_uint32_T        ((uint32_T)(0xFFFFFFFFU))
#define MIN_uint32_T        ((uint32_T)(0))

/* Logical type definitions */
#if !defined(__cplusplus) &&
              !defined(__true_false_are_keywords)
#  ifndef false
```

```
#   define false                          (0U)
#  endif

#  ifndef true
#   define true                           (1U)
#  endif
#endif

/*
 * Maximum length of a MATLAB identifier
 * (function/variable)
 * including the null-termination character.
 * Referenced by
 * rt_logging.c and rt_matrx.c.
 */
#define TMW_NAME_LENGTH_MAX               64
#endif
#endif

/*
 * File trailer for rtwtypes.h
 *
 * [EOF]
 */
```

6.2.3 Application code

The application is intended to simulate an acquisition system, periodically, the CPU reads a value, performs the filtering action, and when accumulates M value, begins the FFT of the signal. The code for the main application is the following

```
/*
 * Copyright (c) 2015, Freescale Semiconductor, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms,
 * with or without modification,
```

```
* are permitted provided that the following
* conditions are met:
*
* o Redistributions of source code must retain
* the above copyright notice, this list
*   of conditions and the following disclaimer.
*
* o Redistributions in binary form must reproduce
* the above copyright notice, this
*   list of conditions and the following disclaimer
* in the documentation and/or
*   other materials provided with the distribution.
*
* o Neither the name of Freescale Semiconductor,
*  Inc. nor the names of its
*   contributors may be used to endorse or promote
* products derived from this
*   software without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
* AND CONTRIBUTORS "AS IS" AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT
* NOT LIMITED TO, THE IMPLIED
* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE ARE
* DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER
* OR CONTRIBUTORS BE LIABLE FOR
* ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
* EXEMPLARY, OR CONSEQUENTIAL DAMAGES
* (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON
* ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
* ANY WAY OUT OF THE USE OF THIS
* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
```

```
 * OF SUCH DAMAGE.
 */

#include "MK70F12.h"
#include <stdio.h>
#include <stdlib.h>
#include "my_fft.h"

#define FILT_ORDER 64
#define N_SAMPLES  5001
#define POINTS_FFT 256

float fast_filter(float, float*, float*, int*);
float slow_filter(float, float*, float*);

int main(void)
{
  int k, cont_FFT = 0, i;
  float b[FILT_ORDER+1] = {'values'};
  float sig[N_SAMPLES] = {'values'};
  float out;
  float buffer_FFT[1024];
  creal_T y[1024];
  float buffer[FILT_ORDER+1] = {0};
  int cont = 0;

  for (;;) {
    cont = FILT_ORDER;
    for(k=0; k<N_SAMPLES; k++){
      //Filter the signal
          out = fast_filter(sig[k], b, buffer, &cont);

      //Preparing the buffer for FFT
          buffer_FFT[cont_FFT] = (double)out;
      cont_FFT++;

          //When buffer gets full points, FFT starts
          if(cont_FFT == POINTS_FFT){
```

```
            my_fft(buffer_FFT, y);
                cont_FFT=0;
        }
      }
    }

    return 0;
}
```

6.3 Validation

Results have been validated through Matlab. In particular, output results of the CPU have been stored as output and plotted with Matlab.

6.3.1 FIR and FFT results

The obtained filtered signal has been compared with the input signal, the result are shown in the following figures. It is trivial to observe that, increasing the order of the filter, a better filtered signal is obtained in output. Of course, there is no difference in the output results between the two different implementations of the filter. In the figure, the blue signal is the input signal, while the magenta is the filtered signal.
The same approach has been exploited to validate FFT results. To check FFT results the order of the filter has been kept constant to $N = 64$, that is the best filter (in terms of filtered signal). Even in this case is trivial to notice that the quality of the FFT improves increasing the number of point for which the FFT is evaluated. In the picture it is shown the behaviour of the different FFTs.
It must be kept in mind that increase the order of the filter or the number of points of the FFT, means increase considerably the number of operations that the CPU must perform, thus increasing the execution time of the functions and the workload of the CPU, and this can be critical for real-time applications, since increasing the execution time of functions could lead to deadline miss. The performance of these algorithm will be discussed in details in the following section.

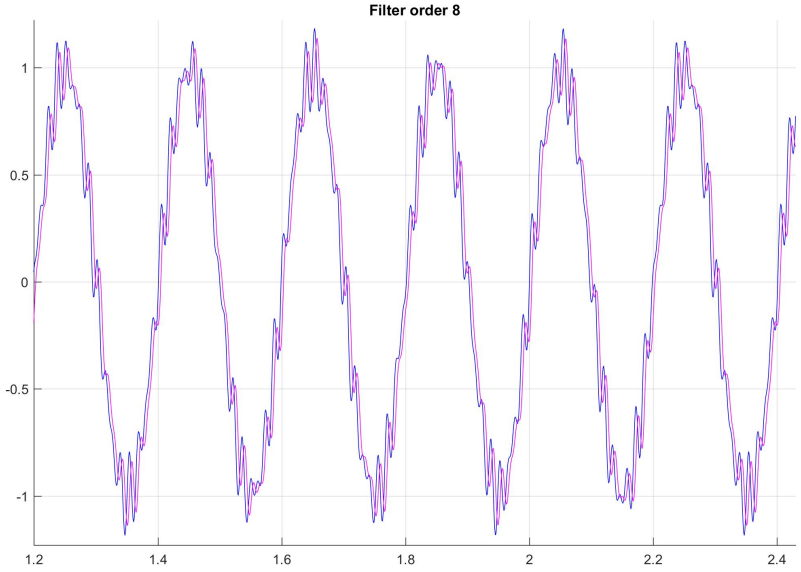

Figure 6.8: Filtered output ($N = 8$)

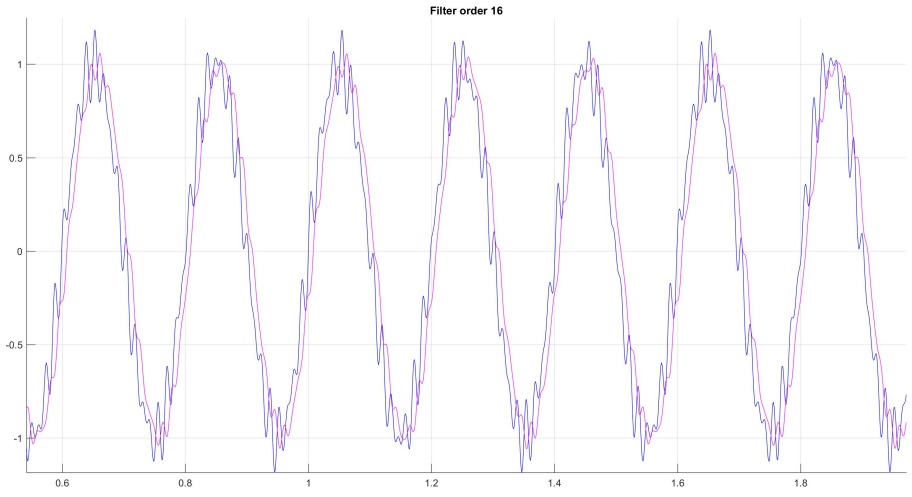

Figure 6.9: Filtered output ($N = 16$)

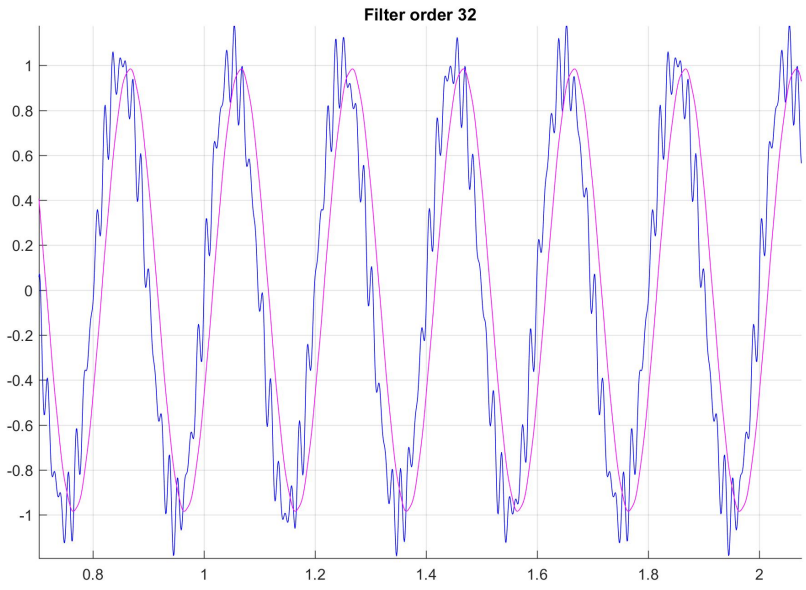

Figure 6.10: Filtered output ($N = 32$)

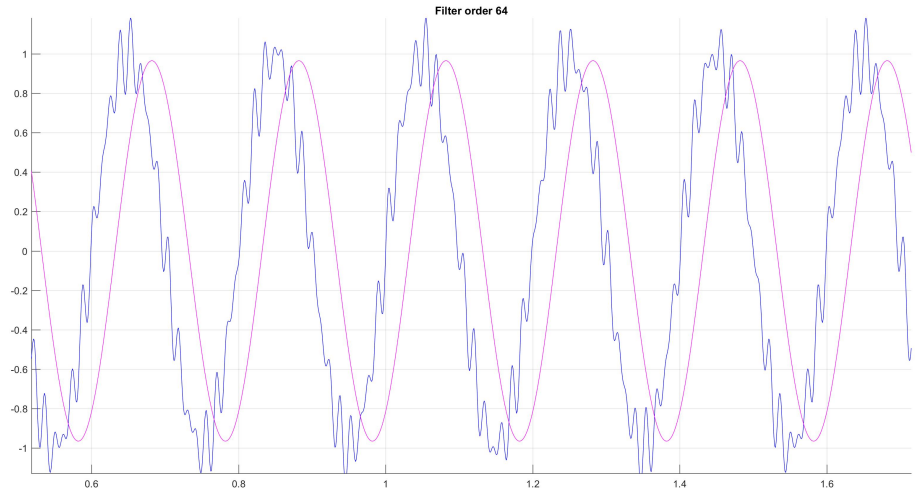

Figure 6.11: Filtered output ($N = 64$)

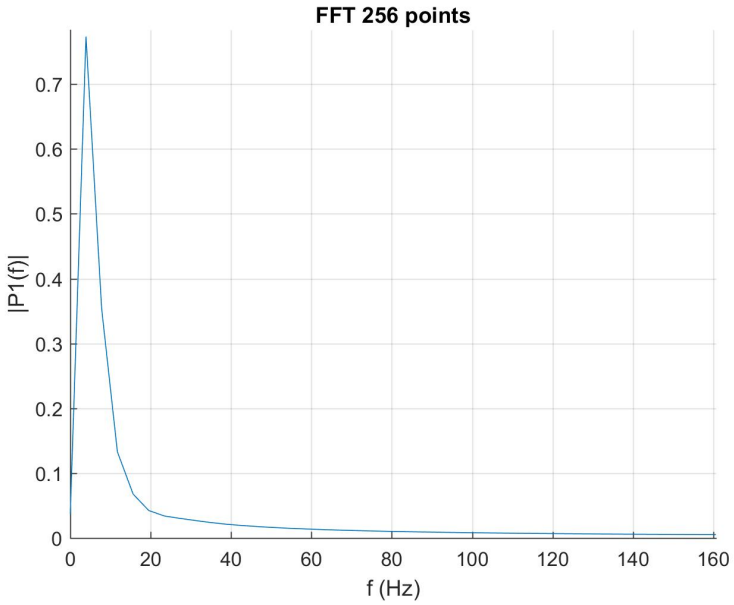

Figure 6.12: Filtered output ($M = 256$)

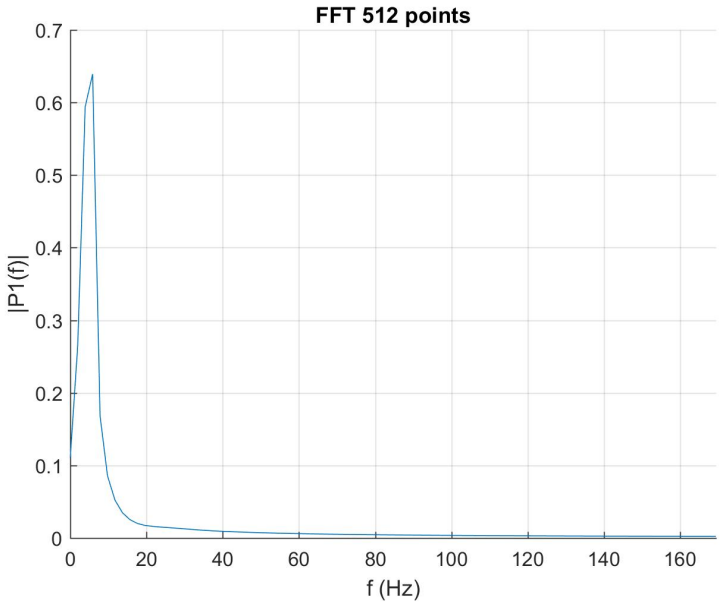

Figure 6.13: Filtered output ($M = 512$)

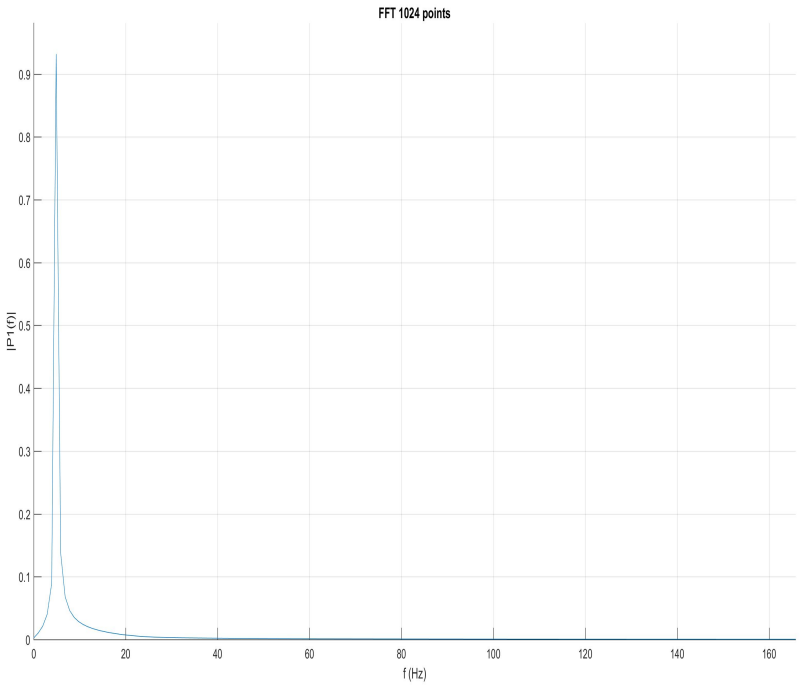

Figure 6.14: Filtered output ($M = 1024$)

6.4 Performance analysis

Now it is performed the analysis of the performances of FIR filter and FFT, that is the part of most interest for this thesis. Numerical results will be given for the single execution times, and it will be proven that the *fast* implementation of the filter is effectively faster than the *slow* implementation.

6.4.1 FIR filter analysis

By using the technology discussed in the previous chapters it is possible to perform the analysis of the different implementations and orders of the FIR filter. Results are shown in figure. To make a comparison between them, the average execution times have been measured. The measurement has been done over an execution time of $10s$, thus means over an huge number of filter functions execution. Numerical results are reported in the table. From this result it is possible to notice that the amount of time saved is not negligible, and there are not drawbacks in terms of used memory, since for the *fast* implementation of the filter, only an extra integer variable is needed. Then, from this analysis it is possible to conclude that there is no reason to chose the *slow* implementation of the filter. Instead it is not possible to say a priori which is the best order for the filter, since no requirements on execution timing are given. If is consider a sampling period of $1ms$, it is possible to conclude that even the most complex designed filter is feasible, because the sampled data can be processed before another data is incoming to the system. This consideration can be done under the assumption that the CPU has only the filtering operation to do, but, if the CPU has to perform other operations, maybe $143.156\mu s$ of execution could be not feasible. It is possible to notice that doubling the order of the filter, cause a doubling in the execution time for the filter. Then, a trade-off between filtering accuracy and execution time must be done.

Filter order	Fast avg time(µs)	Slow avg time(µs)	saved time(%)
8	19.063	27.084	29.6
16	40.740	51.281	20.6
32	75.390	106.953	29.5
64	143.156	210.401	31.9

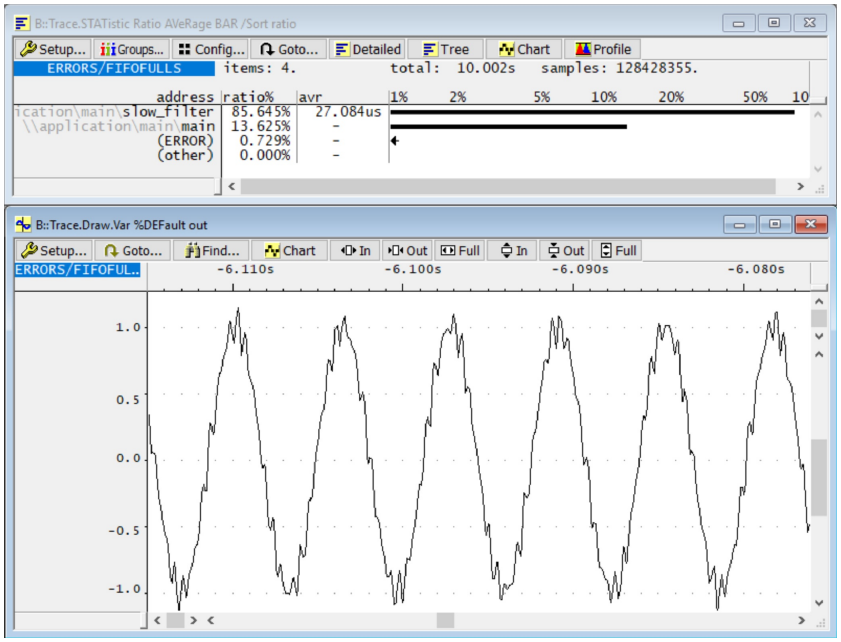

Figure 6.15: Slow filter analysis ($N = 8$)

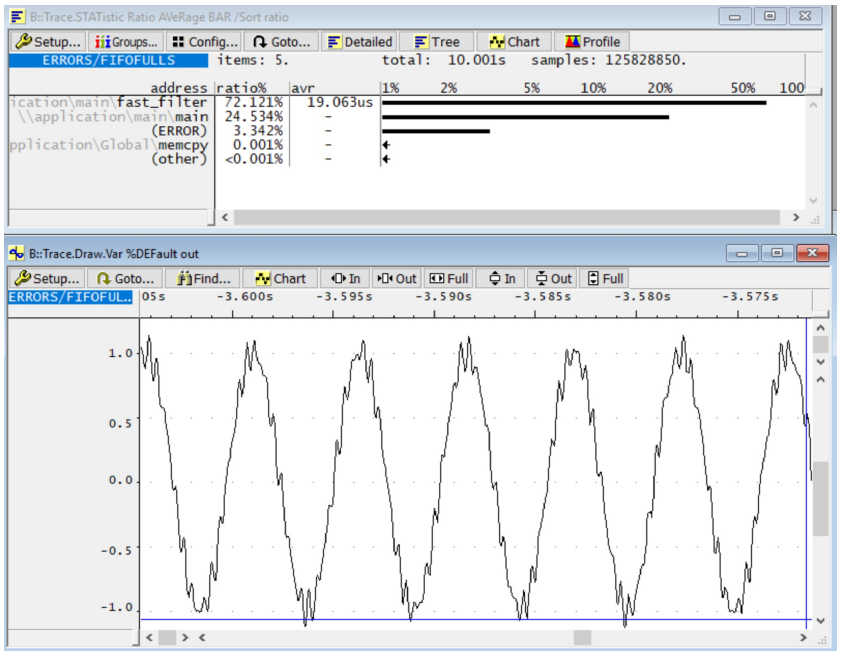

Figure 6.16: Fast filter analysis ($N = 8$)

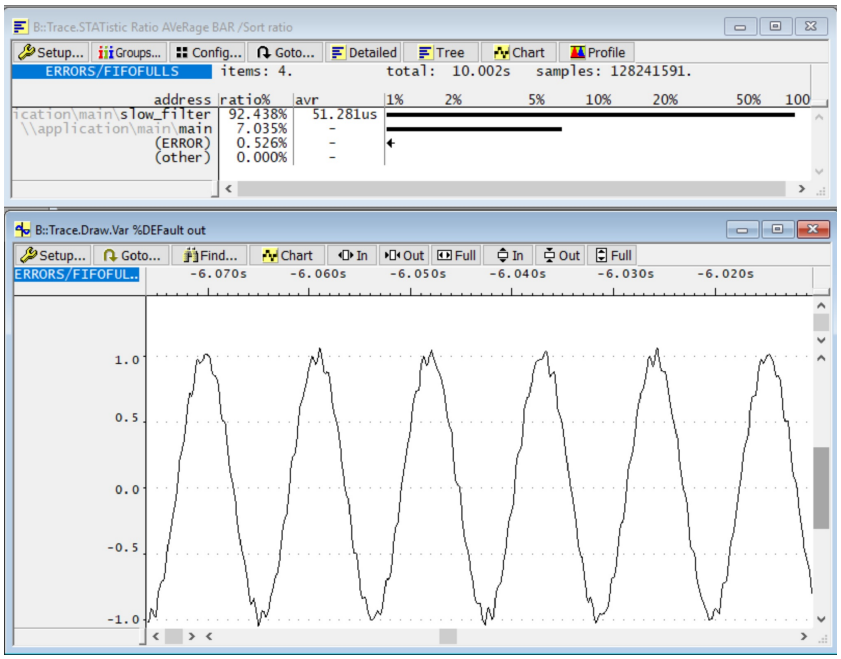

Figure 6.17: Slow filter analysis ($N = 16$)

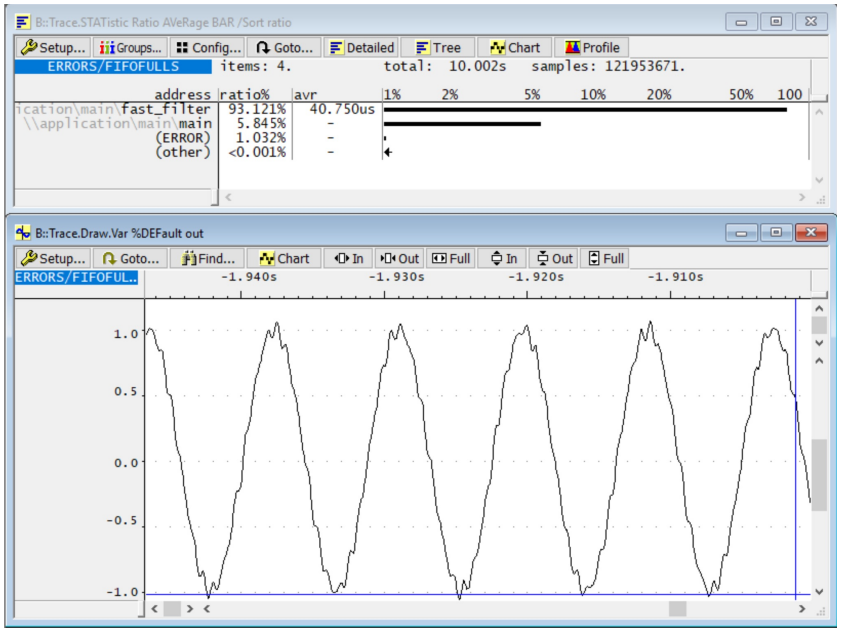

Figure 6.18: Fast filter analysis ($N = 16$)

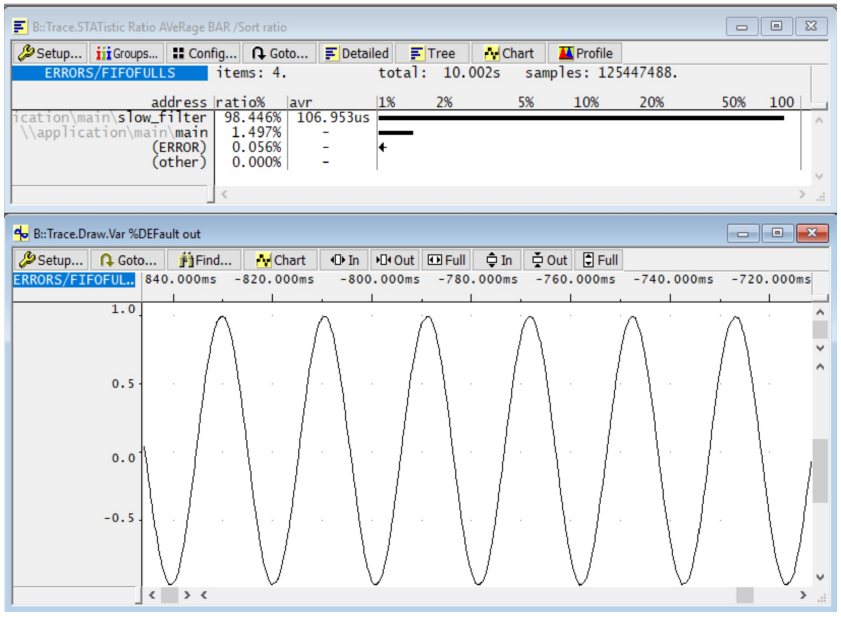

Figure 6.19: Slow filter analysis ($N = 32$)

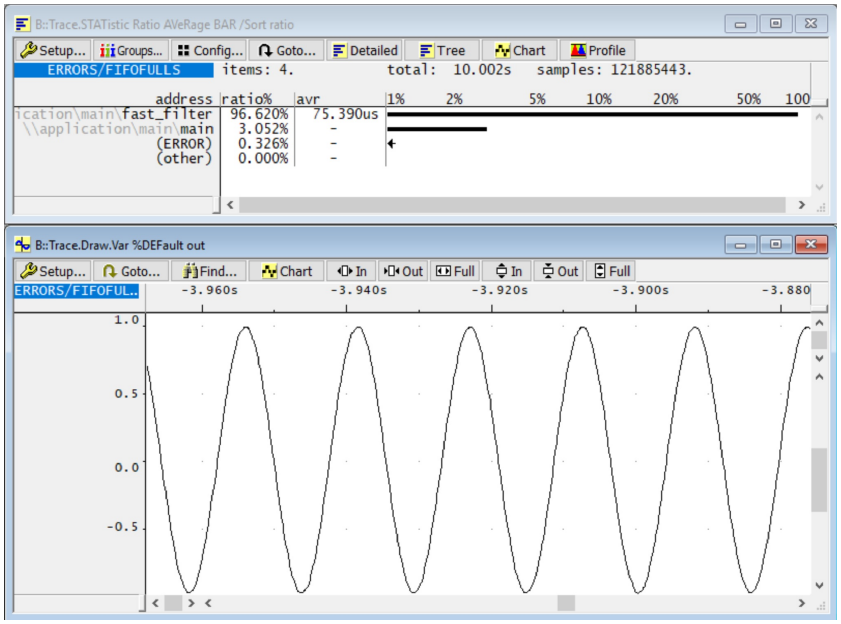

Figure 6.20: Fast filter analysis ($N = 32$)

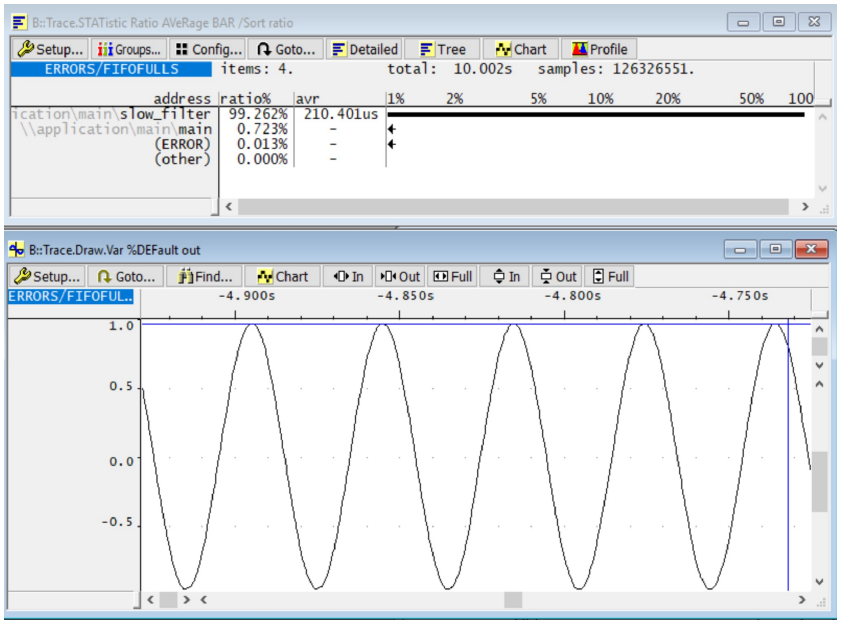

Figure 6.21: Slow filter analysis ($N = 64$)

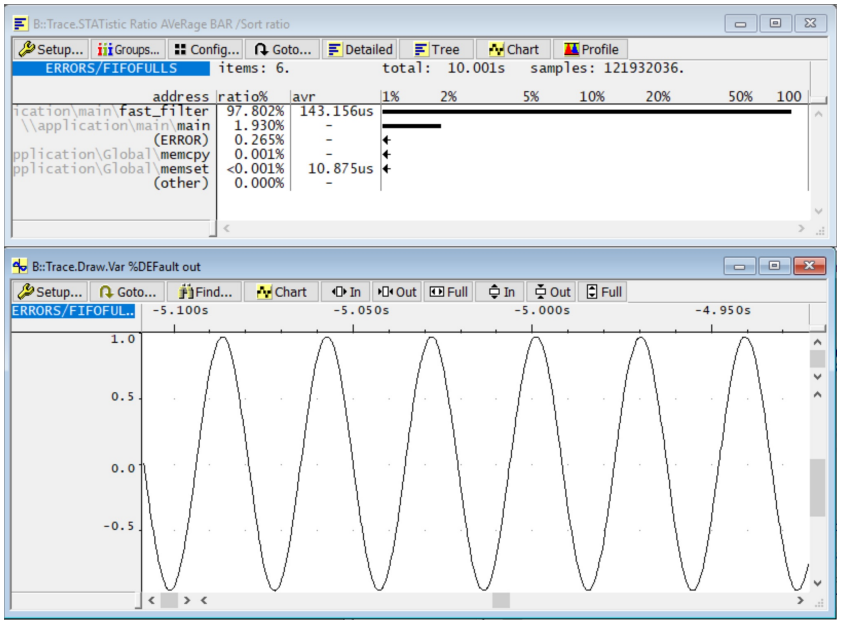

Figure 6.22: Fast filter analysis ($N = 64$)

6.4.2 FFT analysis

While the test of the filter has been done having only the filter operation running on the target hardware, FFT analysis has been performed having the full chain of FIR filter and FFT running on the target. Running only the FIR filter on the target for the previous analysis, do not affect the reliability of the measurement, since only the execution time of the function was the objective of the analysis. In figure is shown that the target runs the filter operation until the number M of samples needed for FFT computation is reached. At this point the execution of the FFT algorithm starts. It is possible to notice that, until this computation ends, the filtering operations on the signal are not executed. This happens because in the bare metal application there is not concept of tasks to execute concurrently the filter and FFT operation. To achieve this result, an operating system should be running on the development board. By the way, this is not affecting the obtained result, since, even this time, the measurement of interest is not the real-time behaviour of the system running both filtering and FFT algorithms concurrently, but only the execution time of the FFT algorithms.
For the measurement of FFT average execution time, it is not possible to trust to the same results used for FIR filter analysis, since FIR algorithm was implemented as a monolithic function, thus the average execution time computed automatically by Trace32 corresponds to the exact value, instead, FFT algorithm is not implemented as a monolithic function due to a great number of calls to different routines because double precision is used. Thus, the average time automatically computed by Trace32 would not take into account these calls to functions. For such a reason, to compute the average execution time of FFT algorithm, since it is a time consuming operation, the runtime method discussed in previous chapter has been adopted. Obtained single execution times have been averaged, and results are reported in the table (precision of millisecond used due to the inaccuracy of runtime method). It is trivial to see that doubling the FFT points, the average execution time is more than doubled.

FFT points	Avg time(ms)
256	45
512	104
1024	272

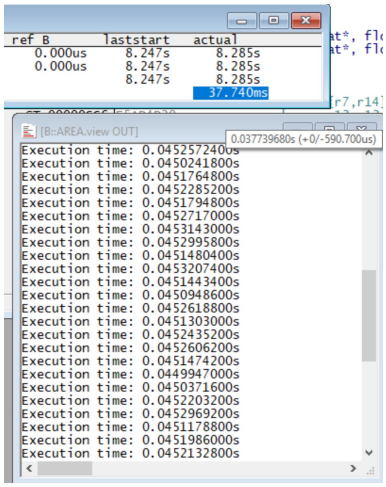

Figure 6.23: FFT runtime analysis ($M = 256$)

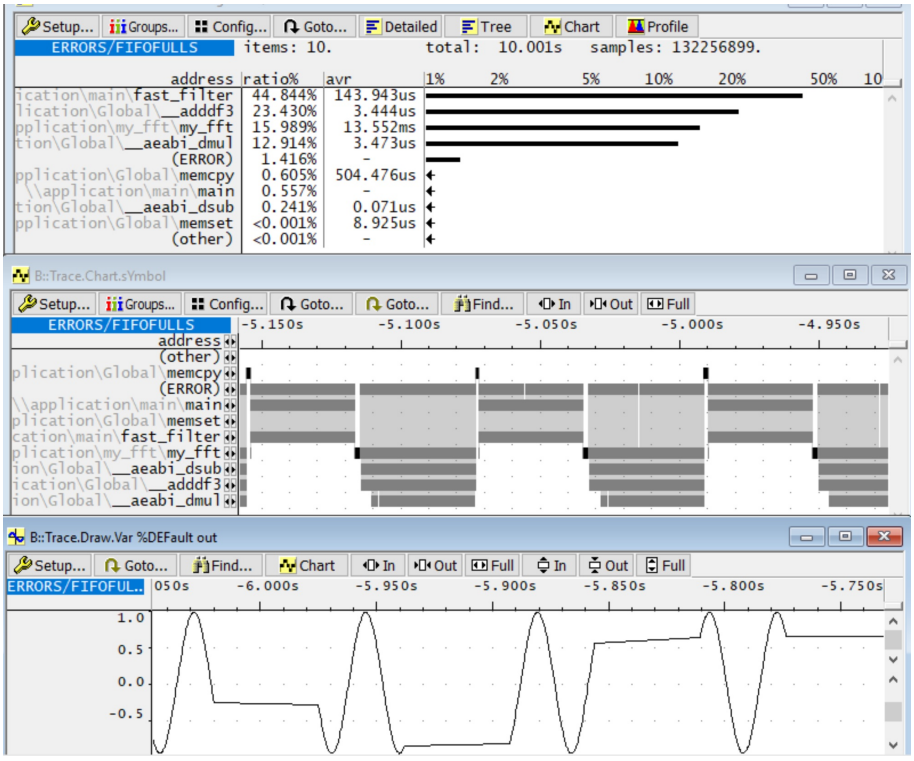

Figure 6.24: FFT analysis ($M = 256$)

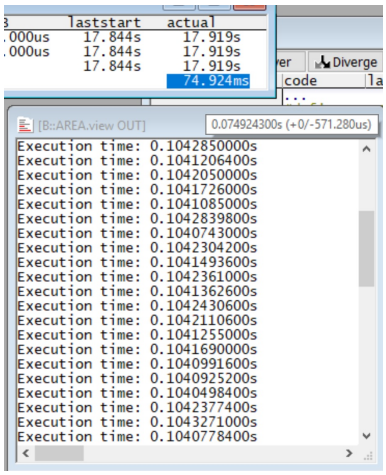

Figure 6.25: FFT runtime analysis ($M = 512$)

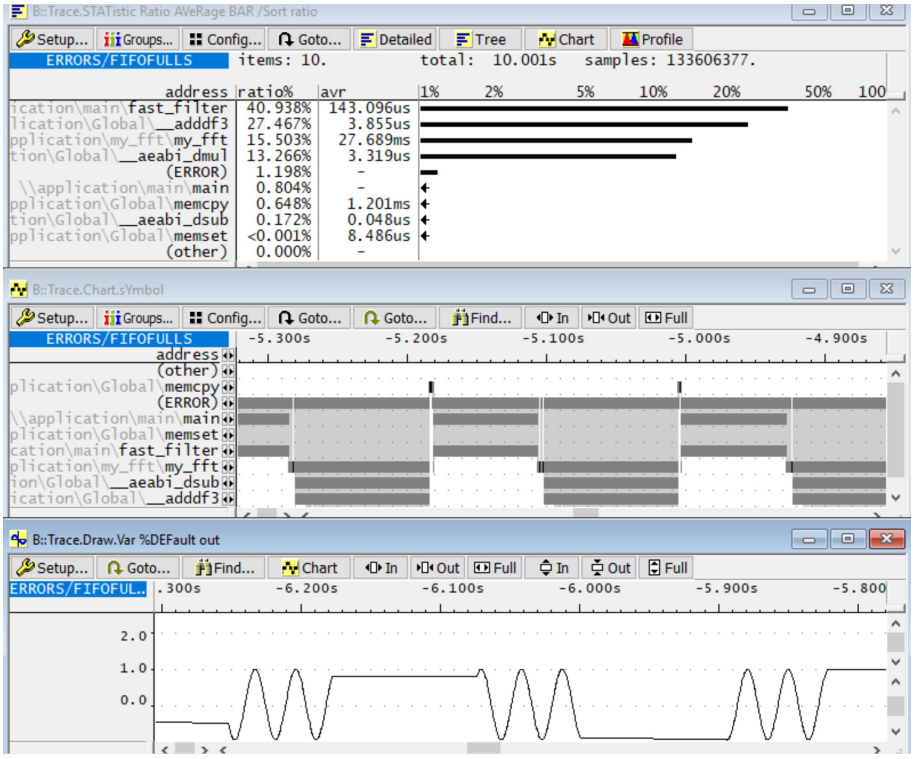

Figure 6.26: FFT analysis ($M = 512$)

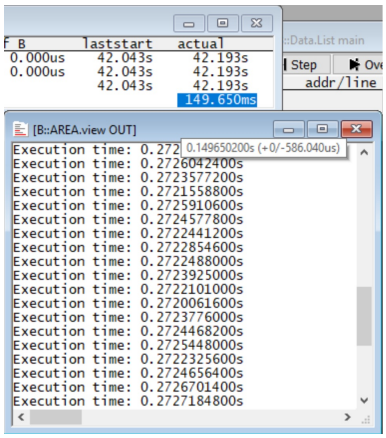

Figure 6.27: FFT runtime analysis ($M = 1024$)

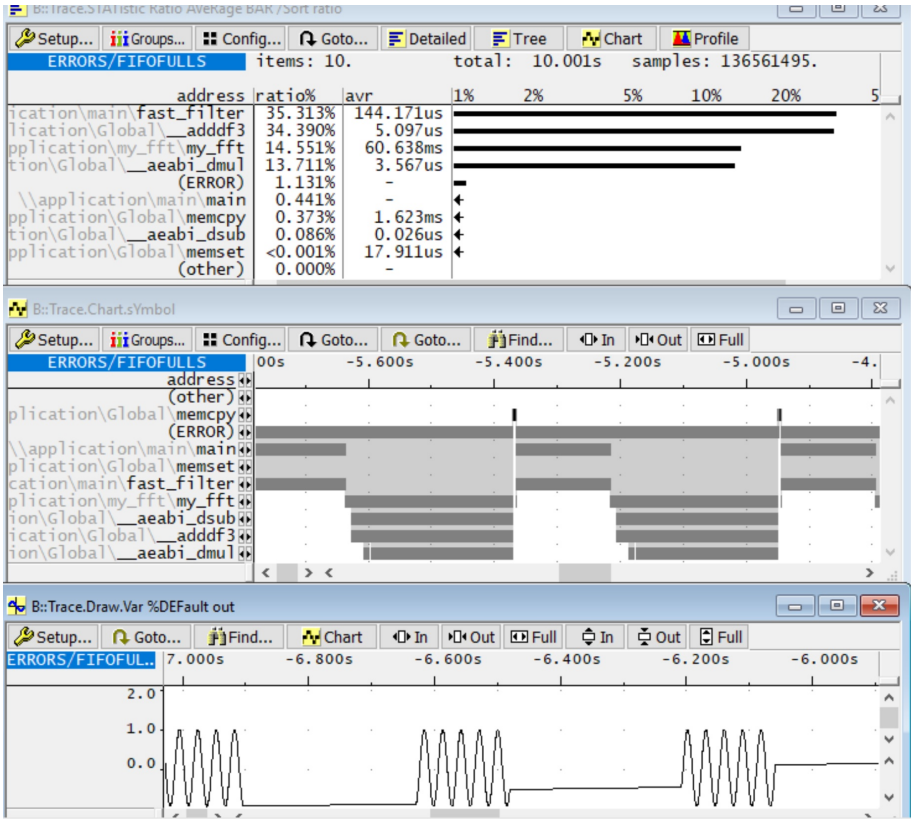

Figure 6.28: FFT analysis ($M = 1024$)

7

Conclusions

The methodology to perform the debugging and tracing operations with this kind of technology it is something that cannot be avoided, because, facing with a safety-critical application, ISO 26262 prescribes that these analysis techniques are mandatory for the development of the application. Using this kind of technology, by the way, still presents some critical points:

- A training period of time is needed to be able to master Trace32 environment
- Set-up script to have a working debug environment is strongly hardware dependant, then changing hardware platform could be time consuming

On the other hand it is possible to conclude that, once the training period has ended, this kind of technology turns out to be useful not only to perform the mandatory tests prescribed by ISO 26262, but it can be useful to save an huge amount of time while debugging the application and performing automatic analysis, due to the possibility to set-up a script that is able to run automatically the desired operations. Of course this turns out to be a great advantage because, thinking that about half of the time for software development is spent in debugging operations, this results into a big economical advantage, with a resulting possibility to reduce the time-to-market.

References

[1] IEEE 1149.7
https://ieeexplore.ieee.org/document/5412866/

[2] CoreSight components reference manual
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0314h/DDI0314H_coresight_components_trm.pdf

[3] TWR-K70F120M User's manual
https://www.nxp.com/docs/en/user-guide/TWRK70F120MUM.pdf

[4] Micrium μC/OS download
https://www.micrium.com/downloadcenter/download-results/?searchterm=pa-arm-cortex-m4&supported=true

[5] Lauterbach Trace32 flyer
https://www.lauterbach.com/product-overview_flyer_web.pdf

[6] μClinux
http://www.uclinux.org/description/

[7] μLinux BSP for TWR-K70F120M
https://www.emcraft.com/products/95

[8] Linux Cortex-M User's Manual
https://www.emcraft.com/docs/linux-cortexm-um-1.12.0.pdf

[9] Lauterbach Debugger Training
http://www2.lauterbach.com/pdf/training_debugger.pdf

[10] Kinetis Design Studio User's guide
https://www.nxp.com/docs/en/user-guide/KDSUG.pdf

[11] Lauterbach μTrace for Cortex-M User's Guide
http://www2.lauterbach.com/pdf/microtrace_cortexm.pdf

[12] Lauterbach μTrace flyer
https://www.lauterbach.com/flyer_microtrace_web.pdf

[13] Kinetis Design Studio Overview
https://www.nxp.com/support/developer-resources/software-development-tools/kinetis-design-studio-integrated-development-environment-ide:KDS_IDE

[14] RTOS Linux Debugging
http://www2.lauterbach.com/pdf/training_rtos_linux.pdf

[15] Lauterbach General Reference - A
http://www2.lauterbach.com/pdf/general_ref_a.pdf

[16] Lauterbach General Reference - B
http://www2.lauterbach.com/pdf/general_ref_b.pdf

[17] Lauterbach General Reference - C
http://www2.lauterbach.com/pdf/general_ref_c.pdf

[18] Lauterbach General Reference - D
http://www2.lauterbach.com/pdf/general_ref_d.pdf

[19] Lauterbach General Reference - E
http://www2.lauterbach.com/pdf/general_ref_e.pdf

[20] Lauterbach General Reference - F
http://www2.lauterbach.com/pdf/general_ref_f.pdf

[21] Lauterbach General Reference - G
http://www2.lauterbach.com/pdf/general_ref_g.pdf

[22] Lauterbach General Reference - H
http://www2.lauterbach.com/pdf/general_ref_h.pdf

[23] Lauterbach General Reference - I
http://www2.lauterbach.com/pdf/general_ref_i.pdf

[24] Lauterbach General Reference - J
http://www2.lauterbach.com/pdf/general_ref_j.pdf

[25] Lauterbach General Reference - K
http://www2.lauterbach.com/pdf/general_ref_k.pdf

[26] Lauterbach General Reference - L
http://www2.lauterbach.com/pdf/general_ref_l.pdf

[27] Lauterbach General Reference - M
http://www2.lauterbach.com/pdf/general_ref_m.pdf

[28] Lauterbach General Reference - N
http://www2.lauterbach.com/pdf/general_ref_n.pdf

[29] Lauterbach General Reference - O
http://www2.lauterbach.com/pdf/general_ref_o.pdf

[30] Lauterbach General Reference - P
http://www2.lauterbach.com/pdf/general_ref_p.pdf

[31] Lauterbach General Reference - Q
http://www2.lauterbach.com/pdf/general_ref_q.pdf

[32] Lauterbach General Reference - R
http://www2.lauterbach.com/pdf/general_ref_r.pdf

[33] Lauterbach General Reference - S
http://www2.lauterbach.com/pdf/general_ref_s.pdf

[34] Lauterbach General Reference - T
http://www2.lauterbach.com/pdf/general_ref_t.pdf

[35] Lauterbach General Reference - U
http://www2.lauterbach.com/pdf/general_ref_u.pdf

[36] Lauterbach General Reference - V
http://www2.lauterbach.com/pdf/general_ref_v.pdf

[37] Lauterbach General Reference - W
http://www2.lauterbach.com/pdf/general_ref_w.pdf

[38] Lauterbach General Reference - X
`http://www2.lauterbach.com/pdf/general_ref_x.pdf`

[39] James Campbell, Valeriy Kazantsev, Hugh OKeeffe *Real-time Trace: A Better Way to Debug Embedded Applications*

[40] International Organization for Standardization *ISO 26262 Road vehicles – Functional safety*

Acknowledgements

I would firstly like to thank my thesis supervisor prof. Massimo Violante, to gave me the opportunity to accomplish this work, under his supervision, and the time spent to give me indications to successfully obtain expected results. I would really like to thank him for introducing myself in the world of embedded systems, that will part of my first professional experience in the following months.
I would like to thank my family, that gave me the precious opportunity to study in this university, involving a huge investment in terms of money, patience and trust.
Thanks to Mario Lauritano to be an exceptional room mate during my first year in Turin, an exceptional reference guide for my studies and an exceptional reference for my life.
Thanks to all my friends that supported my during all these years.