Faculdade de Engenharia da Universidade do Porto

Eclipse based Environment for Multi-Architecture Cross Developments

Flávio Emanuel Santos Dias

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Supervisor: João Paulo de Sousa

July 29th, 2016

Abstract

As the pressure to meet stricter deadlines increases in order to deliver a project in time, hardware and software developers have been struggling to cut the time wasted on activities not directly related to the project. One of tools used to improve the productivity is the Integrated Development Environment (IDE), generally provided by the vendors of embedded system(s).

This dissertation looks into the current state of the art of compilers, IDEs, programmers and debuggers on the attempt of unifying the development environments of some embedded systems architectures, under an Eclipse based IDE, thus providing an open-source and operating system agnostic platform that can be improved without the need to directly engage with the embedded systems vendors. A special focus is given to the current status of the debugging tools, since detecting defects is one of the main contributors to the increasing difficulties in estimating the delivery time of an hardware/software project.

An additional study is performed on non-critical tools, such as version control systems and tools that generate documentation from annotated source files. When correctly used, they can provide a huge boost on the productivity of a developer or a team of developers.

A proof of concept IDE is proposed, proving that the idea of integrating tools for different hardware architectures is feasible and showing that, unfortunately, the weakest part in that integration is the debug component.

Acknowledgments

I would like to acknowledge the support of professor João Paulo de Sousa for his availability to supervise this project and the help of the technical assistants Vítor Pinto and Pedro Galvão.

I also want to express my gratitude to my family, due to their constant support and patience for the last few months.

And to the countless developers that dedicate their free time and knowledge on developing open-source tools without ever asking for money or recognition and are always available to help those in need.

Flávio Emanuel Santos Dias

Contents

List of Figures

List of Tables

List of source codes

Chapter 1

Introduction

1.1 Context

As in many other consumer oriented industries, the embedded systems industry is suffering an increasing pressure to reduce the time to market the products. However, recent market studies [1, 2] confirm an unfortunate trend - more than half of the projects end up being late and a small number is even canceled. In fact, while in 2012 44% of all the projects were completed on schedule or ahead of schedule and 56% were late or canceled [1], from 2013 to 2015 these ratios became 43/57, 41/59 and 38/62 respectively (figure 1.1).

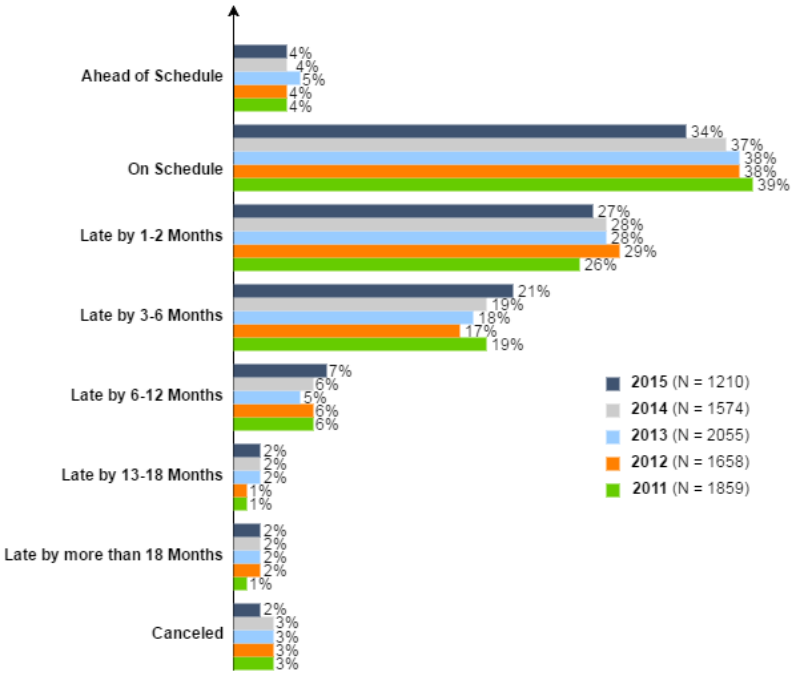

Figure 1.1: Project completion ratios [2, p.23]

Consequently there is an urgent need to correct this situation, by optimizing the more resource-intensive tasks. Currently, on average, 61% of a project's resources are devoted

to software development [1, p.14], making this task the more resource intensive one, and a natural target for optimization.

One of the possible paths to optimization is the use of Integrated Development Environments (IDEs), which consist of a set of software applications, integrated under a unified graphical user interface and incorporating a comprehensive list of functionalities designed to maximize the productivity of the embedded developer.

In its simplest form, an IDE consists of a source editor bundled with a compiler specific to a processor family and programming language, a linker and in some cases a debugger, where everything is abstracted under one interface. As such, the details of each tool are hidden from the user and handled directly by the IDE, considerably reducing the learning curve and the apparent complexity of the development process.

In addition, an IDE can include other functionalities such as intelligent code completion and code analysis software such as standard compliance checkers, introducing a very light concept of *Correctness by Construction*, which is based on two fundamental principles: to make it difficult to introduce errors in the first place, and to detect and remove any errors that do occur as early as possible after introduction [3], instead of waiting to observe the operational behavior of the developed code.

From all the existing IDEs, the software industry in general is gravitating towards either the open source Eclipse framework, or the proprietary Visual Studio, whose main differences are high-lighted on table 1.1, and the reason why only these two will most probably survive in the future is the leverage effect of increasing returns: an IDE that offers already a lot of integration between component tools has an easier job attracting more users, and more users attract more development opportunities. Although this effect works irrespective of whether or not the tool is open source, the embedded systems industry has made to a large extent the choice for Eclipse.

Table 1.1: Main Differences between Eclipse and Visual Studio

Properties	**IDE**	
	Eclipse	**Visual Studio**
Type	Open-Source	Proprietary
Written in	Java	C++ and C#
Operating Systems Supported	Windows, Linux, Mac OS	Windows
C Compiler	External	Internal

1.2 Motivation and goals

Besides the pressure to be more productive, developers often face the need to choose the processor(s) that will be used, in order to maximize performance and to minimize power consumption, while keeping the costs down. In many cases the choice goes to use different

processor chips, from different vendors [2, p.39] on the same project, or to frequently change processors between projects [2, p.61].

And even though there are already multiple solutions aimed to satisfy the typical developers' needs, most of them only support a small set of existing processor architectures and are based on proprietary code, meant to be commercialized. Open-source solutions, on the other hand, usually focus only on one architecture at a time. This situation ends up reducing the productivity of developers of multi-architecture projects by increasing the setup time of the project, including the time required to adapt to new tools.

This is precisely the problem addressed by this work - there is space for a unified IDE, which integrates different tool-chains, as much as possible independent of the operating system and capable of providing developers with the flexibility to tackle the most adequate platform to the task at hand. This will fill a gap not only in industry, where multidisciplinary teams would benefit from a unified programming user interface and environment, regardless of the target hardware platform and underlying operating system, but also in academic environments where students often have to learn different tools for accomplishing similar programming tasks, depending on the chosen hardware architecture.

This IDE is meant to support cross developments for embedded systems in C, the most used programming language in the embedded systems industry [1, p.18], to support different micro controller architectures (Atmel AVR, Microchip PIC, TI MSP430 and ARM) and to be as much as possible independent of the underlying operating system. The three major aspects that were pursued on this work are the following:

1. **OS independence**: many available tools are OS specific. In this work the objective is to integrate tool-chains capable of running on top of different operating systems. This way the resulting development environment can be used regardless of the underlying OS. The goal is to have applications running on Windows and Linux. MacOS is left aside just because it is not possible to guarantee a computer for an adequate testing environment.

2. **Multi-architecture**: most available tools are specific to a certain target architecture. In this work the objective is to integrate, under the same graphical user interface, a set of different tool-chains in order to support cross development for different families of embedded processors. The initial goal is to assess the feasibility of including AVR, PIC, MSP430 and ARM hardware platforms.

3. **Tool integration**: in this respect the goal is to use Eclipse itself, or an IDE based on Eclipse, as a tool wrapper for different open source components, with a special emphasis on the integration of a debugger component for each hardware platform, if possible, since testing and debugging can take up to 20% of the development time [2, p.20].

Additional tools are also considered to be included in the final build of the proof of concept IDE:

1. Documentation generator directly from annotated source files, since the first aspect to suffer when a project is late is usually the proper documentation of the code. Needless to say that the few minutes saved by not properly documenting every day of work will only make the crunch harder as the deadline comes closer [4].

2. A compliance checker for MISRA-C [5] given the recent trend to meet safety & development process standards.

3. Version control capabilities, since it facilitates sharing source code among team members.

4. Last, but not least important, a serial console directly embedded on the IDE, allowing communication with applications without a Graphical User Interface.

1.3 Reading Guide

Following this introductory chapter, the present document reports the review of the current state of the art on free or open-source tool-chains and IDEs in chapter 2. In chapter 3 a related work is described and the differences to this work are explained. Following that, the structure of the Eclipse IDE and why it fits the task at hand is explained in chapter 4. On chapter 5 the focus shifts towards the debugger and its internal components, since it will lead to the functional validation of some of the test algorithms used on chapter 6. Finally, chapter 7 summarizes what has been accomplished and discusses a few still open issues.

Chapter 2

State of the Art

Taking into account the objectives listed in chapter 1, and in order to better understand the functionalities required to implement a set of tool-chains robust enough to attract developers, it was decided to take input from different alternatives in the market. Not only they demonstrate which type of features current developers levitate to, but they also help to better steer the project.

In order to compile an appropriate number of existing alternatives, two key features were used to make an initial selection of candidates. The first one was the possibility of running both in Windows and Linux. Mac OS was not considered for this work since there wasn't a Mac OS computer available to use.

The second key feature was the possibility of incorporating in the IDE a real-time debugger. Based on this criteria a few tools were selected to be further investigated. They are listed in table 2.1 and the findings on each one of these tools are documented in the following sections.

Table 2.1: List of tools to be further investigated

Architecture	**Tools**
AVR	Atmel Studio AVR Eclipse Plug-in
PIC	MPLAB X; Arriba IDE PIC C Builder; Piklab
MSP430	Code Composer Studio MSP430 Eclipse
ARM	ARM DS-5 Development Studio; GNU ARM Eclipse MDK Microcontroller Development Kit 5

2.1 Development tools for AVR micro controllers

2.1.1 Atmel Studio

Atmel Studio (currently at version 7) is without doubt the most complete IDE for AVR micro controllers. Built by Atmel on top of Microsoft's Visual Studio, it inherits the characteristic of only running on Windows Operating Systems. As shown in figure 2.1 the Atmel Studio can be decomposed in several components, with two of them of major interest to this work.

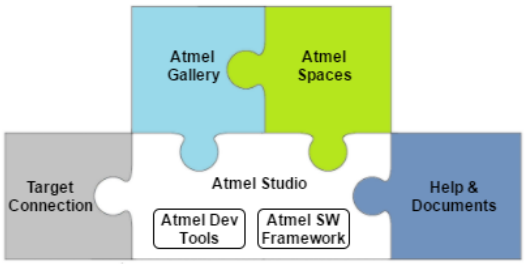

Figure 2.1: Atmel Studio 7 Integrated Development Platform Composition [6]

The first component is the Atmel Dev Tools, which is nothing more than a stand-alone tool-chain based on the free and open source GCC compiler [7] that is used to build AVR dedicated projects. Unlike the IDE, this tool-chain can also run on Linux operating systems.

The other component of interest is the Atmel Software Framework, which is a collection of embedded software for Atmel flash micro controllers, designed to help develop and glue together the different modules of a software project [8]. Although it doesn't exactly fit the proposed objective of this work, it is still an interesting proposal that could be integrated in the final build of the IDE.

Other features include the Atmel Spaces, a cloud-based collaboration work space for hosting software and hardware projects targeting Atmel micro controllers which can be added as an extension to Atmel Studio through the Atmel Gallery [6, p.3]. Likewise, other extensions such as SVN and/or GIT support can also be added. Similarly, serial monitors are also implemented through this chain of suppliers and generating documentation from source files is possible thanks to the integration of the Doxygen Tool (further described on subsection 4.3.1), either as an external tool, or as a plug-in obtained through the Atmel Gallery.

But most important, it offers complete debugging support complemented with simulators of the AVR micro controllers, enabling it to set watchpoints and to inspect registers and I/Os in real-time. Additionally, it is fully compatible with all of Atmel debuggers and programmers, such as the AVR Dragon and JTAGICE [9].

2.1.2 AVR Eclipse Plug-in

Like its name indicates, the AVR Eclipse Plug-in is a plug-in designed to provide Eclipse with tools and settings for developing C programs for AVR micro controllers. Thus, this plug-in is an obvious subject to be further explored as it overlaps most of the requirements of this work.

Since it is simply an add-on to the Eclipse platform, this plug-in guarantees compatibility to every major operating system, as long as the underlying tool chain is compatible with it. This is the case of the AVR GCC Toolchain. Which within the context of the AVR Eclipse plugin the word AVR-GCC tool chain is used quite liberally for all tools and files required to develop and deploy applications for AVR processors with this Plugin [10]. Other characteristics like version control, serial console management and standard compliance checking can be provided by the IDE itself through adequate plug-ins.

In terms of debugging capabilities, the AVR Eclipse Plug-in works with *GDB*, the GNU project debugger, that implements a client-server interface with the client being the user interface and the server either a simulator or a JTAG interface [11], and with a custom program, the AVaRICE, acting as a bridge for both.

Unfortunately, this side of the tool-chain is currently underdeveloped, due to changes on the communication protocol used by the Atmel in the newer ICE platforms. Further investigation of this plug-in can be found on chapter 6.

2.2 Development tools for PIC micro controllers

2.2.1 MPLAB X

MPLAB X is the proprietary IDE from Microchip that runs on Linux, Windows and Mac OS X [12] and it is the reference program to develop for PIC micro controllers and one of the few IDE for embedded systems based on Oracle's Netbeans IDE.

Like Eclipse, multiple productivity oriented plug-ins can be obtained through the IDE plug-in manager or bought using the MPLAB X store. Unfortunately the software offer is reduced, and plug-ins that integrate tools like Doxygen and a serial console still need to be more polished.

In order to enable the debugging of the target embedded systems, the MPLAB X supports both the use of PIC simulators as well as the use of an in-circuit debugger/programmer, such as the PICkit 3. The debugger provides breakpoints, single stepping, Watch windows and all the features of a modern debugger [13]. Supporting all these functions are the MPLAB XC compilers, which can also be downloaded apart from the IDE, offering different optimization levels accordingly to the edition selected, as shown on figure 2.2.

Figure 2.2: Optimization Levels of different versions of the compiler [14]

2.2.2 PikLab

PikLab is an open-source IDE for programs designed to work with Microchip PIC series of micro controllers. It supports Windows and Linux Operating Systems and provides access to a wide array of tool-chains, which unfortunately most are currently discontinued or badly supported.

Additionally, a considerable amount of programmers are supported, although the ability to debug the behavior of the micro controllers is rather small, with only support to the PIC18F series and a few PIC16F series. On the other hand, software simulator support ranging from the PIC10F to the PIC18F family of micro controllers is included [15].

As it stands, this IDE has no utility for this work or for the community built around Microchip products. Moreover, it fails to integrate software to document the source code and version control software, forcing the developer to use those tools outside of the IDE.

2.2.3 Discontinued Eclipse Plug-ins

The Eclipse platform has seen it's fair share of microchip PIC oriented plug-ins in the last years, however none has a defined road map for the next years.

One of them was the Arriba IDE. It was a general IDE that also could be obtained as an plug-in developed by VioSoft Corporation that supported Microchip PIC32 micro controller units due to its MIPS based architecture. That way, it provided support for all three major Operating Systems and replicated most of the functionalities found on MPLAB X through the integration of Microchip's MPLAB XC C compilers, hardware debug tools and simulator [16]. Additional features could be obtained through the use of general Eclipse plug-ins, such as the support for automatic documentation and version control.

However, this IDE hasn't seen any development on the last couples of years, being even incompatible with newer versions of Eclipse.

On the other side there was the PIC C Builder, which was a very small open-source project to implement a tool-chain on Eclipse capable of building projects for Microchip PIC micro controllers [17]. Like the other Eclipse plug-ins here presented, it supported Linux, Windows and Mac OS and could be expanded through the use of third-party software.

However, the existing documentation about its debugging capabilities is scarce and incompatible with newer versions of Eclipse. For this reason further exploration of this tool was abandoned.

2.3 Development tools for TI MSP430 micro controllers

2.3.1 Code Composer Studio

The Code Composer Studio (currently at version 6) is the leading IDE from Texas Instruments to develop solutions to the MSP430 family of micro controllers. It can be obtained through free or paid licenses, with the free version imposing restrictions to the maximum size of a project and type of debug probe [18].

Based on the Eclipse IDE, this software inherits from its base platform the ability to run on Linux and Windows operating systems. Besides this, it also inherits the access to the Eclipse marketplace, as shown on figure 2.3, but with the capability to also access their own App Center. This also guarantees support for other tools like a serial console and version control.

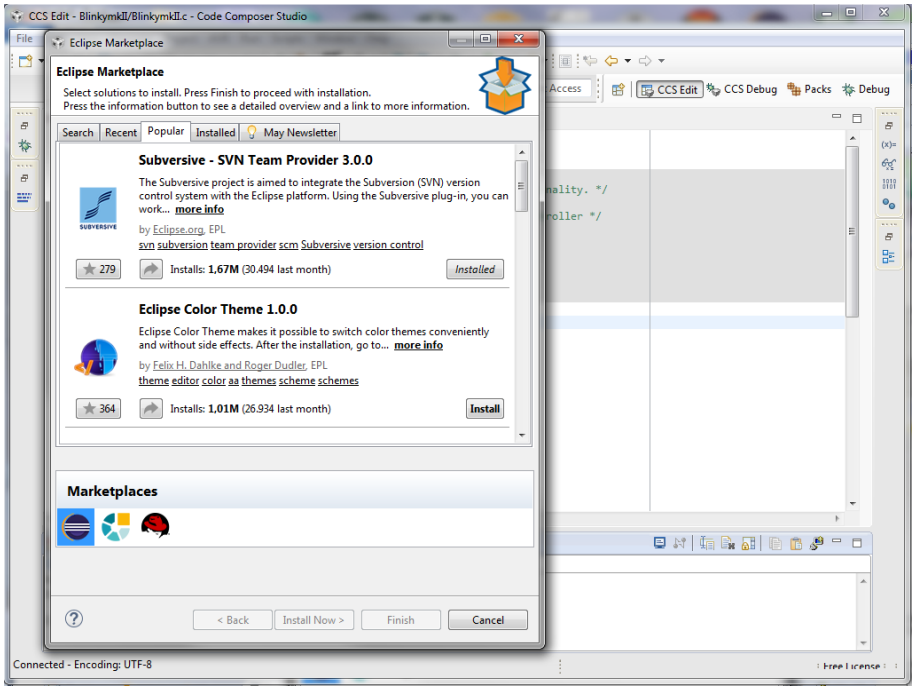

Figure 2.3: Code Composer Studio accessing the Eclipse Marketplace

Under the hood of this IDE is the Texas Instruments proprietary compiler, which brings some extras that differentiates from other alternatives. One of those extras is the EnergyTrace Technology which is an energy-based code analysis tool that measures and displays the application's energy profile and helps to optimize it for ultra-low power consumption [19, p.20]. Other tools like an Optimizer Assistant and an Ultra-Low Power Advisor are also included.

It includes a MISRA-C 2004 standard compliance checker within the compiler, allowing the developer to adapt his personal coding style in order to be compliant with the MISRA C rules, if necessary [20, p.87].

In addition, it can also use MSP430 GCC which is a open source compiler resulting from a partnership between Texas Instruments and Somnium Technologies (formerly maintained by Red Hat until the first quarter of 2016) to bring a new and fully supported open source compiler as the successor to the community driven MSPGCC [21, p.7]. Note that syntax for the Code Composer Studio C compiler and syntax for the Somnnium GCC for MSP430 compiler are not fully compatible so some code needs to be ported between compilers [19, p.61].

It also provides the expected debugging capabilities, such as allowing access to registers and memory of the micro controller, the setup of breakpoints, watchpoints and statistic counters, independently of the compiler used.

2.3.2 MSP430 Eclipse

MSP430 is a plug-in that adds to the Eclipse IDE the capabilities to compile and debug code for Texas Instruments embedded systems [22]. It combines the GNU Project Debugger (GDB) with the MSPDebug tool used as a proxy for GDB, allowing Eclipse to program and disassemble MSP430 projects.

Like Code Composer Studio, it supports breakpoints and memory inspection, without the memory's size restriction imposed by the free version of Code Composer. As any other Eclipse plug-in presented, code version control and automatic documentation are implemented thanks to third party plug-ins.

However, the accepted compiler is still the community driven MSPGCC, not the updated MSP430 GCC created by Texas Instruments and Somnium Technologies/Red Hat, which this plug-in does not support yet.

2.4 Development tools for ARM based micro controllers

2.4.1 ARM DS-5 Development Studio

ARM DS-5 Development Studio is the leading IDE for building and designing applications targeting ARM micro controllers. Based around the Eclipse IDE, this development tool is provided on three different packages, ranging from a free community edition to two paid versions geared towards the business environment with differences ranging between supported compilers, and debugging capabilities.

And although based on the Eclipse IDE, it only provides full support for windows and Linux distributions [23, p.22].

At the heart of this software application are their proprietary compilers, ARM C Compiler 5 and 6, with 6 being exclusively used with the ARMv8 architecture [24]. Unfortunately those compilers are only available on the paid versions. For the users of the community edition there are the Linaro GNU GCC Compilers [25] or the GNU Tools for ARM Embedded Processors [26].

The DS-5 Development Studio also includes their proprietary debugger, with a graphical interface. It supports the software development on ARM processor-based targets and Fixed Virtual Platform (FVP) targets [23, p.14], which enables development of software using a simulator. The simulator, however, does not have absolute timing accuracy, barring the developer from relying on the accuracy of cycle counts, low-level component interactions, or other hardware-specific behavior [23, p.15]. The debugger has all the characteristics presented by a modern debugger, with some extra functionalities reserved for the paid versions like the ability to rewind the program execution and to perform Trace-based function profiling.

2.4.2 GNU ARM Eclipse

GNU ARM Eclipse is a well-rounded and open source family of plug-ins for Eclipse IDE, based on GNU tool-chains. And as many Eclipse plug-ins, it supports Windows, Linux and Mac OS X environments, and third-party plug-ins can be added to integrate source control and other tools mentioned on section 1.2.

There are two approaches to debug projects. The first is to use a JTAG probe with a GDB server running on the computer. This server can be implemented using either the J-link plug-in, specific to the SEGGER J-link probe [27] or the GNU ARM Eclipse OpenOCD, a customized version of OpenOCD [28]. The second can use source machine emulator, named GNU ARM Eclipse QEMU which provides the support for the emulation of Cortex-M series micro controllers [28].

These functionalities are complemented with the integration of the CMSIS Pack technology, a vendor-independent hardware abstraction layer for the Cortex-M processor [29]

which is used to assist the debugger in examining and modifying the micro controller memory and provides access to various documents such as data sheets, reference manuals, and schematics [27].

2.4.3 MDK Micro Controller Development Kit 5

MDK Micro controller Development Kit 5 is a solution from Keil for ARM devices, split between four editions; three paid and a free, but more limited, solution [30, p.7]. Unlike the other development tools for ARM micro controllers, this IDE is based on the μVision IDE and only supports the Windows Operating System.

However, every edition can be characterized by what is called the MDK Core which includes all the components that are needed to create, build, and debug an embedded application (figure 2.4).

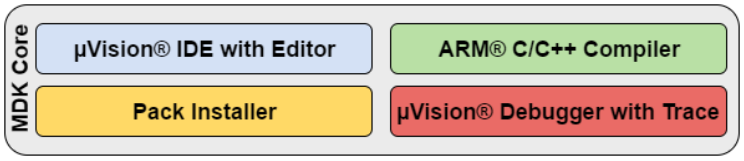

Figure 2.4: Fundamental Components of the MDK core [30, p.7]

Software packs contain information about each micro controller and code examples that are available to be included on a application as building blocks. The previously mentioned CMSIS packs are included in this component but software packs also contain middleware and low-level device drivers [30, p.18]. It uses the ARM C compiler exclusively, even the free version.

As shown in figure 2.4, the IDE also has a debugger with trace, allowing features like sampling, data trace, exceptions including program counter (PC) interrupts, and instrumentation trace as long as they're available in the device under inspection [30, p.64]. Usual features such as stepping, breakpoints and memory inspection are also available.

Unfortunately, this isn't as flexible as other options since it doesn't allow the possibility to add compliance checkers, serial terminals and documentation generators.

2.5 Preliminary Conclusions

Taking into account the information gathered during this survey, it can be concluded that, as already mentioned in section 1.2, the existing open-source projects that implement dedicated tool-chains for embedded systems, usually focus on a single architecture and most of the times lack functionalities found on proprietary integrated development environments.

Fortunately, almost every single option excels on providing support for all the three major Operating Systems on the market. Atmel Studio and Keil MDK 5 are the exceptions. As such, they end up failing one important objective of this work and even though they are well-rounded IDEs, they will be excluded from further investigation, even though Atmel's documentation will play a great part on the development of this project.

Table 2.2 summarizes the conclusions of this survey. In each case we restricted the evaluation to the main platform that is supported, since Code Composer Studio and Atmel Studio also support some ARM micro controllers.

Table 2.2: Summary of each development environment feature

IDE/Toolchain	**Underlying OS**	**Architecture**	**Version Control**	**Doxygen**	**Real-time Debugger**
Atmel Studio 7	Windows	AVR/ARM	Yes	Yes	Yes
AVR Eclipse Plug-in	Windows/Linux	AVR	Yes	Yes	Yes
MPLAB X	Windows/Linux	PIC	Yes	Yes	Yes
Ariba IDE	Windows/Linux	PIC(32 bits)	Yes	Yes	Yes
PIC C Builder	Windows/Linux	PIC	Yes	Yes	Unknown
PikLab	Windows/Linux	PIC	No	No	No
Code Composer Studio	Windows/Linux	MSP430/ARM	Yes	Yes	Yes
MSP430 Eclipse	Windows/Linux	MSP430	Yes	Yes	Yes
ARM DS-5	Windows/Linux	ARM	Yes	Yes	Yes
GNU ARM Eclipse	Windows/Linux	ARM	Yes	Yes	Yes
MDK 5	Windows	ARM	No	No	Yes

Chapter 3

Related Work

This chapter describes a software development environment that has some similarities with this work. It was discovered during the survey of the state of the art.

In spite of the similarities it remains sufficiently different in its goals and approaches but is reported here for the sake of transparency.

3.1 PlatformIO

PlatformIO is a cross platform builder and library manager for a high variety of embedded systems. Initially designed to work with a command line interface, allowing it to be integrated on other IDEs, such as Eclipse, it recently developed their IDE based on GitHub's Atom Text Editor [31, p.09].

The IDE itself, was designed as a special variant of Chromium, an open-source web browser, modified to act as a text editor rather than a web browser since it restricts access to the local system for security reasons. Every Atom window is essentially a locally-rendered web page [32, ch.1].

It supports all three major operating systems, and requires a Python Interpreter to be installed. Likewise, Eclipse requires a java virtual machine to achieve it interoperability between the operating systems.

The core of PlatformIO is a *.ini* configuration file, an informal standard for configuration files for some platforms or software, named *platformio.ini*. This file is divided in sections, denoted by a [header] and key / value pairs within the sections.

In each of these sections the necessary parameters are defined to successfully build the application. For instance, one of the sections stores the directory paths to the source and libraries while other stores the embedded system to be used, its clock frequency and the amount of flash memory. Other section may list source code examples available for that system. A part of this configuration can be seen on listing 3.1, which shows one of the sections for an Arduino Uno development board.

Code 3.1Platformio.ini file

```
# [env:mybaseenv]
# platform = %INSTALLED_PLATFORM_NAME_HERE%
# framework =
# board =
#
# Automatic targets - enable auto-uploading
# targets = upload

[env:uno]
platform = atmelavr
framework = arduino
board = uno
```

As the inclusion of the Arduino Uno board indicates, PlatformIO supports Atmel AVR 8- and 32-bit micro controllers. The rest of the families of embedded systems are also supported with the exception of Microchip PIC line of 8-bit micro controllers. Other families of embedded system are also supported, but are beyond the scope of this work, unlike the 8-bit Microchip PIC products [31, ch.4.8].

Other interesting approach taken by PlatformIO is that its main targets are development boards, not bare metal micro controllers. However, the PlatformIO tool chain is flexible enough and designed to support added custom development platforms.

For example, it creates a target to the Arduino Uno board, but not necessarily to the Atmel Atmega328p that the board uses. That may be because hobbyists and some academic courses which require a light introduction to embedded systems are the main focus of the developers of the tool. Fact supported with the additional support for high-level languages, like *Wiring*, used to program Arduino boards.

Using development boards to quickly prototype something on the embedded systems industry and some advanced courses is quite beneficial, but our work is ultimately more directed to a more deeper approach, using standard C.

PlatformIO also includes a Built-in Terminal with PlatformIO CLI tool and support for a Serial Port Monitor [33]. Since the designer of the the base IDE is *Github*, it uses Git as it's primary tool of version control. SVN compatibility is also attainable through a package (Atom IDE version of Eclipse plug-ins).

No known MISRA-C compliance checker was been included on the IDE. And due to its small market share, this scenario will not likely change in the foreseen future due to expected low return of investment.

Chapter 4

Eclipse IDE

As stated earlier, the Embedded Systems Industry has been using the Eclipse IDE to a great extent. It is not a surprise that many of the vendors researched are using Eclipse as their preferred development environment. However, a detailed answer on why Eclipse is so popular was not given yet. This chapter tries to give an answer to this question by providing an overview of the Eclipse Platform and the aspects that are required to integrate custom tools and make the IDE so popular.

It then proceeds to explain which productivity tools were selected to be bundled with the proof of concept build and when one was not available, the corrective actions taken, leading to the explanation of the replacement of the basic Eclipse IDE by the Texas Instruments Code Composer Studio as the underlying platform for the rest of this work.

4.1 Architecture

The Eclipse IDE is structured as a set of subsystems which are implemented in the form of one or more plug-ins, running on top of a small runtime engine, as shown in figure 4.1, that represents the runtime platform and the generic plug-ins, Workbench, Workspace, Help and Team. The functionalities of this runtime engine are generic by nature, with the plug-ins being the ones to bring specific functionalities. Even more specific plug-ins can be added, one of those being the C/C++ Development Tools (CDT Project) that grants the capabilities of a full-featured C IDE to Eclipse.

This concept is implemented using extension points defined on xml files. This method is used in order to achieve loose coupling between the modules since if components are tightly integrated, late modifications can cause a ripple of changes across the whole system to the point of turning it not viable to work with anymore. So when a plug-in wants to allow other plug-ins to extend its features, it will declare an extension point. The extension point defines a set of rules that other plug-ins must conform to.

When activated, the Platform Runtime discovers the set of plug-ins that are ready to be deployed, reads their manifests, and builds an in-memory plug-in registry, instead of

instantly trying to load them [34]. So a plug-in is only activated when its code actually needs to be run, using the plug-in registry to discover and access their settings. This concept is known as *lazy loading*.

However once activated, a plug-in remains active until it is explicitly deactivated or the IDE is terminated.

4.1.1 Workspace

Returning to figure 4.1, the two aspects that normal users will have most contact with will be the workbench and the workspace. The workspace is the file path were the user is typically working in and where the source files, image files and other artifacts are stored. It can also contain preference settings, meta data and logs. Multiple workspaces can coexist within the same computer.

Too reduce the risk of someone or something accidentally deleting files, a workspace history mechanism keeps locally a track of previous content of the files. The user has some control over this mechanism, managing it via space and date based preference settings.

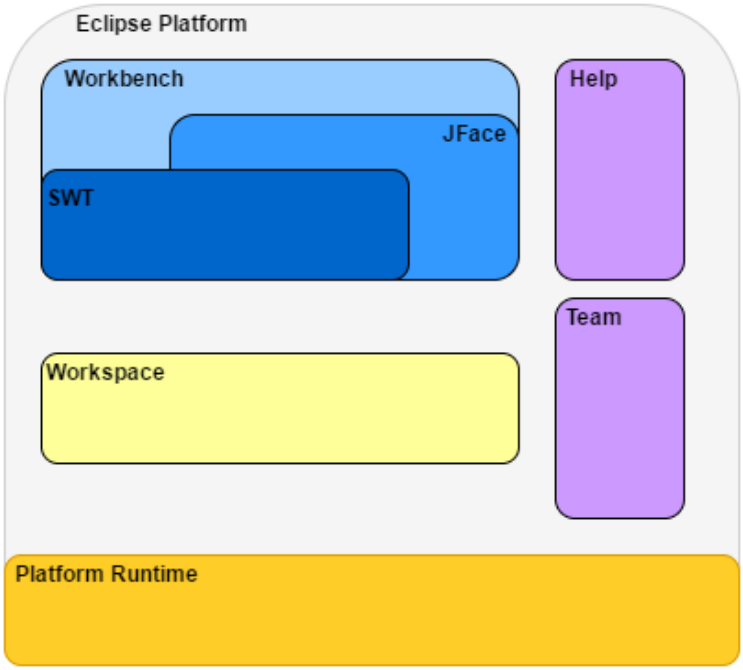

Figure 4.1: Simplified view of Eclipse subsystems [35]

4.1.2 Workbench

The term Workbench refers to the desktop development environment that provides the Eclipse UI personality and the required structures for the interaction of the user with the plug-ins. Its API and implementation are built from two toolkits [34]:

- *SWT*: a widget set and graphics library integrated with the native window system but with an OS-independent API.
- *JFace*: a UI toolkit implemented using SWT that simplifies common UI programming tasks.

SWT is the foundation on which the entire Eclipse UI is built upon. It uses native widgets whenever possible on a platform and supplements them with emulated widgets on platforms where they don't exist; a good example of this is the tree widget that exists in native form under Windows, but is emulated under Linux [36, p.137]. This method allows SWT to keep the same programming style consistent between different systems and homogenizes the look and feel of the Eclipse platform.

JFace is the other toolkit and it is designed to work with SWT without masking it from the developers. It includes the usual UI toolkit components of image and font registries, dialog, preference, and wizard frameworks, and progress reporting for long running operations [34].

It then takes the Workbench API to use those two general purpose toolkits to implement the UI. The interface is based on editors, views and perspectives, although for the user point-of-view it only consists of editors and views since perspectives deal with the arrangement and visibility of these components. Multiple perspectives can exist within the same workbench, but with only one active at any given moment. For example compiling and debugging are two different perspectives of the same project.

As their name states, editors allow the user to open, edit and save objects while views provide information about the objects that are being edited.

4.2 Plug-ins and Development Environment

The popularity of the Eclipse platform with third-party vendors is due to the fact that its modular structure is fully available to them, so that each vendor can concentrate on developing the plug-in fit to their needs, without having to worry about all the details of the platform.

In addition, the plug-in development process itself is very well defined: the Plug-in Development Environment (PDE) is the tool-chain responsible for providing the required tools to create, develop, debug, build and deploy Eclipse plug-ins [37]. This environment can be broken onto three main components, its UI, the API tools and PDE Build.

The PDE UI provides the editors and views that create the full featured environment to develop the plug-ins. The Form-Based Manifest Editors, are one of the most interesting parts of this UI since they are multi-page editors that centralize the management of all manifest files of a plug-in.

The API Tools aids in the documentation and maintenance of the APIs implemented by plug-ins by producing binary compatibility analysis relative to previous versions of a

plug-in and providing specific Javadoc tags that explicitly define restrictions associated with plug-in's methods.

Lastly, PDE Build facilitates the automation of plug-in build processes using Apache Ant scripts, a Java library used to build applications [38]. Those scripts can retrieve the relevant projects from Control Version Systems repositories, jar files and Javadoc files in order to put it in a format apt to be shipped.

4.3 Selected plug-ins and productivity tools

For each hardware platform there must be a plug-in that manages compilation, build and debugging functionalities. In addition to these basic features, the major hurdles that can compromise the productivity of developers must be addressed by a set of specific plug-ins common to all the hardware platforms:

- A tool for generating documentation from annotated C source code tackles the negative effect that poor documentation causes when trying to maintain legacy code or when introducing new elements in a team. A common misstep when any company is working extra hours for extended periods of time in order to finish a project or meet a deadline.

- A version control system, such as git or SVN to improve the ability to securely share source code with other team members and provide a reliable way of restoring projects from failed modifications.

- A tool for style compliance checking, very important in large projects and in case the application requires strict compliance with a given style. In this work, the MISRA-C [5] guide is chosen given its increasing importance and popularity.

- Finally, a serial console is an indispensable way of communication with the embedded hardware platform to retrieve the data it produces or when a debugger is not available, to act as a primitive method of application debugging.

The following plug-ins were selected to implement these functionalities.

4.3.1 Doxygen

Doxygen is the de facto standard tool for generating documentation from annotated C source files [39]. Since the documents are extracted directly from the source code, it is easier to maintain consistency between code and document and besides simple text, Doxygen is also capable of drawing dependency graphs automatically. This function can be expanded by using other programs to design state machines, trees, lists and message sequence charts.

The tool can run externally or using its own user interface, but this method can become cumbersome to some developers. The Eclox plug-in solves this issue, acting as a Doxygen front end and providing a high-level graphical user interface over Doxygen. Figure 4.2 shows the relationship between the two applications.

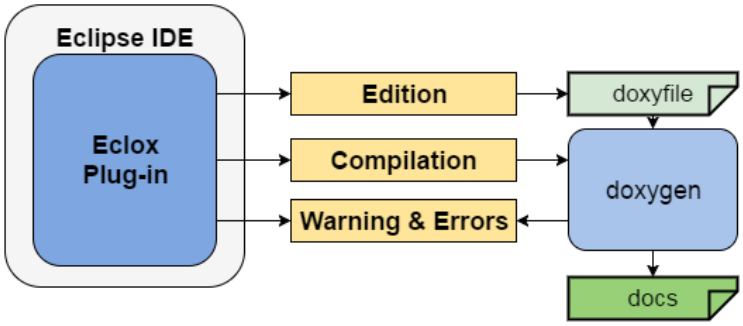

Figure 4.2: Relationship between Eclox and Doxygen [40]

4.3.2 Version Control Systems: Git and SVN

Version Control Systems (VCS) are the recommended tools to deal with the development and maintenance of source code over time by a team of developers. By correctly using a VCS, developers gain access to a complete long-term change history of every file, enabling them to know when a file was created, deleted or edited and by whom.

If by accident or bad directory management the files are deleted or corrupted, turning the whole project too unstable to work, version control systems enable its users to recover these lost files and revert the project to previous saved versions.

Additionally, it allows a team to work concurrently and when a conflict is detected it allows the problem to be discovered and solved without blocking the work of those who were not related to the incident.

As of 2014, the two major version control systems used in the embedded systems industry are the Apache Subversion (SVN) and Git [1, p.82]. Interestingly enough, both tools use different protocols to save and distribute the files between the collaborators. SVN is a Centralized Version Control System (CVCS), where a single server contains the repository, while Git is a Distributed Version Control System (DVCS), where every client mirrors the repository [41, p.31].

Git is an open-source project originally developed in 2005 by the Linux Development community to develop their own VCS in order to maintain the Linux kernel. Eclipse comes already with the Egit plug-in pre-installed, which provides a fairly-complete interface to Git Operations, enabling the user to create repositories and commit changes without using the command-line.

SVN was designed with the objective of correcting the flaws of the Concurrent Version System (CVS), an earlier tool for CVCS type of systems [42, p.XIII]. *SVN* integration is achieved using the Subversive plug-in for Eclipse, enabling the developers to use it

directly from the workbench. An additional download must be done to get the SVN connectors, SVN libraries used by Subversive to communicate with SVN repositories, which are distributed externally and by different providers due to differences in their software licenses.

4.3.3 Serial Console

To implement a fully working serial console inside the IDE, the TM terminal plug-in was selected. When setting up the application, the developer can quickly select the baud rate, the number of bits per frame and other parameters associated with the communication protocol.

Besides acting as a serial console connected to the hardware platform, it can also work as a virtual console accepting commands from the underlying Operating System. Other additional features include support for SSH and Telnet protocols [43].

4.4 Transition to Code Composer Studio

Unfortunately, no open-source or at least free plug-in that could verify the compliance with the MISRA-C standard is available. The solution found to this problem was to use the Texas Instruments (TI) Code Composer Studio proprietary compiler, which has a MISRA-C compliance checker built in.

This led to the decision of using the Code Composer Studio (an Eclipse based environment) instead of a native Eclipse environment as the base IDE to which the rest of the plug-ins would be installed. Decision which did not changed in any order the objectives that were proposed on section 1.2.

From this decision also came the added benefit of inheriting the functionalities added by the TI tool chain described on section 2.3.1, a boon to the developers that were interest in maximizing the performance of their applications or alternatively, minimizing the power consumption on their MSP430 applications. Access to the Code Composer App Center is also possible with this move.

4.4.1 Configuration of the MISRA-C compliance checker

In order to use the compliance checker with projects not concerned with the MSP430 family of micro controllers, two conditions had to be met. To be able to mask the non-compliant standard libraries of some compilers and to be able to call the compiler just to take advantage of the compliance checker and not to compile the source code.

The first condition is all about hiding the standard libraries provided with the micro controller compiler from being verified, since if those libraries are non-compliant, they will only clutter the output with warnings about code the developer is not not capable of modifying. This is achieved using pragma directives, a method specified by the C standard

for providing additional information to the compiler. A generic example is shown on listing 4.1.

Code 4.1Masking libraries with pragma directives

```
#if __GNUC__ /* Replace with your compiler identifier here */
   #include <headerfiles.h>
   #include "yourheaderfiles.h"
#else
   /* This setup was designed to use the TICL430 MISRA-C functionality. */
   #pragma CHECK_MISRA("none")
   #define __PREPROCESSOR_VARIABLES__
   #include <headerfiles.h>
   #pragma RESET_MISRA("all")
   #include "yourheaderfiles.h"
#endif
```

The second condition is achieved by calling the compiler as an External Tool on Eclipse, or implementing a plug-in with that functionality, indicating the directory of the standard libraries and files to be verified.

Unfortunately, even with this procedure, the interrupt declarations are still prone to appear as errors mixed with the compliance analysis. Methods to improve this situation, if possible, are yet to be found and constitute an obvious area for further research.

Chapter 5

Embedded System Debugger

With the increasing complexity of modern embedded systems and the consequent deeper integration of hardware and software, the ability to debug applications on embedded systems is a very critical stage of the development cycle. As previously mentioned, testing and debugging can take up to 20% of development time [2, p.20], so it is in the developer best interest to accomplish this task as efficiently as possible.

This entire chapter is dedicated to develop the concepts associated with the act of debugging.

First a general overview of what is Source Level Debugging is given, before proceeding to explain some concepts related to the GNU Project Debugger (GDB).

The next sections are dedicated to the characteristics that are unique to the world of embedded systems. It starts by discussing the two approaches for Embedded Systems debugging used currently and then the focus is shifted towards the hardware interfaces to enable the communication between the debugger (typically a standard pc) and the debugee (the embedded system): the JTAG interface and the Atmel's proprietary interface, Debugwire, used with low pin-count devices

5.1 Source Level Debugging

One of the most important things a developer needs during a debugging session is the context about the program being debugged. When a bug happens, the user wants the debugger to show the line in the source code where the fault happened as clearly as possible.

But the code that is actually being run by the micro controller is the source code translated to machine code. So the debugger needs to give the illusion that the source code is being directly executed by the underlying embedded system [44, p.12]. That is the concept of source level debugging.

To achieve this, the compiler is instructed to produce extra information about how the source code is mapped into the machine code.

This way of working creates a problem due to compiler optimization, since the compiler will modify the source code to maximize the performance of the resulting machine code, which is specially helpful on systems with limited resources such as small embedded systems. As such, inconsistencies between the original source code and what was translated to machine code may arise.

One possible approach is to simply turn off any optimization during debugging and then deliver the final product optimized, with the belief that if the unoptimized version of the program executes correctly then the optimized will also execute correctly.

However, differences in behavior between unoptimized and optimized program version can be caused by either the application of an unsafe optimization, an error in the optimizer, or an error in the source code that is exposed by the optimization [45, p.1]. So this approach is not always ideal, like when dealing with safety-critical systems.

Other approaches are also possible but are out of the scope of this work [45, ch.02].

Whichever approach is used and in spite of how much can be gained with source level debugging, which is the most effective and frequently used technique for debugging, it is also important that this type of debugger provide an optional access to low-level information, for instance, disassembly views, direct access to hardware registers and memory contents.

5.2 Debugging using GDB

GDB is a portable source level debugger initially written in 1986 that runs on Unix-like native or emulated systems. Microsoft Windows and Linux Operating System fall under this case. It provides a text-based interface to which the user can directly interact, but there are also graphical front ends. The Eclipse IDE C/C++ Development Tooling (CDT) plug-in is one of those cases [46, p.9], invoking GDB unbeknownst to the developer.

5.2.1 Technical Overview

The GDB structure can be divided in two independent parts as shown on figure 5.1, the "symbol" side and the "target" side. They are independent because it is possible to verify the content of the program without running it - the "symbol" side - or debugging it using anything but pure machine code - the "target" side. Nonetheless, the ideal use case is when both are used together.

The "symbol" side is essentially concerned with extracting and displaying the symbolic information of the program. This information includes variable names and types, line numbers and registers used. This is the part of the debugger that is tasked to create the illusion to the user that the source code is actually being run on the embedded processor.

The extraction starts with the Binary File Descriptor (BFD) library, which is used to handle binary and object files. It can handle about fifty different formats, such as those shown on figure 5.1: a.out, COFF and elf, where the last one is commonly used by most embedded systems.

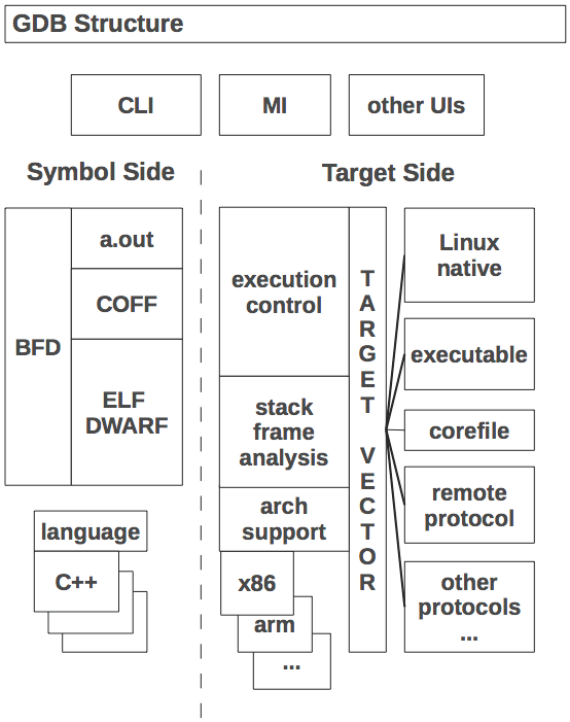

Figure 5.1: Overall Structure of GDB [47, ch.04]

So, porting the BFD is the first key step on tackling a new target architecture, a very important feature for the developers of embedded systems.

The information extracted is then grouped in symbol tables, that sometimes grow up to occupy multiple gigabytes of RAM. Indeed, each local variable, each named type, each value of an enumeration, is a separate symbol. Even though this case won't happen frequently with small systems, GDB also employs partial symbol tables, allowing it to start up in only a few seconds, even for large programs. This is achieved by selecting which symbols to load, looking for just the globally visible symbols to record them in the symbol table. Complete symbolic info for a function is only filled if the user stops inside the function. Bit fields and structures can also be used to reduce the size of the symbolic table.

The "target" side is concerned with the control of the target system. It starts and stops the program, catches signals and manages registers. The method that it chooses to apply this control depends on the system. For Unix-like systems, it uses a special call named *ptrace* to read and write the state of the process.

But for cross-debugging an embedded system the target side constructs message packets to send over the physical connection between both, and waits for response packets in return.

To handle the intrinsic structure of every target system, there is a need to endue the debugger with enough flexibility to handle each one of them. This is accomplished with target vectors where each target vector is implemented as a structure of several dozen function pointers, whose purposes range from the reading/writing of memory and registers, to resuming program execution, to setting parameters for the handling of shared libraries.

The potential of this method can be enlarged by mixing methods from several target vectors. So before a program starts to run, GDB uses the target vector for executables and reads from the binary file. But while the program is running, the bytes should instead come from the process's address space, using the target vector for live processes, named Target Stack.

But in order to reach this flexibility the debugger needs to have an in-dept knowledge of the system, defined as gdbarch objects, which details each target architecture.

5.2.2 Interfaces for GDB

GDB, when used as a standalone environment, offers two text-based user interfaces: the traditional command-line interface (CLI) or a more expanded text user interface (TUI), with separate zones to display the source code, register values, and so on [48, p.3], as seen on figure 5.2

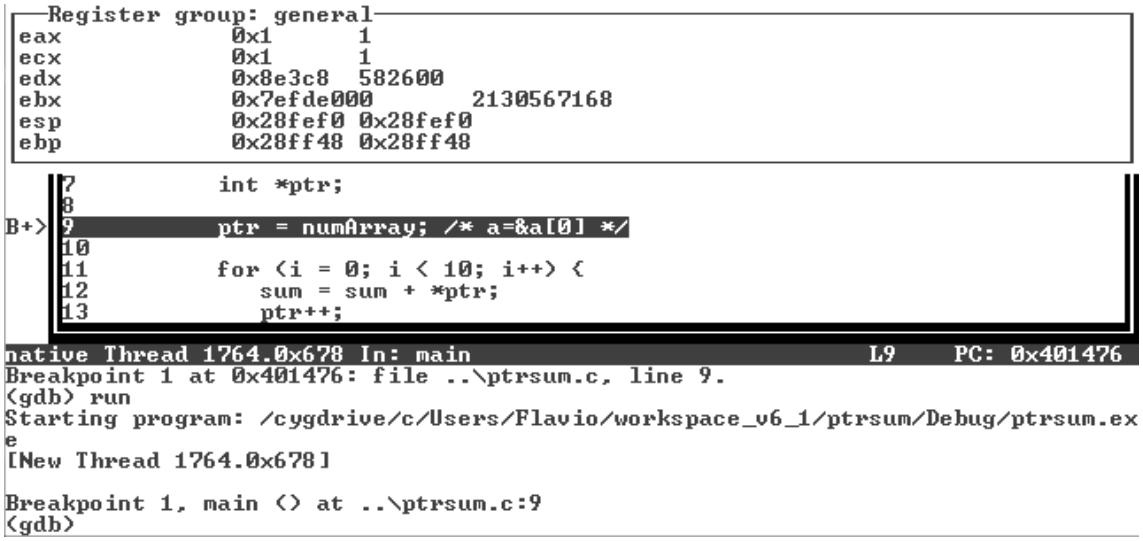

Figure 5.2: Text User Interface of a simple adder

An alternative "user" interface, known as the Machine Interface or MI for short, is also provided. It still is fundamentally a command-line interface, but both commands and results have additional syntax to format them to recognizable data. This is the way that the GDB is integrated into Eclipse, for example [47, ch.02].

Other alternative is possible, with GDB implementing a structure similar to a backend of a graphical interface program, translating mouse clicks into commands and formatting print results into windows.

5.2.3 Standalone debugging of embedded systems

As mentioned on section 5.1, the first step that needs to be taken is to inform the compiler to map the source code to machine code. This is done by passing some arguments when

the compiler is executed.

On a native computer application, the next step would be to invoke the debugger. However with embedded systems applications, a remote server is first used to connect the system with the debugger. GDB offers a remote mode designed to debug programs running on a machine that cannot run GDB on the usual way [49, p.203]. This is the case with the embedded systems studied here with the exception of Microchip PIC family of micro controllers.

Using as an example a MSP430 micro controller as the debuggee, a remote server over IP will be initiated, listening to port 55000. The debugger can then connect itself and `load` the program to be debugged, or attach itself to the program being currently run by the micro controller. The first case is demonstrated in figure 5.3.

```
(gdb) target remote :55000
Remote debugging using :55000
__crt0_start ()
    at /opt/redhat/msp430-15r1-105/sources/tools/libgloss/msp430/crt0.S:60
60      /opt/redhat/msp430-15r1-105/sources/tools/libgloss/msp430/crt0.S: No suc
h file or directory.
(gdb) load
Loading section .rodata2, size 0xc lma 0xc000
Loading section .text, size 0x2b6 lma 0xc00c
Loading section .data, size 0x4 lma 0xc2c2
Loading section __reset_vector, size 0x2 lma 0xfffe
Start address 0xc00c, load size 712
Transfer rate: 1 KB/sec, 178 bytes/write.
(gdb) _
```

Figure 5.3: GDB connected to remote target on port 55000

After this, the user can issue several commands to control and inspect the state of the application. The usual first step is to set breakpoints around the zone where the user suspects the fault will happen. To do so, the user issues the `break` command whose argument specifies a location, which can be a function name, a source line number, or even a machine address. GDB assigns a small positive integer to the breakpoint object, which the user subsequently uses to operate on the breakpoint.

However the number of breakpoints that can be set is finite, specially on flash based embedded systems. This issue is exacerbated by the fact that breakpoints share the same structure with other similar functions, such as watchpoints [47, ch.02], which breaks when data is accessed rather than when some instruction is executed.

Typically, the following step is to run the application until some event is triggered. The command `run` or in certain cases `continue` are used in these cases.

If the event is triggered, the user can then use the debugger to print the variables values, evaluate complex expressions or even analyze the stack. If necessary, the user can utilize this break to have a more granular control of the program execution, with the `step into` or `step over` commands. Both make the program run a statement at a time, with the difference being that the `step over` command skips over functions calls.

When the task to debug the application is completed, the user can `quit` to end the session. While other commands and functionalities are also available, they are only used on

more complex situations. In fact, more advanced uses of the debugger will depend of not only the interface used to communicate with the embedded system but also the embedded system itself.

5.2.4 Integration into Eclipse

A command-line interface is a serviceable tool for debugging, but for some developers and in the context of this work, integration with the IDE is a desirable feature.

As mentioned before, an interface between Eclipse and GDB can be created thanks to the Machine Interface (MI). Although this interface is to complicated to a normal user, it's ideal for communication between software processes, which the Eclipse CDT enables by creating a pseudo-terminal (pty) that sends and receives data. It then starts GDB and creates two session objects to manage debug data [50].

So when the user clicks the Debug button, an instance is called to provide the debugger with the launch object containing configuration parameters, the name of the executable to be debugged, and a progress monitor.

It then creates two session objects to manage the rest of the process. `MISession` and `Session`. The first manages communication with the GDB while the second one connects the GDB session to the CDT debugging framework.

All of this work is essentially transparent to the user, since he will only have contact with the Eclipse's debug perspective. Many of the views — Breakpoints, Modules, and Expressions — are provided by Eclipse, but CDT adds three views to the perspective: Executables View, Disassembly View, and Signals [50].

If the debugger isn't already integrated on the IDE, it still can be called using Eclipse *Debug Configurations...* tool. If the target to be debugged is an embedded system, the *GDB Hardware Debugging* launch configuration is the one used, since it enables the user to connect to the target remotely. Similarly, the GDB server can be configured and launched as an external tool, if necessary.

When the debug session is launched the IDE will swap it's current perspective by the debug perspective, if necessary. This perspective can be seen on figure 5.4. There, the user can use various functionalities that reduce the complexity and time of the debugging session described on the following paragraphs.

A more detailed view over the source code is available, with the statement where the debugger stopped highlighted. Hovering the mouse over a variable will show their current value, as long the variable as been previously initiated.

Since viewing the source isn't enough in some more complex cases, a disassembly view can also be used in parallel with the source code to inspect the execution in more detail. This situation is is exemplified on figure 5.4, with the next step on the code being highlighted.

Generic breakpoints can be placed on the left side of the source code view, seen on figure 5.4 as blue circles on lines 8 and 10, marking the location where the user wishes

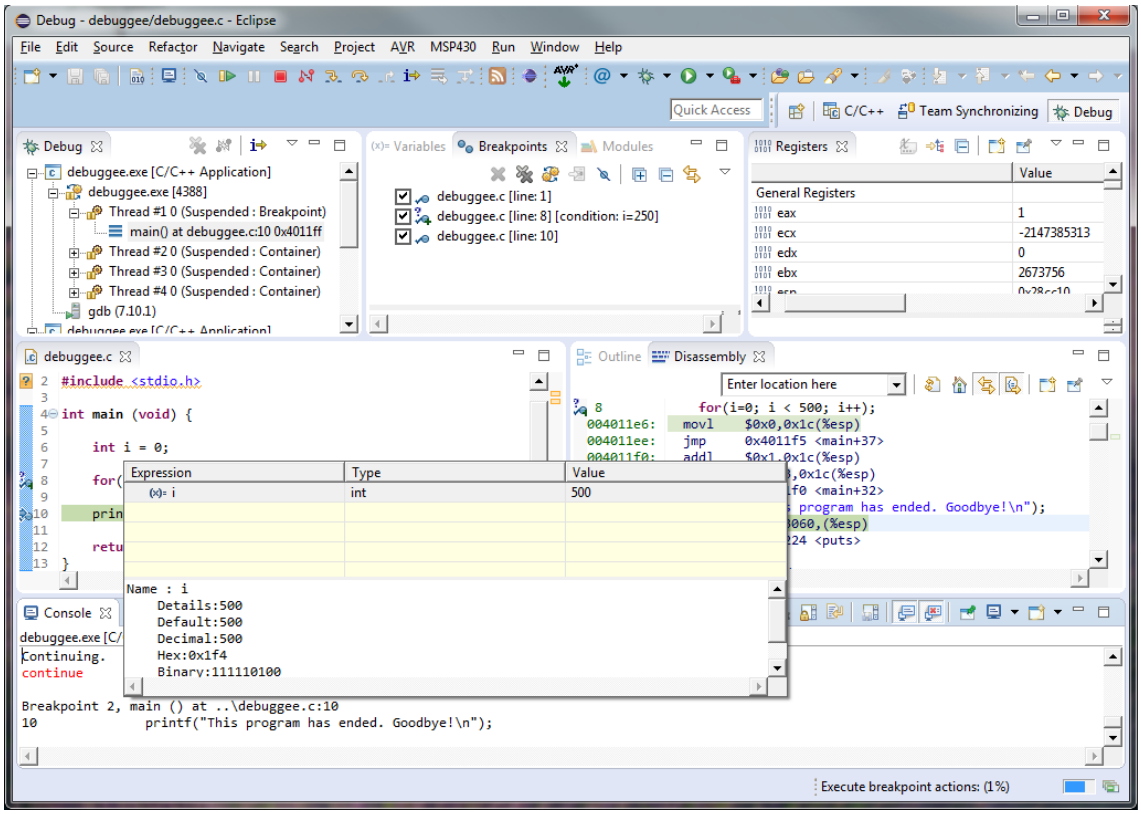

Figure 5.4: A debug session on Eclipse

the program to stop. Right-clicking enables/disables them and sets conditions that must be fulfilled before the breakpoint can fire. Alternatively, a specific view to set breakpoints and edit them is also available.

Commands to stop, start and resume are easily available on a toolbar and can be even more quickly accessed with key shortcuts. If that is not enough, the user can also control the debugger with the Eclipse built-in console, having access to all of its commands through there.

5.3 Approaches for Embedded Systems Debugging

Due to the inherent difficulty in debugging embedded systems, various types of probes were engineered to increase the amount of information a debugger can collect without interfering on the system's behavior.

Two implementations rose over the others, and are the most frequently used today: *In-circuit Emulators* and *On-chip Debuggers*.

5.3.1 In-Circuit Emulator

An In-circuit Emulator (ICE) is a tool that allows the replacement of the embedded processor by emulating it inside the probe. This emulator can monitor everything that goes

on in the on-board CPU, giving complete visibility into the target code's operation [51].

The first models were were quite cumbersome, requiring the developers to replace the processor temporarily with a hardware emulator - a more powerful although more expensive version. Nowadays, modern emulators use the target system's processor directly, accessed through a JTAG connection [52].

This grants the developer the ability to examine and change the contents of registers, memory, and I/O. Placing breakpoints in the source code is also possible. These breakpoints can be of two different types: *Software Breakpoints* or *Hardware Breakpoints*.

While Software Breakpoints work by replacing the destination instruction by a software interrupt, or trap, or instruction, Hardware Breakpoints act as comparators, comparing the program counter with the specified address that is stored on some special debugging registers. This utility is highly dependent on the micro controller since it requires physical components, they are extremely limited in number.

Whenever possible, its preferable to set Hardware Breakpoints, since they can be set on both volatile and non volatile memory. Software Breakpoints, on the other hand, can only be easily set in RAM memory.

Real-time trace is another functionality that the ICE provides. It captures a snapshot of the executing code to a buffer, at full speed, saving thousands of machine cycles, displaying the addresses, the instructions, and transferred data, which can be translated to assist the developer [51].

And since this tools doesn't use any resource from the embedded system under test, it becomes a critical tool when a real-time application needs to be tested.

5.3.2 On-Chip Debugger

While the previously mentioned In-Circuit Emulator services a wide range of debugging opportunities, it can be in some cases too expensive to develop and more difficult to use. So in order to find a different method to debug embedded systems, many semiconductor vendors started to integrate dedicated debug circuitry into their chips [53]. Most of the times, it consist of adding software debug capabilities to the existing JTAG ports.

This way, it is possible to monitor and control the execution of the program inside the embedded system. Like the ICE, the *On-Chip Debugger (OCD)* is capable of using Software Breakpoints as well as Hardware Breakpoints to modulate the run time of the program. Again, Hardware Breakpoints are limited.

Likewise, real-time trace is technically possible to implement using the OCD, since on-chip debug agents may reside inside cache and memory management units, granting them the capability to see addresses and data values just like the CPU sees them.

5.4 Debugging through JTAG

Although the original intent of the JTAG interface was for providing a technology for testing Printed Circuit Board Assemblies (PCBAs), the embedded systems industry saw the underlying technology - the four-wire JTAG technology communications protocol - as an opportunity to gain access to registers, enabling functionalities such as debugging and programming.

This implementation required not only the embedded system vendors to equip their products with the JTAG interface, but also the probes to connect the system to a computer and the necessary software to interface their probes with the debugger.

Two open-source projects aimed at implementing this interface are of special interest. *OpenOCD* which aims to act as an universal application for various embedded systems architectures, and *AVaRICE* which is an interface designed to support the Atmel AVR family of micro controllers. This last tool is of special interest since it can also emulate an Atmel proprietary debug method, the *Debugwire*.

5.4.1 OpenOCD

OpenOCD aims to support debugging, in-system programming and boundary-scan testing for embedded target devices [54, p.1]. Besides supporting the JTAG signaling, it is also compatible with the Serial Wire Debugging (SWD) signaling, a low pin count and high-performance alternative to JTAG [55].

When this software is run, it starts by processing the configuration commands provided, either by command-line or a configuration file. If the information that it received is valid and a JTAG compatible device is connected to the computer, OpenOCD starts running as a daemon [1] [54, p.13].

When a client finally connects, such as GDB, their commands are processed and then issued to the JTAG compatible device. The GDB and OpenOCD connection is made possible due to the latter compliance with the remote gdbserver protocol.

Setting up this connection implies starting up the OpenOCD configured to support GDB, and then configuring GDB to connect to OpenOCD. This connection can be established either using a socket over the TCP/IP protocol, or by using pipes [54, p.132].

However, there are some aspects that still need to be ironed. One of those is the limit to the number of breakpoints that can be set. While OpenOCD can read this parameter, it is the user that needs to inform the debugger of this number.

[1] A program or process that runs in the background but remains inactive until invoked.

5.4.2 AVaRICE

Even though nothing on the structure of the OpenOCD prohibits it from supporting Atmel AVR family of embedded systems, AVaRICE stands out for supporting an Atmel proprietary interface, named Debugwire, a two line communication protocol ideal for micro controllers with low-pin counts.

AVaRICE runs on any POSIX compatible machine (Linux or Linux-like environment for Windows) and it is designed to connect to GDB via a TCP socket. The commands issued by the debugger are translated to the protocol recognized by Atmel JTAG compatible devices.

Unfortunately, the development of AVaRICE has been lagging behind the introduction of new products because the vendor changed a communication protocol on more recent models, such as the *Atmel ICE*[2] or Atmel Xplained mini series.

Debugging with the AVaRICE and any JTAG ICE limits the maximum number of Hardware Breakpoints to three with no support for Software Breakpoints at all [56, p.05]. Even worse is the fact that only one byte hardware watchpoints are supported, each of them counted as a hardware breakpoint on the limit of three breakpoints.

Debugging with Debugwire is even more limited, compared to JTAG, since it does not offer the functionality to place hardware breakpoints, so they have to be implemented by directly rewriting flash pages using `BREAK` instructions. Some memory spaces (fuse and lock bits) are also not accessible through the DebugWire protocol [56, p.05].

This is the price to pay for such a low-cost system.

[2]Even though it is called an Atmel ICE, it is fact an On-Chip Debugger probe type

Chapter 6

Project Development

In order to test the tool-chains and IDEs presented on chapter 2, two aspects of this work need to be addressed. First, the acquisition of a sample of each embedded system architecture - Atmel AVR, Microchip PIC, Texas Instruments MSP430 and ARM - as well as programmers and debuggers. Secondly, the source code to test if the considered tool-chains were in fact functional and, in no way, affected the behavior of the application when programmed.

Therefore, the two next sections deal with these aspects, presenting the embedded systems used and the source code used to test them.

Then, each suitable tool-chains is evaluated regarding the possibility of them being included on the final build which as mentioned on chapter 4, will be the Code Composer Studio. Considerations about possible modifications, effectively applied or suggested as future work, are also presented.

6.1 Hardware acquisitions

For each of the architectures under study, at least one micro controller was acquired in order to test the tool-chains. Since not every single micro controller has a direct connection to the computer, so that it could be programmed and/or debugged, additional tools were also acquired when deemed necessary. Budget was a concern during this selection phase thus, whenever it was possible, hardware already existent was used.

For testing tool-chains for Atmel AVR platform, an Arduino Uno sporting a *Atmega328P* and a *ATtiny85* micro controller were used. And while the Arduino Uno has an programmer built-in, no internal debugging functionality was found. To solve this issue, an Atmel JTAGICE was acquired, soon followed by an AVR JTAGICE MK II, an older model, but better supported. This also solved the problem on how to properly program the ATtiny85 without the support of the Arduino Uno board.

Microchip PIC tool-chains were tested using a chipKIT Uno32 board, equipped with a PIC32MX320F micro controller. Since the tool-chain for 32-bit embedded systems is

separated of the tool-chain for 8-bit embedded systems, a PIC16F1825 micro controller was also used. For programming and debugging, the Microchip PICKit 3, an on-chip debugger, was selected.

The MSP430 tool-chains were tested using the popular Launchpad board series by Texas Instruments. The MSP-EXP430G2 Launchpad was primarily used until a bug on the Code Composer Studio compiler on Linux forced the acquisition of the MSP-EXP430F5529LP Launchpad. No additional programmer or debugger were required since both boards provide a debug communication pathway between a host computer and the target micro controller. Access to an Olimexino-5510 board using an MSP430F5510 micro controller was also granted, but the board wasn't used besides some preliminary tests.

Finally, ARM tool-chains were tested using an Olimexino-STM32 board equipped with STM32F103RBT6 micro controller, which is based on the ARM Cortex-M3. No additional programmer or debugger were used. Additional hardware was expected to arrive under the ARM University Program contract, but unfortunately the process suffered many delays resulting on a poorer experience with ARM tools and micro controllers.

Table 6.1 summarizes this information.

Table 6.1: Acquired hardware

Architeture	**Development Platform**	**Debugger/Programmer**
Atmel AVR	Arduino Uno ATtiny85	Atmel JTAGICE AVR JTAGICE mkII
Microchip PIC	chipKIT Uno32 PIC16F1825	Microchip PICkit 3
Texas Instruments MSP430	MSP-EXP430G2 MSP-EXP430F5529LP Olimexino-5510	None
ARM	Olimexino-STM32	None

6.2 Test Code Algorithms

Four test programs were designed to test the validity of the tool-chains, taking into account the capabilities of the micro controllers used. The simplest one only implements an infinite loop, powering one of their pins on a fixed interval of 1 second, using a software delay. This code was then evolved, combining a timer with an interrupt.

An external interrupt algorithm was also developed, to assert how the debuggers handled interruptions. While the previous algorithm could be used for the same purpose, the scenario created by this implementation is more easy to debug, since it depends on the user to explicitly trigger the interruption.

Finally, a new program was designed to use a hardware peripheral. That peripheral was chosen to be the Analog to Digital Converter.

With the objective of aiding the development of the programs and collection of data, LEDs, buttons and potentiometers were employed sparingly and the IDE of each embedded system vendor was used to develop the code, even they weren't considered fit to be properly studied like *Atmel Studio*, in order to speed up the development of the applications.

On the following sections, each one of these program's algorithms are explained. The source code of each example can be found on appendix A.

6.2.1 Software Delay

The first algorithm implements an infinite loop that at every one second changes the state of one of the pins of the micro controller. The delay is implemented using software, that is, a finite cycle with a predictable time period. If a LED is connected to that pin, it will blink with the period of one second.

Then in order to change the output value of the pin, an exclusive OR (xor) is used. This logical operation outputs true only when inputs differ. The algorithm 1 is as follows:

Algorithm 1 Software Delay Algorithm.

1: $PORT \leftarrow Output$
2: $PIN \leftarrow 0$
3: **while** true **do**
4: $\quad SoftwareDelay(1s)$
5: $\quad PIN \leftarrow PINxor1$
6: **end while**

6.2.2 Timer Interrupt

This algorithm is an evolution of the previous one, that instead of using software delay, uses the micro controller timer, which counts in sync with the micro controller clock, hence being more precise.

Additional steps need to be ensured so that the program behaves correctly. The frequency of the clock needs to be properly declared, and the prescaler defined, that is, the number of clock ticks to skip. For example, if the prescaler is set to 64, the timer will only count every time the clock ticks 64 times.

From balancing these two factors, a value for the timer can be derived that when reached, will trigger the interruption that needs to be preconfigured to fire when the comparison is between the intended value and the actual value of the timer is reached.

The algorithm 2 is as follows:

Algorithm 2 Timer Interrupt Algorithm.

1: $PORT \leftarrow Output$
2: $PIN \leftarrow 0$
3: $Clock \leftarrow SetValue$
4: $Prescaler \leftarrow SetValue$
5: $TimerTop \leftarrow TopValue$
6: $Interrupt \leftarrow SetTriggerOnCompareMatch$
7: $Interrupt \leftarrow Enabled$
8: $Timer \leftarrow Enabled$
9: **while** true **do**
10: **end while**

11: **if** $Timer = TimerTop$ **then**
12: $Interrupt \leftarrow Triggered$ ▷ The program exits the cycle when interrupted
13: $PIN \leftarrow PINxor1$ ▷ Consequently the output changes
14: **end if**

6.2.3 External Interrupt

One of the most challenging aspects of debugging embedded systems is controlling the behavior of the system during an interrupt. This is what this algorithm is for. Truthfully, the previous algorithm also allowed to verify this type of behavior, but the user wouldn't have any control on when the interruption would be triggered. it would become quite cumbersome.

This algorithm solves this issue by relinquishing control of the interruption to the user. It depends on the user changing the value at the input on one of the Pins of the micro controller, for example with a button, for the interruption to fire.

Using the configuration with a button connected between the ground and the interrupt pin, two aspects need to be taken care of. First, select the interruption to trigger when the input changes. Secondly, to only fire when a certain transition occurs. On the case described, when the button is pressed the input changes from High to Low.

Algorithm 3 displays it as follows:

6.2.4 Internal Peripheral

The final algorithm was designed to configure and use one of the many internal peripherals that normally exist on a micro controller, such as an *Analog to Digital Converter* (ADC) or a *Pulse-width modulation* (PWM). In the end the ADC was chosen.

There are multiple steps that need to be taken in order to configure and enable the ADC. First, a pin must be configured as an analog input and the ADC must know which one of them it will use.

Then, the frequency at which the conversion is done must be set as well as the reference voltage, the maximum value that the ADC can convert. Lastly, the ADC is enabled. The main execution will wait for the conversion to end, and then depending if the values

Algorithm 3 External Interrupt Algorithm.

```
1: PORTA ← Output
2: PORTB ← Input  ▷ To simplify the algorithm, the input is located on another port.
3: PINA ← 0
4: Interruption ← SetTriggerInputB
5: Interruption ← SetTransitionInputB
6: Interruption ← Enabled
7: while true do
8: end while

9: if PINBchangedfromHightoLow then
10:     Interrupt ← Triggered          ▷ The program exits the cycle when interrupted
11:     PINA ← PINAxor1                ▷ Consequently the output changes
12: end if
```

surpasses any threshold, activate one of two LEDs. Following that, a new conversion is promptly started again. The algorithm 4 is as follows:

Algorithm 4 Peripheral Algorithm.

```
1: PORTA ← Input                       ▷ This port will control the inputs
2: PORTB ← Output                      ▷ This port will control the outputs
3: PORTA ← SetInputAsAnalog
4: PINB ← 0                            ▷ Output is initially driven low.
5: ADCclock ← SetClock
6: ADCreference ← VReference
7: ADCpin ← PINA                       ▷ The pin that will be read
8: ADC ← Enabled
9: while true do
10:     if ADC = DONE then
11:         if ADCValue > Threshold then
12:             PORTB ← PB1
13:         else
14:             PORTB ← PB2
15:         end if
16:     end if
17: end while
```

6.3 Evaluation of the tool-chains

6.3.1 AVR Eclipse Plug-in

The first tool-chain that was tested was the AVR Eclipse Plug-in for Eclipse and consequently Code Composer Studio that can be directly acquired from the Eclipse Marketplace.

It interfaces the IDE with the open-source avr-gcc, a subset of the gcc tool-chain, and the avrdude, a utility to download programs to AVR micro controllers. They can be built

on Linux or Windows, if an emulated POSIX environment is presented. Alternatively, the suite of pre-built tools can be downloaded for Windows operating Systems with the WinAVR suit.

However, the later procedure isn't currently recommendable, since it is quite out-of-date. This has some repercussions on the ability to use some of the newer versions of the Atmel JTAGICE, mainly because only newer versions of avrdude support them.

After the avr-gcc is successfully recognized by Eclipse, the tool-chain is ready to be used with any project. Creating a new project is in no way different from a normal C/C++ project. Eclipse has the feature of being able to load from the AVR micro controller its architecture and clock frequency, as long as only one micro controller is connect at time, as can be seen on figure 6.1.

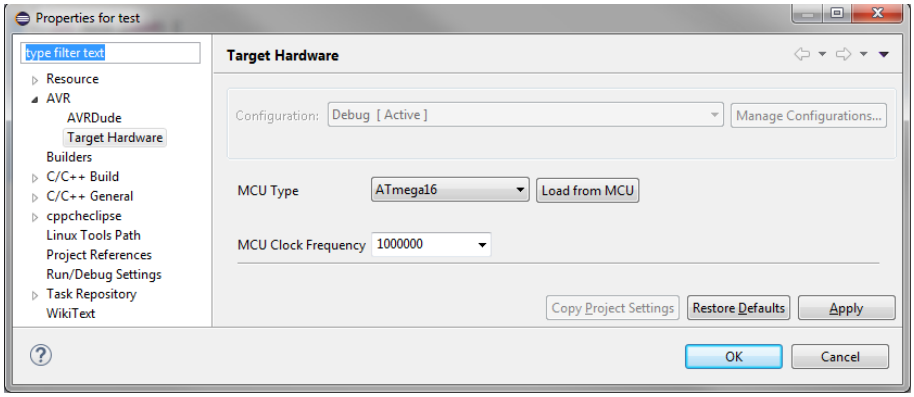

Figure 6.1: Loading from the target hardware

This feature is implemented by avrdude, and this tool is clearly the center of all the tool-chain. Furthermore, as a programming tool, it is compatible with every Atmel JTAGICE probes. On the other side, the debugger isn't as developed, mainly due to AVaRICE, the tool that acts as a server to avr-gdb. As mentioned on chapter 5 it is quite limited.

Thus this tool needs to be improved in near future to be usable, or alternatively, port its defining features to OpenOCD. In the end, the evaluation was positive and the tool-chain was deemed important enough to be included on the IDE.

6.3.2 MPLAB X

MPLAB is the proprietary IDE from Microchip to develop for their line of PIC micro controllers. Unfortunately, it is based on the Netbeans IDE, and the other open-source alternatives for Eclipse were deemed unsuitable and too unstable to work with.

Fortunately, the MPLAB XC Compilers can be obtained independently from the IDE and as such interface with Eclipse. Some preliminary test indicate that the compiler can indeed be integrated on the IDE, but debugging capabilities of this set up are yet to be fully investigated.

Nevertheless, without any other alternative, the compiler will be included on the final version of the IDE. Further developments on it's integration, such as developing a plug-in to ease in the integration are left to future work.

6.3.3 Code Composer Studio

As mentioned on chapter 4, Code Composer replaced Eclipse as the base IDE. Since it was the only way found of easily accessing a MISRA-C compliance checker.

Additionally, the ability to use either Texas Instruments or the free MSP430-GCC tool-chain on the same IDE, grants the developers more freedom on how to approach their projects.

Code Composer Studio is guaranteed to be part of the final build.

6.3.4 ARM DS-5 Development Studio and GNU ARM Eclipse

From all the embedded systems architectures studied here, only ARM offered the opportunity to really choose between both competing tool-chains.

The community edition of DS-5 is quite restricted in relation to their paid brethren. However, it still offers the same core experience has the open-source GNU ARM Eclipse, since both use the same underlying tool-chain: The Linaro GNU GCC Compilers.

Lamentably, the ARM board that was available wasn't very friendly to work with, since it had been designed to be used with a deprecated IDE with a high-level programming language. If it wasn't for the ARM University application that the college had applied for, an acquisition of another development board would have been the natural course of action. However, with the University Program looming on the horizon, it was decided to hold off any new acquisition.

As such, major differences between both plug-ins weren't really discernible. Especially, since no debugging session wasn't made. So the support fell on the plug-in that seemed more likely to be up to date: The ARM DS-5 Development Studio Community Edition.

Chapter 7

Conclusions

The aim of this work was to find out if it was possible to build an Eclipse based Environment for multi-architecture developments, while still being independent from the underlying Operating System, and packing the necessary tools to reduce the productivity hurdles that current embedded systems developers face.

Extra effort was made in understanding how the debugger works, and why was Eclipse so popular as a development platform.

Although not everything went according to plan, it is possible to state right now that it is in fact feasible to use native Eclipse, or Eclipse-based frameworks like Code Composer Studio, as a single umbrella to the tool-chains for the architectures studied. However, there are still some improvements that can be made on the overall usability and stability of the final build of the IDE.

The most difficult part was the integration of a debugging component for each platform. In fact this was only possible in two of the four cases. Regarding their important aspect the conclusion is that we are still far away of achieving an acceptable level of integration. The ARM debugging environment is yet to be fully tested, due to unexpected delays, in the implementation of the ARM University program in the Department.

As a final remark, the emergent move towards open source tools by most of the vendors will pave the way to much better results in the future.

7.1 Contributions

This work contributed to the development of a proof of concept IDE to provide embedded systems developers less hassle configuring the development environment for projects with multiple families of embedded systems, or whose embedded system is constantly changing between projects, by:

- Surveying the currently free or open-source tool-chains.
- Categorizing them on how helpful they can be to the developer

- Integrating productivity tools in the IDE
- Identifying the areas where a strong push to future development should be made
- Engineering a method capable of providing compliance checking with the MISRA-C standard free of charge.

7.2 Future Work

This section lists possible paths to further explore this area.

7.2.1 AVaRICE

Currently, the AVaRICE project needs to be revamped to present a set of functionalities that could be considered acceptable for today standards.

Improvements on how it handles Debugwire are perhaps the most important, given the popularity of Atmel's line of 8-bit micro controllers with low pin count.

Better support for the new Atmel JTAGICE debuggers is another point has yet to be addressed. Some small advances were made in this area recently, but there is still a lot to do in the future.

Alternatively, transposing the AVaRICE project to the OpenOCD could be the path to take, since it would simplify the number of tools to manage at the IDE level.

7.2.2 Microchip PIC tool-chain integration

As discussed, the Microchip PIC tool-chain is yet to be successfully integrated on an Eclipse based IDE. The quickest path would be to integrate it as an external tool on the IDE.

However, the best option is to develop a proper plug-in that could handle all the intrinsic details of such integration as well as providing the right tools to integrate the debugger with the Eclipse's Debugging Perspective.

The recent acquisition of Atmel by Microchip might play an important role in the future regarding standardization of tools.

7.2.3 Detailed test of ARM tool-chains

Unfortunately, it wasn't possible to properly test both plug-ins for the ARM-based development board due to administrative delays in joining the ARM University program.

As of now, there is no point in doubting the capabilities presented by the debugger of the DS-5 Development Studio Community Edition. After all, it is the IDE provided by the vendor itself.

However, it would be of the developers best interest finding out the true extent of the debugging capabilities of the GNU ARM Eclipse.

Appendix A

Listings of test cases

This appendix lists the source code of the test programs developed for each family of micro controllers tested. The programs for the Microchip Pic line of 8-bit micro controllers are quite different from their 32-bit line, so both are listed separately.

Doxygen compatible annotations were included in the source.

A.1 Software Delay

A.1.1 For Atmel AVR micro controllers

Code A.1A simple loop for Atmel AVR micro controllers

```
1  /*!\file simple_loop.c **********************************
2   *  \brief Delays the execution by 1 s before inverting its output on pin PB5
3   *  \details A led will be connected to PB5 to give some visual feedback
4   *  \mainpage Simple Loop
5   *  \brief Check the \b Files tab to see the contents of this document. */
6
7  #include <avr/io.h>
8  #include <util/delay.h>
9
10 /*! \brief Runs indefinitely the delay application
11  * @return Nothing, since there is no designated point of exit. */
12 int main(void){
13         DDRB |= (1 << PB5); /* The pin where a led while be connected. */
14         PORTB &= ~(1 << PB5); /* Starts initially at low */
15
16     for(;;){
17             _delay_ms(1000); /*Delays the program execution by one second */
18                 PORTB ^=(1 << PB5); /* Applies the XOR operation */
19     }
20 }
```

A.1.2 For Microchip PIC line of 8-bit micro controllers

Code A.2A simple loop for Microchip PIC line of 8-bit micro controllers

```
/*!\file simple_loop.c
 * \brief blinks a led placed on pin 8 with a frequency of 1 Hz.
 * \mainpage Blink for PIC16
 * \brief Please refer to the \b Files tab to see the contents of this document. */

#include "configPIC8.h" /*Configures some hardware aspects of the micro controller */
#include <xc.h>
/*! \brief Defines the clock frequency the micro controller will run. */
#define _XTAL_FREQ 16000000uL

#if (_XTAL_FREQ == 16000000uL)
        #pragma config FOSC = INTOSC
#elif  ( _XTAL_FREQ == 11059200uL )
        #pragma config FOSC = XT
#elif  ( _XTAL_FREQ ==  3686400uL )
        #pragma config FOSC = XT
#endif

 /*!\brief Defines the pin where the led will be connected to.*/
#define RC2 (1 << 2)

/*!\brief Runs indefinitely the main application. */
void main(void) {

    #if (_XTAL_FREQ == 16000000uL)
        // IRCF<3:0> - 1111 16 MHz
        OSCCON |= (1 << 6 ) | (1 << 5) | (1 << 4) | (1 << 3);
    #endif

    //Everything is an output
    TRISA = 0x0;
    TRISC = 0x0;

    int i = 0;

    for(;;){
        for(i = 0; i < 100; i++) /* Runs the delay 100 times */
            __delay_ms(10); /*Delays the program execution by 10 milliseconds */
        LATC ^= RC2;
    }

    return;
}
```

A.1.3 For Microchip PIC line of 32-bit micro controllers

Code A.3A simple loop for Microchip PIC line of 32-bit micro controllers

```
/*!\file simple_loop.c*************************************************
 * \brief Blinks a set of leds with a frequency of 1 hz.
 * \mainpage Simple Loop
 * \brief Please refer to the \b Files tab to see the contents of this document.
 */

#include <p32xxxx.h>
#include <plib.h>
#include "BoardConfig.h"

/*! \brief DelayMs creates a delay of given milliseconds using the Core Timer.
 * @param delay The desired delay in milliseconds.*/
 void DelayMs(int delay)
 {
     unsigned int int_status;
     while( delay-- )
     {
         int_status = INTDisableInterrupts();
         OpenCoreTimer(F_CPU / 2000);
         INTRestoreInterrupts(int_status);
         mCTClearIntFlag();
         while( !mCTGetIntFlag() );
     }
     mCTClearIntFlag();
 }

 /*! Runs indefinitely the main application.
  * @return Nothing, since there is no designated point of exit. */
int main(void) {

    TRISGCLR = 0x0064; //LED 4 - connected to RG6
    TRISFCLR = 0x0001; //LED 5 - connected to RF0

    //Make sure they start with the same value
    LATGCLR = 0x0064;
    LATFCLR = 0x0001;

    for(;;) {
        LATGINV = 0x0064;
        LATFINV = 0x0001;
        DelayMs(1000);

    }
}
```

A.1.4 For Texas Instruments MSP430 micro controllers

Code A.4A simple loop for Texas Instruments MSP430 micro controllers

```
/*!\file simple_loop.c ******************************************
 * \brief Blinks a led connected to P1.0 with an arbitrary frequency.
 * \details On an olimexino-5510 P1.0 corresponds to pin D2.
 * \mainpage MSP430 Blinky
 * \brief Check the \b Files tab to see the contents of this document.*/

#include <msp430.h>

int main(void){
  volatile unsigned int i;

  WDTCTL = WDTPW+WDTHOLD;                    /* Stop WDT */
  P1DIR |= BIT0;                             /* P1.0 set as output */
  P1OUT &= ~BIT0;                           /* Set the LED off */
  for(;;)                                    /* continuous loop */
  {
    P1OUT ^= BIT0;                           /* XOR P1.0 */
    __delay_cycles(1000000); // SW Delay of 100000 cycles at 10Mhz
  }
  return (1);
} //THE END
```

A.2 Timer Interrupt

A.2.1 For Atmel AVR micro controllers

Code A.5A timer interrupt for Atmel AVR micro controllers

```
/*!\file interruption.c
 * \brief Creates an interruption every second in order to blink an led.
 * \details The led will be connected to pin PB5.
 * \mainpage Interruption
 * \brief Please refer to the \b Files tab to see the contents of this document. */

#include <avr/io.h>
#include <avr/interrupt.h>

int main(void){
        DDRB |= (1 << PB5);
        PORTB |= ~(1 << PB5);
        TCNT1=0x0BDC; /* Initial Value */
        TCCR1A = 0;
        TCCR1B = 0;

```

```
17          OCR1A = 0xF423; // compare match register 16MHz/256/1Hz-1
18          TCCR1A |= (1 << WGM12); /* CTC mode */
19          TCCR1B |= (1 << CS12); /* 256 pre-scaler */
20          TIMSK1 |= (1 << OCIE1A); /* enable timer compare interrupt */
21
22          sei(); /* Enables the interruption */
23      for(;;);
24  }
25
26  /*! Catches the generated interrupt and blinks the  led as a consequence.*/
27  ISR (TIMER1_COMPA_vect){
28          PORTB ^= (1<<PB5);
29  }
```

A.2.2 For Microchip PIC line of 8-bit micro controllers

Code A.6A timer interrupt for Microchip PIC line of 8-bit micro controllers

```
1   /*!\file timer.c
2    * \brief blinks a led placed on pin 8 with a frequency of 1 Hz.
3    * \mainpage Timer
4    * \brief Please refer to the \b Files tab to see the contents of this document. */
5
6   /*! \brief Defines the clock frequency the micro controller will run. */
7   #pragma config FOSC = INTOSC
8   #pragma config PLLEN = OFF       // PLL Enable (4x PLL enabled)
9
10
11  #include "configPIC8.h" /*Configures some hardware aspects of the micro controller */
12  #include <xc.h>
13
14  /*!\brief Counts the number of times there was a timer0 overflow.
15   * \details Every time it reaches 4, a second as passed. */
16  short ovf_counter;
17
18  /*!\brief Configures timer 0. */
19  void Timer0_init(void){
20
21      //Calculations:
22      //FOSC/4  = 240 kHz
23      //Overflows occurs at 255
24      //1/500khz = 4 us
25      //If a prescaler of 256: 4us*255*256 = 0.26 sec
26      TMR0 = 0;
27      INTCONbits.TOIE = 1; //Enables interrupt overflow
28      OPTION_REGbits.TMR0CS = 0; //Internal instruction cycle clock (FOSC/4).
29      OPTION_REGbits.PSA = 0; //Prescaler is assigned to the Timer0 module.
30      OPTION_REGbits.PS = 0b111; //1 : 256
```

```
}

/*!\brief Runs indefinitely the main application. */
void main(void){
    OSCCONbits.IRCF = 0b1011;

    ovf_counter = 0;

    Timer0_init();

    // set up IO pin
    TRISCbits.TRISC2 = 0; // output
    TRISCbits.TRISC3 = 0; //output

    ANSELCbits.ANSC2 = 0;
    ANSELCbits.ANSC3 = 0;

    LATCbits.LATC2 = 1; // high
    LATCbits.LATC3 = 0; // high

    INTCONbits.PEIE = 1; // enable peripheral interrupts
    INTCONbits.GIE = 1;  // global interrupt enable

    for(;;);
} //THE END

/*!\brief Increments the ovf_counter every time there is a overflow.
 * \details If one second has passed. Inverts the led. */
void interrupt ISR(void){

    if(INTCONbits.TMR0IF == 1){
        if(4 == ++ovf_counter){
            LATCbits.LATC2 = ~PORTCbits.RC2;
            LATCbits.LATC3 = ~PORTCbits.RC3;
            ovf_counter = 0;
        }
        INTCONbits.TMR0IF = 0;
    }
}//THE END
```

A.2.3 For Microchip PIC line of 32-bit micro controllers

Code A.7A timer interrupt for Microchip PIC line of 32-bit micro controllers

```
/*!\file interruption.c**********************************
 * \brief Blinks a set of leds with a frequency of 1 hz.
 * \mainpage Interruption
 * \brief Please refer to the \b Files tab to see the contents of this document. */
```

```

#include <p32xxxx.h>
#include <plib.h>
#include "BoardConfig.h"

/*! \brief The number of cycles it needs to count to reach 10 ms (?). */
#define END 31250 // 80MHz/256/10

/*!\brief Catches the compare interrupt every 10 milliseconds.*/
void __ISR(_TIMER_1_VECTOR, ipl2) Timer1Handler(void) {
    int static c = 0;
    if (c++ == 10) { //So it inverts every 1 second
        c = 0;
        LATFINV = 0x0001; // troca o LED
    }
    mT1ClearIntFlag();    // clear the interrupt flag
}

 /*! \brief Runs indefinitely the main application.
  * @return Nothing, since there is no designated point of exit. */
int main(void) {
    TRISFCLR = 0x0001;    // RF0 pin as an output
    ConfigIntTimer1(T1_INT_ON | T1_INT_PRIOR_2 | T1_INT_SUB_PRIOR_0);
    INTEnableSystemMultiVectoredInt();
    OpenTimer1(T1_ON | T1_PS_1_256, END);

    for(;;);
}
```

A.2.4 For Texas Instruments MSP430 micro controllers

Code A.8A timer interrupt for Texas Instruments MSP430 micro controllers

```
/*!\file TimerLaunchPad.c ************************************************
 * \brief Uses Timer A to blink a led with a frequency of 1 Hz.
 * \mainpage Timer A on MSP430
 * \brief Check the \b Files tab to see the contents of this document.
 *
 * \details   TACTL = TASSEL_2 + ID_3 + MC_1 + TACLR means:\n
 * Select SMCLK as the clock for Timer_A, select input divider of 8
 * (1 MHz SMCLK becomes 125 kHz), select up mode, and clear the count in TAR. */

#include <msp430.h>

/*!\brief Handles the interruption caused by a compare match. */
__interrupt void Timer_A (void);
/*!\brief Main body of the program.*/
void main(void);
```

```
16
17  void main(void)
18  {
19          /* To save energy */
20          P3OUT = 0;
21          P3DIR = 0xFF;
22          P2OUT = 0;
23          P2DIR = 0xFF;
24
25    WDTCTL = WDTPW + WDTHOLD;                 /* Stop WDT */
26    P1DIR |= 0x01;                            /* P1.0 output */
27    TACCTL0 = CCIE;                             /* CCR0 interrupt enabled */
28    TACCR0 = 62500 - 1;
29    TACTL = TASSEL_2 + ID_3 + MC_1 + TACLR;
30
31    __bis_SR_register(LPM0_bits + GIE);       /* Enter LPM0 w/ interrupt */
32
33    for(;;){}
34  }
35
36  #if __GNUC__
37          __attribute__((interrupt(TIMER0_A0_VECTOR))) void timer0_isr(void) {
38                  P1OUT ^= 0x01;
39          }
40  #else
41          /* Timer A0 interrupt service routine */
42          #pragma vector=TIMER0_A0_VECTOR
43          __interrupt void Timer_A (void){
44                  P1OUT ^= 0x01;                              /* Toggle P1.0 */
45          }
46  #endif
```

A.3 External Interrupt

A.3.1 For Atmel AVR micro controllers

Code A.9An external interrupt for Atmel AVR micro controllers

```
1  /*! \file avr_external_interruption.c ***********************************
2   *\brief Creates a interruption on the falling edge of a button press.
3   *\mainpage External Interrupt
4   \brief Please refer to the \b Files tab to see the contents of this document. */
5
6  #include <avr/io.h>
7  #include <avr/interrupt.h>
8
9  /*!\brief Runs indefinitely the main program.
10  *\return Nothing, since there is no designated point of exit. */
```

```
int main(void)
{
        DDRD &= ~(1 << DDD2);     // Clear the PD2 pin.
        // PD2 (PCINT0 pin) is now an input

        DDRB |= (1 << PB5);
        PORTB &=~(1 << PB5);

        PORTD |= (1 << PORTD2);    // turn On the Pull-up
        // PD2 is now an input with pull-up enabled

        EICRA |= (1 << ISC01) ;    // set INT0 to trigger on the falling edge
        EIMSK |= (1 << INT0);     // Turns on INT0

        sei();                    // turn on interrupts

        for(;;){}
}

/*! \brief Catches the generated interrupt and blinks the led
                if it wasn't caused by noise. */
ISR (INT0_vect)
{
        cli();        // turn off interrupts
        //Need to double check due to noise
        if((PIND & (1 << PB3)) == 0) {               //if the button is still pressed
                        PORTB ^=(1<<PB5);
                }
        sei();        // turn on interrupts
}
```

A.3.2 For Microchip PIC line of 8-bit micro controllers

Code A.10An external interrupt for Microchip PIC line of 8-bit micro controllers

```
/*!\file external_interruption.c
 * \brief Creates a interruption on the negative edge of the button.
 * \mainpage External Interruption
 * \brief Please refer to the \b Files tab to see the contents of this document.*/

/*! \brief Defines the clock frequency the micro controller will run. */
#define _XTAL_FREQ 16000000uL
#pragma config FOSC = INTOSC
#pragma config PLLEN = ON       // PLL Enable (4x PLL enabled

#include "configPIC8.h" /*Configures some hardware aspects of the micro controller */
#include <xc.h>

```

```
/*! \brief Interrupt triggered by the press of the button. */
void interrupt ISR(void) {

    if(INTCONbits.IOCIF == 1) {
        IOCAFbits.IOCAF2 = 0;
        LATC3 = ~RC3;
        __delay_ms(10);
    }
}

 /*!\brief Runs indefinitely the main application. */
void main(int argc, char** argv) {

    // IRCF<3:0> - 1111 16 Mhz
    OSCCONbits.IRCF  = 0b1111;
    OSCSTAT = 0x00;
    OSCTUNE = 0x00;


    ANSELCbits.ANSC3= 0; // RC3 is digital
    TRISCbits.TRISC3= 0; // RC3 is an output
    APFCON0 |= (1<<5); // Don't use special features on Pin RC3
    APFCON1 |= (1<<2); // Don't use special features on Pin RC3

    /*=========INTERRUPT CONFIGURATION=========*/
    ANSELAbits.ANSA2 = 0;
    TRISAbits.TRISA2 = 1;
    IOCANbits.IOCAN2  = 1;  // Generate Interrupt on Negedge
    IOCAP  = 0x00;  // Disable Interrupt on Posedge
    INTCONbits.IOCIE = 1; //interrupt on change
    INTCONbits.PEIE = 1; // enable peripheral interrupts
    INTCONbits.GIE = 1;  // global interrupt enable

    LATC3 = 1;

    for(;;);
}
```

A.3.3 For Microchip PIC line of 32-bit micro controllers

Code A.11An external interrupt for Microchip PIC line of 32-bit micro controllers

```
/*!\file external_interrupt.c ********************************************
 *\brief Creates an interruption on the falling edge of the button
 * \mainpage "External Interruption"
 * \brief Please refer to the \b files to see the contents of this document. */

#include <p32xxxx.h>
```

```
#include <plib.h>
#include "BoardConfig.h"

/*!\brief Defines the pin where the button is connected to*/
#define RB8 (1 << 8)

/*!\brief Runs indefinitely the main application
 * \return Nothing, since there is no designated point of exit. */
int main(void) {

//Configure Output
TRISFCLR |= 0x0001;     // Pin RF0 as an output
LATFSET = 0x0001;        //Start turned on

 INTEnableSystemMultiVectoredInt();

//Configure external interrupt
ConfigINT3(EXT_INT_PRI_7 | FALLING_EDGE_INT | EXT_INT_ENABLE);

for(;;){}

return 1;
}

/*!\brief Captures the interruption when the button is pressed.*/
void __ISR(_EXTERNAL_3_VECTOR, ipl7) INT3Interrupt() {
   if(!(PORTB & RB8))
        LATFINV = 0x0001; // troca o LED

   mINT3ClearIntFlag();
}
```

A.3.4 For Texas Instruments MSP430 micro controllers

Code A.12An external interrupt for Texas Instruments MSP430 micro controllers

```
/*!\file externaInterruption.c ***********************************
 * \brief Every time the button is pressed, causes an interruption that flips the LED.
 *
 * \mainpage External Interrupt on a MSP430
 * \brief Check the \b Files tab to see the contents of this document.\n
 *
 * \details The (yellow) LED is connected to P1.0
 * \details The button is connected to P1.3.
 * \details The button is on a pull-up configuration. */

#include <msp430.h>
#include <signal.h>
```

```
13
14  /*! \brief Handles the interruption caused when the button is pressed. */
15  __interrupt void P1_ISR(void);
16
17  /*! \brief Main body of the program. */
18  int main(void);
19
20  int main(void) {
21      WDTCTL = WDTPW | WDTHOLD;         /* Stop watchdog timer. */
22
23      P1OUT = 0; /* Start by putting everything at zero. Predictable behavior. */
24      P1DIR |= BIT0; /* P1.0 is now the output. */
25      P1IES |= BIT3;    /* high -> low is selected with IESx = 1. */
26      P1IFG &= ~BIT3;  /* To prevent an immediate interrupt, clear the flag for
27                                      P1.3 before enabling the interrupt. */
28      P1IE |= BIT3;     /* Enable interrupts for P1.3 */
29
30      _enable_interrupts();
31      for(;;){} /* Loops forever */
32      return 1;
33  }
34
35  #if __GNUC__
36  __attribute__((interrupt(PORT1_VECTOR))) void port1_isr(void){
37          if((P1IFG & BIT3) == BIT3){
38                          P1IFG &= ~BIT3;
39                          P1OUT ^= BIT0;
40                  } else{
41                          P1IFG &= ~BIT3;
42                  }
43  }
44
45  #else
46  #pragma vector=PORT1_VECTOR
47  __interrupt void Port_1(void) {
48          if((P1IFG & BIT3) == BIT3){
49                          P1IFG &= ~BIT3;
50                          P1OUT ^= BIT0;
51                  } else{
52                          P1IFG &= ~BIT3;
53                  }
54  }
55  #endif
```

A.4 Internal Peripheral

A.4.1 For Atmel AVR micro controllers

Code A.13Peripheral (ADC) for Atmel AVR micro controllers

```
/*! \file main.c ************
 *\brief Reads an analog input and lights the corresponding LED.
 *\mainpage ADC
 *\brief Check the \b Files tab to see the contents of this document. */

#include <avr/io.h>

/*!\brief Selects the reference voltage and enables the ADC. */
void ADC_init() {
        /*Select reference voltage*/
        ADMUX = (1<<REFS0);
        /*AVCC with external capacitor on AREF selected*/

        /*ADC enable | Pre scaler selected : 128, thus giving the ADC a
    frequency of 16MHz/128 = 125 KHz */
        ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}

/*!\brief Reads the value converted from a channel.
 * \param ch The channel number to be read.
 * \return The value read. */
uint16_t adc_read(uint8_t ch) {
        // select the corresponding channel 0~7
        // "ANDing" with 7 will always keep the value
        // of ch between 0 and 7
        ch &= 0b00000111;  // AND operation with 7
        ADMUX = (ADMUX & 0xF8)|ch; // clears the bottom 3 bits before ORing

        // start single conversion
        // write 1 to ADSC
        ADCSRA |= (1<<ADSC);

        // wait for conversion to complete
        // ADSC becomes 0 again
        // till then, run loop continuously
        while(ADCSRA & (1<<ADSC));

        return (ADC); /*The result of an ADC converter is stored here*/
}

/*!\brief Runs the main application definitely.
 *\return Nothing since there is no designated point of exit. */
int main(void) {

        DDRB |= (1 << PB4) | (1 << PB5);
        PORTB &=(1 << PB4) | (1 << PB5);
```

```
47
48          ADC_init();
49
50          uint16_t adc_value;
51
52      for(;;) {
53                  adc_value = adc_read(2); //Reads channel 2 -> PA2
54                  /* Since this as 10 bit ADC, 2.5 volts are
55                          approximately 511 when converted. */
56                  if(adc_value < 511) {
57                          PORTB |= (1 << PB4);
58                          PORTB &= ~(1 << PB5);
59                  }
60                  else  if(adc_value >=511 && adc_value < 1023) {
61                          PORTB |= (1 << PB5);
62                          PORTB &= ~(1 << PB4);
63                  }
64                  else {          //Only if there is an error
65                           PORTB |= (1 << PB4);
66                           PORTB |= (1 << PB5);
67                  }
68          }
69
70          return 0;
71  }
```

A.4.2 For Microchip PIC line of 8-bit micro controllers

Code A.14Peripheral (ADC) for Microchip PIC line of 8-bit micro controllers

```
1   /*!\file ADC.c
2    * \brief Reads an analog value.
3    * \mainpage ADC
4    * \brief Please refer to the \b Files tab to see the contents of this document. */
5
6   #include "configPIC8.h" /*Configures some hardware aspects of the micro controller */
7   #include <xc.h>
8
9   /*! \brief Defines the clock frequency the micro Controller will run. */
10  #define _XTAL_FREQ 11059200uL
11
12  #pragma config FOSC = HS
13  #pragma config PLLEN = OFF   // PLL Enable (4x PLL enabled)
14
15  /*!\brief Runs indefinitely the main application. */
16  void main(void) {
17
18      unsigned int conv = 0;
```

```
19
20      //RC0 and RC1 will be inputs; RC2 and RC3 an output.
21      //0b00000011 -> 0x03
22      TRISCbits.TRISC0 = 1;
23      TRISCbits.TRISC1 = 1;
24      TRISCbits.TRISC2 = 0;
25      TRISCbits.TRISC3 = 0;
26
27      //RC0 will be an analog input 0b00000001 -> 0x01
28      ANSELCbits.ANSELC = 0b0001;
29
30      //Chose FOSC/16 - ADCS<2:0> = 101
31      ADCON1bits.ADCS0 = 1;
32      ADCON1bits.ADCS1 = 0;
33      ADCON1bits.ADCS2 = 1;
34
35      //ADCON1 register provides control of the voltage references
36      ADCON1bits.ADPREF = 0;
37      ADCON1bits.ADNREF = 0;
38
39      //the output format.
40      ADCON1bits.ADFM = 1;
41
42      ADCON0bits.CHS = 0b00100;
43
44      //will start the Analog-to-Digital conversion.
45      ADCON0bits.ADON = 1;
46      ADCON0bits.GO_nDONE = 1;
47      __delay_us(8);
48          for(;;){
49              if(0 == ADCON0bits.GO_nDONE){ //Sampling done
50
51                  conv = (ADRESH << 8) + ADRESL;
52                  //conv = conv << 8 + ADRESL;
53
54                  //comparison between values
55                  if(0 <= conv && conv < 511) {
56                      LATCbits.LATC3 = 1;
57                      LATCbits.LATC2 = 0;
58                  } else if(511 <= conv && conv < 1024) {
59                      LATCbits.LATC3 = 0;
60                      LATCbits.LATC2 = 1;
61                  } else {
62                      LATCbits.LATC3 = 1;
63                      LATCbits.LATC2 = 1;
64                  }
65
66                  //Ready for another conversion.
```

```
67                  ADCON0bits.GO_nDONE = 1;
68              }
69          }//FOR CYCLE HAS ENDED
70
71  }//THE END
```

A.4.3 For Microchip PIC line of 32-bit micro controllers

Code A.15Peripheral (ADC) for Microchip PIC line of 32-bit micro controllers

```
1   /*!\file ADC.c ********************************************************
2    * \brief Reads the analog input lights the corresponding LED.
3    * \mainpage ADC
4    *  \brief Check the \b Files tab to see the contents of this document. */
5
6   #include <p32xxxx.h>
7   #include <plib.h>
8   #include <inttypes.h>
9   #include "BoardConfig.h"
10
11  /*! \brief Stores the value read by the ADC. */
12  uint16_t data;
13
14  #define SAMP 1 /*!< \brief Status bit to order the ADC to start sampling. */
15  #define DONE 0 /*!< \brief Status bit to check if the sampling was done.  */
16
17  /*! \brief DelayMs creates a delay of given milliseconds using the Core Timer.
18   * \param delay The desired delay in milliseconds.
19   */
20   void DelayMs(int delay) {
21       unsigned int int_status;
22       while( delay-- ) {
23           int_status = INTDisableInterrupts();
24           OpenCoreTimer(F_CPU / 2000);
25           INTRestoreInterrupts(int_status);
26           mCTClearIntFlag();
27           while( !mCTGetIntFlag() );
28       }
29       mCTClearIntFlag();
30   }
31
32   /*! \brief Configures the ADC. */
33  void init_ADC(void) {
34      AD1PCFG = 0x7FFF;     // all PORTB = Digital but RB15 (AN15) = analog
35      AD1CON1 = 0x00E0;   // SSRC bits = 111 implies internal counter ends
36                                          //sampling and starts converting
37      AD1CON2 = 0;
38      AD1CON3 = 0x1F02;     // Sample time = 31 TPB,
```

```
39                      //Conversion time = 12*TAD = 12*[2*(ADCS+1)] TPB = 12*6 TPB
40      AD1CSSL = 0;
41      AD1CHS = 0x000F0000; // Connect RB15/AN15 as CH0 input
42      AD1CON1SET = 0x8000; // turn ADC ON
43  }
44
45  /*! \brief Acquires a new reading. */
46  void acquire_ADC(void) {
47      int v;
48
49      AD1CON1SET = (1<<SAMP);          // start Sampling (SAMP = 1) and then converting
50      while (!(AD1CON1 & (1<<DONE))); // conversion done?
51      v = ADC1BUF0;                    // yes then get ADC value
52      data = v;
53  }
54
55  /*! \brief Runs indefinitely the counter application
56   * @return Nothing, since there is no designed point of exit.
57   */
58  int main(void){
59
60      TRISGCLR = 0x0064; //LED 4 - connected to RG6
61      TRISFCLR = 0x0001; //LED 5 - connected to RF0
62
63      //Make sure they start with the same value
64      LATGCLR = 0x0064;
65      LATFCLR = 0x0001;
66
67      init_ADC();
68
69      for(;;){
70          acquire_ADC();
71          if(data <= 511) {
72              LATGINV = 0x0064;
73              LATFCLR = 0x0001;
74          } else if(data > 511) {
75              LATFINV = 0x0001;
76              LATGCLR = 0x0064;
77          } else {
78              LATGSET = 0x0064;
79              LATFSET = 0x0001;
80          }
81          DelayMs(1000);
82      }
83
84  }//END OF MAIN
```

A.4.4 For Texas Instruments MSP430 micro controllers

Code A.16Peripheral (ADC) for Texas Instruments MSP430 micro controllers

```
/*!\file ADC10.c ************************
 * \brief Demo - ADC10, Sample A1, AVcc Ref, Set P1.0 if > 0.5*AVcc
 * \mainpage MSP430 LaunchPad ADC
 * \brief Check the \b Files tab to see the contents of this document.*/

#include <msp430.h>

/*!\brief Main body of the program.*/
void main(void);
/*!\brief ADC10 interrupt service routine */
__interrupt void ADC10_ISR(void);

void main(void){
  WDTCTL = WDTPW + WDTHOLD;                 /* Stop WDT */
  ADC10CTL0 = ADC10SHT_2 + ADC10ON + ADC10IE; /* ADC10ON, interrupt enabled */
  ADC10CTL1 = INCH_1;                       /* input A1 */
  ADC10AE0 |= 0x02;                         /* PA.1 ADC option select */
  P1DIR |= 0x01;                            /* Set P1.0 to output direction */

  for (;;) {
    ADC10CTL0 |= ENC + ADC10SC;             /* Sampling and conversion start */
    __bis_SR_register(CPUOFF + GIE);        /* LPM0, ADC10_ISR will force exit */
    if (ADC10MEM < 0x1FF){
      P1OUT &= ~0x01;                       /* Clear P1.0 LED off */
    } else {
      P1OUT |= 0x01;                        /* Set P1.0 LED on */
    }
  } /*END OF FOR */
}

#if __GNUC__
        __attribute__((interrupt(ADC10_VECTOR))) void adc_isr(void) {
                  __bic_SR_register_on_exit(CPUOFF);  /* Clear CPUOFF bit from 0(SR) */
                }
#else
        #pragma vector=ADC10_VECTOR
        __interrupt void ADC10_ISR(void){
                __bic_SR_register_on_exit(CPUOFF);   /* Clear CPUOFF bit from 0(SR) */
        }
#endif
```

References

[1] UBM Tech. 2014 Embedded Market Study. Then, Now: What's Next? *UBM Electronics*, 2014.

[2] Rich Quinnell. 2015 Embedded Market Study: Changes in Today's Design, Development and Processing Environments. *EDN/EE Times UBM Electronics*, 2015.

[3] Martin Croxford and Roderick Chapman. Correctness by Construction: A Manifesto for High-Integrity Software. *The Journal of Defense Soft. Engr*, pages 5–8, 2005.

[4] Bernhard Spuida and Senior Word Wrangler. The fine Art of Commenting. *Tech Notes, general Series. SW Wrangler*, 2002.

[5] MIRA Ltd. Misra c: A brief history of MISRA C. `http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx`, 2013.

[6] Atmel Corporation. Atmel AVR10006: XDK - User Guide. Technical Report 42050B-AVR/ARM-11/2012, 2012.

[7] Atmel Corporation. GNU Toolchain for Atmel AVR8 Embedded Processors. Technical Report 42372A-MCU-02/2016, 2016.

[8] Atmel Corporation. *Atmel AVR4030: Atmel Software Framework - Reference Manual*, 2012.

[9] Atmel Corporation. Atmel Studio 7.0 Release Note, 2016.

[10] Thomas Holland and Kees Bakker. The AVR GCC Toolchain. `http://avr-eclipse.sourceforge.net/wiki/index.php/The_AVR_GCC_Toolchain`, 2011. (Last visited on 13/06/2016).

[11] Holland, Thomas and Bakker, Kees. Debugging - AVR Eclipse. `http://avr-eclipse.sourceforge.net/wiki/index.php/Debugging`, January 2012. (Last visited on 13/06/2016).

[12] Microchip Techology Inc. MPLAB X Integrated Development Environment (IDE). `http://www.microchip.com/mplab/mplab-x-ide`, 2016. (Last visited on 13/06/2016).

[13] Microchip Techology Inc. *MPLAB X IDE User's Guide*, 2015. (Last visited on 13/06/2016).

[14] Microchip Techology Inc. MPLAB XC Compilers. `http://www.microchip.com/mplab/compilers`, 2016. (Last visited on 13/06/2016).

[15] gpsim homepage. http://gpsim.sourceforge.net/, September 2015. (Last visited on 14/06/2016).

[16] Microchip. Overview - Arriba. http://www.microchip.com/pagehandler/en-us/products/arriba/, 2014. (Last visited on 13/06/2016).

[17] Sourceforge. PIC C Builder for Eclipse. https://sourceforge.net/projects/piccbuilder/, 2010. (Last visited on 13/06/2016).

[18] Texas Instruments Incorporated. Licensing - CCSv6. http://processors.wiki.ti.com/index.php/Licensing_-_CCSv6, December 2015. (Last visited on 14/06/2016).

[19] Texas Instruments Incorporated. *Code Composer Studio v6.1 for MSP430 User's Guide*, June 2016. (Last visited on 14/06/2016).

[20] Texas Instruments Incorporated. *MSP430 Optimizing C/C++ Compiler v15.12.0.LTS User's Guide*, January 2016. (Last visited on 14/06/2016).

[21] Texas Instruments Incorporated. *MSP430 GCC User's Guide*, March 2016. (Last visited on 14/06/2016).

[22] MSP430 Eclipse. http://xpg.dk/projects/msp430/msp430-eclipse/, October 2013. (Last visited on 14/06/2016).

[23] ARM Limited. *ARM® DS-5 Version 5.24: Getting Started Guide*, 2016. (Last visited on 14/06/2016).

[24] ARM Limited. ARMv8-A Architecture. http://www.arm.com/products/processors/armv8-architecture.php, 2016. (Last visited on 14/06/2016).

[25] Linaro. Toolchain Working Group Flyer. https://wiki.linaro.org/WorkingGroups/ToolChain/Flyer, 2012. (Last visited on 15/06/2016).

[26] GCC ARM Embedded Maintainers. GCC ARM Embedded. https://launchpad.net/gcc-arm-embedded, 2016. (Last visited on 15/06/2016).

[27] Liviu Ionescu. GNU ARM Eclipse: GNU ARM Eclipse Plug-ins Features. http://gnuarmeclipse.github.io/plugins/features/, November 2014. (Last visited on 15/06/2016).

[28] Liviu Ionescu. GNU ARM Eclipse - Welcome to GNU ARM Eclipse! http://gnuarmeclipse.github.io/, February 2016. (Last visited on 15/06/2016).

[29] ARM Limited. CMSIS - Cortex Microcontroller Software Interface Standard. http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php, 2016. (Last visited on 15/06/2016).

[30] ARM Germany GmbH. *Getting Started: Create Appliations with MDK Version for ARM® Cortex®-M Microcontrollers*, 2015. (Last visited on 15/06/2016).

[31] Ivan Kravets. PlatformIO Documentation. `https://media.readthedocs.org/pdf/platformio/latest/platformio.pdf`, June 2016.

[32] Github Inc. Atom Flight Manual. `http://flight-manual.atom.io/getting-started/sections/why-atom/`.

[33] Ivan Kravets. PlatformIO IDE for Atom. `http://docs.platformio.org/en/latest/ide/atom.html#installation`, 2016.

[34] International Business Machines Corp. Eclipse Platform Technical Overview. `http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html`, 2006. (Last visited: 17/06/2016.

[35] Eclipse Foundation. Eclipse Documentation - Current Release (sr1). `http://help.eclipse.org/mars/index.jsp`, 2015. (Last visited on 16/06/2016).

[36] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins Third Edition.* Pearson Education, 2008.

[37] Eclipse Foundation. Plug-in Development Environment Overview. `http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fguide%2Fintro%2Fpde_overview.htm`, 2015. (Last visited on18/06/2016).

[38] Apache Ant ANT. Welcome: Apache Ant™. `http://ant.apache.org/`, April 2016. (Last visited on 18/06/2016).

[39] Dimitri van Heesch. *Doxygen Manual for version 1.8.11*, 2015.

[40] Eclox. `http://home.gna.org/eclox/#download`, December 2009.

[41] Scott Chacon and Ben Straub. *Pro git.* Apress, 2014.

[42] C. Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick. *Version control with subversion.* O'Reilly Media, Inc., 2008.

[43] Martin Oberhuber. TM Terminal. `https://marketplace.eclipse.org/content/tm-terminal`, February 2014.

[44] Jonathan B Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture.* John Wiley & Sons, Inc., 1996.

[45] Clara Ines Jaramillo. *Source Level Debugging Techniques and Tools for Optimized Code.* PhD thesis, University of Pittsburgh, 2000.

[46] Norman S Matloff and Peter Jay Salzman. *The Art of Debugging with GDB, DDD, and Eclipse.* No Starch Press, 2008.

[47] Amy Brown. *The Architecture of Open Source Applications: Structure, Scale, and a Few More Fearless Hacks*, volume 2. Kristian Hermansen, 2012.

[48] Arnold Robbins. *GDB Pocket Reference.* " O'Reilly Media, Inc.", 2005.

[49] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*, April 2008.

[50] Interfacing with the CDT debugger, Part 2: Accessing gdb with the Eclipse CDT and MI, author=Scarpino, Matthew, year=2008, month=June, howpublished=`http://www.ibm.com/developerworks/library/os-eclipse-cdt-debug2/`, language=English, notes=(Last visited on 25/06/2016).

[51] Jack Ganssle. Introduction to In-Circuit Emulators. `http://www.embedded.com/electronics-blogs/beginner-s-corner/4024647/Introduction-to-In-Circuit-Emulators`, November 2001.

[52] XJTAG. What is JTAG? and how can i use it? `https://www.xjtag.com/wp-content/uploads/xjtag-ebook_what-is-jtag-en.pdf`.

[53] Arnold Berger and Michael Barr. Introduction to On-Chip Debugging. `http://www.embedded.com/electronics-blogs/beginner-s-corner/4024528/Introduction-to-On-Chip-Debug`, February 2003.

[54] The OpenOCD Project. *Open On-Chip Debugger: OpenOCD User's Guide*, May 2015.

[55] ARM Ltd. Serial Wire Debug, 2016.

[56] Joerg Wunsch. Avarice, December 2011.