Technical University of Crete

Department of Electronic and Computer Engineering

"On Runtime Environments for Reconfigurable Logic:

A Dynamically Reconfigurable Architecture for Adaptive Environmental Monitoring"

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER IN ELECTRONIC ENGINEERING

by

Efstathiou Dionissios

Supervising Professor:	**Professor Apostolos Dollas**
Thesis Committee:	**Professor Kostantinos Kalaitzakis**
	Professor Dionysios Pnevmatikatos

March 2011

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

Preamble

The technology of environmental monitoring systems is becoming a mature research field. The need to monitor and understand complex environmental phenomena and biochemical processes that take place in various temporal and spatial domains cannot be addressed simply by deploying (spreading) tens of data logging nodes in the area of interest. The complexity of these environmental systems originates from their nonlinear dynamics, scale-dependent behavior and heterogeneity of the interacting processes.

Given the physical reality of many environmental applications, especially the size and expense of sensors and their cost, we are now seeing a movement toward networks that are comparatively lower in population and density but much smarter. The transformation of an Environmental Monitoring System (EMS) from an elaborate logging system to an intelligent adaptive platform that adjusts its sampling and processing features in the context of evolving data acquisition would revolutionize our understanding of the monitored environment.

Within this context, this work presents an adaptive reconfigurable platform for environmental monitoring. It is based upon the close coupling of an 8-bit microcontroller and an FPGA acting as a co-processor. The key concept of the proposed system is that the extra processing power provided by the reconfigurable hardware is accessed on demand. Reserving processing resources for the extreme conditions such as flood events can be extremely helpful in the process of monitoring and understanding complex environmental phenomena while at the same time preserving the system's energy efficiency.

Chapter 1
Introduction

1.1 Introduction

The technology for sensing and control has the potential for significant advances, not only in science and engineering, but equally important, on a broad range of applications relating to critical infrastructure protection and security, health care, the environment, energy, food safety, production processing, quality of life, and the economy.

Environmental monitoring with Wireless Sensor Networks is one of the most challenging research areas in the last decade. It represents a class of sensor network applications with enormous potential benefits for scientific communities and society as a whole.

In recent years, technological advances in the miniaturization of electronics, wireless communications and embedded microprocessors have decreased the size, weight, and cost of sensors and sensor arrays by orders of magnitude and at the same time increased their spatial and temporal resolution and accuracy. They also tend to transform environmental monitoring systems (EMS) from simple logging devices that record and transmit raw environmental data at specific time intervals, to "smart" low-cost, multifunctional, event-driven, monitoring systems.

The design of environmental monitoring applications based on WSNs requires the integration of many disciplines including embedded systems, telecommunications, software engineering, data bases and management as well as data modeling. At the same time designers have to address in situ and operational problems (node deployment, sensors calibration etc). The latter are associated with the interaction of the monitoring system with the environment and, somehow, with the extension of the network lifetime (e.g., energy harvesting and management, faults and failures, thermal drifts, ageing effects).

1.2 Components of a Environmental Monitoring System

A typical EMS is made up of five basic components. Processor, storage unit, radio, sensors, and power supply:

- **Processing Unit:** Typically a microprocessor coupled with a small amount of memory for signal processing, storage and transmission. Next generation 16/32-bit embedded processors and reconfigurable logic are being introduced that provide order-of-magnitude increases in computational throughput over 8-bit microcontrollers.
- **Power Supply:** In an environmental monitoring system where the primary requirement is long-term field monitoring, the power source of the system can only be sustained through harvesting energy available in the environment mainly through solar cells and rechargeable batteries for storage.
- **Storage Unit:** The amount of storage needed in a node, depends on the overall network structure. If the architecture of the entire system dictates that all data should be transmitted instantaneously on the central station, then the amount of local storage needed can very little and is mainly used as a temporary buffer. If on the other hand, the main concern is the limitation of data being transmitted through the communication medium, then there is a greater need for local storage. The latter scenario requires the existence of greater processing capabilities locally.
- **Sensors:** The primary purpose of EMS is neither computing nor communicating, but rather sensing. The sensing component of SN nodes is the current technology bottleneck. The sensing technologies are not progressing as fast as semi-conductors. Also, sensors are being applied to the real physical world, while the computing and communicating units are dealing with a somewhat controlled environment. One of the main challenges of environmental monitoring is selecting the appropriate type and quantity of sensors for an application. There are numerous types of

sensors with different properties such as resolution, cost, accuracy, size, and power consumption. A single chemical sensor can add a few thousands of dollars to the cost of the entire system. Also some of the most crucial tasks such as fault tolerance, error control, calibration, and time synchronization are associated with the sensors of the system. The analysis of all aspects of sensing technologies is beyond the range of this thesis.

- **Communication Unit:** One of the key features of EMS is the communication layer. In a remote monitoring location, we are referring to wireless communication for data relaying. Communication demand depends on the level of local data processing and control [6] but the communication layer must be able to cope with high and even burst data (worst-case) transfers, especially when real-time tasks are involved. Let's assume, for example, a system is capable of producing -ready to transmit-data at a rate of at least 100 Kbps throughout the network. There is also a communication control overhead, which increases the data rate to 110 Kbps. If the communication layer cannot serve the above requirements then there will be a bottleneck, which will affect the entire system.

1.3 Issues and Constrains

The implementation of an efficient and reliable EMS for ecological research imposes a number of design issues and constraints that need to be addressed. Challenges and limitations of wireless sensor nodes include, but are not limited to, the following:

- **Energy efficiency:** Since we are referring to autonomous nodes in distributed systems, each node must have the ability to maintain balanced (if not minimal) energy consumption. Obviously, the power consumption of a system is closely related to its computational task. Therefore, a computationally demanding task "consumes" more energy than a computationally effortless one. There is a lot of literature in power considerations [1,2] and it is also beyond the range of this thesis.

- **Small physical size and weight:** Reducing physical size has always been one of the key design issues. It is imperative for the system to have the smallest effect in the sensing environment. The overall weight and size of a node is mainly affected by the power unit (battery and solar cells). The choice of having a small-sized battery would be beneficial in terms of size but would compromise dramatically the energy efficiency of the entire system.

- **Robustness:** Sensor nodes for ecological research often have to be deployed in harsh environments, where they need to survive the elements of nature (humidity, moisture etc). They will be unattended, and are expected to be power efficient and operational for a long period of time. The system must also be resilient to errors and malfunctions. Since the existence of redundant subsystems is prohibited due to space, cost and power limitations, special attention must be given to the reliability of the individual units even through local or even remote repair, calibration and test.

- **Scalability – Upgradability:** Due to continuous technological advances in sensors (chemical sensors, cameras etc) and in other fields (microcontrollers, network) the system must be easily upgradeable with the least effort (both in hardware and software redesign) and with minimal cost. This can only be achieved by maintaining a modular architecture.

- **Task Concurrency:** A sensor node for environmental monitoring is actually a multifunctional platform for data capturing, processing, storing and transmission. The overall performance of the system can be achieved through task parallelism. For example, information may be simultaneously captured from sensors, processed, and transmitted over the network in a pipelined manner, instead of sequential action. There are two conceptual approaches to address this requirement: (i) partitioning the processing unit into multiple units where each is responsible for a specific task; and (ii) reduction of the context switching time between tasks.

- **Functional diversity:** Apart from the obvious benefits of modular architecture, the system must also be able to offer functional modularity. We refer to dynamic functional in situ reconfiguration without the need for system recovery, reprogramming and redeployment. This dynamic diversity in design and usage requires an unusual degree of software and hardware modularity which can be accomplished through: (i) remote system reprogramming (if the processing unit is microcontroller based). (ii) dynamic, on the fly, system redesign (if the processing unit is based on reconfigurable logic) [3]

1.4 Thesis Objectives

This thesis presents an adaptive reconfigurable platform for environmental monitoring which is based upon the close coupling of an 8bit microcontroller and an FPGA acting as a co-processor. The proposed platform transforms a conventional data logging device that records and transmits raw environmental data at specific time intervals into an intelligent, event-driven platform that adjusts its behavior at run-time in the context of the acquired data. Additionally it acts as a pre-processor of the collected data, thus minimizing the volume of data transmitted.

The natural architectural choice for such a system is the combination of a general purpose processor and a reconfigurable processing element. In such a coupled node where a host processor (microcontroller) and reconfigurable logic are present, the microcontroller is mostly suitable for implementing sequential tasks (control flow oriented tasks), while on the other hand the reconfigurable logic is preferred for computationally intense tasks with high degree of parallelism. This close coupling of hardware and software in a communication – demanding environment is a major factor in the development of a reconfigurable environmental sensor node.

1.5 Thesis Contribution

The key innovations of the proposed thesis are the following:

- Development of an environmental monitoring platform where a low-power microcontroller is coupled with an FPGA for computationally demanding tasks.
- Design and development of a platform that is technology and vendor independent of the incorporated reconfigurable logic (FPGA)
- Developing mechanisms for implementing runtime dynamic system reconfiguration both in terms of hardware and of software.
- Implementation of a microkernel where the end-user takes advantage of the platforms resources seamlessly through user defined tasks.

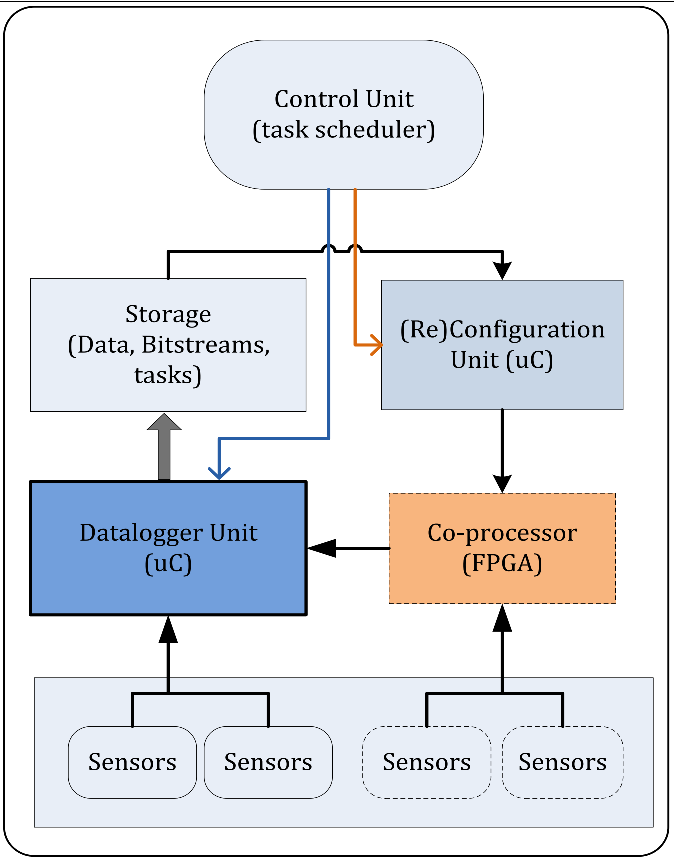


Figure 1: Proposed architecture

In terms of overall system operation, the major innovations are the following:

- It allows dynamic remote reconfiguration without the need to withdraw it from the sampling field, both in software and in hardware
- Immediate response to extreme events. The system can self-adapt to the acquired readings from the environment and change its operation (for example sampling frequency) at runtime
- From an architectural point of view, the modular design of this platform offers a level of transparency between the different layers allowing the replacement of various units like the reconfigurable hardware without the need for complete system redesign.

1.6 Motivation of the Work

Hydrologic and geochemical processes that take place in Mediterranean watersheds have variable temporal and spatial scales. The hydrographs of both temporary and permanent rivers are flashy with response times ranging from minutes to hours. “Temporary River” is a general term for all intermittent, ephemeral and episodic streams. Temporary river watersheds constitute 30% of the Mediterranean region and at least 42% of the Greek territory. Based on recent trends, the number of temporary rivers will likely increase in the future due to human impacts such as climate change and increased water abstraction. Temporary river hydrographs are flashy and exhibit characteristic response times ranging from minutes to hours such as experienced during first flush and storm events. After dry periods, the first flash floods carry significant quantities of suspended solids and pollutants (Fig 1.2). Compared to perennial flow conditions, temporary rivers deliver most of the annual pollution load during only a few flood events typically lasting a few hours. [4]

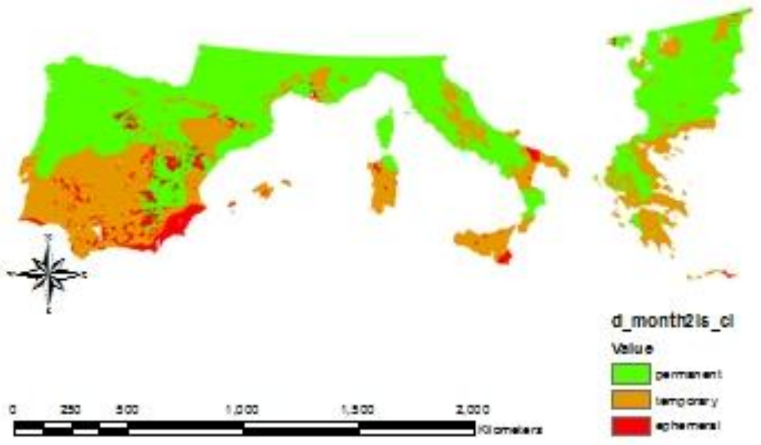
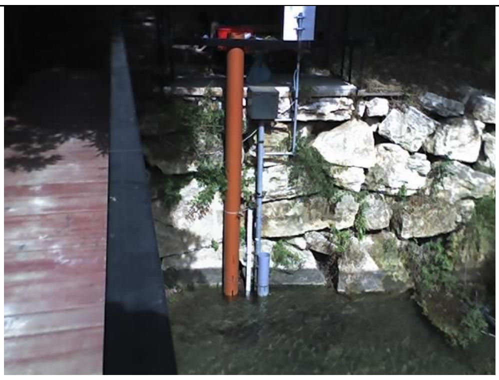
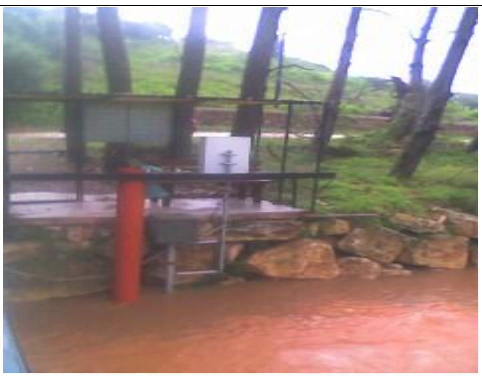


Figure 2: Mediterranean Ephimerality

Given the fact that most EMS are battery powered and with limited resources, the key concept of the proposed system is that the extra processing power provided by the reconfigurable hardware is accessed on demand and only during extreme environmental conditions like those previously described. Reserving processing resources for the worst-case scenario can be extremely helpful in the process of monitoring and understanding complex environmental phenomena while at the same time preserving the system's energy efficiency.





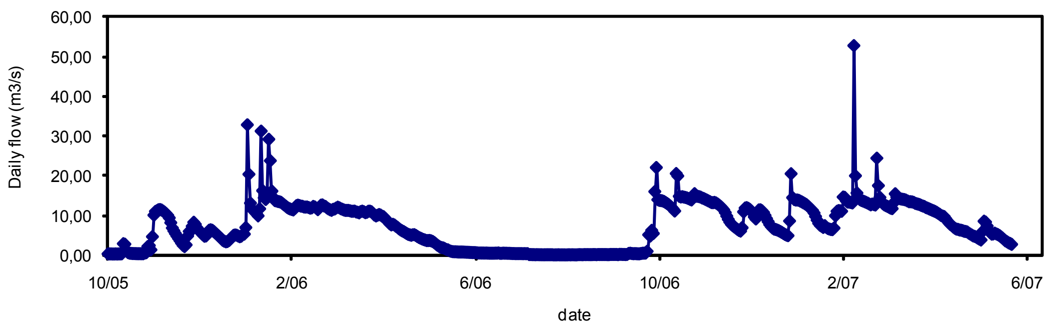


Figure 3: Flood Event in the Koiliaris River

1.7 Thesis Outline

The current thesis is organized as follows:

- **Chapter 2** provides the theoretical background of the technical discussions and terms in this thesis. Basic concepts of reconfigurable hardware, system architectures and technologies are presented.
- **Chapter 3** provides the literature survey on reconfigurable platforms.
- **Chapter 4** discusses the proposed system architecture.
- **Chapter 5** refers to the system's software architecture with a detailed analysis on the implemented task scheduler
- **Chapter 6** focuses on the reconfigurable co-processor architecture along with the platform implementation.
- **Chapter 7** gives a synopsis of the current work and results.
- **Chapter 8** discusses directions for future upgrades of the proposed system.

Chapter 2
Background

2.1 Introduction

This chapter offers the reader the background information for a complete understanding of the technical discussions and terms in the following chapters. Basic concepts of reconfigurable hardware, system architectures and technologies are presented.

2.2 Reconfigurable Hardware Technology

A Field Programmable Gate Array (FPGA) is a matrix of configurable logic cells, called Configuration Logic Blocks (CLBs). These blocks are embedded in a general routing structure (also configurable) which allows their interconnections (inputs and outputs of each CLB). As opposed to Application Specific Integrated Circuits (ASICs) where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements [5].

The Configurable Logic Block is the basic logic unit in an FPGA. Exact numbers and features vary from device to device, but every CLB consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc.), and flip-flops.

The FPGA configuration is specified with the use of a Hardware Description Language (HDL). The programming technology in an FPGA determines the type of basic logic cell and the interconnect scheme. The logic cells and interconnection scheme, in turn, determine the design of the input and output circuits as well as the programming scheme.

Although one-time programmable (OTP) FPGAs are available, the dominant type is SRAM based. The advantages of SRAM based FPGAs is that designers can reprogram them. The disadvantage of using SRAM programming technology is the need to keep power supplied for the volatile SRAM to retain the connection information. Alternatively, one can load the configuration data from a permanently programmed memory (typically a programmable read-only memory or PROM or Controller) every time the system is turned on.

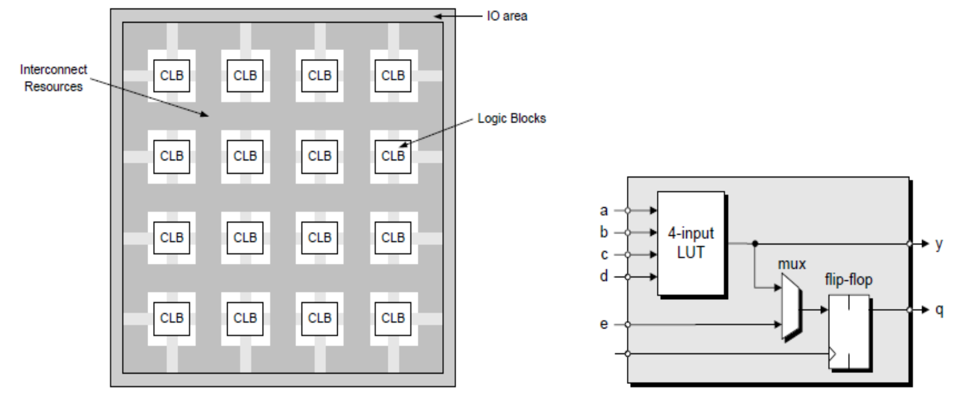


Figure 4: FPGA Architecture and CLB

2.3 Configuration Techniques

Configuration is the process of loading design-specific configuration bitstream data into one or more FPGAs/CPLDs to define the functional operation of the internal blocks and their interconnections. An SRAM-based FPGA can be configured on its power-up or even on demand, depending on the architecture of the device. The EEPROM based CPLDs can be programmed on demand and they keep their configuration data even after power-off.

For Xilinx FPGAs, those changes to structure and functionality are made by loading configuration bitstream data through one of several configuration ports.

External configuration ports such as the SelectMAP and JTAG interfaces are typically driven by an external controller. This has the potential of allowing applications to have a smaller operational area on the FPGA and of consuming less power.

1. **Slave Serial/SelectMAP (Slave Parallel)**: Slave Modes use external control logic to generate the configuration clock. It allows the FPGA to be configured using other logic devices such as microprocessors, or in a daisy-chain. The device is incorporated into a system with an intelligent host that controls the

configuration process. The intelligent host transparently selects a serial or parallel data source and the data is presented to the device on a common data bus. Such systems can store the configuration data on a mass-storage device, such as a hard disk. This way, installing new configuration data becomes easier and the number of Integrated Circuits (ICs) required for a system is reduced.

2. **JTAG**: The Joint Test Action Group has developed a specification for boundary scan testing. The Boundary Scan Test (BST) is an industry standard (IEEE 1149.1, or 1532) and it offers the capability to efficiently test components on PCBs with tight lead spacing. JTAG has gained popularity due to its standardization and ability to program both FPGAs and CPLDs. In this mode external logic is also required but this time to drive the JTAG specific pins, Test Data In (TDI), Test Mode Select (TMS) and Test Clock (TCK), and one optional the Test Reset (TRST). All other pins are tri-stated during JTAG configuration. JTAG configuration can start at any time, even during configuration through another mode.
3. **ICAP:** In contrast to JTAG and Slave Modes, the internal configuration access port (ICAP) can be directly accessed by application circuits configured on the FPGA (not available on all FPGAs), allowing them to change their own structures and functionalities at run time. To achieve this, different circuits with different functionalities are loaded onto the FPGA when needed by those applications.

2.4 Reconfigurable WSN nodes architectures

There has been considerable research into reconfigurable architectures and coupling. Todman et al. [6] extended the work of Compton and Hauck [7] and have classified the reconfigurable hardware architectures into 5 main classes (Figure X): It is outside the scope of this thesis to emphasize on all reconfigurable architectures. The main focus is those used in reconfigurable WSN nodes.

In typical, reconfigurable WSN nodes also have a microprocessor, memory, and possibly other structures. Whether the reconfigurable logic (RL) is depicted as a separate coprocessor or integrated as a functional unit mostly depends on the system architecture and the coupling between the general purpose processor (in this case microprocessor) and the FPGA.

The key classification is based on their system architecture and the nature of their RL unit. They are categorized as:

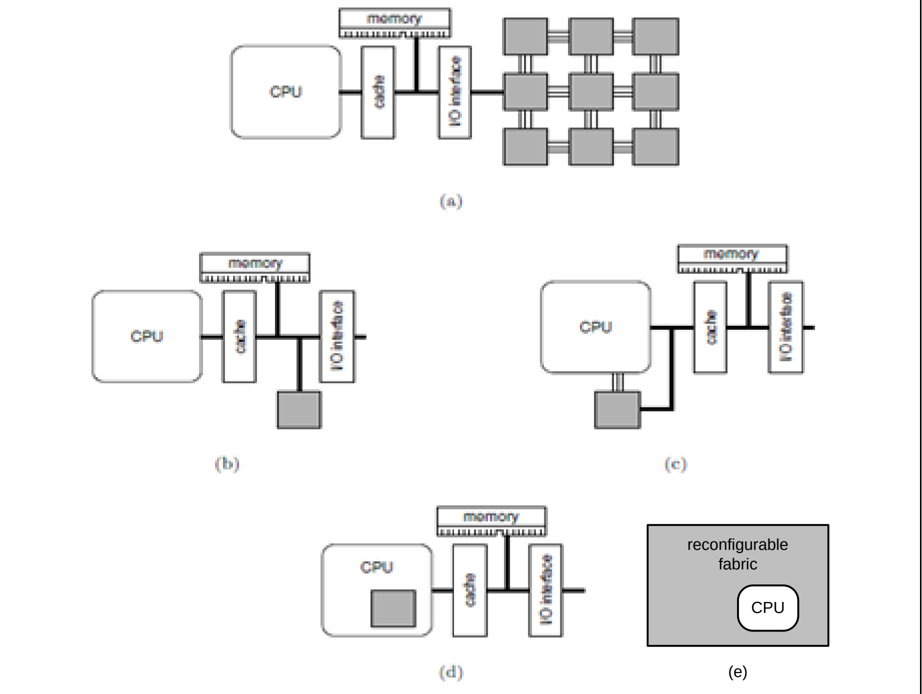


Figure 5: Reconfigurable hardware architectures and coupling

2.4.1 Microcontroller plus Reconfigurable Co-processor

In these platforms the RL is coupled to the processor through a) its system I/O bus or b) its system memory bus. The first case is the easiest implementation with the drawback of providing the least data bandwidth between the processor and the RL, which is usually a system's performance bottleneck. In the second case, all data communications take place through the main memory. The RL performs its computations and returns the results back to main memory thus increasing the system's bandwidth.

In general, since the RL has no direct transfer link to the processor even a conventional microprocessor platform can be extended to a reconfigurable system by simply inserting an add-on card with reconfigurable logic to the system's peripheral bus. In the second case, the extension is a bit more complicated but equally feasible. The main disadvantage in this coupling is that the overall data bandwidth of the system is limited.

For this reason independent co-processor RLs are best suited for application that require data-streaming, like image processing and encryption where the RL acts independently from the processor.

2.4.2 Reconfigurable hardware plus embedded processor

Advances in reconfigurable logic technologies have made possible the tight coupling between processor and reconfigurable logic. Instead of deploying the RL to a processor system, new reconfigurable ICs embed the processors in their fabric. Such architectures allow the direct access to the reconfigurable logic from the processor. Nowadays, almost all vendors provide reconfigurable fabrics with embedded processor cores. These processors can be implemented physically or as soft processors. A processor built from dedicated silicon is referred to as a “hard” processor. A “soft” processor is built using the FPGA’s general-purpose logic. The soft processor is typically described in a Hardware Description Language (HDL) or netlist. Unlike the hard processor, a soft processor must be synthesized and fit into the FPGA fabric [8]. Examples of soft processors are the MicroBlaze and PicoBlaze processors by Xilinx, as well as the Nios and Nios-II processors by Altera. As for hard processors, Altera’s Excalibur family embeds the ARM processor and inside Xilinx’s Virtex family is the PowerPC processor. Additionally, ATMEL offers FPSLIC family with an AVR core embedded. [1,9,10]

This close coupling between the processor and the RL increases the efficiency of the system by increasing communication and data transfers. On the other hand, it limits the RL’s independence. By assigning the RL the role of a functional unit, means that it is placed directly in the pipeline of the processor, potentially stalling execution until it terminates its task.

2.5 Chapter Summary

This chapter highlights technologies regarding reconfigurable hardware configuration processes and system architectures used in reconfigurable wsn architectures.

Chapter 3
Literature Survey

3.1 Introduction

Current technological advances have led to the availability of environmental sensors that are smaller, cheaper, intelligent and more reliable. On the other hand, traditional WSN platforms based solely on microcontrollers fall short of providing flexible and adequate solutions in response to the increased processing and data demand. These drawbacks have led, among other things, in the adaptation of reconfigurable logic (RL) as hardware accelerators in WSN platforms. Due to the diverse nature of environmental monitoring applications and the importance of reconfigurability at the hardware platform level as illustrated in previous sections, this chapter is mainly focused on the FPGA-based reconfigurable platforms currently available.

3.2 Reconfigurable WSN nodes architectures

In typical, reconfigurable WSN nodes also have a microprocessor, memory, and possibly other structures. Whether the reconfigurable logic (RL) is depicted as a separate coprocessor or integrated as a functional unit mostly depends on the system architecture and the coupling between the general purpose processor (in this case microprocessor) and the FPGA.

3.3 Microcontroller + reconfigurable co-processor platforms

In this section platforms based on the microcontroller + reconfigurable co-processor approach will be discussed.

3.3.1 mPlatform

At Microsoft Research Labs they developed a modular stackable platform [11]. The mPlatform was designed so that a wide range of processors can coexist on the same platform and efficiently communicate in any possible configuration.

The coupling of the heterogeneous microprocessors is achieved by a reconfigurable HW (CPLD in this case). The local processor on each module interacts with a parallel bus through the bus controller, implemented in a low-power CPLD thus achieving communication abstraction through the entire modular platform.

In the mPlatform the term reconfigurability is used only to emphasize on the platform's Lego-like nature where the number and type of processors put together depends on the requirements of a given research project.

With this in mind, the CPLD is configured only at design time in order to meet the specific communication demands of a particular project.

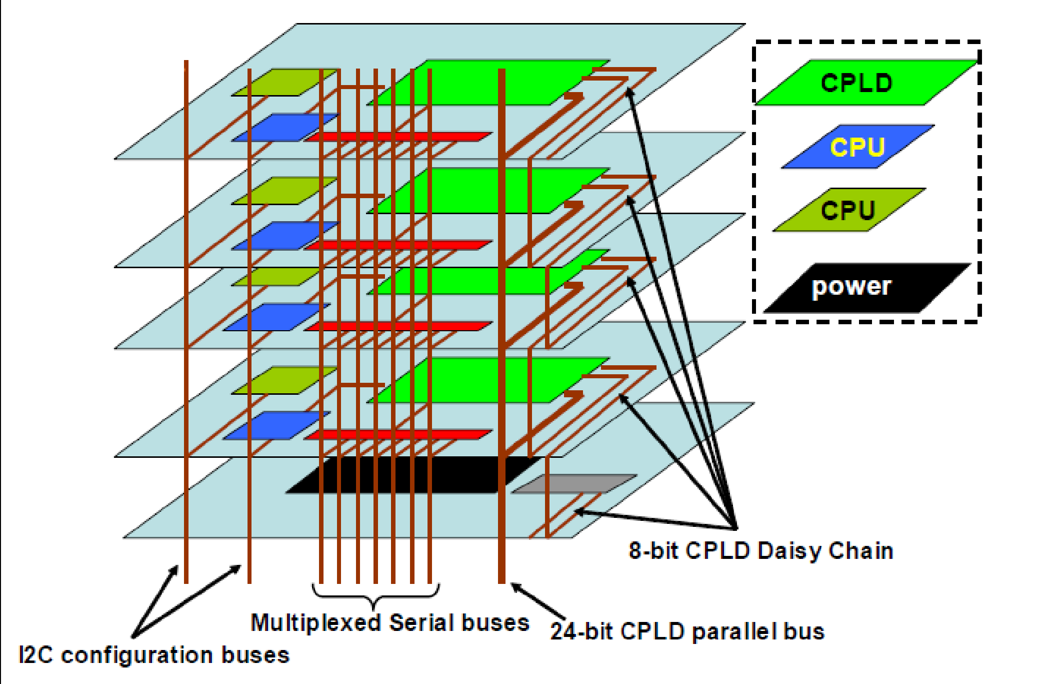


Figure 6: mPlatform architecture

3.3.2 Cookie

At the CEI-UPM, researchers have developed a platform based on a 8052 uC from Analog Devices (ADuC841) and a Xilinx XC3S200 Spartan 3 FPGA [12]. It is composed of four main layers: processing, communication, power supply and sensors. The FPGA acts as an independent co-processor solely for taking measurements from digital sensors (Figure X2). The uC sends triggers to the FPGA, specifying the sensor from which the measure has to be taken, and the FPGA activates the corresponding sensor interface.

The Cookie platform supports dynamic reconfiguration of the FPGA. It uses the uC to reconfigure the FPGA from a library of general HW digital interfaces for sensors (as I2C, 1-Wire, SPI, etc.). The uC will use the JTAG port of the FPGA to accomplish this task. The uC will receive the bitstream from the communication layer (ZigBee module) and will download it in the FPGA configuration memory.

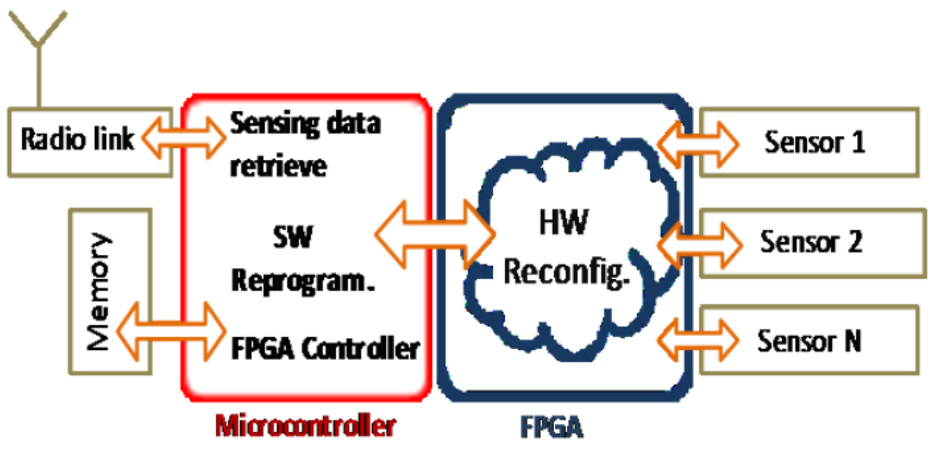


Figure 7: HW-SW Reconfigurable Sensor Node Diagram

3.3.3 RESENSE

RESENSE is a reconfigurable WSN platform, developed at the Technical University of Crete [13]. In the RESENSE platform there are two distinct architectures that incorporate reconfigurable logic. In the sensor node level researchers have coupled a commercial WSN mote (Micaz/IRIS - ATMega1281 uC) and a low-power CPLD (Xilinx CoolRunner-II). On the base station they have combined an Intel Atom processor (general purpose CPU) with a Virtex-5 FPGA (XC5VLX110T).

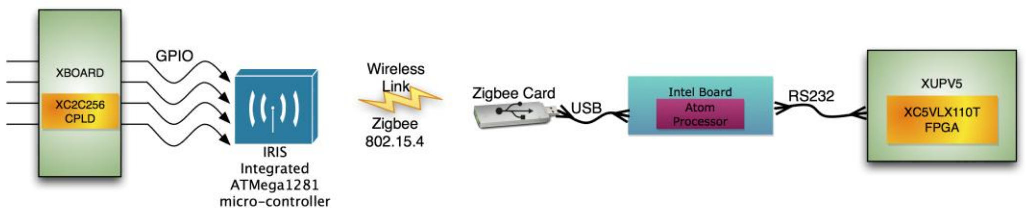


Figure 8: RESENSE general scheme

Both Reconfigurable Nodes are treated as hardware accelerators by executing security (encryption/decryption/authentication) algorithms for providing a network-level security framework. They have proven that the overall energy consumption of the new infrastructure is reduced by up to 98%, when compared with the consumption of a widely used commercial CPU-based WSN node executing those same WSN processing tasks in software.

3.3.4 Tyndall (25mm Cube)

The Tyndall Mote was developed at the Tyndall National Institute by the Wireless Sensor Networks Team [14]. The Tyndall Mote is a compact, reconfigurable and modular platform. The design is based around several 25×25mm boards that are interconnected by means of two standard connectors placed on contiguous sides of each of the square boards.

FPGA technology and additional processing capability can easily be incorporated into the system stack when required by simply adding the required layer, similarly appropriate power supplies and battery layers or coin cell battery layers can be stacked one on top of each other, also in a modular fashion

All of the communication layers are designed with an on-board ATmega128L microcontroller developed to integrate the radio and transceivers, as well as being compatible with TinyOS, and other standard.

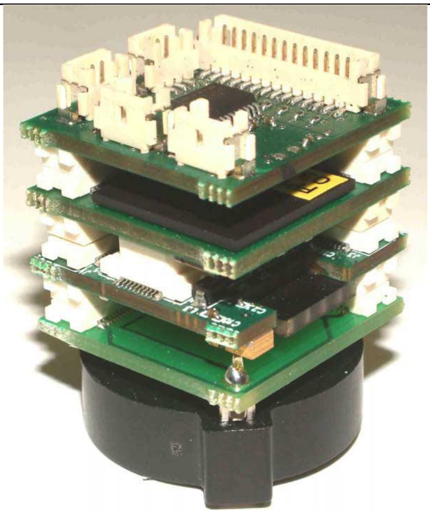


Figure 9: Tyndall's 25 mm stackable platform

In this case also, the main research is not focused on FPGA reconfiguration, but rather in reprogramming the microcontroller and the FPGA before deployment.

3.3.5 REWISE node

Wilder et al. [15] present a reconfigurable wireless sensor network (REWISE), showing that FPGAs reach an optimum balance between flexibility, energy requirements and processing power. Their node is based on a 16bits microcontroller (MSP430) from Texas Instruments. Their system is an implementation of what they call wireless JTAG, which is independent of the final FPGA that will be used. Each node includes a repository of HW/SW configurations and programs to avoid the high cost of sending these files through the network.

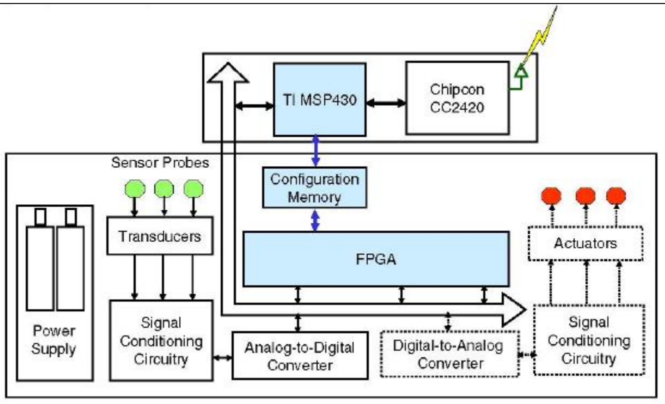

TI MSP430
Chipcon CC2420
Sensor Probes
Configuration Memory
Transducers
FPGA
Actuators
Power Supply
Signal Conditioning Circuitry
Analog-to-Digital Converter
Digital-to-Analog Converter
Signal Conditioning Circuitry

Figure 10: REWISE reconfigurable node

3.4 Reconfigurable hardware + embedded processor

In this section platforms based on embedded processors in the reconfigurable fabric will be considered.

3.4.1 Darmstadt platform

At the Institute of Microelectronics System of Darmstadt University, they have developed a reconfigurable prototyping platform based only on a single Spartan3-2000 FPGA as the main processing [16]. The board has a CPLD only for reconfiguration purposes. In their application example, the heart of the system is a soft core 32-bit LEON2 RISC processor with the rest of the reconfigurable logic acting as functional unit (RFU) integrated directly into the processor's data path. The FPGA is used to integrate debugging and system monitoring in the logic, and to emulate the digital part of the final node. Therefore, their main target is prototyping. There is no reference to any kind of system reconfiguration of the platform, neither static nor dynamic.

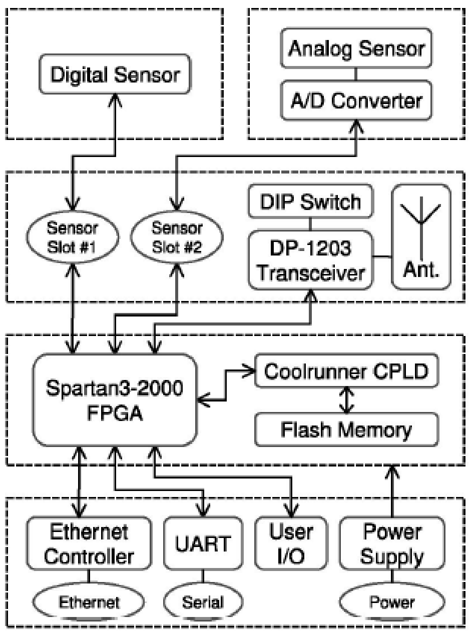


Figure 11: Schematic view of the Darmstadt platform

3.4.2 Warangal platform

In the National Institute of Technology, Warangal in India, Muralidhar and Rao, developed a Reconfigurable WSN node using an ALTERA Cyclone II FPGA [17]. The processing unit (treatment unit by the authors' terms) is a NIOS II soft core. They proposed a real-time forest fire detection system by using reconfigurable wireless sensor networks. Their target is to reuse the reconfigurable functional unit (RFU) through dynamic reconfiguration.

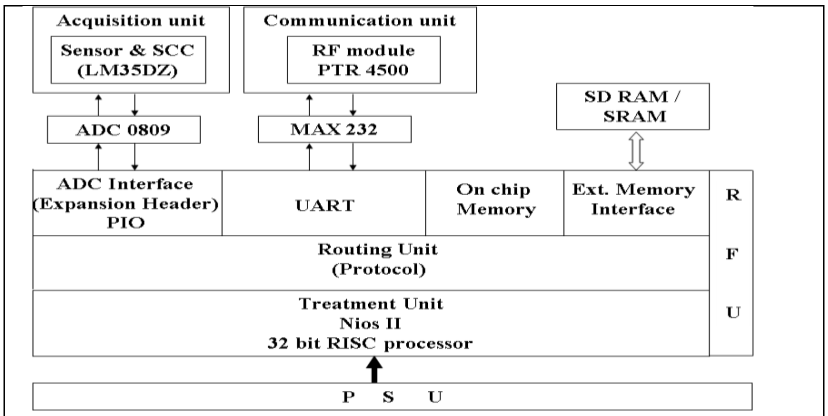


Figure 12: Warangal platform node architecture

3.5 Other Reconfigurable WSNs

Latha et al [18] from College of Engineering, Anna Univ., Chennai, India, have proposed a two-level system for surveillance and intrusion detection. At the first level, relatively simple nodes with basic sensing devices are used as primary detectors of intrusions. The second-level sensor node is composed of a high performance FPGA that can be dynamically reconfigured for different applications, interfaced with the camera.

There are no additional references to the way the FPGA in the node is dynamically reconfigured.

3.6 Summary

In this chapter we gave a detailed survey on existing reconfigurable platforms. **In contrast to our system, most platforms treat the reconfigurable logic as a functional accelerator with no reference to run-time dynamic reconfiguration.**

Chapter 4
System Architecture

4.1 Introduction

This chapter focuses on the description of the platform which includes a microcontroller and an FPGA for acting as a coprocessor in terms of its hardware architecture.

4.2 System Architecture (Layers)

The reconfigurable platform is partitioned into three layers, for sensing, communication and processing as proposed in [19] and illustrated in fig. 12.

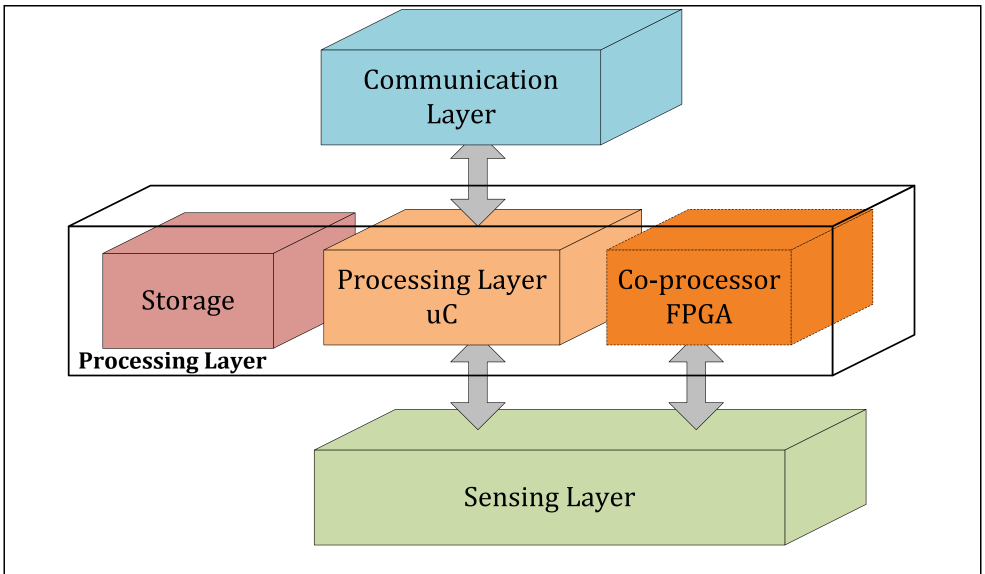


Figure 13: Layers of the reconfigurable Platform

This layered- modular design approach is adopted by many researcher [20,21,11,22] mainly because it can provide scalability and reusability. It offers a level of transparency between the different layers. Each layer only communicates with its adjacent layer. The "communication" layer interfaces only with the control medium ("processing" layer), inside the node. The existence of the reconfigurable section, its functionality and even its presence, is completely hidden from the "communication" layer. The platform can be easily upgraded with the least effort (both in hardware and software redesign) and with minimal cost. For example, the communication layer or the sensors can be replaced without the need to redesign the entire platform, as long as the interfaces remain unaltered. All of the above are depicted in Figure 12.

4.2.1 Communication Layer

The "Communication" layer is the physical and data-link layer. It typically consists of the wireless communication module and its controller. All incoming data and control instructions that reach the node from the network are collected by this layer. They are parsed and stripped from any communication related information and are forwarded to the "Processing" layer. Additionally, all processed or raw data collected from the sensors are transmitted through the communication layer to a base-station on demand or at specific time intervals, depending on the communication policy of the deployed network and the platforms appointed tasks.

The modular approach of the platform permits the use of different communication protocols and physical layers. In the past, a platform-compatible Communication Layer consisting of a Bluetooth module and its controller has been implemented [19].

For the purpose of this thesis, the communication layer has been implemented as a point to point serial, communication interface through the platform's UART. The platform can receive a series of data and control commands form the communication layer which, refer to the following operations:

1. **Replace TASK INI instructions:** A sequence of commands and data for replacing the "task-ini" file located in the storage unit. This file contains the user-defined task parameters, necessary for controlling the entire system's behavior.

2. **Replace PTL Script instructions:** Commands followed by a sequence of data, which replace the PTL script controlling the FPGA (re)configuration process
3. **System Reboot command**: Control instruction for performing a system reset.
4. **Write Bitstream instructions**: Command and data for placing a new (or replacing) an FPGA application file (bitstream).

The entire platform can be remotely configured at runtime, by sending it a sequence of control instructions and data.

The following table contains all the control instructions the platform can receive from the communication layer. The opcode of the instructions in Mnemonics syntax is displayed in the left side of the Table followed by the number of operands of each instruction.

Mnemonics	Operand	Description	Opcode
RESET		System's "Software" Reset	0x01
Replace TASK INI instructions			
TASK_PROG_START		Accessing the"Task_ini" file for write and truncating the file to zero length	0x02
TASK_PROG_BYTE	X	Receive X bytes of "Task_ini" file contents	0x03
TASK_PROG_KBYTE	X	Receive X Kbytes of "Task_ini" file contents	0x04
TASK_PROG_STOP		Accessing and closing the "Task_ini" file (like fclose)	0x05
Replace PTL INI instructions			
PTL_PROG_START		Accessing the "ptl" script file for write and truncating the file to zero length	0x06
PTL_PROG_BYTE	X	Receive X bytes of "ptl" script file contents	0x07

Mnemonics	Operand	Description	Opcode
PTL_PROG_KBYTE	X	Receive X Kbytes of "ptl" script file contents	0x08
PTL_PROG_STOP		Accessing and closing the "ptl" script file.	0x09
Replace bitstream instructions			
BSTR_START		Accessing a bitstream file for write and truncating the file to zero length (create it if it doesn't exist)	0x0A
BSTR _PROG_BYTE	X	Receive X bytes of bitstream file contents	0x0B
BSTR _PROG_KBYTE	X	Receive X Kbytes of bitstream file contents	0x0C
BSTR _PROG_STOP		Accessing and closing the bitstream file.	0x0D

4.2.2 Processing Layer

The "processing" layer is the platform's core. It consists of an 8-bit microcontroller, a storage unit and the FPGA co-processor for enabling efficient data retrieval and processing. More details about the architecture of the "processing" layer will be provided in the following sections. The processing layer is the control medium of the entire system. It is responsible for a) raw data collection from the "sensing" layer, b) data processing, c) data storage and d) data transmission to the network though the communication layer.

In cases where the system cannot rely solely on the microprocessor for data gathering, the processing layer is also responsible for triggering and dynamically (re)programming the FPGA to act as a co-processor

The layer's "storage" unit acts as the system's main memory. It is partitioned into a) **Data Memory**: Stores all the measurements (raw or processed) from the sensing layer, b) **RL Configuration memory**: stores all configuration files (bitstreams and configuration instructions) for the FPGA (re)configuration process and c) **Instruction Memory**: System initialization files.

4.2.3 Sensing Layer

This layer is connected with the “processing” layer and hosts all the sensors necessary to take measurements from the environment. It provides the necessary communication interfaces of different sensors (pH, nitrate, DO, temperature or camera), with the processing layer.

The topology of the attached sensors and their interfaces with the microcontroller or the FPGA is of course dictated by the application for which they are deployed. As a rule of thumb though, it is better to have the sensors with the lowest data throughput (temp, flow and chemical sensors) controlled by the microcontroller in the processing layer. On the other hand, cameras are best managed by the FPGA since it has the capability to handle more complex interfaces.

4.3 Platforms’s Datapath

The platform contains 5 main modules as depicted in the following figure.

4.3.1 Control Unit

The control unit is responsible for issuing and scheduling tasks on the data-logger unit (uc-tasks) or the co-processor (fpga-tasks). When an fpga-task is to be executed, the control unit addresses a request to the “configuration” unit. Along with the request, it passes information concerning the entire configuration process of the FPGA (bitstream location, priority level, configuration process). The allocation of all the tasks along with the scheduling algorithm will be discussed in a following section.

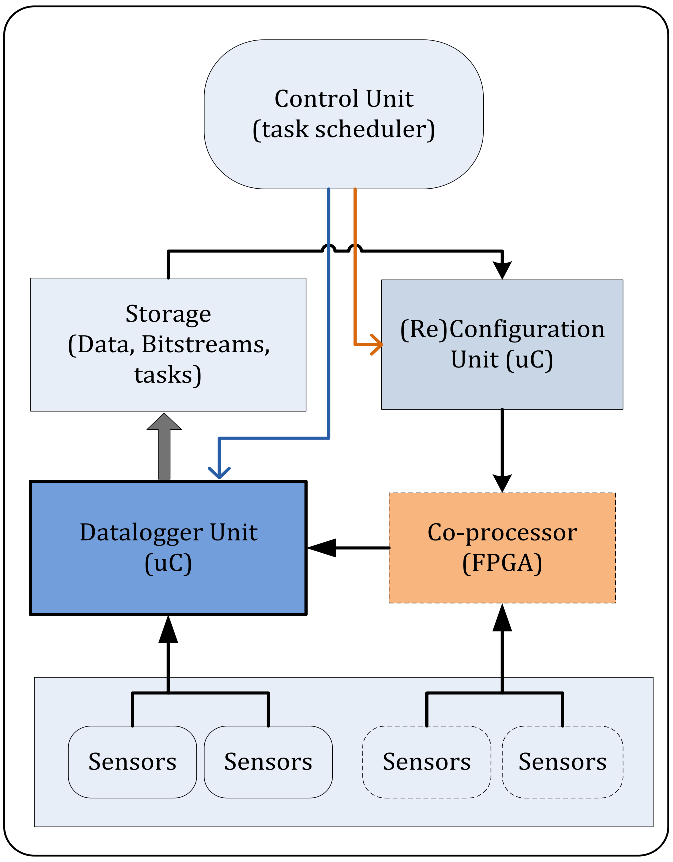


Figure 14: Platform's Datapath

4.3.2 Configuration Unit

As described earlier, the configuration unit is responsible for placing a given task in the FPGA by configuring it with the provided bitstream. It receives and places all incoming requests for FPGA configuration in the "FPGA Arbitration Buffer" (FA BUFFER). Inside the "FA Buffer" all pending requests are sorted based on their accompanied priority level. This way, FPGA configuration requests with higher priority levels are executed before requests with lower configuration levels.

It is actually an updated version of the Hardware Programmer & Tester (H.P.T.) which was originally developed as a download and testing apparatus in close coupling with the Reconfigurable Run – Time Environment ReRun [23,24] developed also in our lab. In this version the HPT is completed detached from the PC-based ReRun and is solely controlled by the "scheduler" unit. Yet, it continues to support the PTL formal language, created for the RTE ReRun.

The HPT communicates with the co-processor through two dedicated 8-bit configuration ports. They used for dynamic reconfiguration and readback purposes, also attached to the reconfigurable section of the node.

4.3.3 Co-processor Unit (FPGA):

The co-processor (FPGA) is dynamically triggered by the “configuration” unit upon request from the control unit. When the FPGA is programmed, it performs the task for which it has been activated and is then deactivated.

The co-processor only communicates with the data-logger (microprocessor) through a serial interface for sending and receiving data. In other words, the microcontroller “sees” the FPGA as an external source of data.

4.3.4 Data logger Unit

The data-logging unit is the main module for collecting data either from the sensors attached to it or from the co-processor. The tasks executed on the data-logging unit (microprocessor) can be event or time driven.

4.4 PTL Language

The PTL language (Programming and Testing Language) can be used to write scripts to describe the (re)configuration or testing process of a reconfigurable unit. PTL has a BNF grammar, well-defined semantics, and is compiled with flex and bison.

One of the advantages of having a formal language for the environment is that the behavior of the system as a whole is unambiguous. PTL instructions are categorized into two different types. The “programming” instructions, needed for implementing any (re)configuration, readback or test process at the reconfigurable layer and the “protocol” instructions. Our platform only supports the PTL “programming” instructions. The HPT module is connected through an SPI interface to the memory unit. The RL Configuration memory (controlled exclusively by the HPT) stores the configuration bitstreams necessary for the FPGA (re)configuration, and the appropriate scripts [25].

Each PTL instruction is divided into 2 main fields: the opcode and the operand. The following table contains all the programming instructions.

Mnemonics	Operand	Description	Opcode
CLEAR_BITS	X	Clear “CNTRL PORT” bit(s)	0x03
CLK_HIGH		Initialise Configuration Clock at logic level ‘1’	0x1E
CLK_LOW		Initialise Configuration Clock at logic level ‘0’	0x1D
CNTRL_BITS	X	Send Byte to “CNTRL PORT”	0x0A
CNTRL_MAP	X	Define “CNTRL PORT” bit direction	0x02
DATA_SERIAL		Send Data from “DATA PORT” serially	0x05
DATA_PARALLEL		Send Data from “DATA PORT” parallel	0x06
LSB		Send data serially (Least SB first)	0x1F
MSB		Send data serially (Most SB first)	0x20
NOP		No operation	0x0D
PROG_BYTE	X	Send X bytes of data through “DATA PORT”	0x08
PROG_KBYTE	X	Send X Kilobytes of data through “DATA PORT”	0x09
SET_BITS	X, Y	Set “CNTRL PORT” bit(s)	0x04
Branch Instructions			
CSEQ_CNTRL_A	X	Compare “CNTRL PORT” contents with X. Skip if Equal	0x22
FOR_LOOP_START	X, Y	For Loop Start	0x0B
FOR_LOOP_END		End of For Loop	0x0C
SBC_CNTRL		Skip if bit(s) in “CNTRL PORT”	0x24

Mnemonics	Operand	Description	Opcode
		is (are) clear	
SBS_CNTRL		Skip if bit(s) in "CNTRL PORT" is (are) set	0x27

Table 1: Programming Instructions

4.5 System's Interfaces

The platform's interfaces are divided into 5 main sections as shown in the following figure.

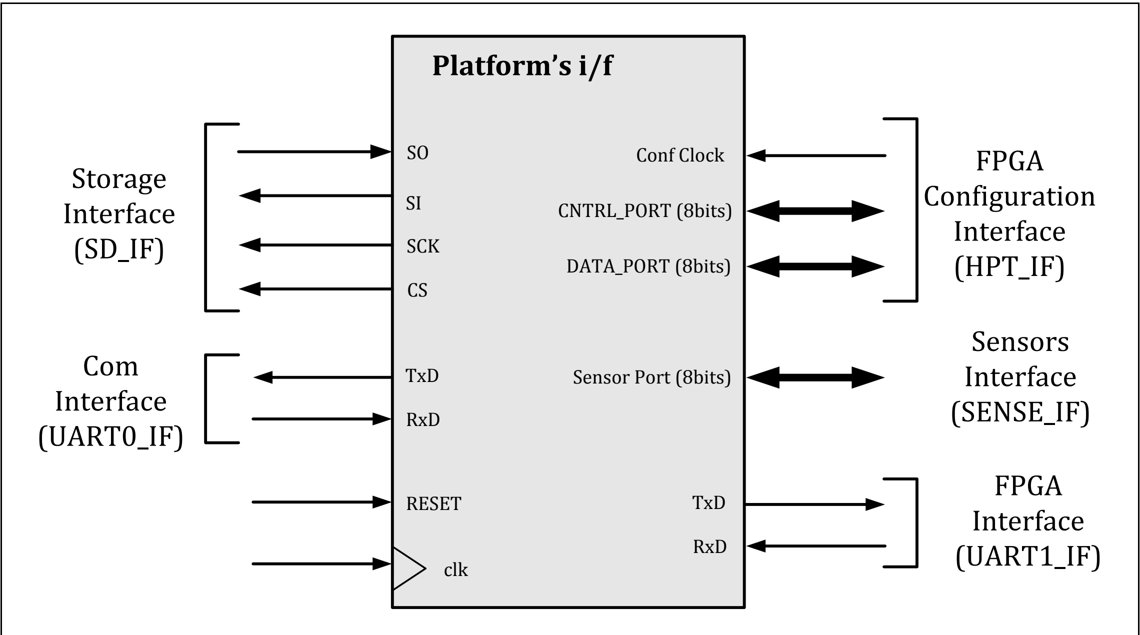


Figure 15 Platform's Interfaces

4.5.1 Communication layer Interface (UART0_IF)

This serial interface is provided as standard and is maintained internally by the control unit. It uses the UART0_IF during normal operation in order to communicate with the network and acquire data and instructions necessary for its operation.

4.5.2 Storage Interface (SD_IF)

The storage module that has been selected is a serial flash memory. Although slower than parallel interface, the serial interface is ideal in space-sensitive systems such as the current one. The sequential access serial interface scheme employed through a Serial memory pinout enables a practically limit-free upgrade path for either density or word-width. Conventional random access, parallel interface Flash must use dedicated address pins to interface with the system. The Serial Peripheral interfaces with other devices using only seven signal leads, three of which are dedicated to the serial bus (SCK, SI and SO). The remaining signal leads on the Serial memory include a chip select (CS), chip reset input (RESET), a write protect input (WP), and a ready/busy output (RDY/BUSY).

The Serial memory can be used with any type of micro controller, but the interface of the device is also compatible with SPI modes to provide simple interconnections with the increasingly popular SPI micro controllers. SPI is a serial interface protocol, utilizing 8-bit words, useful in communicating with external devices such as serial EEPROMs and the Serial DataFlash.

4.5.3 Sensor Interface (SENSE_IF)

The sensing interface is responsible for communicating with the “sensing” layer. It consists of an 8 bit port used to send and receive data and commands to several sensors directly attached to the microcontroller. The sensors can be digital and are accessed through several supported protocols like 1-Wire Protocol, or analogue and can use the system’s built-in ADC.

4.5.4 FPGA Configuration interface (HPT_IF)

The configuration interface is responsible for sending data and control signals to the target FPGA. It consists of two 8bit ports and an independent pin the Conf CLK.

The data port, when in programming mode, sends configuration data to the target FPGA(s) in parallel or serially. In serial configuration mode, data can be sent starting from the MSB or the LSB pin, depending on the configuration protocol.

The Control port (CNTRL_PORT) can be used to manipulate the configuration process by issuing appropriate patterns. Each pin in the Control Ports can be used independently.

The Conf CLK is the main clock source of the target FPGA(s) during the configuration cycle. With each clock pulse on the Conf CLK line, configuration data are transferred, serially or in parallel, through the data port. Since the configuration algorithms differ between FPGAs, even in the use of the configuration clock source, the Conf CLK is capable of producing pulses with minimum high level duty cycle or vice versa as shown in the following figure.

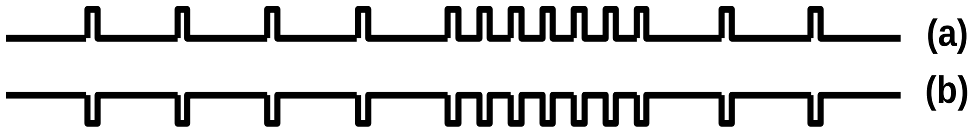


Figure 16: CONF CLK pulses with:
(a) Low level duty cycle >50% and **(b)** high level duty cycle >50%.

4.5.5 FGPA Data Interface (UART1_IF)

The FPGA communicates with the uC through a serial interface. The advantages of using a low cost and robust communication serial bus between the FPGA and the microcontroller are:

a) It is an easily implemented unified communication medium, mainly on the FPGA.
b) It doesn't burdens the microcontroller's limited resources and pin count.
c) The architecture and the interface of the microcontroller remain unaffected and independent of the hardware characteristics of the FPGA.
d) Retains data synchronization. For example, the reconfigurable section (FPGA) can be driven by a 200MHz system clock. On the other hand, the 8bit microcontroller is driven by a ~12MHz system clock. In this case, asynchronous serial communication is a simple and ideal communication

interface to overcome the synchronization problems imposed in a system where each component is driven from a different clock source.

e) It offers the easiest scalability mechanism

The drawback though is that it provides the least data bandwidth between the microcontroller and the FPGA. If the designer doesn't utilizes the FPGA solely as a unit for data acquisition but rather as a source of high processing capabilities, the serial communication bottleneck can be alleviated. For example, the FPGA can transmit through the serial i/f, only processed and compressed data.

Chapter 5
Task Scheduler Architecture and Implementation

5.1 Introduction

Most conventional Real-time Operating Systems (RTOSs) are not suitable for platforms based on the CPU-FPGA architecture. This is mainly due to their inability to integrate dynamic runtime (re)configuration of the FPGA in their scheduling algorithm. Many RTOSs provide support for a Functional Unit or a co-processor simply by seeing it as a static (one-time configurable) module. In this chapter, we present the software we have developed for our system.

5.2 Software Architecture (Abstract View)

Our software architecture follows the microkernel concept. By the term microkernel we refer to a near minimum set of services implemented in software in order to provide to the user a hardware resources level of abstraction.

More specifically, our kernel is composed by a set of services that run exclusively on the microcontroller. These services call device drivers for accessing different hardware components of the platform. Among others, the Storage Area through a FAT file-system, the serial ports and the different sensors attached to the platforms.

The user is responsible for writing and formulating application tasks based on these predefined services. In our approach, the tasks utilize the microcontroller resources and the adjacent sensors form the “sensing layer” and only in application critical tasks do they use the FPGA resources.

5.3 Kernel Modules and Services

The implemented kernel consists of the following primary modules and services, as illustrated in figure 19:

- **Task Parser Module:** It is activated during the system's initialization phase for reading the "task-ini" file and parsing the user defined tasks and their parameters.
- **Task Allocation Module**: This module controls the queue that a newly appointed or rescheduled task is going to be placed in.
- **Task Dispatcher Module**: Based on the incoming interrupts (from the timer and event handling services) it decides to which task it is going to give control of the microcontroller resources.
- **Task Scheduler Module**: The scheduler is the main core of the entire microkernel. It is controlling the execution and termination of the user appointed tasks along with their accompanied services at runtime and based on the scheduling model and algorithm that will be discussed in a following section.
- **FPGA Arbiter Module**: This module is responsible for controlling the FPGA configuration process. It is triggered by the scheduler and according to the user defined tasks. The configuration algorithm will be analyzed in a following section.
- **Sensor Services**: These are preinstalled kernel functions that provide the mechanism for data acquisition and storage.
- **Device Drivers**: They provide access to the system's hardware resources, like the memory unit, the serial ports. They are also responsible for providing access to the different sensors attached to the platform.
- **FPGA Configuration Service**: Although the FPGA can be categorized with the rest of the resources, the complexity and the importance of this service permit us to treat it as a discrete one. It is responsible for providing the configuration mechanism for the attached FPGA. It is actually the software model of the reconfiguration unit (HPT)

- **Timer-handling service**: This service is responsible for offering time-based triggers to services and tasks.
- **Event-handling service**: It offers event-based triggers to the scheduler based on interrupts from external or internal resources.

Figure 19 shows the modules that provide the operating system services.

5.4 Scheduler Model - Multitasking

The adopted scheduler and correlative algorithm is capable of scheduling and executing tasks on the microcontroller and utilize the platforms resources, including the FPGA, dynamically and at runtime.

The scheduling model for the tasks executed in the microcontroller is based on a cooperative scheduler [26] in order to avoid the need for a preemptive one which is costly in memory usage (a stack per task) and context switching times. Additionally we wanted to preserve “sensitive” services like FPGA configuration services from being arbitrarily switched (preempted) while they are accessing the FPGA.

The main drawback in using a cooperative scheduling algorithm is that each task must ‘voluntarily’ give up its execution and allow the kernel to make a task switch, in order to maintain task level parallelism. If a task takes too long to execute then the entire task sequence is stalled until the task has finished execution and has released the microcontroller resources.

In our platform, this task level parallelism is maintained by dividing the system’s execution time into time slices. The running services are decomposed into a number of sequential states as in finite state machines (FSMs) that are executed in these predefined time slices. This way a task is executed in many time slices. This alleviates the problem of task stalling. Since the tasks that are executed in our platform are mainly services that require many I/O transfers (data from sensors and data storage) it means that they take a lot of time to finish. So the task scheduling time granularity is carefully picked in order to be able to accommodate lengthy I/O Transfers (i.e. 1/8th of a second). Of course there can be task states that exceed the designated time slice, but these only occur in rare conditions, like for example when programming the FPGA.

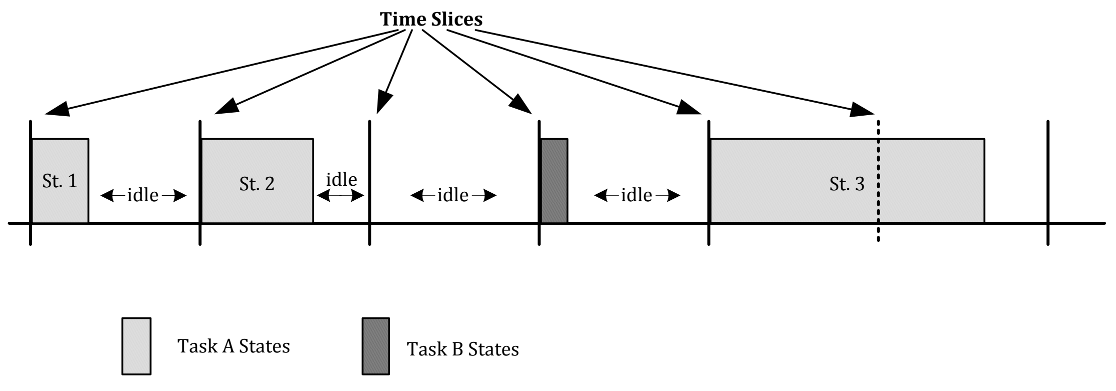


Figure 17: Cooperative scheduling with time slices

5.5 Kernel Services

As mentioned earlier, a service is a generic kernel module that is structured as a set of sequential states as in finite state machines.

There are two basic services in the system as shown in the following figure, the "**FPGA Configuration"** service and the "**Data Acquisition**" service.

The "**Data Acquisition"** service is the basis for implementing user defined data-logging tasks. It controls the entire process of acquiring readings from a sensor, processing and storing them. It receives as arguments the type of sensor (for data acquisition), the storage filename (for data-logging) and whether it will be executed periodically (time-driven) or at an event. It is partitioned in the following states:

- State1: Reads a value from a sensor
- State2: Stores the value
- State3: Compares the reading with the threshold value
- State4: Execute an "FPGA Configuration" Task if not already activated
- State6: Reschedule

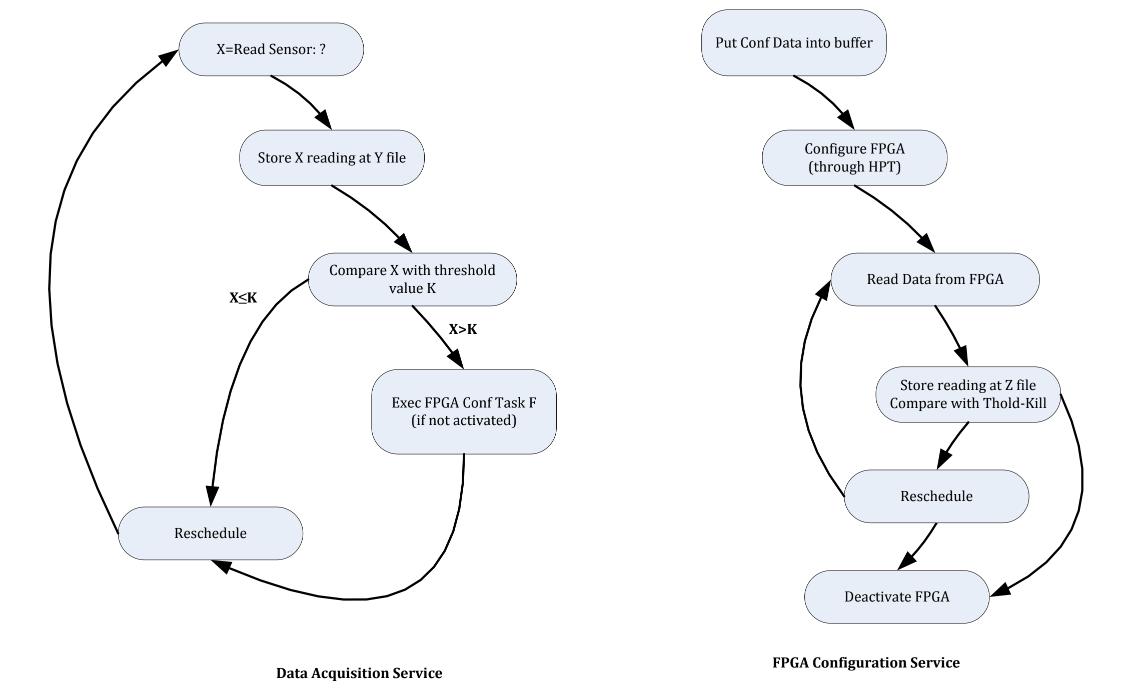

X=Read Sensor: ?
Store X reading at Y file
Compare X with threshold value K
X≤K
X>K
Exec FPGA Conf Task F (if not activated)
Reschedule
Data Acquisition Service
Put Conf Data into buffer
Configure FPGA (through HPT)
Read Data from FPGA
Store reading at Z file Compare with Thold-Kill
Reschedule
Deactivate FPGA
FPGA Configuration Service

Figure 18: State machine kernel services

The "**FPGA Configuration**" service is responsible for controlling an FPGA. Basically it controls the configuration process, the data acquisition process and finally its deactivation. It receives as arguments the configuration files needed for programming an FPGA (ptl script and bitstream), the storage filename (for data-logging) and the event (or the time) for deactivating the FPGA. It is partitioned in the following states:

- State1: Read configuration files
- State2: Initiate the FPGA configuration process based on the acquired configuration files
- State3: FPGA is activated. Reads a value from the FPGA through the uC-FPGA communication i/f (UART)
- State4: Stores the value and compare it with a predefined value. This value is used in order to decide if the FPGA has finished its operation and the service must proceed to the FPGA deactivation process (State 6) or continue reading values from it (State 3).
- State5: Go to State3 in case of an event (interrupt from uart)
- State6: Deactivate (reset) the FPGA and terminate.

5.6 Task Definition - Syntax

A Task is an executable instance of a service with user defined parameter values. At run-time, any task may create or terminate (kill) another instance of itself or any other task. All tasks are executed:

- At fixed or variable time intervals,
- at a specific date and time,
- upon an event (interrupt, serial I/O, internal signal from another task)

Tasks are formed by the combination of the system's services and the user defined parameters which are stored in a "task-ini" text file in the storage unit. During the systems' initialization phase the "task-ini" file is read by the parser module. According to the values that it reads, it forms and queues tasks based on the "data acquisition" and the "FPGA configuration" services.

All values stored in the "task-ini" file have a simple syntax with name/value pairs. The name must be unique (per task) and the value must fit on a single line. A Task is a name between square brackets, like "[Task_A]" in the example below. The task parameters and their values are separated by an equal sign ("=").

Parameter Name	Definition	Values
Task Name	Unique value which defines the name of the Task. It is placed between brackets.	[TASK-A]
Sensor Type(*)	Defines if the new Task will be based on the "data acquisition" or the "FPGA conf" service.	0: data acquisition, 1: FPGA conf
Sensor Resource(*)	Defines the device driver for accessing the appropriate sensor.	pH, flow, temp etc
Store filename	Name of the text file for storing acquired values from the sensor	Filename.txt (if null, no storage)
Threshold(*)	Value for comparing all incoming data with.	Constant value (if null, no comparison)
Activate-TASK (*)	If the acquired value is greater than the threshold value, activate a task.	<Task-name> (if null, no activation)

Parameter Name	Definition	Values
Schedule (ReSCH)	Declares if the Task is going to be rescheduled	0: no, 1: yes
Sched-type	Defines the type of task scheduling followed by a value indicating the time (days, hours, minutes, seconds) or the ISR (i.e. uart)	0: time intervals, 1: at a specific time, 2: upon an event (ISR,flag)
BITSTR(**)	Name of the bitstream for the FPGA configuration	<filename>
FPGA-Priority(**)	Priority value for FPGA arbitration buffer ordering	higher value = high priority
Thold-Kill (**)	Special data character received from UART upon which the FPGA must be deactivated (like threshold)(***)	Constant value. (if null, FPGA runs infinitely)
(*): only for tasks based on the "data acquisition" services (**): only for tasks based on the "FPGA configuration" services (***): In order to distinguish the "Thold-kill" value received from uart from the data values, a 9 bit protocol is implemented. During data exchange the 9th bit is always '0'. When the FPGA wants to inform the task that it has finished execution, it transmits the "Thold-Kill" byte and the 9th bit is set to '1'.		

Table 2: User – defined task values/parameters

In the following example there are two task scripts. The first script refers to a task based on the data-acquisition service which reads data from a flow meter and saves them to a file named "afile.txt". If the reading from the flow sensor exceeds the threshold value of "40", "Task_F" is activated. Finally the task is rescheduled to be executed every 1 minute. The second script refers to a task based on the "fpga-configuration" service, as indicated by the "Type" value. It configures the FPGA with the bitstream file "fpga.bit" based on the configuration process indicated in the ptl-script file "ptl.txt". After the FPGA is enabled, "Task_F" is activated upon interrupt from the UART connecting the uC with the FPGA, for data retrieval. All data received from the FPGA are stored in the "filez.txt" file. In order for the task to terminate its execution, the FPGA must inform Task_F that it has finished its application. This is done by sending the "Thold-Kill" byte as described in the previous parameter definition table.

At this point it is important to mention that the designer of the FPGA application must take into account the data exchange and the communication interface between the microcontroller and the FPGA when creating a new application

“Data acquisition” Task	“FPGA configuration” Task
[Task_A] **Type** = 0 **SENSOR** = flow **STORE** = afile.txt **THOLD** = 40 **Act-TASK=** Task_F **ReSCH** = 1 *#Reschedule at 1 minute* **Sched-type** = 0,0,0,1,0	**[Task_F]** **Type** = 1 **BITSTR**: fpga.bit **STORE** = filez.txt **Thold-Kill** = 5 *#Reschedule at 1 minute* **ReSCH** = 1 *#Reschedule at uart event* **Sched-type** = 2, uart **FPGA_PR**=3

Table 3: Examples of user defined parameters in “ini-task” file

5.7 Task Allocation - Placement

The system’s task scheduler architecture is depicted in the following figure. When the system enters its initialization phase, the parser module reads the “Task Ini” file, located in the Storage Unit. This file contains all the user defined parameters for the services that need to be executed during the system’s operation.

- Whether the task will be executed periodically (time-driven) or under specific circumstances (event-driven) and their values,
- The libraries (resources) it will utilize (sensors, storage area, etc)
- The location of the configuration file (bitstream) in the **RL Configuration memory**, the configuration commands and the configuration mode by which the FPGA will be programmed (if we are referring to a task being executed at the FGPA)
- The storage file in the **Data Memory**

The “Task Allocation” Module places the newly formed tasks in the scheduler’s queues. There are two main task queues. The “Time Wait” and the “Event Wait” Task queue. If the “Sched-type” value of a Task is ‘0’ or ‘1’ (meaning time – driven task), it is placed in the “Time Wait” Task Queue. If the value is 2 (indicating an event driven task) it is placed in “Event-Wait” Task Queue. So, at first all tasks are placed in the scheduler’s queues. The tasks that are placed in the “Time Wait” Task Queue are ordered by date and time with the youngest first. The tasks that are placed in the “Event Wait” queue are also ordered chronologically. When an event (interrupt or flag from another task) for which a task in the event-queued task is waiting for, arrives, it is placed by the Task Dispatcher module in “pending event” task queue. The “pending event” task queue holds all the tasks that need to be placed on top of the “Time Wait” Queue for immediate execution.

As mentioned in the previous section, a task is decomposed into states which are executed independently in time slices. When a task’s state has finished execution, the task allocation module reads the task’s “Sched-type” and “ReSCH” values and decides whether to place in one of the queues or discard it. This is the case when a task is only executed once (ReSCH=’0’).

If an “FPGA Configuration” task is ready to be executed for the first time, it checks to see if the “FPGA Configuration” Queue is empty. The “FPGA Configuration” queue holds the user defined information about the FPGA configuration process.

If the queue is empty, it places all configuration user parameters (priority, bitstream filename) in this queue and activates the FPGA configuration procedure. If the “FPGA Configuration” Queue is not empty, it means that there is another TASK waiting to take control of the FPGA, or the FPGA is used by another Task. In this case the configuration user parameters are placed in the “FPGA Configuration” Queue and the Task is queued in the “Event Wait” queue waiting for the release of the FPGA Queue. All data in this queue are ordered based on the user appointed priority value.

An issue that must be taken into account while designing the task execution scripts is the priority levels of the “FPGA conf” tasks. In order to avoid “starvation” conditions in situations where many “FPGA configuration” tasks are queued inside the FPGA Configuration queue, varying priority levels must be kept to a minimum.

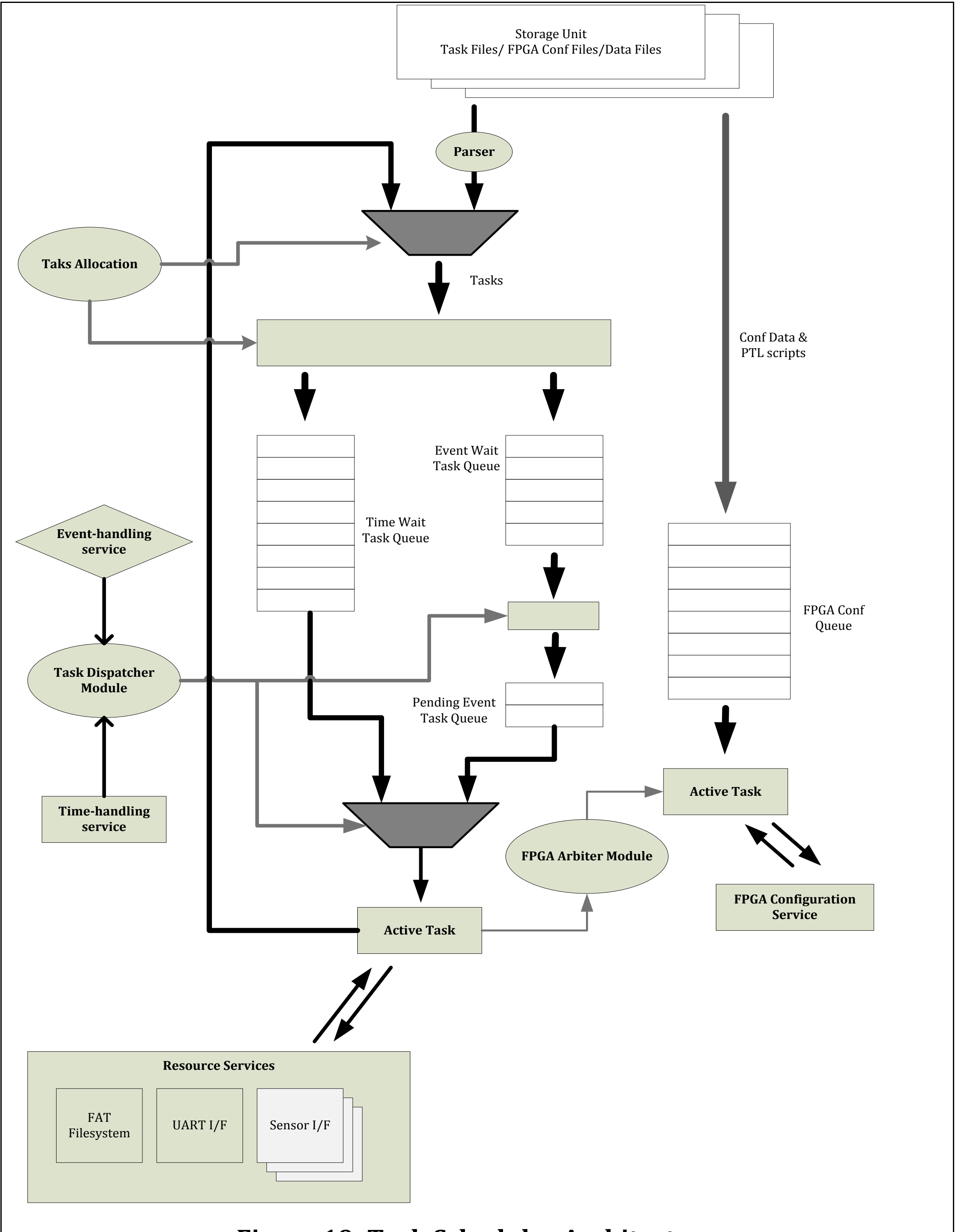


Figure 19: Task Scheduler Architecture

5.8 Task Execution Example

In the following figure there is an example of a task execution scenario.

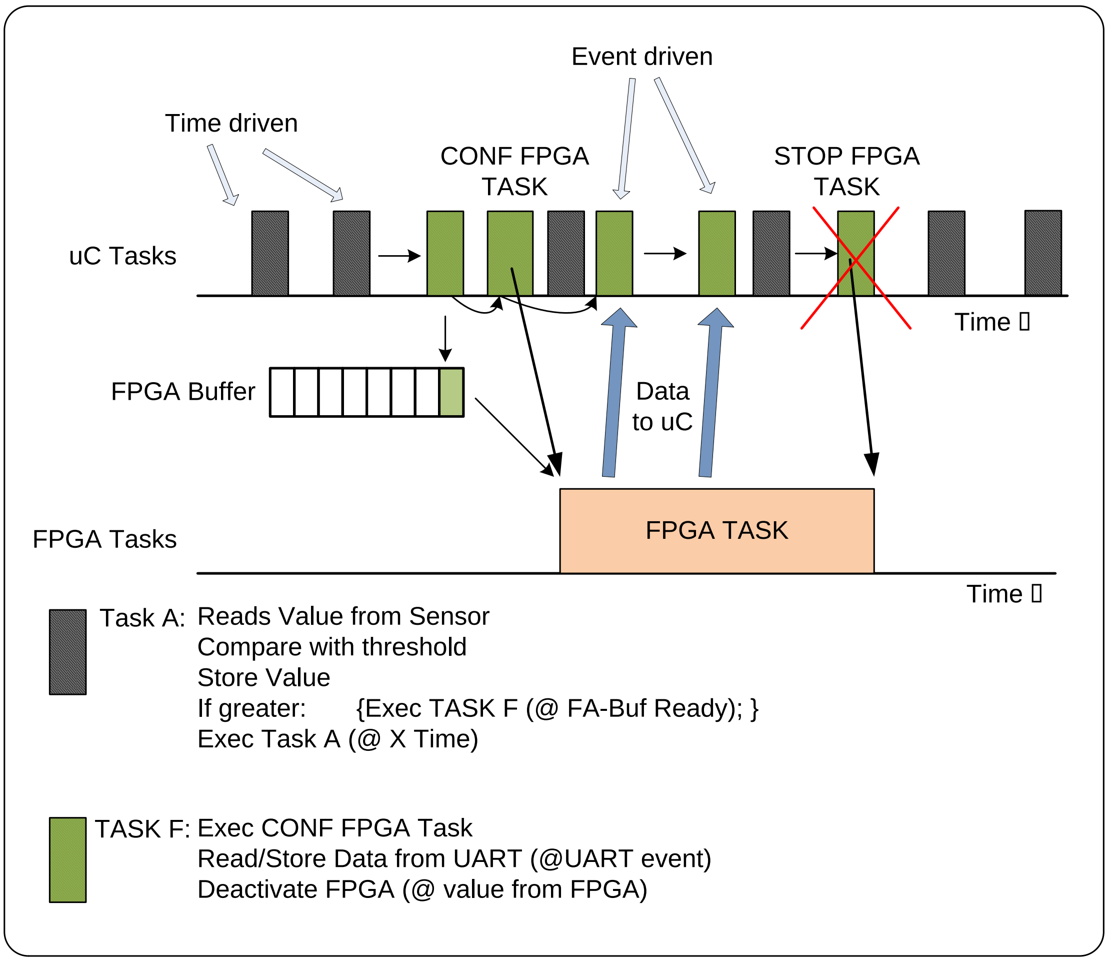

Event driven
Time driven
CONF FPGA TASK
STOP FPGA TASK
uC Tasks
Time
FPGA Buffer
Data to uC
FPGA Tasks
FPGA TASK
Time
Task A: Reads Value from Sensor
Compare with threshold
Store Value
If greater: {Exec TASK F (@ FA-Buf Ready); }
Exec Task A (@ X Time)
TASK F: Exec CONF FPGA Task
Read/Store Data from UART (@UART event)
Deactivate FPGA (@ value from FPGA)

Figure 20: Task Allocation and Scheduling Example

TASK A's states are executed at specific time intervals.

1. In the first step, it reads a value from a sensor by calling the appropriate sensor library.
2. It stores the value in a predefined file in the memory unit, along with the current timestamp
3. Compares the value with a predefined threshold value.
    - If the value exceeds the threshold, it calls Task F
    - If the reading is lower than the threshold, Reissues itself for execution at the next interval

TASK F is defined with the following steps:

1. Puts all information regarding the FPGA configuration process (priority, bitstream) in the “FPGA Configuration” Queue. This queue holds all pending FPGA configuration requests. If the queue is empty, it commences the FPGA configuration process else “TASK F” is placed in the pending queue and waits for the release of the FPGA Configuration Queue.
2. If the FPGA is configured, “TASK F” enters the “datalogging state” where it reads data from the FPGA (through the uart), stores them
3. Additionally it compares the newly arrived data with the “Thold-Kill” value in case the FPGA has issued a termination signal. In that case, TASK F released the FPGA Configuration Queue, deactivates the FPGA and issues a “sig_kill” for itself

Chapter 6
Reconfigurable Co-Processor Architecture and Example Implementation

6.1 Introduction

This chapter refers to the architecture of the system's reconfigurable co-processor. We discuss the design considerations taken into account for implementing the proposed architecture and we conclude by illustrating an implementation example with a Spartan 3 FPGA,.

6.2 Co-processor Design Considerations

As mentioned earlier the proposed environmental monitoring platform must be scalable and adaptive to the application and the monitoring requirements for which it is being deployed.

Since the FPGA plays a vital role in terms in the platform's overall processing capabilities, from the processing layer perspective, it is very important to be able to choose from a variety of FPGAs with different capabilities as the platform's co-processing unit, without the need for full system redesign, in terms of software and hardware.

Our platform is technology and vendor independent of the incorporated reconfigurable logic (FPGA). Every SRAM-based FPGA currently available in the market can be incorporated as a co-processor without the need for redesign due to:

a) The FPGA configuration mechanism by the microcontroller
b) The FPGA communication interface with the microcontroller

6.3 Co-processor Architecture

The FPGA (co-processor) is coupled to the microcontroller through its system I/O bus as shown in the following figure. The FPGA can only access the sensors directly attached to it and the storage unit through the microcontroller.

Although the FPGA is part of the "processing" layer, the microcontroller only sees the FPGA as a source of data to be collected. In other words, from a behavioral point of view, the FPGA is part of the "Sensor" Layer.

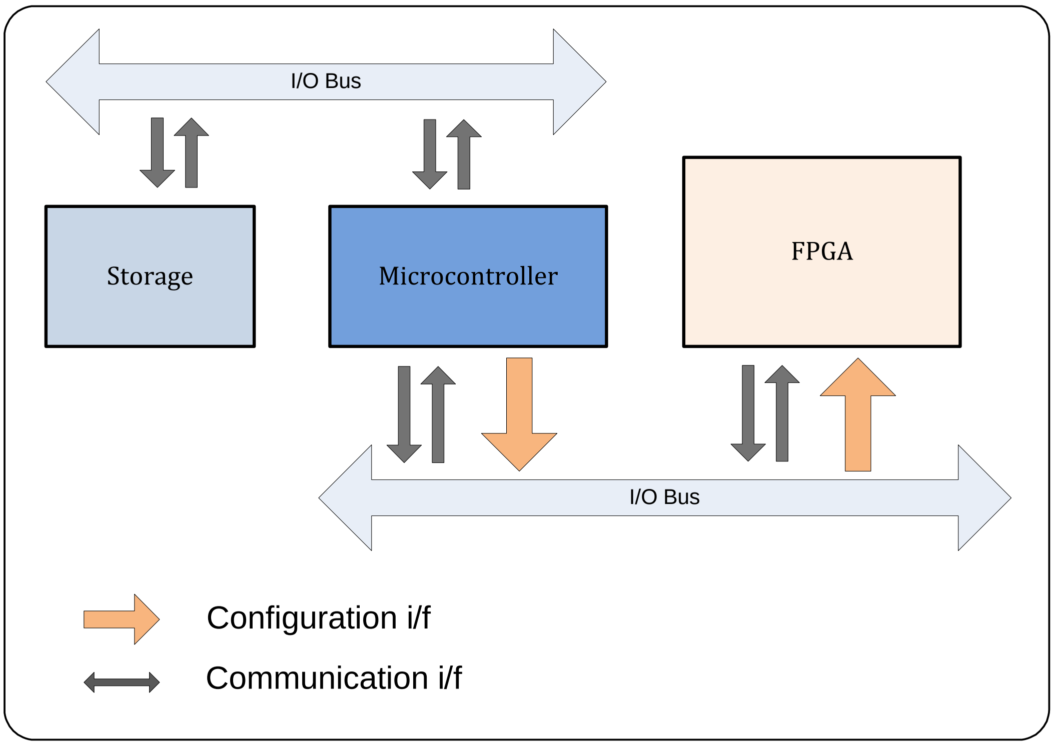


Figure 21: FPGA – uC coupling

When the FPGA is enabled and for its entire lifecycle, it remains completely autonomous from the microcontroller. The communication bus between the two remains idle until the FPGA is ready to transmit data to the microcontroller. The corresponding sw-task running on the uC, which is responsible for collecting data from the FPGA, is interrupt – driven. It remains inactive until data arrive in the FPGA-uC communication bus. It then collects the newly arrived data from the bus and stores them in the reserved storage space, indicated by the “parent” task that triggered the FPGA. It is then deactivated once more until the next interrupt, or until the FPGA has been deactivated, in which case it is also terminated.

6.3.1 FPGA’s Configuration Mechanism

The configuration mechanism of the FPGA is controlled by the reconfiguration module (HPT) integrated into the system, which is a vendor – independent universal programmer. It is a centralized configuration solution which can provide both the bit density and the control logic to manage configuration for all FPGAs within a system.

The configuration bitstream along with the accompanied PTL instructions that indicate the FPGA’s configuration process, are retrieved from the storage area designated Task that issued the FPGA configuration.

6.4 Platform Implementation

The prototype platform has been implemented based on the previously mentioned concepts. It is shown in the following figure. It includes a low-power Atmel 8-bit AVR RISC-based microcontroller (ATmega644) [27]. The ATmega644 was chosen because it contains , 64KB ISP flash memory with read-while-write capabilities, 2 USARTs, byte oriented 2-wire serial interface, 8-channel/10-bit A/D converter with optional differential input stage with programmable gain, SPI serial port and 6 software selectable power saving modes. The platform’s co-processor has been realized by a commercial low cost development board, the Digilent Spartan-3 Board [28,29]. It contains a Xilinx SPARTAN-3 FPGA (XC3S1000) [30], an RS232 transceiver, LEDs, buttons and switches. The reason for selecting the specific FPGA was cost, available literature and configuration diversity. The XC3S1000 FPGA contains 1M system gates (~2K CLBs) and 391 user I/Os. The platform also comprises a SD card as the storage unit and other peripheral components.

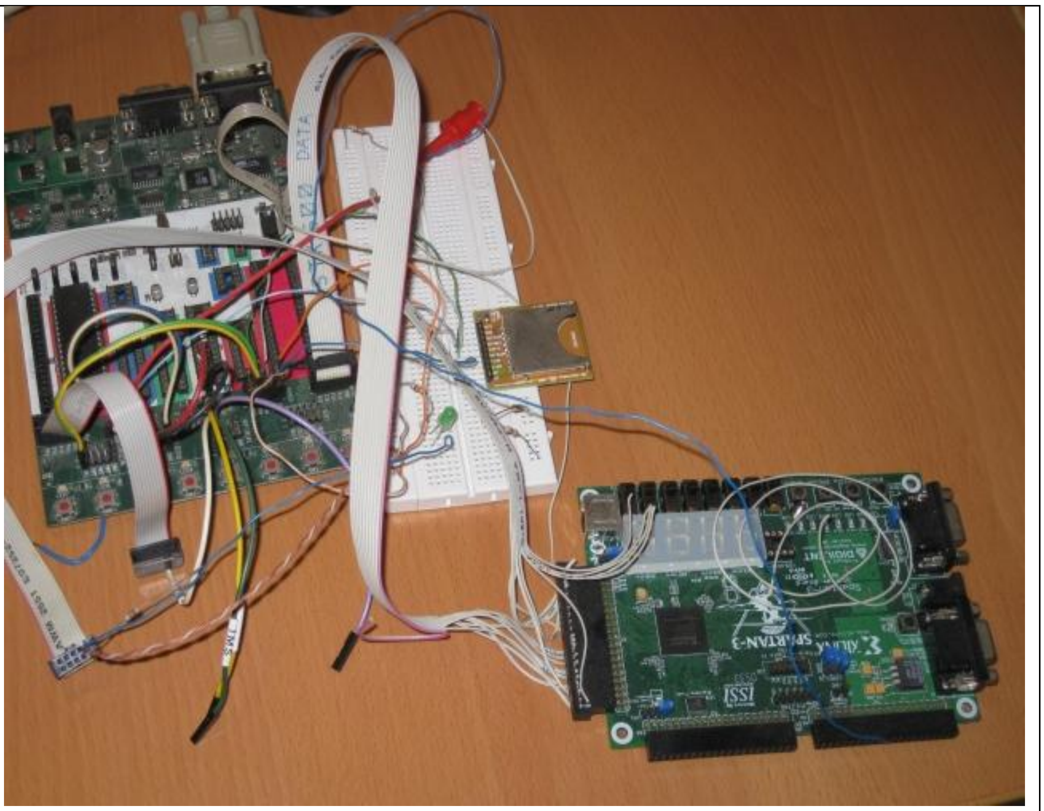


Figure 22: Platform's schematic view

6.4.1 Platform's pinout

The platform is connected with the communication layer through one of the microcontroller's a serial ports. The other serial port is used to communicate with the FPGA board. The implementation of a serial interface between the reconfigurable section, the communication layer and the microcontroller offers a unified communication medium. In this way the architecture and the interface of the platform remains unaffected and independent of the hardware characteristics of the reconfigurable section or the communication layer. Another reason is the need for data synchronization. For example, the reconfigurable section (FPGA) can be driven by a 50MHz system clock. On the other hand, the 8bit microcontroller is driven by a much slower system clock (~12MHz). In this case, asynchronous serial communication is a simple and ideal communication interface to overcome the synchronization problems imposed in a system where each layer is driven from a different clock source.

Platform's Interfaces			
Pin Name	Type	Pin Count	Function
HPT to FPGA (PROG I/F)			
CONF CLK	Output	1	Configuration clock connected to the target board.
CNTRL_PORT	Bidir	8	Port for controlling the configuration (or test process).
DATA_PORT	Bidir	8	Port for sending configuration or test data to the target board
Memory Module Interface (FLASH I/F)			
MOSI	Input	1	Serial Data Input Signal
MISO	Output	1	Serial Data Output Signal
SS	Output	1	Slave Select
SCK	Output	1	Serial Clock
uC to Communication Layer (RS232 I/F)			
RxD	Bidir	1	Receive data pin
TxD	Bidir	1	Transmit data pin
uC to FPGA (RS232 I/F)			
RxD	Bidir	1	Receive data pin
TxD	Bidir	1	Transmit data pin
uC to Sensing Layer			
SENSOR_PORT	Bidir	8	Port for communicating with the sensors attached to the uC
uC to Sensing Layer			
FPGA_SENSOR PORT(s)	Bidir	(*)	Port(s) for communicating with the sensors attached to the FPGA.
(*) Depends on the application and the configuration of the FPGA			

Table 4: Platform's Interfaces

Chapter 7
System Validation and Performance Evaluation

7.1 Introduction

For testing purposes, the platform was partitioned into two main sections. In the first section the Spartan3 FPGA has been successfully configured by the platform's HPT module through Slave Serial configuration mode. This mode provides the advantage of utilizing only 5 pins compared with the 12 pins used in the SelectMAP mode, thus freeing valuable microcontroller's pins. This of course comes with the cost of extra configuration delay, since the entire configuration file downloaded in the FPGA is serialized. In the other section, the system's overall behavior has been tested with a primary focus on task operations and responses to external events.

7.2 FPGA Configuration

Spartan3 devices are configured, like most SRAM based FPGAs, by loading application – specific configuration data into internal memory. The bitstream is organized into 32-bit words. These words carry instructions for the configuration logic, as well as data that will be stored in the configuration memory. Configuration is carried out by using a subset of the device pins, some of which are dedicated, while others can be reused as general – purpose inputs and outputs after configuration is completed. The FPGA's configuration memory can be programmed in different ways by using either serial or parallel data. Additionally by having an external device like the microcontroller supplying the clock signal along with the data to the FPGA, it can be configured in what is known as "Slave Serial" or "Slave Parallel" modes. The mode is specified by setting values on the Spartan-3 mode pins (M0,M1 and M2).

Several of the Spartan3 configuration modes are selectable via mode pins. The mode pins M0, M1, M2 are dedicated pins. Other dedicated pins are:

- CCLK: the configuration clock pin
- DONE: configuration status pin
- PROG_B: configuration reset pin

Device	*Total Number of Configuration Bits (including header)*
XC3S50	439,264
XC3S200	1,047,616
XC3S400	1,699,136
XC3S1000	3,223,488
XC3S1500	5,214,784
XC3S2000	7,673,024
XC3S4000	11,316,864
XC3S5000	13,271,936

Table 5: SPARTAN3 Bit – Stream Lengths.

Configuration Mode	***M2***	***M1***	***M0***	***CCLK Direction***	***Data Width***	***Daisy Chain***
Master Serial	0	0	0	Out	1	Yes
Slave Serial	1	1	1	In	1	Yes
Master SelectMAP	0	1	1	Out	8	No
Slave SelectMAP	1	1	0	In	8	No
Boundary Scan	1	0	1	N/A	1	No

Table 6: Spartan3 Configuration mode pin settings.

7.2.1 3.3V Tolerant configuration interface

The connection of the microcontroller board to the FPGA required some attention since it connects a board with 2.5V logic to one with 3.3V logic like the STK500 board with the ATMega644. This is mainly because the DONE and PROG_B pins are powered by the FPGA's 2.5V VCCAUX supply. A Xilinx application note describes this interface where series of resistors have to be inserted on the dedicated configuration pins to account for the voltage drop [31].

The following figure shows the required connections for slave-serial configuration.

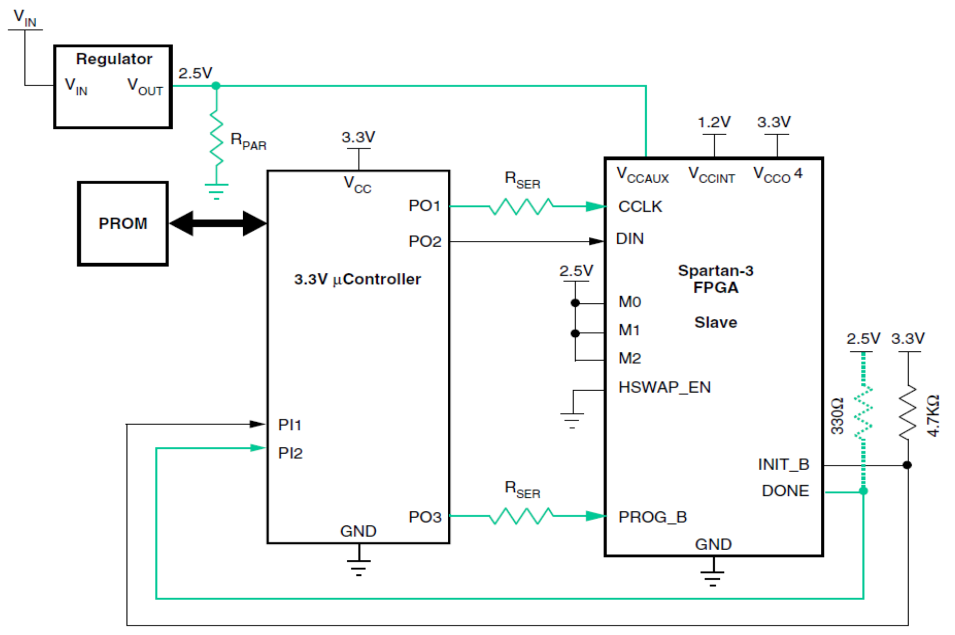


Figure 23: 3.3V configuration of a Spartan-3 device in slave-serial mode

7.2.2 Spartan3 Slave Serial Configuration

In serial configuration mode, the FPGA is configured by loading one bit per CCLK cycle. The FPGA's CCLK pin is driven by an external source. In this case, by the microcontroller's CONF_CLK pin. The MSB of each data byte is always written to the DIN pin first.

There are four major phases in the configuration process:

1. Clearing Configuration Memory
2. Initialization
3. Loading Configuration Data
4. Startup

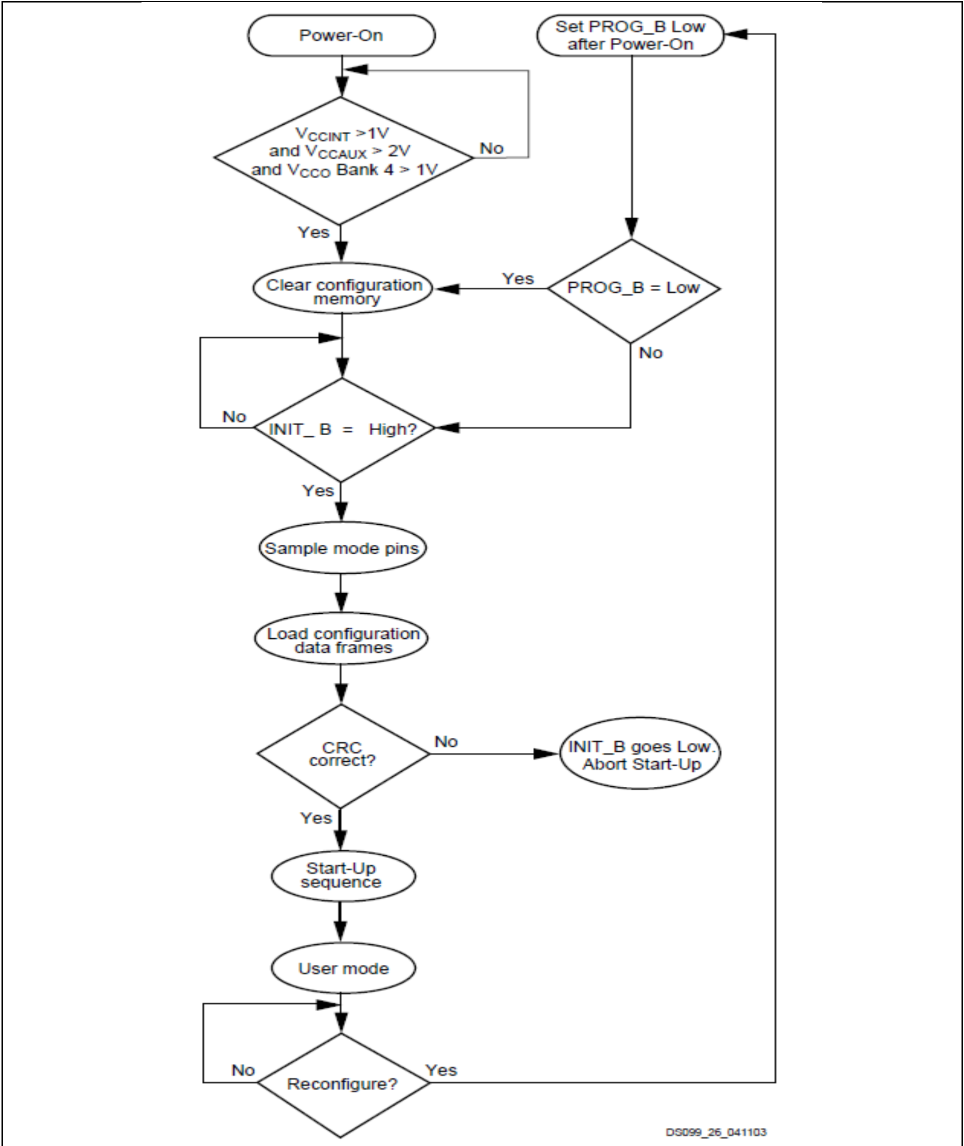

Power-On
Set PROG_B Low after Power-On
V_{CCINT} >1V and V_{CCAUX} > 2V and V_{CCO} Bank 4 > 1V
No
Yes
Clear configuration memory
Yes
PROG_B = Low
No
No
INIT_ B = High?
Yes
Sample mode pins
Load configuration data frames
CRC correct?
No
INIT_B goes Low. Abort Start-Up
Yes
Start-Up sequence
User mode
No
Yes
Reconfigure?
DS099_26_041103

Figure 24: Configuration Flow Diagram for the Serial Mode

In order for the device to enter the “Clearing Configuration Memory” phase, the microcontroller must issue a HIGH to LOW to HIGH pattern in the PROG_B pin thus resetting the device. The minimum time the microcontroller holds the PROG_B pin low, is determined by the device’s datasheet and for the current device is approximately 300 ns. The INIT_B pin of the FPGA transitions HIGH when the clearing of the configuration memory is complete. For the XC3S1000 device, this time is 3ms.

7.2.3 PTL Scripts

The following script was used as a “global” PTL script in order for the platform to be able to (re)configure the attached SPARTAN3 FPGA

```
//PTL script for the configuration of
//a XC3S1000 device (3,223,488 bits = 402,936 Bytes)

program "serial";            // Set the programming mode to Serial

msb;                         // The MSB will be loaded first

clk high;                    // The configuration clock has a
                             // minimum low time

signal prog_b,done,init_b;// Three signals will be used

 map                         // Signal Mapping
{
done <= 0;                   // done is an input and connected to
                             // CNTRL_PORT pin0 of the platform
prog_b => 1;                 // prog_b is an output and connected to
                             // CNTRL_PORT pin1 of the platform
 init_b <= 2;                // init_b is an input and is connected
                             // CNTRL_PORT pin2 of the platform
}

start                        //Start of script
set prog_b '1';              // Generate a high-low-high
set prog_b '0';              // pulse on the prog_b
set prog_b '1';              // signal

 wait 148                    //Wait 148 msec until init_b
                             // goes HIGH

 get 1                       //Retrieve the Control values
                             //to see that INIT_B is indeed HIGH
                             // Load 402 Kbytes + 936 bytes
 loadb 255;                  // Load   255 bytes
 get 1                       // Get back values to see if
                             // INIT_B remains HIGH (if not error)
 loadb 255;                  // Load   255 bytes
 get 1                       // Get back values to see if
                             // INIT_B remains HIGH (if not error)
 loadb 255;                  // Load   255 bytes
 get 1                       // Get back values to see if
                             // INIT_B remains HIGH (if not error)
 loadb 171;                  // Load   171 bytes
 get 1                       // Get back values to see if
                             // INIT_B remains HIGH (if not error)
loadkb 255;                  // Load   255 Kbytes

```

```
48      loadbk 147;             // Load   147 Kbytes
49                      // This is the size of the bit file
50                      //
51
52      get 1;          // Get back the values to see that
53                      // DONE is HIGH (if so conf = SUCCESS)
53 end
```

In the above PTL script the configuration mode pins M0, M1, M2 are assumed to be pre – defined by the user on the board. If not, then the PTL script must be enriched with the following instructions at line 12:

```
static M0 '1';          // Set the mode pins to
static M1 '1';          // 111 to enable Slave
static M2 '1';          // Serial configuration mode
```

and the following instructions below line 13 in the pin mapping section:

```
M0 => 3;                // The mode pins are statics and
M1 => 4;                // can only be mapped as
M2 => 5;                // outputs
```

7.3 SW Validation

In terms of system operation and validation, a number of simulation and testing scenarios have been tested. Since we refer to a prototype platform tested in a lab, all environmental readings were emulated by a series of input files from the SD card. In all of the tests, the system operated with as expected

Furthermore, in one case, real environmental data have been used in order to test the system's overall behavior. Hydrologic data (level and Temp) gathered from an environmental monitoring station deployed in the Koiliaris River were used as validation input to the platform. The imported time series are from data collected from a 2-year period with a constant sampling rate of 5 measurements per hour (Total: ~70000 measurements).

The simulation scenario referred to two independent tasks reading 1 measurement each every 0.5 secs, in order to shorten the simulation time. The values from the task which read the water level values were compared with a predefined threshold value. In this scenario, if this value was exceeded an FPGA Configuration command would be issued, indicating that an extreme event was occurring. The entire simulation period lasted 10hours during which time the platform issued 6 FPGA configuration commands. This matches exactly with the number of observed flood events as depicted in the following figure.

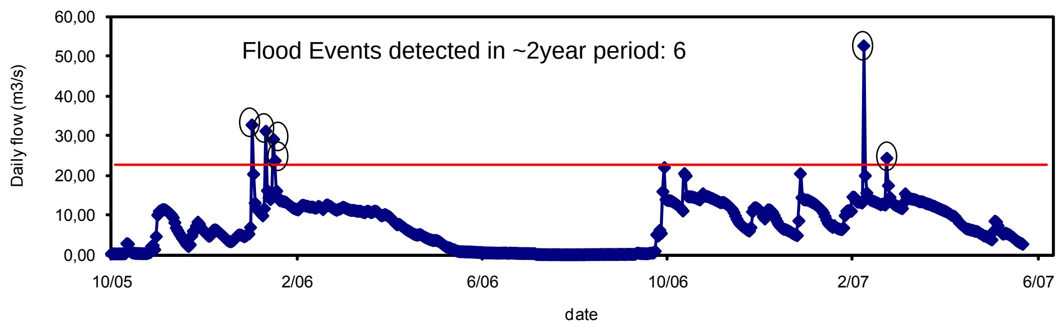


Figure 25: Koiliaris River Flood Events Counts

7.3.1 Accompanied Task-INI Script

In one of the simulation and testing scenarios the Task-Ini file contained the following parameters.

```
#“Data acquisition” Task
[level-LOG]
Type = 0
SENSOR = level
STORE = levelfile.txt
THOLD = 400
Act-TASK= Video
ReSCH = 1
#Reschedule at 5 minutes
Sched-type = 0,0,0,5,0


#“FPGA configuration” Task
[Video]
Type = 1
BITSTR: video.bit
STORE = filev.txt
Thold-Kill = 5
#Reschedule at 1 minute
ReSCH = 1
#Reschedule at uart even
 Sched-type = 2, uart
 FPGA_PR=3


```

```
#"Data acquisition" Task
[PH-LOG]
 Type = 0
SENSOR = ph
STORE = phfile.txt
THOLD = 8
Act-TASK=
ReSCH = 1
#Reschedule every 2 minutes
Sched-type = 0,0,0,2,0
```

In this scenario, there are three user defined tasks, “PH-LOG”, “level-LOG” and “Video”. It must be pointed out at this point, that the values and parameters in this task-ini file are completely arbitrary and do not refer to real–life environmental parameters.

As indicated by their values in the Type fields, the tasks “PH-LOG” and “level-LOG” are Data Acquisition Task and are executed solely on the microcontroller. “level-LOG” Task is rescheduled every 5 minutes and at that time, it takes a reading from the attached flow sensor and stores it along with the current timestamp in the “levelfile.txt” inside the SD Card. Furthermore, this task compares the flow reading (value in centimeters) with the threshold value of 400 and if it is greater, it activates “Video” Task. With this task, we wanted to describe that if there is a rise in the river level which reaches 4 meters, then we are referring to a flood event and we enable the FPGA for further processing and data gathering.

“Video” Task is actually a demo FPGA configuration task, which is triggered by the “PH-LOG” task. As mentioned earlier, the system validation was performed in two distinct sections. In this section, the FPGA configuration process has not been fully tested. As a sign though that the “PH-LOG” task triggers the FPGA Task properly, the “Video” task instead of initiating the FPGA conf process, it writes to a log file (in the SD) the time that it started executing.

The “PH-LOG” task is a separate data-acquisition task that doesn’t affect the FPGA operation. It takes a reading from a ph sensor every 2 minutes and stores it in a file inside the SD card

Chapter 8
Conclusions – Future Work

8.1 Conclusions

This document presented the development of a dynamically reconfigurable system for environmental monitoring. The platform's main objective is to serve a variety of tasks ranging from simple data logging to highly computationally intensive ones. This is accomplished through the dynamic reservation of the processing resources that the reconfigurable co-processor offers only when needed.

The motivation for the current work and the issues and constraints that this system has to address were investigated in order to derive guidelines for the system's hardware and software design. One of the main design goals was the implementation of a dynamically reconfigurable platform both in terms of software and hardware. Additionally the system's layered – modular design allows the use of every SRAM-based FPGA currently available in the market as a co-processor without the need for redesign. This is also achieved because the FPGA configuration mechanism is controlled by the HPT module integrated into the system, which is a vendor – independent universal programmer.

8.2 Future Case Study

Experimental setup for image acquisition and processing tasks by the FPGA is not implemented yet. An ideal real – life application would be the analysis of video captured from the river in order to calculate the water's velocity vectors and river flow rates during a flood event.

One of the major advantages of introducing complex in-situ data acquisition and analysis capabilities in a "water quality" monitoring system, such as automated motion estimation and river-flow calculation is that the volume of data transmitted to the network will be greatly reduced.

8.3 Future Works

As mentioned in Chapter 5, the current FPGA configuration algorithm is based on the FCFS (first-come, first-serve) policy. Despite the fact that all queued tasks are prioritized, an active FPGA Task cannot be replaced by a queued one with higher priority. In rare but critical conditions this could lead to task “starvation” and ultimately in loosing valuable data. This can be resolved through FPGA task (hw task) preemption. On preemption the state of the task should be saved either by readback (if the FPGA supports it) or through some other method. The current state of the FPGA would be saved along with the accompanied software tasks running on the microcontroller and the FPGA would be configured with a new configuration. After that, the FPGA would be reconfigured with the configuration data retrieved through the readback process. It resembles the PUSH, POP instructions of an interrupt driven routine in a microcontroller.

For a hardware technology point of view, the major restriction of having only 8 pins for the sensor interface can be alleviated by deploying a microcontroller with more I/O pins, or by tripling its I/O capabilities with a 8255, Programmable Peripheral Interface, chip.

References

[1] M. Glesner, T. Hollstein, L. S. Indrusiak, et al., “Reconfigurable platforms for ubiquitous computing”, Conf. Computing Frontiers 2004: 377-389.

[2] C. Plessl, R. Enzler, H. Walder, et al.,“Reconfigurable Hardware in Wearable Computing”, Nodes. ISWC 2002: 215-222.

[3] J. Feng, F. Koushanfar, M. Potkonjak, "System-architectures for sensor networks issues, alternatives, and directions," Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on , vol., no., pp. 226- 231, 2002

[4] D. Moraetis, D. Efstathiou, F. Stamati, O. Tzoraki, N. P. Nikolaidis, J. L. Schnoor, K. Vozinakis, “High-frequency monitoring for the identification of hydrological and bio-geochemical processes in a Mediterranean river basin”, Journal of Hydrology, Volume 389, Issues 1-2, Pages 127-136, 28 July 2010.

[5] Xilinx Inc: http://www.xilinx.com

[6] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, “Reconfigurable computing: architectures and design methods,” in IEE Proceedings: Computer & Digital Techniques, vol. 152, no. 2, pp. 193–208, March 2005.

[7] K. Compton and S. Hauck, “Reconfigurable computing: a survey of systems and software,” ACM Computing Surveys, vol. 34, no. 2, pp. 171–210, 2002.

[8] B. Fletcher: “FPGA Embedded Processors,” in Embedded Systems Conference San Francisco, CA, p. 18, 2005.

[9] Altera Corporation, http://www.altera.com

[10] Atmel Corporation, http://atmel.com

[11] D. Lymberopoulos, N. Priyantha, and F. Zhao, “mPlatform: A reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes,” Information Processing in Sensor Networks, 2007, IPSN 2007, 6th International Symposium on, pp. 128–137, April 2007.

[12] Y.E. Krasteva, J. Portilla, J.M. Carnicer, E. de la Torre, T. Riesgo: “Remote HW-SW reconfigurable Wireless Sensor nodes” in 34th Annual Conference of IEEE, Industrial Electronics, IECON 2008, p. 2483 – 2488, Orlando, FL 2008.

[13] A. Brokalakis, G-G. Mplemenos, K. Papadopoulos and I. Papaefstathiou, "RESENSE: An Innovative, Reconfigurable, Powerful and Energy Efficient WSN Node", IEEE International Conference on Communications (ICC 2011 Adhoc, Sensor and Mesh Networking Symposium), 5-9 June 2011, Kyoto, Japan.

[14] S. J. Bellis, K. Delaney, B. O'Flynn, J. Barton, K. M. Razeeb, C. O'Mathuna, “Development of field programmable modular wireless sensor network nodes for ambient systems”, Computer Communications, Volume 28, Issue 13, Wireless Sensor Networks and Applications - Proceedings of the Dagstuhl Seminar 04122., 2 August 2005, Pages 1531-1544.

[15] J.L. Wilder, V. Uzelac, A. Milenkovic, E. Jovanov, "Runtime Hardware Reconfiguration in Wireless Sensor Networks," System Theory, 2008. SSST 2008. 40th Southeastern Symposium on , vol., no., pp.154-158, 16-18 March 2008

[16] H. Hinkelmann, A. Reinhardt, S. Varyani, and M. Glesner, "A reconfigurable prototyping platform for smart sensor networks," in Proceedings of the 4th Southern Conference on Programmable Logic (SPL '08), pp. 125–130, San Carlos de Bariloche, Argentina, March 2008.

[17] P. Muralidhar and C. B. R. Rao, "Reconfigurable Wireless sensor network node based on NIOS core," in Proceedings of the 4th International Conference on Wireless Communication and Sensor Networks (WCSN '08), pp. 67–72, Allahabad, India, December 2008

[18] P.Latha, M.A. Bhagyaveni, "Reconfigurable FPGA based architecture for surveillance systems in WSN," Wireless Communication and Sensor Computing, 2010. ICWCSC 2010. International Conference on , vol., no., pp.1-6, 2-4 Jan. 2010

[19] D. Efstathiou, K. Kazakos, A. Dollas, "Parrotfish: Task Distribution in a Low Cost Autonomous ad hoc Sensor Network through Dynamic Runtime Reconfiguration," fccm, pp.319-320, 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06), 2006

[20] H. Hinkelmann, A. Reinhardt, M. Glesner, "A Methodology for Wireless Sensor Network Prototyping with Sophisticated Debugging Support," Rapid System Prototyping, 2008. RSP '08. The 19th IEEE/IFIP International Symposium on , vol., no., pp.82-88, 2-5 June 2008

[21] J. Portilla, T. Riesgo, and A. de Castro, "A Reconfigurable FPGA-Based Architecture for Modular Nodes in Wireless Sensor Networks" In Proc. 3rd Southern Conference on Programmable Logic, pages 203–206, 2007.

[22] B. O'Flynn, S. Bellis, K. Delaney, J. Barton, S. C. O'Mathuna, A. M. Barroso, J. Benson, U. Roedig, and C. Sreenan, "The Development of a Novel Minaturized Modular Platform for Wireless Sensor Networks", In Proc. 4thInt. Symp. on Information Processing in Sensor Networks, pages 370–375, 2005.

[23] D.Efstathiou, Diploma Thesis, "Design and Implementation of a Vendor-Independent Universal Programmer for FPGA Technology", MHL, ECED Department, Technical University of Crete, Greece, July 2002.

[24] T.Kyriakides, Diploma Thesis, "Development of a Language and Universal Run – Time Environment for FPGA programming", MHL, ECED Department, Technical University of Crete, Greece, July 2002

[25] A. Dollas, D. Efstathiou, and T. Kyriakides, "A universal low cost runtime and programming environment for reconfigurable computing," in Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping (RSP'03), IEEE Computer Society, 2003.

[26] OPEX Scheduler v1.0 http://www.atmanecl.com/EnglishSite/opex.htm, Atman Electronics,

[27] Atmel Corporation, ATMega644 Datasheet: http://www.atmel.com/dyn/resources/prod_documents/doc2593.pdf

[28] S3 Board User Manual, “Spartan-3 Starter Kit Board User Guide”, available online at: http://www.digilentinc.com/Data/Products/S3BOARD/S3BOARD_RM.pdf, UG130 (v1.1) May 13, 2005

[29] Digilent Inc.: http://www.digilentinc.com/

[30] Xilinx Spartan 3 FPGA Family Datasheet.: http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf

[31] K. Goldblatt, “XAPP453 (v1.1.1): The 3.3V configuration of Spartan-3 FPGAs,” Xilinx,” Application note, June 2008.