POLITECNICO DI MILANO
Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

A REAL-TIME, RELIABLE AND REPROGRAMMABLE FRAMEWORK FOR HYBRID SENSOR NETWORKS

Relatore: Ing. Manuel Roveri
Correlatore: Ing. Romolo Camplani

Tesi di Laurea Specialistica di:
Luca VACCARO
Matr. 755789

Anno Accademico 2012-2013

to my little big love,

Contents

Bibliography **71**

List of Figures

List of Tables

Abstract

In questa tesi presentiamo la progettazione ed implementazione del framework REEL, un framework per reti di sensori orientato alla riprogrammazione remota. Il framework REEL si basa sul sistema operativo FreeRTOS, estendendone le funzionalità di base, fornendo in particolare dei servizi aggiuntivi per gestire la gestione remota delle applicazioni installate sul nodo di una rete di sensori.

La riprogrammazione remota del nodo consiste nell'iniettare un nuovo codice applicativo sul nodo a run-time, senza necessità di intervento diretto dell'operatore umano. Questa caratteristica permette, ad esempio, di cambiare il comportamento del nodo, la configurazione e gli algoritmi, direttamente dalla stazione di controllo, senza modificare il sistema di base del'OS. La politica di riprogrammazione a livello di applicazione intrapresa permette di caricare sul nodo una nuova applicazione, o sovrascrivere una esistente, senza effetti collaterali sulle applicazioni eseguite in modo concorrente. A questa precedura si affiancano servizi avanzati per fornire:

- contesto di esecuzione protetto delle applicazioni: permette di rilevare i software faults, notificare le eccezioni e bloccare le relative applicazioni difettose evitando il blocco dell'intero sistema
- monitoraggio remoto del sistema: rileva il livello di utilizzo del sistema, le applicazioni installate, la memoria disponibile, ed eventuali valori caratteristici quali il livello della batteria
- gestione remota delle applicazioni: consente il caricamento, l'esecuzione e l'arresto delle applicazioni senza riavvio della macchina
- procedura di riavvio: ripristina la configurazione iniziale del sistema e riavvia le applicazioni schedulate, portando il sistema in uno stato consistente.
- metodo di riprogrammazione sicuro: verifica l'integrita' e l'autenticita' delle applicazioni prima di effettuare l'installazione sul nodo

Non di minore importanza sono gli strumenti software forniti da REEL per la stazione di controllo:

- ambiente di sviluppo di alto livello per sviluppare nuove applicazioni basate sul REEL framework
- compilare ed rilocare il codice, linking statico, delle applicazioni in modo automatico
- gestire la manutenzione e la riprogrammazione dei nodi remoti

- effettuare il debug delle applicazioni
- strumenti per il monitoraggio delle attività su ciascun nodo e lo stato della rete
- interfaccia utente per monitorare ed interrogare il nodo remoto, noonchè caricare nuove applicazioni

Il tema della riprogrammazione remota e' abbastanza recente in letteratura e sono stati proposti numerosi approci. REEL si propone buon compromesso fra il consumo energetico e prestazioni, e puo' essere adattato per svariate tipologie di scenari. La politica di riprogrammazione si basa sul linking statico delle applicazioni nella stazione di controllo, la diffusione dell'immagine verso il nodo remoto, e la relativa esecuzione. Questa procedura permette da un lato di minimizzare le informazioni scambiate durante la comunicazione rispetto al linking dinamico; dall'altro lato, fornisce un'esecuzione nativa del codice macchina, senza introdurre overhead temporali tipici di soluzioni basate su macchina virtuale.

1. Introduction

The goal of this thesis is to design and implement REEL, an novel framework for Wireless Sensor Network. The REEL framework is based on the lightweight, real time, operating system FreeRTOS aiming to extends its basic functionalities, which are limited to the scheduler and the memory management. In particular, REEL is a complete framework oriented to the remote reprogramming, which consists in injecting new codes in to nodes at run-time. This feature, for example, allows to change the configuration and the algorithms on the remote node directly from the Control Room. Since the remote reprogramming is not a completely new subject on the WSN literature, we want to focus on the following key features:

1. the ability to load more than one single application without affecting the others running applications,
2. the ability to kill a bugged application and thus to "protect" the execution from exceptions.

Other aspects such as the code dissemination optimization, while important, are not directly addressed by REEL in the sense that we relies on existing solutions. Moreover REEL framework provides a development environment to write and debug applications, a complete tool-chain and a simple mechanism to propagate the new code over the network.

The thesis is organized as following. In the first chapter we present the problem of the remote reprogramming in the WSN. In the second chapter, we discuss the main operating systems for WSN and the reprogramming methods present in the literature. In the Design and Implementation chapters, we present the architecture of REEL and our practical realization. In the fifth chapter, we present the experimental results and finally, in the last chapter, we show the final remarks.

1.1. WSN overview

We consider a monitoring system for environmental and structural applications composed by a set of remote monitoring systems (RMSs) deployed in the environment to observe the physical phenomena of interest and by a Control Room that collects data from remote systems for processing, analysis and storage. The hybrid wired-wireless architecture of the RMS is composed by:

- a cluster of sensing and processing units (sensor nodes): acquire and locally process signals (e.g., MEMS, geophones, temperature sensor, strain gauges)
- a base station (gateway node) connected to all sensor nodes of his RMS: gateway nodes collects the acquisition from the sensor nodes and remotely transmit them to a control room for storage and analysis.

The proposed architecture uses different hardware platforms and system software for two class of nodes: DSPIC33F microcontroller with FreeRTOS for sensor nodes and 200MHz ARM9 CPU with Linux for gateway nodes. Our idea is different: a generic and stable hardware/software embedded system, configurable for a specific domain. The hardware platform is defined in the "NetBrick: a high-performance, low-power hardware platform for wireless and hybrid sensor network" [4] article, developed at Politecnico di Milano.

1.2. Reel objectives

Nowadays an update in a Wireless Sensor Network requires to disassemble the entire network. This procedure is really expensive and not always feasible. In this scenarios we are not only interest about the value detected by the sensors, but we want to know also the internal state of a remote node: running applications, configuration parameters (clock frequency), battery life, software faults and etcetera. Moreover we want to install and debug our applications on the remote node without disassemble the entire network. This is REEL OS. The main targets of REEL are:

- an efficient and secure reprogramming method
- an safe execution context per applications

In this thesis we explain in details all the mechanism of REEL solution. In specify, the main peculiarities are:

- Reprogrammability
- Maintenance
- Security
- Robustness and Safety
- Performance
- Usability and scalability

1.2.1. Reprogrammability

The main focus of REEL is the ability of reprogramming. The term reprogramming means the ability to load a new application or overwrite an existed one over a remote

sensor node. The load procedure looks like the installation methods of traditional computer system. The node supports a collection of applications which define the tasks and the algorithms to perform. The application can be insert, or install, delete or overwritten but the basic system must be unchangeable. The kernel and the services must be solid components to provide a high level of reliability. In the next section we present the reprogramming scenarios and analyze different kinds of situations.

1.2.2. Maintenance

In the software life cycle, the maintenance phase is the last stage of the cycle. During the maintenance phase of the software life cycle, software programmers regularly issue software patches to address changes in the needs of an organization, to the main reasons:

- correct issues relating to bugs in the software or resolve potential security issues
- prevent any hindrance to the expected performance of the software
- add increased functionality to the software

In the WSN domains, the application must satisfy the constrains due to the embedded platform.

REEL provides two main levels of maintenance:

- system: the kernel, device drivers, services are unchangeable
- user applications: can be update through reprogramming procedure

The system maintenance allows to manager the internal state of the remote machine. The control room can remotely access to the node and set the current configuration. In specify, the system maintenance lets to:

- monitoring the internal information of a node: battery status, used memory
- monitoring the user applications: list of installed, scheduled or running applications
- notify the software faults and select the politics to use

1.2.3. Security

The application can be signed for a specific devices, so nobody, unless the owner, can modify the binary. This security feature does not allow malicious person or virus to infect the application. A mechanism with cryptographic signatures avoids this hole in the security and increase the availability of the system.

1.2.4. Robustness and safety

In the perfect world, the applications have no bugs and the system never crash. But we are in the real word and sometimes the things don't go as you want. Sometimes the application profiling and simulating are not enough and they do not satisfy all possible scenarios. For example the application faults are not unusual: division by 0, access to a protected or invalid memory address. In the classic systems for WSN, the kernel and the applications are merged together, and if the application gets a fault, the system gets a fault, and the platform stops to work and resets itself. After reboot, the system execute the application again and it resets itself one more time. So we built a useless machine that consumes a lots of energy. A robust framework should recognize the software faults and block the bugged application. The system marks the application as not-executable, and it is never executed anymore.

1.2.5. Performance

The sensor node applications usually performs cyclic operations: sampling the values, local process and eventually transmit the results. These operations are performed in a well-defined interval of time. At the diminution of time interval length, follows the increase of the frequency increase and in high frequency applications, the frequency becomes one of the criticality of the system. In general a sensor node can sampling data at rate of 10/100Herz or more. To guarantee this rate we define two conditions for OS. First, an application, with cyclic behavior, must start exactly at the begin of the time period; the Real-Time scheduler of REEL satisfies this requirement. The second condition consists to verify that the application does not violate the frequency rate. A single cycle of the application must not exceed its period of time. The sampling frequency is set ad-hoc by the operator. The system should not introduce time overhead during the execution of the application. For this reason, the REEL reprogramming method, based on binary code, is much performed than virtual machine approach.

1.2.6. Usability and scalability

Many solutions in literature provides good reprogramming methods but many of them are hard to use. The complexity is due to: poor documentation, insufficient software tools and few library functions. Moreover the framework often doesn't help the operator with a common interface to exchange information over the network. We can exchange various information about the WSN node and collect them in the Control Room. In the Control Room different kinds of operator can interface with the system:

- Developers: write and debug their applications with a development environment, build the binary image with the tool-chains, and test the remote system

- Administrators: configure the nodes and monitor the network
- Observers: after an high-level synthesis, read the values received from the sensor nodes.

We focus our project for the firsts two kinds of operator that directly interface with the WSN nodes and REEL framework.

1.3. Reprogramming scenarios

Software updates for sensor networks are necessary for a variety of reasons ranging from implementation and testing of new features of an existing program to complete reprogramming of sensor nodes when installing new applications. In this section we review a set of typical reprogramming scenarios and compare their qualitative properties.

- Software development is an iterative process where code is written, installed, tested, and debugged in a cyclic fashion. Being able to dynamically reprogram parts of the sensor network system helps shorten the time of the development cycle. During the development cycle developers typically change only one part of the system, possibly only a single algorithm or a function. A sensor network used for software development may therefore see large amounts of small changes to its code.
- Sensor network testbeds are an important tool for development and experimentation with sensor network applications. New applications can be tested in a realistic setting and important measurements can be obtained. When a new application is to be tested in a testbed the application typically is installed in the entire network. The application is then run for a specified time, while measurements are collected both from the sensors on the sensor nodes, and from network traffic. For testbeds that are powered from a continuous energy source, the energy consumption of software updates is only of secondary importance. Instead, qualitative properties such as ease of use and flexibility of the software update mechanism are more important. Since the time required to make an update is important, the throughput of a network-wide software update is of importance. As the size of the transmitted binaries impact the throughput, the binary size still can be used as an evaluation metric for systems where throughput is more important than energy consumption.
- Correction of Software Bugs in sensor networks was early identified. Even after careful testing, new bugs can occur in deployed sensor networks caused by, for example, an unexpected combination of inputs or variable link connectivity that stimulate untested control paths in the communication software. Software bugs can occur at any level of the system. To correct bugs it must therefore be possible to reprogram all parts of the system.

- Application Reconfiguration in an already installed sensor network, the application may need to be reconfigured. This includes change of parameters, or small changes in the application such as changing from absolute temperature readings to notification when thresholds are exceeded. Even though reconfiguration not necessarily include software updates, application reconfiguration can be done by reprogramming the application software. Hence software updates can be used in an application reconfiguration scenario.
- Dynamic Applications in many situations where it is useful to replace the application software of an already deployed sensor network. One example is the forest fire detection scenario presented where a sensor network is used to detect a fire. When the fire detection application has detected a fire, the fire fighters might want to run a search and rescue application as well as a fire tracking application. While it may possible to host these particular applications on each node despite the limited memory of the sensor nodes, this approach is not scalable. In this scenario, replacing the application on the sensor nodes leads to a more scalable system.

Scenario	Update frequency	Update fraction	Update level	Program longevity
Development	often	small	all	long
Testbeds	seldom	large	all	long
Bug fixes	seldom	small	all	long
Reconfiguration	seldom	small	app	long
Dynamic App.	often	small	app	long

Table 1.1.: The reprogramming scenarios on WSN

2. State of the art

In this section we present a brief introduction of the available operating systems for embedded devices and in particular for wireless sensor network nodes. In particular, we will analyze the reprogramming methods supported by several OSs. Among the techniques present in the literature, we investigate:

- complete reprogramming: the entire program image, which contains the application *and* the OS is replaced by a new version; the size could be very large;
- incremental patches: only a delta is transmitted and replaced; the size is reduced;
- virtual machines: the application is based on a virtual machine byte-code, optimized in space for reduced transmission overhead
- loadable modules: applications and services are loaded at run time without modifying the OS

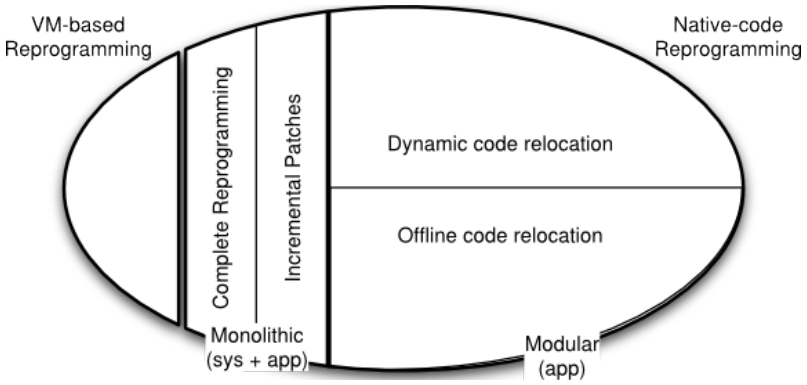

Figure 2.1.: Reprogramming methods

2.1. Operating systems for WSN nodes

The WSN node operating system are a key component in of the WSN-based application software stack. The internal architecture of an OS has an impact on the size on the overhead in terms of memory footprint and, from the development point of view, how it provides services to the application programs. The common OS architectures are the following:

- **monolithic architecture**: in the monolithic architecture (i.e., TinyOS, Nano-RK), the kernel provides all the services (e.g., device drivers, communication protocols, memory management, etc.). Such an architecture allows bind all the exported services together into a single system image, thus allowing to optimize the OS memory footprint. An advantage of the monolithic architecture is that the module interaction costs are low because the kernel space shares functions and variables among modules.

- **modular architecture**: modular architecture (used in FreeRTOS) exposes only basic set of functionality (i.e., memory management, scheduling). Other services (e.g., device drivers, network protocols, etc.) are relegated to external processes (such as the UNIX daemons). in modular kernels the modules are in practice separated processes that communicate each other by using the inter process communication (IPC) kernel service. In particular, the IPC provides mechanisms such as communication queues, semaphores and shared memories for both external services and applications. The main advantage of the modular kernel to complete decouple the modules via IPC, which eases the testability and maintenance of the system. The drawback of such approach consists in the augmented complexity in terms of memory required an CPU time to handle the IPC service w.r.t. the typical sys-call of the monolithic kernel.

Another important key to compare OSs are the programming model supported, which has a significant impact on the application development. There are two main programming models provided by typical WSN OSs:

- **event driven programming**: it is optimized for low-end devices, but difficult and error prone not from the developer point of view. It lets the micro-controller sleep as much as possible (it wake-up only to react to external events), thereby achieving energy-efficiency.

- **multi-threaded programming**: it is the classical application development model, but it require a not negligible overhead in terms of memory required and then not well suited for resource constrained devices such as sensor node. Statically allocating per thread stack is often too expensive in terms of memory space. Many contemporary on developing a light-weight multi-threading programming model for WSN OSs.

TinyOS [18] is one of the most widely used monolithic operating systems for WSN. TinyOS is written in NesC [9], a programming language based on C, and is a component-based OS, allowing modular programming. Its level of abstraction is very low and it is often difficult to implement even simple programs. In addition, rigorously event-based style and exclusion of blocking operations are often the sources of complexity. Fiber [30]is a lightweight concurrency model for TinyOS and allows a single blocking execution context along with TinyOS event-driven concurrency model.

Mantis OS [1] is at the other end of spectrum: it provides preemptive, time-sliced multi-threading on MICA2 motes. TinyThreads [30] , Proto-Threads [8] and Y.Threads [24] are in the intermediate level between Fiber and Mantis OS. TinyThread is a multi-thread library for TinyOS. Proto-threads are a multi-threading library for Contiki [7], which is an event-based operating system. Proto-threads are similar to TinyThread in that they also adopt the cooperating multi-threading.

2.2. Reprogramming techniques

The reprogrammability feature is something realized in middleware. Some examples are Impala [22] and SensorWare [2]. Impala is a middleware designed for ZebraNet project [14] and its goal is to enable application modularity, adaptability to dynamic environments, and repairability. Its modular design allows easy and efficient on-the-fly reprogramming via wireless channel. SensorWare supports Tcl-based control scripts for the language used for reprogramming. Compared to virtual machine, SensorWare is designed for a richer hardware platform such as iPAQ.

Some operating systems have started to feature a dynamic reprogramming capability. Deluge [12], a dissemination protocol, with TOSBoot boot-loader enables an in situ code update for TinyOS, Mantis OS [1] , Contiki[7], and SOS [10] also support dynamic update in finer resolution such as module and thread for more efficiency.

2.2.1. Complete reprogramming

The full image replacement is the easiest way to reprogram a sensor node by redistribution of the program. However, reprogramming cost is high because the image size can be large several kilobytes. Moreover, after the completion of downloading process, the sensor nodes must reboot themselves to activate the new image, causing the operation of sensor nodes to be stopped and the current execution state to be lost. Although an external mechanism can be provided to save the execution state to non-volatile memory and restore it after the reboot, this approach is still an expensive operation.

TinyOS [18] is the primary example of an operating system that does not support loadable program modules in the standard release. Several protocols provide reprogramming of the full system, such as Deluge[12], Stream [26], Freshet [16], MOAP [28], and MNP [17].

Deluge[12], a reliable data dissemination protocol for disseminating a large data object (i.e. larger than can fit into RAM) from one or more source nodes to many other nodes over a multi-hop wireless network. Deluge's density-aware, epidemic properties help achieve reliability in unpredictable wireless environments and robustness when node densities can vary by factors of a thousand or more. Representing the

data object as a set of fixed-size pages provides a manageable unit of transfer which allows for spatial multiplexing and supports efficient incremental upgrades.

2.2.2. Incremental patches

The differential patching schemes, or incremental code update, allow to reduce the amount of data transmitted during reprogramming. Based on the assumption that image changes such as bug fixes are usually small, they only distribute the binary differences between the original system image and the new one. After receiving the differences, the new image could be generated from the received differences and the current image stored on the sensor node. However, since previous differential patching schemes are based on XNP[5], sensor nodes still need to reboot themselves to execute the new image.

To support incremental reprogramming Jeong [13] use Rsync to compute the difference between the old and new program images. However, because it is built on top of XNP [5], it can only reprogram a single-hop network and does not use any application-level modifications to handle changes in function locations. In 2003 Reijers and Langendoen, the authors modify Unix's diff program to create a edit script to generate the delta. They identify that a small change in code can cause many address changes resulting in a large delta. Koshy and Pandey use slop, empty, regions lead to fragmentation and inefficient use of the Flash memory. Also when the function references to change and size of the delta script to increase.

Zephyr[25], a fully functional incremental multi-hop reprogramming protocol, is an extension of Rsync algorithm to reduce the size of the delta. The delta script is transmitted wirelessly to all nodes in the network in a multi-hop manner. The nodes save the delta script in their external flash memory and then build the new image using the delta and the old image and store it in the external flash. Finally the boot-loader loads the newly built image from the external flash to the program memory and the node runs the new software. Zephyr uses techniques like function call indirections to mitigate the effect of function shifts for reprogramming of WSN node.

2.2.3. Virtual Machine-based reprogramming

The idea is to provide an application-specific virtual machine, which is designed for a particular application domain and provides the needed flexibility, so that it can support a safe and efficient programming environment. The motivation for being application specific is based on the observation that WSN are usually deployed for a certain application purpose, unlike the Java virtual machine that is generic enough. Since virtual machines provide a higher level interface, the size of the byte-code is extremely small as compared to the native code image. This is attractive in WSNs since it reduces the energy for code dissemination during reprogramming.

Although reprogramming costs are cheap, the cost of running virtual machines on devices with strict resource constraints such as sensor nodes would be high. This is because dynamically interpreting byte-codes can have a significant computational overhead. Finally, the reprogramming ability is applied only to applications running on the virtual machine for a limited number of instructions. Moreover it is not possible extend on-the-fly the supported instructions.. Neither the virtual machine itself nor the kernel can be reprogrammed. Maté [19]and ASVM [21, 20] are stack-oriented virtual machines implemented on top of TinyOS. Melete [31] extends Maté and supports multiple concurrent applications. VMStar [15] allows the dynamic update of the system software such as VM itself, as well as the application code. In the licterature there are several application specific VM based on Java, example Darjeeling [3].

2.3. Loadable Modules

A loadable module contains the native machine code of the program that is to be loaded into the system. The machine code in the module usually contains references to functions or variables in the system. These references must be resolved to the physical address of the functions or variables before the machine code can be executed. The process of resolving those references is called linking. Linking can be done either when the module is compiled or when the module is loaded; we can identify the two different cases:

- **pre-linked**, or static, module contains the absolute physical addresses of the referenced functions or variables whereas a dynamically linked module contains the symbolic names of all system core functions or variables that are referenced in the module. This information increases the size of the dynamically linked module compared to the pre-linked module.
- **dynamic linking** has not previously been considered for wireless sensor networks because of the perceived run-time overhead, both in terms of execution time, energy consumption, and memory requirements.

With dynamic linking, the object files do not only contain code and data, but also names of functions are variables of the system core that are referenced by the module. The code in the object file cannot be executed before the physical addresses of the referenced variables and functions have been filled in. This process is done at run time by a dynamic linker. SOS [10] and Contiki[7] are examples of operating systems that support dynamically loadable modules in WSNs. Furthermore, when an existing module needs to be upgraded, it must be completely removed and its current state and allocated system resources must be discarded. Thus, the new module needs to be run from the beginning and to acquire the necessary resources again. This not only increases the reprogramming latency but also degrades the system availability.

The main advantage to choose the loadable module solution are:

- high execution performance: the code is compiled for the specific target, at different of virtual machine
- low overhead in transmission: transmit only the module application instead the full image
- higher usability: it is not necessary know exactly the current state of full image in the node, like the differential patching scheme

The high complexity of dynamic linking increases the amount of traffic that needs to be disseminated through the network to transfer the symbol and relocation tables to the node for runtime linking. Moreover the symbol table consumes a considerable amount of FLASH memory. On the hand, in the pre-link solution, the linking procedure is performed by the base station without exchange symbol information. The base station need to know information about the current state of the sensor node: the allocated and free memory segments.

Contiki is a lightweight open source OS written in C for WSN sensor nodes. Contiki is a highly portable OS and it is built around an event-driven kernel. A typical Contiki configuration consumes 2Kilobytes of RAM and 40 Kilobytes of ROM. A full Contiki installation includes features like: multitasking kernel, preemptive multi-threading, proto-threads, TCP/IP networking, IPv6, a GUI, a web browser, a simple telnet client. The Contiki OS follows the modular architecture. At the kernel level it follows the event driven model, but it provides optional threading facilities to individual processes. The Contiki kernel comprises of a lightweight event scheduler that dispatches events to running processes, Protothreads[8]. The Contiki architecture is composed by:

- Core: kernel, device drivers, standard applications, parts of C library and symbol table. It includes the meta-information about the loadable programs
- Loadable programs: load at the top of core; they have full access to the core functions and variables. Each loadable programs is a module, a binary code unloadable into the system.

The reprogramming methods of Contiki is based on the dynamic linking procedure; we can distinguish two main steps of the reprogramming methods:

- **linking**: link the symbol name of the functions and variables into the physic address, before the execution of the code (compile-time o load-time).
  - pre linking : link the reference into physic address
  - dynamic linking: link the reference into symbol name
- **relocation**: bind the reference of functions an variables of a same module. (compile-time o run-time)

The complexity of the system requires the minimum dimension of the system symbol table of 4 Kbytes of ROM.

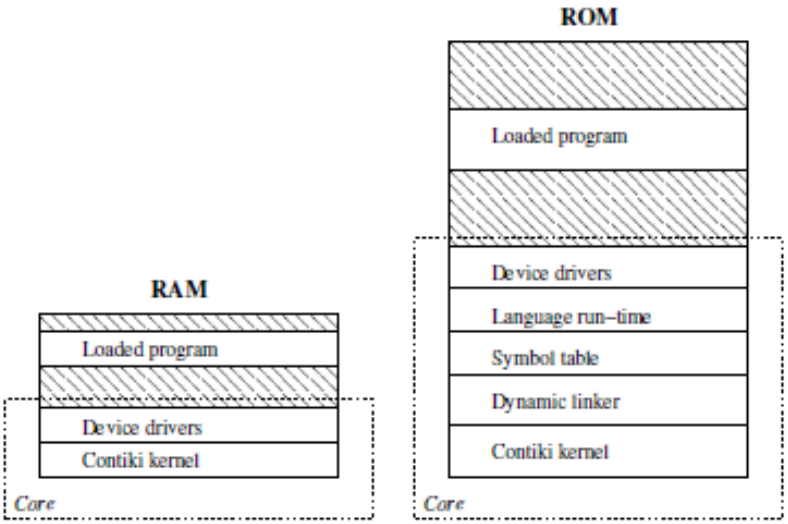

Figure 2.2.: The memory architecture of the Contiki memory [7].

2.4. Proposed solution

In this thesis we propose REEL, a framework oriented for remote reprogramming, based on the FreeRTOS real-time OS. The reprogramming method is based on loadable modules. This approach provides a low code dissemination and high performance on execution. Moreover it allows the developer to write applications completely separated from the underlining kernel and libraries and upload only the applications to the sensor nodes.

In specific, REEL implements the pre-link, or static linking, method to decrease the overhead due to the binary transmission and reduce the energy consumption. Instead dynamic linking, REEL does not dissemination the symbol tables and the linking phase is performed by the base station during the generation of the binary code. So it is necessary that the base station know the free memory regions over the sensor node. This constrain is less strong than differential patching scheme: we need to know exactly the full image on the current system.

On the sensor node side, the reprogramming procedure is very easier: the system stores the binary code exactly into the specified region of memory.

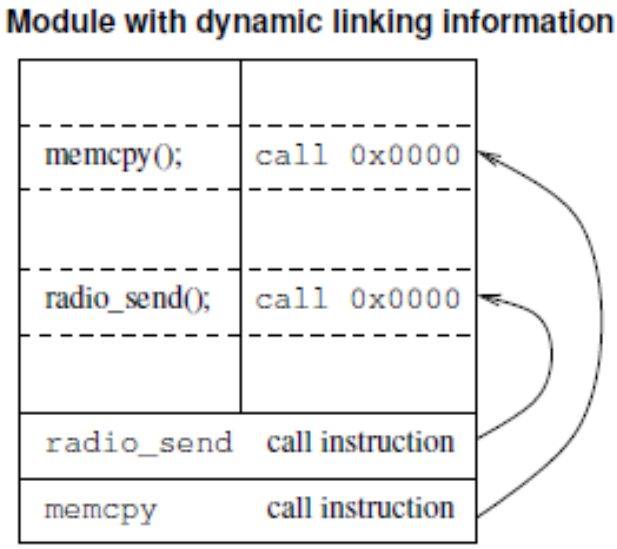

Figure 2.3.: The dynamic linking mechanism of Contiki [7]

3. Design

REEL is a framework based on the FreeRTOS operating system. In particular, the presented framework is a collection of software services:

- Remote reprogramming mechanism
- Safe execution context for applications
- Remote debug of system and applications
- Monitoring system and applications

It is worth noting that the REEL frameworks relies on standard communication protocols (e.g., ZigBee, 6loWPan etc) to manage the local network aspects. In fact, the focus of the entire framework is to provide a robust mechanism to remote reprogramming a WSN nodes; we are not interested to present a reliable way to disseminate the program because they already exist valid solutions in the literature (e.g., SINAPSE [27]).

In the classical WSN system hierarchy exists three main components, in general differing for hardware capabilities and software architecture. The communication back-bone provided by the network-layer allows to exchange data, parameters, commands and either application. Those three components are:

- Control Room: monitors the state of the network and the state of each node
- Gateway node (GW): coordinates the entire network and is able to communicate with the control room
- Sensor node (SN): has the role to sense and locally process the measurements coming from attached transducers; the gathered data are routed to the GW node.

Typically GW and SN have different hardware; however in our system the GW and SN share a similar hardware architecture: the NetBrick mote. Since the NetBrick architecture is modular, the difference between GW and SN consists only in the hardware modules configuration (a further discussion about NETBRICK hardware is presented in "NetBrick: a high-performance, low-power hardware platform for wireless and hybrid sensor network" artivle[4]). Thus, the SN and GW sports a similar software architecture and, in particular, they have the same operating system but different the services (provided by REEL) installed.

In the following paragraphs, we briefly introduce the hardware architecture used as foundation of the REEL framework, then we introduces all the layers composing

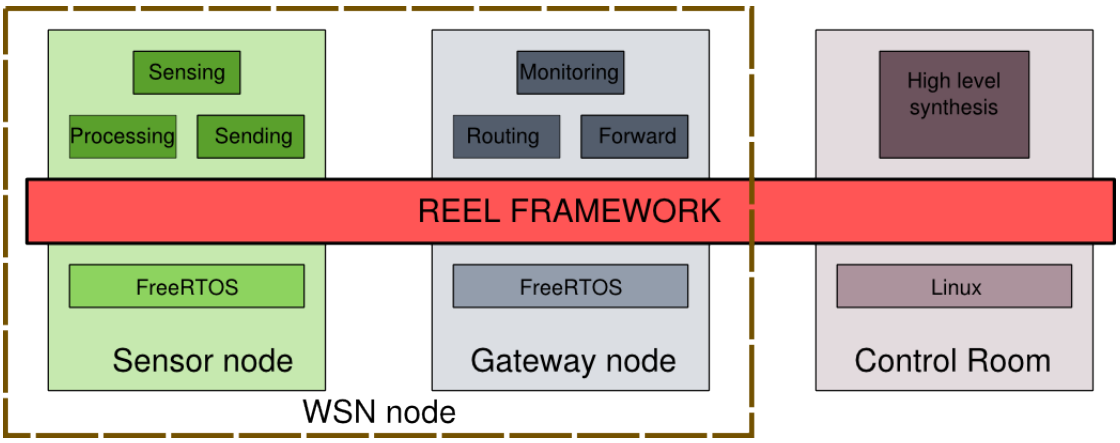

Figure 3.1.: REEL over the network

REEL with a special focus on the remote reprogramming procedure. We present, also the REEL framework at the Control Room level, which allows to write, compile, monitor and transmit an application to a WSN node.

3.1. Sensor and Gateway nodes architecture

The REEL framework provides a Real-Time OS with a robust execution context and protection mechanisms for a secure remote reprogramming procedure for the WSN node. Moreover it provides a complete development environment for writing, debug the applications and a set of tools for the control room to interact and install applications on the WSN node. For this reason without loss of generality, in the rest of the paper we will consider REEL framework for a generic WSN node and the behavior depends on the installed applications. The software architecture layers are the following:

- **hardware**: The NetBrick[4] board is equipment with a STM32F103ZE processor based on ARM CORTEX M3, an low-power architecture for embedded systems.
- **kernel space**: REEL framework abstract from the hardware platform to provide an execution environment for applications. It is composed by two main parts:
  - **kernel**: is responsible to manage the hardware and to provide basic functionalities used by higher levels. It mainly consists of the operating system and low-level libraries.
  - **services**: provides secure reprogramming, robust execution environment and remote command interface services.

- **application space**: is the top of the described software stack and uses all the features provided by the underlying layers.

In the next sections we present in details each REEL layers.

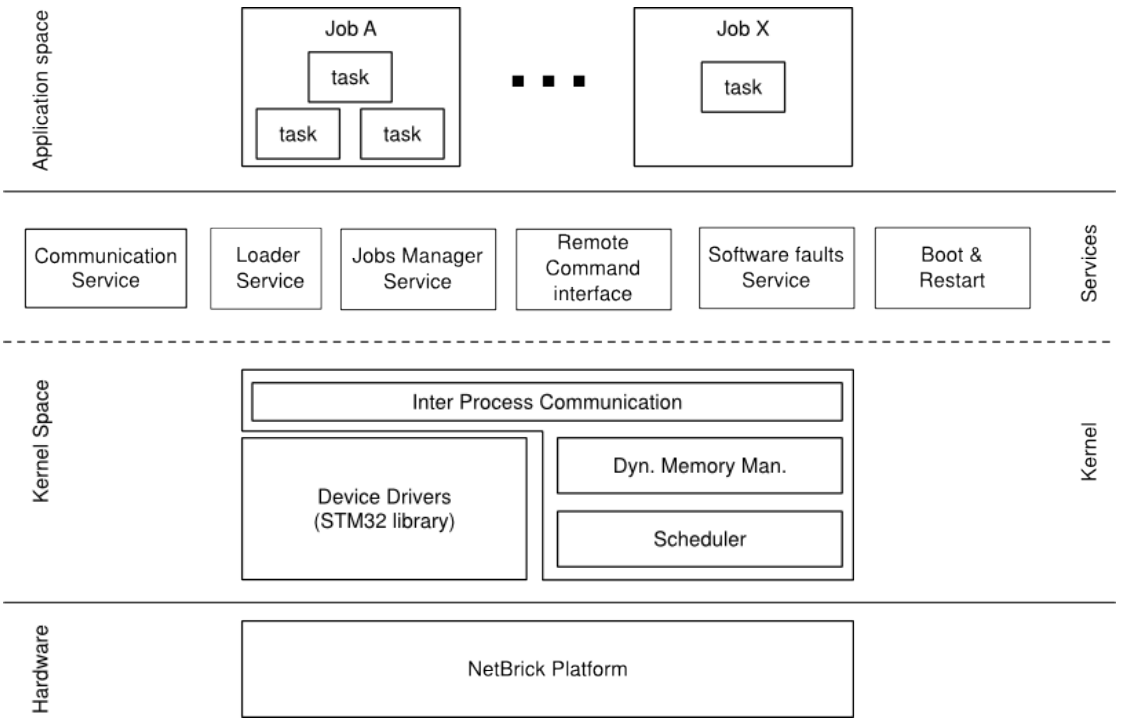

Figure 3.2.: The architecture of the REEL node

3.2. Hardware

An efficient, adaptive and reliable software architecture is essential to locally process signal acquired up to high frequency, to guarantee synchronism among the acquisition units and to provide the mechanism to update the behavior of RMS nodes during the operational life. The choice of the appropriate system layer architecture for each hardware platform with its specific constraints is crucial. The WSN nodes are equipped with the NetBrick board[4], developed at Politecnico di Milanobased. This board is equipped with the STM32F103RB processor by ST Microelectronics. The limited memory does not allow to run a general purpose operating system such as Linux. Therefore, we employed an embedded real-time operating system that guarantees real time elaboration on high frequency and provides tools and services for software managing. The principal component of the architecture is the ARM 32-bit CortexTM-M3 CPU Core. The main properties of this processor are the following:

- 72 MHz maximum frequency,1.25 DMIPS/MHz

- 20 Kbytes of SRAM memory for volatile data
- 256 or 512 Kbytes of Flash memory for permanent data and executable code
- Low power: Sleep, Stop and Standby modes

The FLASH memory is grouped in blocks of 2Kbytes. This observation is fundamental to develop an efficient reprogramming method. Moreover the NetBrick board has a lots of additional peripherals as:

- debug mode: Serial wire debug (SWD) & JTAG interfaces
- communication interfaces: USARTs (ISO 7816) and CAN interface

The NetBrick board is not the only board supported by REEL. We executed and tested successfully REEL under STM3210E-EVAL, the ST Microelectronics development board. So REEL can be installed in several STM32 boards equipped with enough memory for store the kernel and the applications.

3.3. Kernel layer

The kernel is the first software layer to provide an abstraction level from the hardware. It is composed by the ST Microelectronics low-level libraries and the FreeRTOS, an real-time operating system for embedded devices. Our proposed solution uses software components from third parties to guarantee security and reliability. FreeRTOS is mostly used in embedded system and a lots of company used it for business purpose. Although these software components are not provided with the REEL, they can be freely downloadable and usable from their website.

3.3.1. FreeRTOS

The proposed REEL framework is based on FreeRTOS, a real time operating system for embedded devices, with its low overhead in terms of memory foot print (an essential feature in memory constrained devices) and a real time scheduler to verify in advance the scalability of the processes running. Moreover FreeRTOS supports a large set of boards and interacts with the ST low-level libraries which contains functions to access to the hardware devices. We choose FreeRTOS because:

- native support different architectures and development tools.
- reliable and undergoing continuous active development.
- minimal ROM, RAM and processing overhead.
- very scalable, simple and easy to use.
- truly free for use in commercial applications.

In the other hand, FreeRTOS does not include all the concepts of a complete kernel system: it doesn't provide the abstraction of applications, advanced services or device libraries. REEL aims to complexity provides a model of the application, the job, that can be stored, execute and transmit over the network.

3.3.2. Device driver

The NetBrick board is equipped with a processor of the STM32 family. ST Microelectronics provides the support for their components composed by documentation, schematics, libraries and demo projects under copyright. We are interesting in the library of low-level for interacting with registers and peripherals. Each peripheral is linked to the processors with a specific port and each port is mapped in one or more processor register. We make a different from other solutions, because we are not interested to build a new Open Source low-level library, but we want use the ST standard library.

3.4. Services Layer

The services layer extends the functionalities of a common OS, like FreeRTOS, to became a complete framework system. REEL framework provides the abstraction of high-level application as independent program. Our main focus is the ability to reprogram the remote node and manage the current execution of the remote applications. For this purpose we define cross-services between WSN nodes and Control Room:

- **remote command interface service**: interaction between the Control Room and the WSN node.
- **secure reprogramming service**: install an application on the remote note with static linking method and verified the authenticity of the application.
- **software faults manager**: manage the exceptions to guarantee a robust execution environment and notify the fault to the Control Room.
- **communication service**: is the interface to send and receive data over the network

3.4.1. Communication service

The communication layer is responsible to transport data, parameters and commands for the reprogramming procedure between the node and the Control Room. In specify we can split the communication in two different class:

- **data**: the node sends the processed data to the Control Room .

- **commands**: the Control Room interacts with the node to read the current status of the system and reprogramming a specific job.

Although in our realization the local communication is based on ad-hoc protocol, we don't want to focus on a specific the communication protocol. In real system communication protocol is chosen by the topology of the network and the nature of the application domain. In the following we list the main quality of a communication layer:

- the topology of a network determinate a single hop or multi-hop routing algorithm.
- adaptive routing algorithm to topology adaption
- unit can run out of power with subsequent no wished disconnection from the network.
- easy scalability of the WSN network
- quality of service of communication among units is affected both meteorological and environmental
- conditions as well as electromagnetic pollution

3.4.2. Remote command interface service

The RCI provides a standard communication interface between the Control Room and the node through a set of commands that will be will be executed on the remote node. A key aspect of our architecture is the ability to interact with the node to acquire data information, read the current status of the system and reprogram a job. In specific we consider two different access levels for operators:

- **application expert**: retrieve the information of interest from an acquired measurement or processed data.
- **application designer**: manage the maintenance and reprogramming. The main functions are the following:
  - retrieve and change the current status of one or more applications
  - upload and download of a single application
  - notify when happen an exception on the node and select a politics to use
  - restart, suspend or clean the node execution

The RCI service is based on client/server architecture:

- Control Room as Client: the CR sends commands to the WSN node.
- Sensor and Gateway Nodes as Server: the SGN receive the commands from CR, locally process them and transmit a replay to the CS. A replay is usually a ACK/NACK to notify the success/failure of the command execution

3.4.3. Loader service

One of the main focus of the research is the secure reprogramming method. To guarantee the security of the applications we used an hardware/software mechanism based on Trusted Platform Module, TPM. Each application can be signed for a specific devices, so nobody, unless the owner, can modify the binary. This security feature does not allow malicious person or virus to infect the application. For example, two years ago the virus Stuxnet infected the workstations and the computers of nuclear plants in Iran. This virus cloned itself in the internal network and compromised the software (Siemens Step 7 in windows OS) for the reprogramming procedure of PLC devices. A mechanism with cryptographic signatures avoids this hole in the security and increases the availability of the system. We present 3 different scenarios for the security of an application:

- the company builds the application and nobody can change.
- a certified operator builds the ad-hoc application
- a 3-parties authority certified the applications

3.4.4. Software faults handling service

In a scenario of medium criticality applications, the certification body agrees that the proposed architecture and safety claims meet the standards necessary for the assessed system safety integrity level. Moreover the STM32 micro-processor has embraced the concept of a MPU (Memory Protection Unit). This is a simple and fast unit designed for special purpose systems, as opposed to the MMU (Memory Management Unit), suitable for general purpose designs. REEL manages the memory protection unit (MPU) to ensure tasks cannot inadvertently access each others RAM memory space, or the RAM memory space of the kernel. Further, REEL ensures that a task cannot inadvertently execute the kernel code. We are interested in the following class of faults:

- **hard fault**: A hard fault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism.
- **memory management fault**: A memory management fault is an exception that occurs because of a memory protection related fault. The fixed memory protection constraints determines this fault, for both instruction and data memory transactions. This fault is used to abort instruction accesses to Execute Never (XN) memory regions.
- **bus fault**: A bus fault is an exception that occurs because of a memory related fault for an instruction or data memory transaction. This might be from an error detected on a bus in the memory system.

- **usage fault**: A usage fault is an exception that occurs because of a fault related to instruction execution. This includes:
  - an undefined instruction
  - an illegal unaligned access
  - invalid state on instruction execution
  - an error on exception return.

  The following can cause a usage fault when the core is configured to report them:
  - an unaligned address on word and half-word memory access
  - division by zero

We have seen different kind of faults, but we don't discuss about how to manage the fault. We presented different politic strategies:

- restart of the system
- not consider the fault and go on
- suspend the faulty application and delete from restarting scheduled applications
- increment the faulty register and after n-times suspend the application as before
- notify to the Control Room the faulty event and receives the ad-hoc strategies to handle the exception

The operator can choose one or more politics at the same times in case to use different mechanisms with different faults.

3.4.5. Boot and restart services

We distinguish different two kind of system restart:

- **hardware reset**: by pressing the reset button on the board
- **software reset**: by hardware constrain (ex. power-off) or a software command.

Note that we consider the hardware reset as the high priority reset and in this special case we desire to load only the basic system (kernel and services). In the other situation the system return to the state previously the reset event. After the reset event, the main phases of restarting are the following:

1. Initialize and run the kernel.
2. Initialize and run the services.
3. Execute only the previously running applications

a) Copy the from SRAM to FLASH the application code

b) Initialize the application variable

c) Schedule the application

3.4.6. Job service

A Job is a program develop by the user and executed by REEL. Basically it looks as a normal program written in C with ST library methods and FreeRTOS functions. REEL is based on FreeRTOS that implements the methods and strategies to execute multiple programs at the same time by the preemption model. REEL supports the multitasking model as well and extends the concept of multi-Jobs: all tasks of a jobs are executed at the same time. Usually a job is realized as a single task on FreeRTOS. On the other hand, a single job can instantiates multiple tasks to cooperate and make a complex work. The job service implements all kinds of mechanisms to provide the abstraction of jobs.

3.5. Application Layer

In the previously sections we described the architecture and the functionality of REEL framework as a generic purpose system usable in a wide range of domains. The behavior of the software depend on the applications running at the top of the system.

In the WSN infrastructure, the Sensor Node applications focus on the local elaboration of the acquired signals; the Gateway Node applications exploit the management of the WSN; finally, the Control Room applications are sophisticated post processing algorithms. All this kind of operations are organized in applications.

3.5.1. WSN node monitoring applications

REEL framework provides a set of applications for monitoring the current hardware and software status on the remote nodes.

- Hardware monitoring:
  - QoS management o Energy/power management
- Software monitoring:
  - Data memory
  - Status of interprocess communication structures
  - Profiling: CPU and memory consumption

  - Ad-hoc performance solutions

- Connectivity and advanced communication services
  - Routing of communication flow
  - Reconfiguration of path between nodes
  - Reconfiguration of communication frequency for power-consumption.

3.5.2. Control Room applications

The software of the Control Room is composed by high level applications for computation and analysis of the retrieved data. Usually these software make some statistical analysis of the data exchange on the network and process data retrieved from the SNs. A database can store all the data and value for history purpose.

3.6. REEL tool-chain

The REEL tool-chain provides a set of Open Source tools for application development. The development environment tools let the following operations:

- Provide the base for develop and extension of the REEL platform allowing customization for optimize the kernel code for specific application domain
- Develop new application based on REEL framework
- Compile the application and link together with REEL automatically allowing the code reallocation
- Verify the physical constrains: memory space, dependencies, real-time profiling and user constrains
- Manage the maintenance and reprogramming of a remote node
- Debug an application on remote node
- Provide the set of tools for monitoring the activity of every nodes and the status of complete network.

4. Implementation

In this section we discuss the behavior of REEL and show the implementation of each components. We preset the hardware platform Netbrick[4] used in this project and a short introduction to FreeRTOS. Over FreeRTOS we realize the REEL service level to implement all the politics and components presented in the Design chapter: Job manager, Remote command interface, Exception manager, Loader service, Boot and restart service. The last section of this chapter presents the mechanism of static linking technique and tools to generate a re-allocable binary file for a REEL application. The figure 4.1 shows the implementation of the designed architecture of REEL into Flash and Sram memories. The layers of the REEL architecture are mapped in different regions in the memories. The size of the system depend on the configuration of FreeRTOS and the ST libraries which we need.

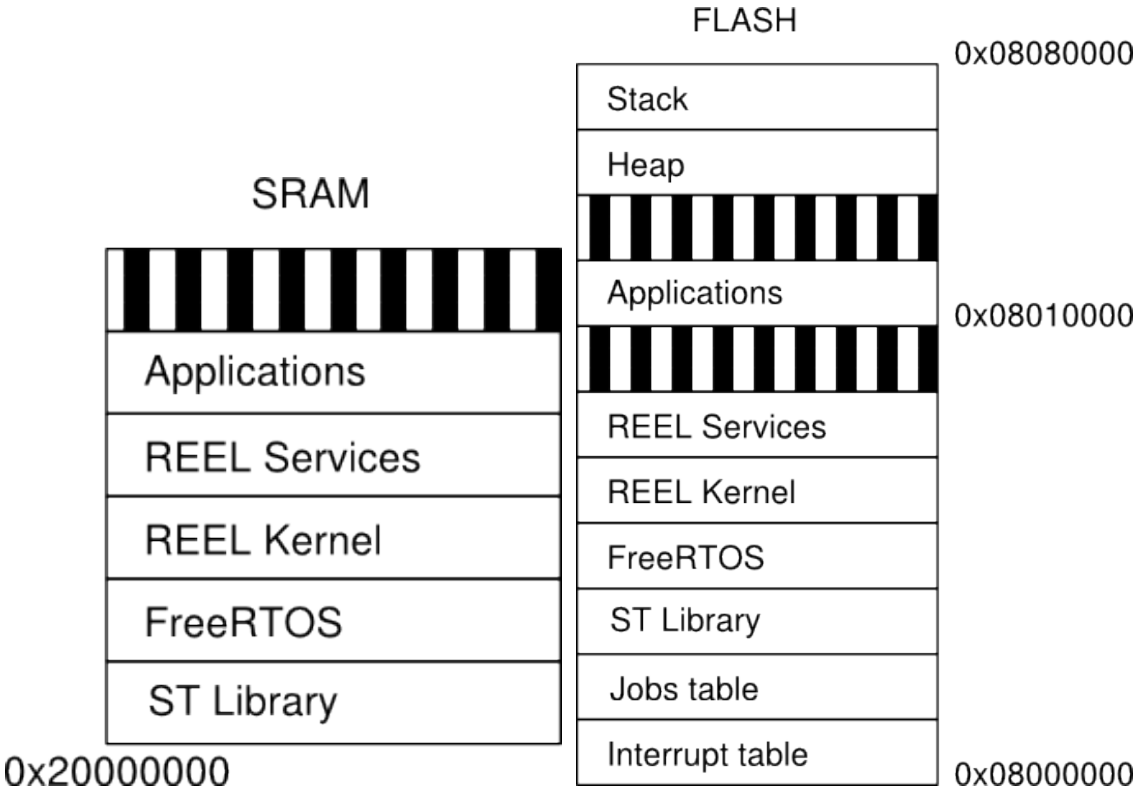

Figure 4.1.: Sram and Flash memories

4.1. Kernel space

FreeRTOS is an high configurable real-time operative system. We can choose:

- clock of cpu processor Type of task scheduler : preemptive or cooperative.
- size of stack memory, name length and property for each task
- schemes of memory allocation

These and other parameters are define in the main FreeRTOS configuration file, "FreeRTOSConfig.h":

```
1 #define configUSE_PREEMPTION 1
2 #define configUSE_IDLE_HOOK 1
3 #define configUSE_TICK_HOOK 0
4 #define configCPU_CLOCK_HZ ( 24000000UL )
5 #define configTICK_RATE_HZ ( ( portTickType ) 1000 )
6 #define configMAX_PRIORITIES ( ( unsigned portBASE_TYPE ) 5 )
7 #define configMINIMAL_STACK_SIZE ( ( unsigned short ) 70 )
8 #define configTOTAL_HEAP_SIZE ( ( size_t ) ( 7 * 1024 ) )
9 #define configMAX_TASK_NAME_LEN ( 10 )
```

4.1.1. Kernel structure

REEL is based on FreeRTOS. We show the REEL kernel structure as a FreeRTOS program. We can defined four main parts of FreeRTOS program:

1. Includes the FreeRTOS headers (FreeRTOS.h, task.h, list.h, queue.h, FreeRTOSConfig.h)
2. Declarations of global task handler variables (xJOBMANAGERTaskHandle, xRCITaskHandle)
3. Initialization of components and creation of tasks: we can choose the priority, stack size and function handler for each task.

```
1 void kernel_init (void){
2     /* Component initialization */
3     EXCEPTIONS_conf();
4     /* Tasks creation */
5     xTaskCreate(RCI_start,(signed char *) "RCI\0",
          APP_TASK_STACK_SIZE/2, NULL, APP_TASK_PRIORITY, &
          xRCITaskHandle);
6     xTaskCreate(JOBMANAGER_task,(signed char *) "JOB\0",
          APP_TASK_STACK_SIZE/2, NULL, APP_TASK_PRIORITY, &
          xJOBMANAGERTaskHandle);
7 }
```

4. Starting the FreeRTOS scheduler: the scheduled tasks will be execute.

```
1 void kernel_start (void){
2     /* Start the scheduler. */
3     vTaskStartScheduler();
4     /* Will only reach here if there is insufficient heap available
          to start the scheduler. */
5     while(1);
6 }
```

4.1.2. FreeRTOS's tasks

A real time application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the RTOS scheduler itself. Only one task within the application can be executing at any point in time and the real time RTOS scheduler is responsible for deciding which task this should be. As a task has no knowledge of the RTOS scheduler activity it is the responsibility of the real time RTOS scheduler to ensure that the processor context (register values, stack contents, etc) when a task is swapped in is exactly that as when the same task was swapped out. To achieve this each task is provided with its own stack.

A FreeRTOS task can be in one of the following state:

- **Running**: When a task is actually executing it is said to be in the Running state. It is currently utilizing the processor.
- **Ready**: Ready tasks are those that are able to execute (they are not blocked or suspended) but are not currently executing because a different task of equal or higher priority is already in the Running state.
- **Blocked**: A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event.
- **Suspended**: Tasks in the Suspended state are also not available for scheduling.

4.2. Reprogramming model

The job is a binary program of the user application compiled for the specific target machine that can be transmitted, installed and executed in REEL. The job is compiled for the stm32 platform in the control station with a special procedure to statically reallocate the object code. This special procedure allows the job manager of REEL to execute the native code of an installed job into the WSN node. FreeRTOS does not support the reallocation procedure to link external binary files: the kernel and the applications are a single monolithic binary file.

4.2.1. Concurrent programming model

REEL is based on FreeRTOS that implements the methods and strategies to execute multiple tasks at the same time by the preemption model. In FreeRTOS a task is a path of program that is independent and is scheduled by OS. Moreover all FreeRTOS tasks have the same properties and does not provide hierarchical mechanism. In other words, an FreeRTOS application is a monolithic binary composed by the kernel and various tasks work independently where each of them is not related to the others. REEL supports the multitasking model as well and extends the concept of multi-Jobs: all tasks of a jobs are executed at the same time. Usually a job is mapped with a single task on FreeRTOS. Moreover a single job can instantiates multiple tasks to cooperate and make a complex work. The multi-task paradigm looks like the multi-threading paradigm in POSIX concurrent programming:

- a job is an executable program for REEL with his own instruction pointer, address space, memory space, stack and registers. REEL stores meta-information about each jobs in the jobs table, as current state or entry-point.
- a task is a path of execution in a job. Each task has its own instruction pointer, stack and various registers. It shares the same memory space with the job that created the task. All of the tasks in the job share the same global memory, but each task is allocated a chunk of memory for its stack space.

The jobs solution has some others advantages:

- the same time of a normal task because a job for FreeRTOS scheduler is a normal task
- a shared global memory for tasks of the same job
- general functions for group of tasks, when a job begins it starts all child tasks

4.2.2. Job state

The states of the job are the following:

- **load**: the job is correctly received but it is not jet installed in the machine
- **ready**: the job is installed in the machine and it is ready to run; the jobs manager stores the code and data of the job in the memory and sets the meta-information.
- **run**: the job is executed; all the tasks of the job are scheduled. At boot time, the job is started.
- **suspend**: the execution of the job is suspended by the user; all the task of the job are delete from the scheduler. At boot time, the job is restarted.
- **block**: the execution of the task is stopped by a fault. If only one task gets a fault, all tasks of the job are suspended. At boot time, the job is not started.

The status information is useful in the reset phase when the boot loader looks the application table and starts only the jobs marked as RUN.

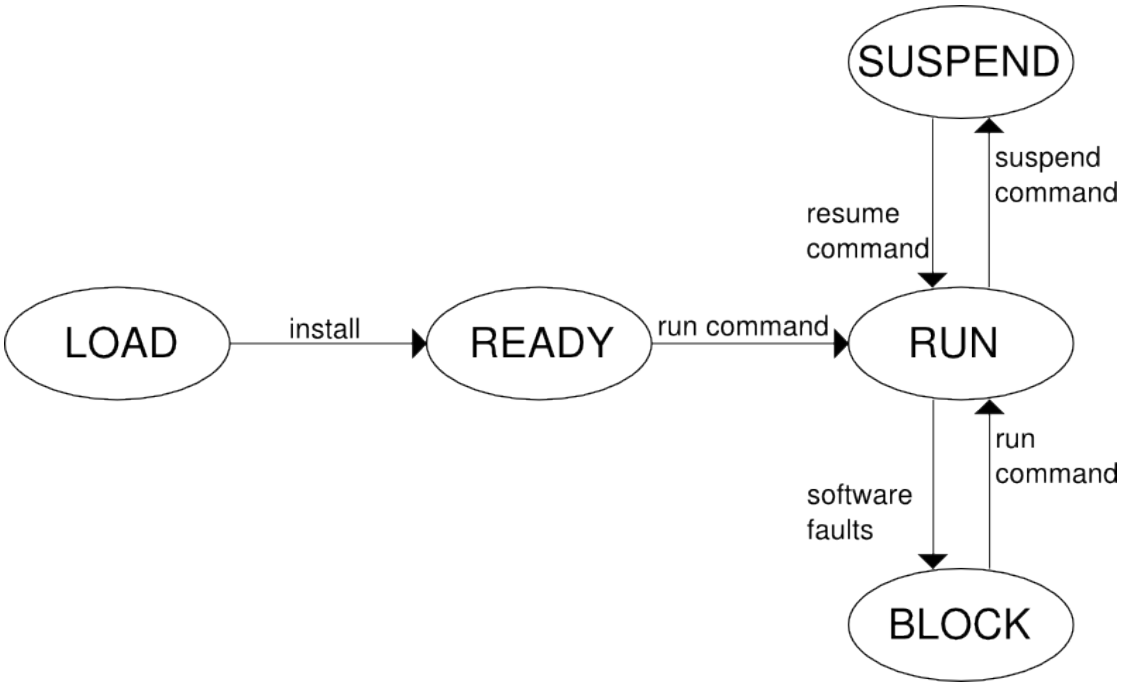

Figure 4.2.: Job states

4.2.3. Job structure

The main meta information associated to a job are the following:

- size and started address of code and init sections on FLASH memory.
- size and started address of data and bss sections on SRAM memory.
- the list of tasks of the job.
- the current state of a job. In the following we present the complete structure of a job.

```
1  struct job_t {
2      uint8_t id;
3      uint32_t size_text;
4      uint32_t size_data;
5      uint32_t size_bss;
6      uint32_t size_init;
7      uint8_t signature[SIGNATURE_SIZE];
8      uint8_t *text;
9      uint8_t *init;
10     uint8_t *data;
11     uint8_t *bss;
```

```
12      Jobstate state;
13      xTaskHandle task;
14      xTaskHandle *taskChild[MAX_TASK_FOR_JOB];
15  };
16  typedef struct job_t Job;
```

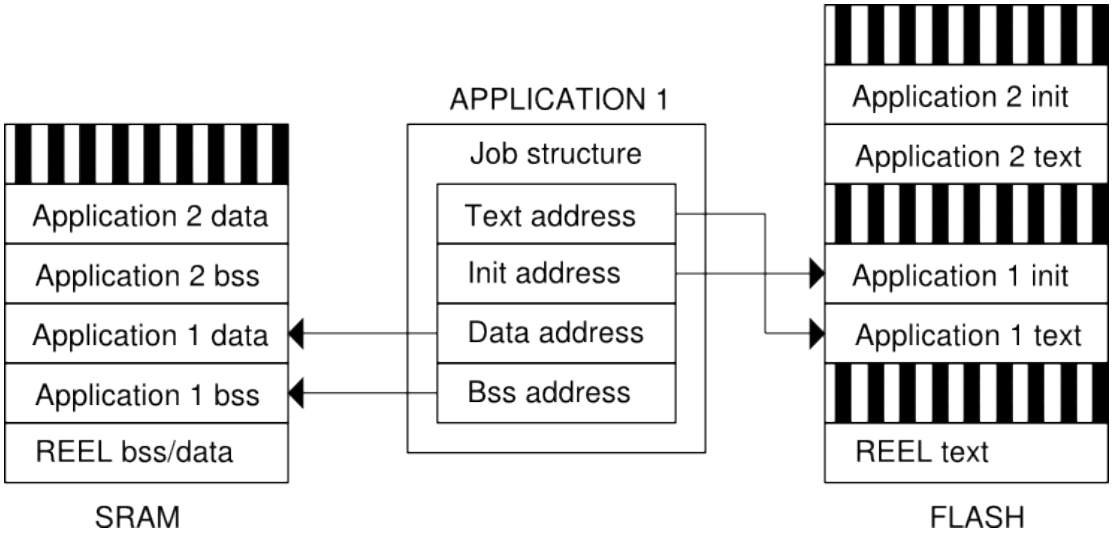

Figure 4.3.: Job structure and memory addressing

4.3. Jobs manager

The high-level application for REEL framework, called Job, is a program develop by the user and executed by REEL. Basically it looks as a normal program written in C with ST library methods and FreeRTOS functions. REEL wants to improve the execution paradigm of concurrent programming and extend the concept of FreeRTOS task. In REEL a job is:

- the binary file of the application that can be transmitted, installed and executed in REEL framework.
- the hierarchical multi-tasks execution model for concurrent programming

The jobs manager component implements all the mechanisms to manage a job on the device. For this purpose the jobs manager stores information about the memory location, the execute status and the some error status for each jobs. These meta-information form the list of jobs stored in a global structure called JobsTable.

4.3.1. Jobs table

The JobsTable is the list of all jobs installed on the machine. This is a critical resource for two reason:

- It can be accessible in writing only by task in the kernel space.
- The values must not change after a reset In the REEL the JobsTable resides in the initial regions of the FLASH memory after the Interrupt Vector. REEL allocates 2kB of FLASH, 16 job-entries, for the JobsTable because 2kB is the minimum block writable by the FLASH.

In other words, REEL standard configuration support at max 6 installed jobs. For performance in SRAM there is a copy of the JobsTable for local update.

```
1 static Job JOBMANAGER_table_FLASH[JOBTABLE_SIZE] __attribute__
    ((section("jobtable")));
```

We present two solution for the implementation of the Jobstable:

- **centralized**: store all the meta information in a single location in memory.
- **distributed**: each job contains his meta information in his part of memory assigned.

The reason of the choice is imposed by hardware constrains. In this implementation we used a centralized job table.

4.3.1.1. Centralized approach

The system allocates one block of the FLASH memory for the job table structure. In the STM32 architecture a block of memory is composed by 2k of bytes. In one hand, all the information about the current status of the entire system are in a single place and we don't need scan the memory to find meta information. In the other hand, the entire block of memory must be rewritten each time that we want to modify the status of a job or insert/delete a job. So we need a buffer of dimension of one memory block.

4.3.1.2. Distributed approach

The distributed solution is based on the idea that each job stores his meta information in his segment of memory. At startup the boot loader scans each segment of memory and retrieves the value of the information. We save memory because we don't use a block of memory for the table and a buffer as in centralized case. In the other hand, a change in the job state field brings the rewriting of all the job memory; this is not want we want.

In the simplest scenario, the system knows only the running jobs. If an application has faults or does not work anymore, it is clear from the system. In this way, the system does not need the byte for the jobs status. So a single row in the job table has read-only access. This solution is the perfect case for a distributed job table where each job contains his own meta information which a read-only access.

4.3.2. Jobs operations

List of functions to scan, insert or delete jobs from the JobsTable:

- JOBMANAGER_task: scan the JobsTable and schedule the jobs to execute at restart.

```
1  void JOBMANAGER_task(void) {
2      //Copy the structure from FLASH to SRAM
3      FLASH_readPage((uint32_t)JOBMANAGER_table_FLASH,(uint32_t)
           JOBMANAGER_table_SRAM);
4      uint8_t i;
5      for (i=0;i<JOBTABLE_SIZE;i++){
6          if(JOBMANAGER_table_SRAM[i].id!=0){
7              switch(JOBMANAGER_table_SRAM[i].state){
8                  case BLOCK:
9                      break;
10                 case READY:
11                     JOBMANAGER_table_SRAM[i].state=LOAD;
12                     JOBMANAGER_ready(&JOBMANAGER_table_SRAM[i]);
13                     break;
14                 case RUN:
15                     JOBMANAGER_table_SRAM[i].state=LOAD;
16                     JOBMANAGER_ready(&JOBMANAGER_table_SRAM[i]);
17                     JOBMANAGER_resume(&JOBMANAGER_table_SRAM[i]);
18                     break;
19                 case SUSPEND:
20                     JOBMANAGER_table_SRAM[i].state=LOAD;
21                     JOBMANAGER_ready(&JOBMANAGER_table_SRAM[i]);
22                     JOBMANAGER_suspend(&JOBMANAGER_table_SRAM[i]);
23                     break;
24                 default :
25                     break;
26             }
27         }
28     }
29 }
```

- JOBMANAGER_save: copy the volatile jobtable from SRAM to FLASH
- JOBMANAGER_create: assign a slot to a job in the jobtable
- JOBMANAGER_free: search a free slot in the jobtable
- JOBMANAGER_get: return the pointer to a job from his id
- JOBMANAGER_lookUpTask: return the pointer to a job from his task
- JOBMANAGER_index: return the position of a job in the jobtable
- JOBMANAGER_delete: clear a job

- JOBMANAGER_write: copy text and data of an job into the memories. It writes the text code and init data inside the job region in the FLASH. Then copy the init data into the SRAM to became the data section. At the end fill the bss section with zero.

```
1 void JOBMANAGER_write(Job *job, uint8_t *textinit, uint8_t *
    init) {
2     //write the text and init into FLASH : copy from the
        textinit buffer into FLASH
3     FLASH_writeBytes((uint32_t)job->text,(uint32_t)textinit,
        job->size_text + job->size_bss);
4     //write the data into FLASH : copy the init into SRAM
5     memcpy ( (void*)job->data , (void*)init , job->size_data
        );
6     //write the bss into FLASH ; 0 initialization into SRAM
7     memset ( (void*)job->bss , 0x00 , job->size_bss );
8 }
```

List of functions to manage the state of a single job:

- JOBMANAGER_resume: resume a job
- JOBMANAGER_suspend: suspend a job
- JOBMANAGER_block: block a job
- JOBMANAGER_ready: create and schedule a new task for the job. The task is suspended until the call of the JOBMANAGER_resume function. The entrypoint of the job is at the beginning of text code region.

```
1 void JOBMANAGER_ready(Job *job){
2     if (job==NULL)
3         return;
4     else if (job->state!=LOAD && job->state!=BLOCK)
5         return;
6     job->state=READY;
7     void (*ptrTask)(void) ;
8     ptrTask=((long int *)((uint32_t)(job->text)+1));
9     char name[5]="JOBX\0";
10    name[3]=job->id+'0';
11    xTaskCreate( *ptrTask, (signed char *) name,
        APP_TASK_STACK_SIZE/10, NULL, APP_TASK_PRIORITY, &job
        ->task);
12    vTaskSuspend( job->task );
13    JOBMANAGER_save();
14 }
```

4.4. Remote command interface service

The remote command interface, RCI, is an interpreter of command. One command is mapped to a specific function build in the RCI components. The architecture is flexible and scalable to support dozens of functions and allow to write ad-hoc functions. In same application domains can be useful specified some instructions to run on the machine at request mode. The purpose of RCI is to provide standard interface for the communication with the network. It is possible view and change the internal state of the system and the installed jobs. The interface of the Remote Command Interface is very simple and it is composed by three methods:

- RCI_start: set the peripheral registers and parameters to open the channel of communication.
- RCI_parse: merge the bytes from the channel and build the packet.
- USART1_IRQHandler: the handle function to retrieve the interrupt from the network channel. It reads a single byte at time and call the RCI_parse function to rebuild the packet. If the packet is complete, it is send to the dispatcher, the EXECUTER_do function, to read and execute the right command.

```
1 void USART1_IRQHandler(void){
2     if(USART_GetITStatus(EVAL_COM1, USART_IT_RXNE) != RESET)
          {
3         /* Read one byte from the receive data register */
4         byteRead = (USART_ReceiveData(EVAL_COM1)); //& 0x7F);
5         if ( RCI_parse(byteRead) == 1) {
6             // Packet Accepted EXECUTOR_do(&executorCmd);
7         }
8     }
9 }
```

We implement the following operations:

- **reset**: restart the system; the function calls the reset procedure in the Boot & restart service.
- **upload**: insert a new job; the function calls the loader service to upload and install the new job.
- **download**: retrieve a job; the function calls the Job manager and replies with the code of the job
- **run**: execute a specific job installed on the machine; the function calls the Job manager.
- **top**: print the list of job and them status; the function calls the Job manager.
- **kill**: suspend a job; the function calls the Job manager to suspend the job with the specific job id.

4.4.1. Communication protocol

RCI provides the methods to encapsulates the information and create a packet for the transmission. The same protocol is used to transmit and receive information.

- **header** (1 byte): the first flag of the packet '0xaa'
- **size** (2 bytes): the dimension of the payload
- **crc** (1 byte): a XOR of all bytes of the packet
- **payload** (n bytes): 1 byte to identify the instruction and Size-1 bytes for the arguments
- **footer** (2 bytes): the end of the packet '0xcc', '0xcc'

The payload 1stbyte identity the operations described in RCI: 0=reset, 1 = upload, 2 = download, 3 = run, 4 = top, 5 = kill.

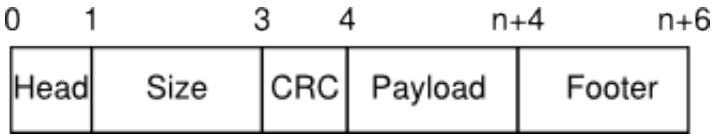

Figure 4.4.: Format of REEL packet

4.4.2. Job payload

The application packet is the binary image of the application with additional information for the static linking procedure. These meta-information define the code and data segment of the application in FLASH and SRAM memory. Moreover an extra field can be used to set the cryptographic signature of the application. This information provides a secure mechanism of reprogramming. The meta-information are the following:

- size and address location of TEXT section in FLASH
- size and address location of DATA section in SRAM
- size and address location of BSS section in SRAM
- size and address location of the initialize values section in FLASH
- cryptographic signature for security verification.

4.5. Exception service

The reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time. [Institute of Electrical

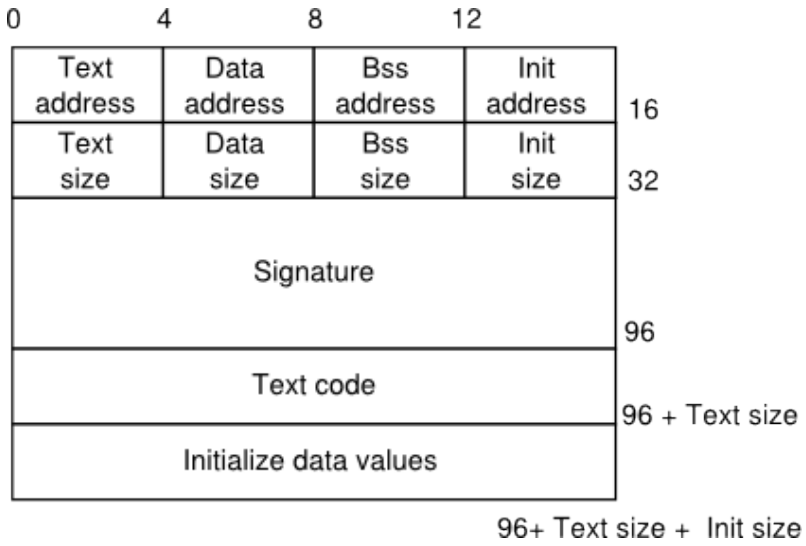

Figure 4.5.: Format of REEL application packet

and Electronics Engineers (1990) IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY ISBN 1-55937-079-3]. In our model a reliable system is a robust system for software faults. A software fault, an exception, is an interrupt generated within a processor by executing an instruction. The exception manager handles the software interrupt to respond the to the occurrence, during computation, of exceptions.

4.5.1. Classification

The list of exceptions to handle:

- **NMI**: A NonMaskable Interrupt (NMI) can be signaled by a peripheral or triggered by software.
- **hard fault**: A hard fault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism.
- **memory management fault**: A memory management fault is an exception that occurs because of a memory protection related fault. The fixed memory protection constraints determines this fault, for both instruction and data memory transactions. This fault is used to abort instruction accesses to Execute Never (XN) memory regions.
- **bus fault**: A bus fault is an exception that occurs because of a memory related fault for an instruction or data memory transaction. This might be from an error detected on a bus in the memory system.
- **usage fault**: A usage fault is an exception that occurs because of a fault related to instruction execution. This includes:

- An undefined instruction
- An illegal unaligned access
- Invalid state on instruction execution
- An error on exception return.

The following can cause a usage fault when the core is configured to report them:

- An unaligned address on word and halfword memory access
- Division by zero

4.5.2. Specification

The exception manager binds the software interrupts to the exception routines. This mapping operation is based on the interrupt vector. The firsts items of the interrupt vector are the functions handler of the interrupt exceptions. Each exception label calls a function that resolve the software faults by a defined politic.

```
1  void (* const Interrupts[])(void) __attribute__ ((section("
       vector"))) = {
2      &_stacktop,  // The initial stack pointer
3      Reset,  // 1 reset handler
4      NMI_Handler,  // 2 NMI handler
5      HardFault_Handler,  // 3 hard fault handler
6      MemManage_Handler,  // 4 MPU fault handler
7      BusFault_Handler,  // 5 bus fault handler
8      UsageFault_Handler,  // 6 usage fault handler
9      ...
10 }
```

Some kind of exceptions, like division by 0 and others, are disabled by default on stm32: the line that throws the exception is ignored and nothing happen. If we want enable all kinds of exceptions, we must set manually the register bit on stm32 cpu. The stm32 library provides several masks to set the register bits of usage fault exceptions:

- **SCB_CCR_STKALIGN_Msk** : Configures stack alignment on exception entry. On exception entry, the processor uses bit 9 of the stacked PSR to indicate the stack alignment. On return from the exception it uses this stacked bit to restore the correct stack alignment.
- **SCB_CCR_BFHFNMIGN_Msk** : Enables handlers with priority -1 or -2 to ignore data bus faults caused by load and store instructions. This applies to the hard fault, NMI, and FAULTMASK escalated handlers. Set this bit to 1 only when the handler and its data are in absolutely safe memory. The

normal use of this bit is to probe system devices and bridges to detect control path problems and fix them

- **SCB_CCR_DIV_0_TRP_Msk** : Enables faulting or halting when the processor executes an SDIV or UDIV instruction with a divisor of 0
- **SCB_CCR_UNALIGN_TRP_Msk** : Enables unaligned access traps
- **SCB_CCR_USERSETMPEND_Msk** : Enables unprivileged software access to the STIR, see Software trigger interrupt register (NVIC_STIR) on page 127 of stm32 manual
- **SCB_CCR_NONBASETHRDENA_Msk** : Configures how the processor enters Thread mode.

The listing shows the configuration function to enable some advanced exceptions disabled by default.

```
1 void EXCEPTIONS_conf (void){
2     SCB_Type   *SCB_DBG;
3     SCB_DBG = (SCB_Type *)
4     SCB_BASE;
5     SCB->CCR = SCB_CCR_DIV_0_TRP_Msk ;
6 }
```

4.5.3. Policies

In the design section we showed a lots of different politics about exception handling. In this section we show the most used politic for a general case. When an exception occurs the routine suspends the faulty job and deletes it from restarting. This politics is realized by setting the state of the job to BLOCK: the execution of the task is stopped by a fault; if only one task gets a fault, all tasks of the job are suspended and at boot time, the job is not restarted. The EXCEPTIONS_do function retrieves the faulty job and sets it to BLOCK.

```
1 void EXCEPTIONS_do (EXCEPTION_Type type) {
2     xTaskHandle *faultyTask = xTaskGetCurrentTaskHandle();
3     Job *faultyJob = JOBMANAGER_lookUpTask(faultyTask);
4     JOBMANAGER_block(faultyJob);
5 }
```

4.6. Loader service

The loader service allows the system to upload a new application into the node. This component implements the reprogramming mechanism with a secure politic. The network packet of the application is received by the RCI and it calls the loader service. The secure reprogramming procedure is composed by 3 main steps:

1. Check the job:
   a) Security mechanism with cryptographic signatures (TPM)
   b) Availability of memory spaces / CPU time
   c) Check the application dependences
2. Install the job:
   a) Set the entrypoint and the meta-information in the JobsTable
   b) Allocate the needed space for the application in FLASH and SRAM memories
      i. Copy the textcode and initialized variables into FLASH
      ii. Copy the values of the variables into SRAM
3. Insert the job: add the new job to the list of available jobs.

```
1  void LOADER_build(uint8_t *RxBuffer){
2     Pkt *pkt; Job *job;
3     pkt=(Pkt *)RxBuffer;
4     /* Create the new job */
5     job=JOBMANAGER_create();
6     job->text = (uint8_t *) pkt->header.address_text;
7     job->init = (uint8_t *) pkt->header.address_init;
8     job->bss = (uint8_t *) pkt->header.address_bss;
9     job->data = (uint8_t *) pkt->header.address_data;
10    job->size_text = pkt->header.size_text;
11    job->size_init = pkt->header.size_init;
12    job->size_bss = pkt->header.size_bss;
13    job->size_data = pkt->header.size_data;
14    memcpy (job->signature, pkt->header.signature,SIGNATURE_SIZE)
          ;
15    /* Write the job in the memory */
16    JOBMANAGER_write(job, (uint8_t *)&pkt->stream, (uint8_t *) (&
          pkt->stream+pkt->header.size_text));
17    /* Insert job into the scheduler */
18    job->state=LOAD;
19    JOBMANAGER_ready ( job );
20 }
```

4.7. Boot and restart service

The boot and restart procedure sets the initial configuration of the system and lunches the kernel. This procedure starts when one of this events happened: push the reset button or receive the reset command. It is composed by the following steps:

1. Initialize the platform:
   a) Initialize system clock
   b) Set the vector table base address at 0x08000000
2. Initialize the SRAM:
   a) Copy initialized system data values from section ".system.init" in FLASH to section ".system.data" in SRAM
   b) Initialize to 0 system bss variables from section ".system.bss" in SRAM
3. Initialize the system:
   a) Initialize the system kernel
   b) Initialize the jobs of the applications
   c) Start the scheduler and the task of the kernel and the jobs

```
1  void Reset(void) {
2      unsigned long *Src, *Dest;
3      // Copy system data variables from section ".system.init" in
           FLASH to section ".system.data" in SRAM
4      Src = &__system_init__;
5      for(Dest = &__system_data__; Dest < &__system_data_end__; ){
6          *Dest++ = *Src++;
7      }
8      // Initialize to 0 system bss variables from section ".system
           .bss" in SRAM
9      for(Dest = &__system_bss__; Dest < &__system_bss_end__; ){
10         *Dest++ = 0;
11     }
12     // Initialize system clock
13     SystemInit();
14     // Set the Vector Table base address at 0x08000000
15     NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
16     // Initialize the system
17     kernel kernel_init();
18     // Initialize the jobs
19     JOBMANAGER_reset();
20     // Start the kernel
21     kernel_start();
22 }
```

4.8. Development tool-chain

In this sections we show the methods to build the system and the applications. The make command starts the compilation procedure composed by:

- configure the REEL base system
- compile and link procedures for REEL base system
- compile and link procedures for an application and build an packet

4.8.1. Configuration script

We provide a small menu script to specify the parameters and components of the REEL base system. We can set a wide range of configurations:

- the path and version of the gnu embedded tool-chain.
- the path of the ST library and the library components to include.
- the FreeRTOS library and the main configuration parameters.
- the list of binary application to include.

Then the compile and link procedure starts automatically.

4.8.2. Compile and link the system

The compilation of Reel is not different from the compilation of standard c program. The make procedure builds the shared object and the elf binary file of the system for the target machine. The link procedure is more complex. We specify a linker script file for the linker to specify the size and the location of the system sections. We assume a FLASH memory composed by 0x80000 blocks (512Kbytes) and a SRAM memory composed by 0xfa00 blocks (64Kbytes).

```
1 /* Target Specific Parameters */
2 MEMORY {
3     FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 0x80000
4     SRAM_DATA (rwx) : ORIGIN = 0x20000000, LENGTH = 0xfa00
5 }
```

The sections in the linker script show the different segments of text, data on the FLASH and SRAM memories:

1. Interrupt vector: the list of all interrupt function handler. It must be the first structure in the FLASH memory. We assign a block of 1 flash page of 2Kbytes.

```
1 .system.vector : {
2     __system_vector__ = .;
3     KEEP(*(vector vector.*))
4     . = ALIGN(0x800);
5     __system_vector_end__ = .;
6 } > FLASH
```

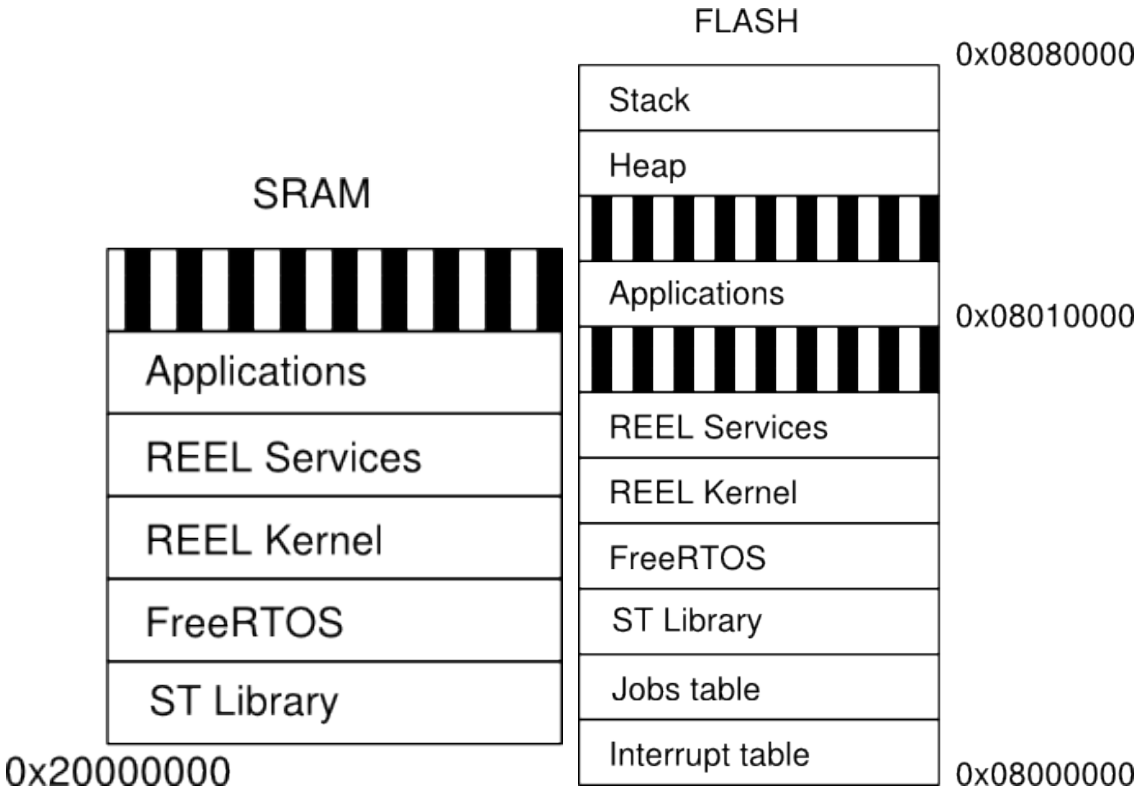

Figure 4.6.: Sram and Flash memories

2. Job table: the list of all job installed on the machine. This data must be stored permanent in the FLASH memory. We assign a block of 1 flash page of 2Kbytes.

```
1  .system.jobtable : {
2      __system_jobtable__ = .;
3     KEEP(*(jobtable jobtable.*))
4      . = ALIGN(0x800); __system_jobtable_end__ = .;
5  } > FLASH
```

3. Text section: the segment of code text of the kernel, libraries and the services.

```
1  .system.text : {
2      __system_code__ = .;
3      ./startup.o (.text .text.*)
4      *(.text .text.*)
5      *(.gnu.linkonce.t.*)
6      *(.glue_7)
7      *(.glue_7t)
8      *(.gcc_except_table)
9      *(.rodata .rodata*)
10     *(.gnu.linkonce.r.*)
11     __system_code_end__ = .;
```

```
12  } > FLASH
```

4. Data section: contains global and static variables used by the program that are explicitly initialized with a value.

```
1  .system.data : AT (___system_code_end___) {
2      ___system_data___ = .;
3      *(vtable vtable.*)
4      *(.data .data.*)
5      *(.gnu.linkonce.d*)
6      . = ALIGN(4);
7      ___system_data_end___ = .;
8  } > SRAM_DATA
```

5. BSS segment: starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

```
1  .system.bss : {
2      ___system_bss___ = .;
3      *(.bss .bss.*)
4      *(.gnu.linkonce.b*)
5      *(COMMON)
6      . = ALIGN(4);
7      ___system_bss_end___ = .;
8  } > SRAM_DATA
```

6. Init section: it contains the initial configuration of constant value and the initialize values for the variables in the data section. When an job is restarted, it starts with its own initial configuration. These values must be stored in a permanent way in the FLASH memory at the end of the text section.

```
1  .system.init : {
2      ___system_init___ = .;
3      . = . + SIZEOF(.system.data);
4      . = ALIGN(4);
5      ___system_init_end___ = .;
6  } > FLASH
```

7. The heap area begins at the end of the BSS segment and grows to larger addresses from there. The heap area is managed by malloc, realloc, and free. The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack.

```
1  .heapstack 0x20005000 : {
2      /* HEAP!!! */
3      _heap = .;
4      *(.heap .heap.*)
```

```
5       . = . + 0x1000;
6       /* reserved for heap */
7       _eheap = .;
8       . = ALIGN(4);
9       *(.stack .stack.*)
10      . = ALIGN(4);
11      _stacktop = 0xa000-4;
12      /* Top of Stack */
13  } > SRAM_DATA
```

4.8.3. Compile and link the job

The compile and link procedure generates from an application source file '.c' the job packet file '.pkt' to send to the network.

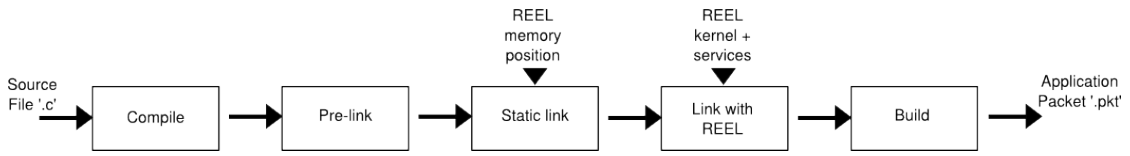

Figure 4.7.: Compile and link chain

This procedure is composed by the following steps:

1. Compile: compile the source code and generate the single object file; the source files can be more than one. The make command lunch the compilation of all applications.

2. Pre-link: regroup and rename the sections of the object file. To regroup all the text sections into only one we used a ldscript with the following sections:

```
1  MEMORY {
2      FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 0x80000
3      SRAM_DATA (rwx) : ORIGIN = 0x20000000, LENGTH = 0xfa00
4  }
5  SECTIONS {
6      .text : { *(.text .text.*) } > FLASH
7      .data : AT ( ADDR(.text) + SIZEOF(.text) ) { *(.data .
           data.*) } > SRAM_DATA
8      .bss : { *(.bss .bss.*) } > SRAM_DATA
9      .init : { . = . + SIZEOF(.data); } > FLASH
10 }
```

   Then we rename the previous sections of the application, as «app_name».«section». We want distinguish sections of different application with the same name.

```
1 ${OC} --rename-section .text=.$name.text main.out ;
2 ${OC} --rename-section .data=.$name.data main.out ;
3 ${OC} --rename-section .bss=.$name.bss main.out ;
4 ${OC} --rename-section .init=.$name.init main.out ;
```

3. Static link: calculate the static address of allocation of all the sections of the application binary file; retrieve the size of the previous installed application and the size of the sections of the current application.
4. Link with reel: resolving the symbol table with the absolute address of the functions of kernel and libraries. The main steps are:
   a) Link the application object with all the libraries and source of REEL in a static way
   b) Extract the text and data sections of the current application from the full system binary

```
1 ${LD} -o main.tmp main.out $flag_app --whole-archive -
    T${LDSCRIPT_KERNEL} ${LFLAGS}
2 ${OC} --only-section=.$name.text main.tmp text.tmp
3 ${OC} --only-section=.$name.data main.tmp data.tmp
4 ${OC} -Obinary text.tmp text.bin ${OC} -Obinary data.
    tmp data.bin
```

5. Build: generate the packet to upload the current application into the WSN node. This packet follows the communication protocol rules and can be send directly over the network.

5. Evaluation

In this character we describe the dynamic behavior of REEL system in a real context. REEL allows to load and execute remotely a new application through the static linking reprogramming method. The more applications we install, the more functionalities the system has. We show a concurrent scenario where two applications works in parallel. Using the results from the measurements, we can quantitative compare the memory and time overhead to calculate approximations the consumptions. We detected the measurements on the STM3210E-EVAL board through an high precision oscilloscope without using a simulator.

5.1. Concurrent applications

The main purposes of the sensor node are to acquire measurements, locally process them to extract information and transmit the processed data to the gateway node over the network. These operations describe the cycled behavior of an sampling application. The more sensors there are, the more measurements we have. The REEL node support more concurrent application for sampling data at different frequency rate.

In the figure we show the parallel execution of two applications for blinking at the frequency of 500Hz. In specify the first application turn up and down an GPIO peripheral, the green line. On contrary the second application is drawn in yellow color. The two applications are executed in parallel as two different REEL jobs. The preemptive behavior is a property of FreeRTOS scheduler.

5.2. Energy Consumption

In a WSN node the main critical component for energy consumption is the wireless radio. To reduce the energy consumption, an efficient reprogramming technique minimizes the data exchanged over the network. For example we test the implementation of the Blink application. We can make only a quantitative comparison between the REEL and other reprogramming solutions:

- REEL static linking: the size of a REEL packet grows of 128Byte for meta-information: linking memory address, signature.

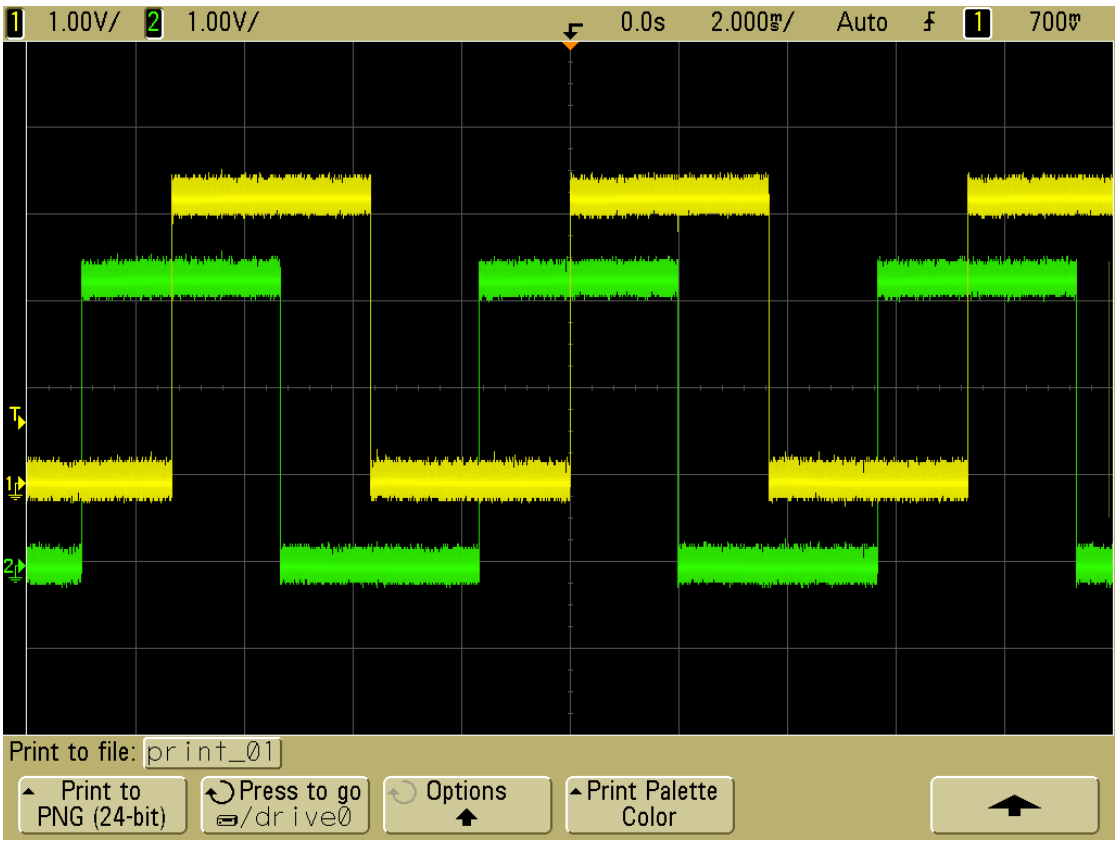

Figure 5.1.: Concurrent blink applications

- complete reprogramming: The size of blinker application including the operating system, FreeRTOS and ST Library, is about 55KB which is about 1100 times the size of the blinker application itself.
- Virtual Machines: the packet size is the dimension of the function call into application library. The famous Virtual Machine solution are based on Java. The bytecode .jar file introduces a lots of memory overhead. In specify Darjeeling uses .di file format which uses no compression but eliminates all string literals.
- Dynamic linking: the packet size of blinker application is due to the length of the application plus the references and the symbol table.

5.3. Memory Consumption

Memory consumption is an important metric for sensor nodes since memory is a scarce resource on most sensor node platforms. The netbrick nodes feature only 512 KB FLASH for program code and 64KB SRAM for data variables. The FLASH memory is grouped in blocks of 2Kbytes. The less memory required for the system

	size
nature code	48B
REEL packet	176B
Java bytecode	856B
Darjeeling VM	146B
ELF Contiki	1056B
CELF Contiki	361B

Table 5.1.: Blink application size for different reprogramming solutions

and services, the more is left for user applications. Each application is stored in block of 2KB Flash memory. We discuss the memory requirements for kernel space, services and user applications.

5.3.1. Kernel space

The kernel space is the unchangeable structure of the system. It is composed by the FreeRTOS, the ST Library and a kernel interface to lunch the task and initialize the system. The most expensive memory consumption is due to the ST library. In our test, we are interest to support a lots of functionalities: more functionalities we add, more functionalities we can reuse in our applications. On the other hand when we insert functionalities to the library, we increase its memory size. We need a trade-off between memory consumption and number of ST functionalities. The make procedure allow a user to select the ST library functionalities to compile in the system.

Moreover the FreeRTOS scheduler is quite bigger, but we can be change a lot of configuration and memory space reserved for each thread. However for FreeRTOS it is not possible decrease the text region under the 2KB of Flash memory. Moreover the internal stack of each task is defined in region on SRAM memory and not in stack system segment. In other words we can change the the dimension of consumption of FreeRTOS as we want.

	Flash	*Sram*
Kernel	936	0
FreeRTOS	12.7K	7.6K
ST Library	43.9K	156
Services	3.9K	2.5K

Table 5.2.: Flash and Sram memory consumption for kernel space.

A more complex problem is to adapt FreeRTOS memory scheme with privileged and restricted regions through MPU component.

5.3.2. Services

The services are based on the kernel space and provide the reprogramming methods and other interesting features. We presented these service and their functionalities in the Implementation chapter. In specify the Jobs manager services consumes 2K of Sram memory, the same amount of memory to store the Jobs table. The jobs manager generate a volatile structure in Sram, a buffer, to store temporarily the Jobs table.

	Flash	*Sram*
Exceptions	176	0
Jobs manager	1988	2K
Loader	176	0
Remote cons.	1584	56

Table 5.3.: Flash and Sram memory consumption for services.

5.3.3. Applications

The memory segment of the application takes place between the service and heap regions. Each application is paged in block of 2KB Flah memory because this memory can be rewritten only in single block of 2KBytes, although the binary image of the application requires less resources. The binary image of the application is composed by the text sections and the initialize values. These information are saved permanently in the Flash memory region of the application.

	Binary text + init values	Binary data	*Flash*	*Sram*
Blink	46	0	2K	0
Timer	540	0	2K	0

Table 5.4.: Flash and Sram memory consumption for applications.

5.4. Execution Overhead

The execution overhead is the time due to the REEL for:

- reprogramming time: to install an application on the node.
- booting time: to initialize the REEL's services.

5.4.1. Reprogramming time

The reprogramming tecnique introduces a time to install an application on the note. The reprogramming process begins when the packet of the application is received and ends when the application is ready to run. In the figure, the green line show the required time to install an application, 109,926ms. It is composed by the following steps:

- add a new entry in the job table,
- copy the binary image into Flash memory, the yellow line
- create a new FreeRTOS task and save the job table into Flash memory,

REEL uses the static linking technique and doesn't introduced additional overhead for resolving symbols and references.

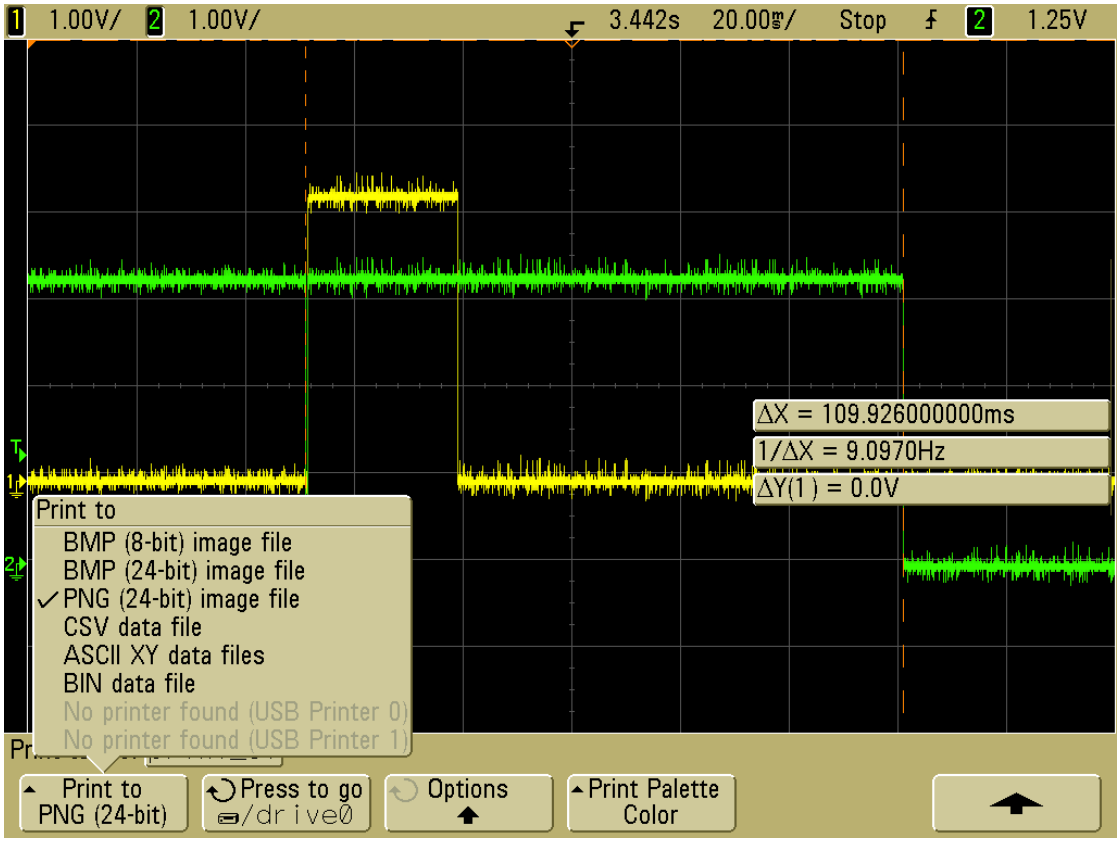

Figure 5.2.: Reprogramming time

5.4.2. Booting time

In the figure we show the time period for booting of REEL. The booting process begins when the device is turned on for the first time or is re-energized after being

turned off, and ends when the device is ready to perform its normal operations. It takes 8.034ms.

In the figure the green lane shows the boot time period, 8.034ms, composed of the init operations: setting the interrupt table, initialization of clock, peripheral, SRAM memory and REEL services. The yellow lane shows the time period, 1.249ms, to initialize the REEL's service components. The different between the two line, 6.785ms, is the indipendent-application time for setting the platform and 1.249ms is the time overhead due to REEL during the boot procedure.

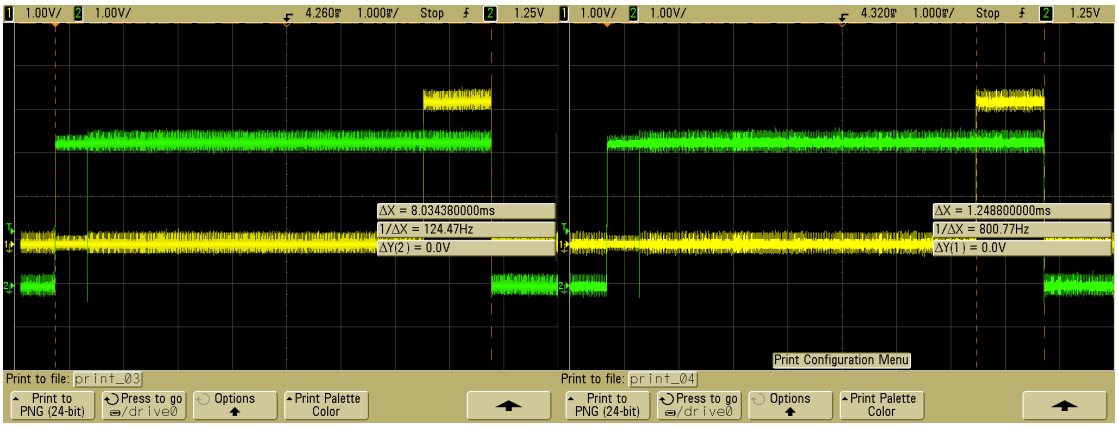

Figure 5.3.: Booting time

5.5. Results

In the table we present a summary of a quantitative comparative between different reprogramming solutions. In specify the static linking tecnique of REEL requires less complexity and memory overhead than a virtual machine or dynamic linking solution. In term of performance binary code is compiled for the target machine and doesn't require any interpreter or address resolving. Moreover we can transmit only the application binary image with a little overhead for meta-information.

	memory overh.	execution	transmission	Reprogramming
Full image replacement	no	native	the full system	yes
Virtual machine	many	interpreter	function calls	limited instruction
Dynamic linking	medium	native	app + symbols	applications
Static linking (REEL)	less	native	app	applications

Table 5.5.: Quantitative comparative of reprogramming methods

We can't quality compare Contiki, because it use a different hardware platform. ESB is equipped with an MSP430 micro-controller with 2 kilobytes of RAM and

60 kilobytes of flash ROM, an external 64 kilobyte EEPROM, as well as a set of sensors and a TR1001 radio transceiver. However we can quantitative compare the measurements which doesn't depend on execution time.

	REEL	Contiki
table	2KB	4KB
blink size	176B	361B

Table 5.6.: REEL vs Contiki

6. Conclusions

We have presented the design and the implementation of REEL framework, a complete real-time reliable and reprogrammable framework for hybrid sensor networks. We discuss in the Evaluation section the memory overhead, the energy consumption, the execution time of our solution and made a comparison with other reprogramming methods. We showed that REEL is a good trade-off between energy consumption and memory overhead without a decrease of performance. The static linking method allows to execute native code, that it is more efficient than a virtual machine, and reduce the transmitted information due to the symbol table in dynamical linking reprogramming. The only requirements is to exactly know the memory free regions on the remote node. For this purpose, REEL provides a remote command interface with additional remote functionalities.

The presentation of the full system, it can be comparable with the solution proposed by other systems in the literature. The future works can be in two directions:

- Robust execution context for the applications: Memory fault handling mechanism with MPU.
- Security reprogramming method: Security mechanism with TPM cryptographic signatures.

A more accurate memory fault handling mechanism uses the Memory Protection Unit, MPU, component inside the processor. This is a simple and fast unit designed for special purpose systems, as opposed to the Memory Management Unit, suitable for general purpose designs. Using a Memory Protection Unit (MPU) can protect applications from a number of potential errors, ranging from undetected programming errors to errors introduced by system or hardware failures. The MPU can be used to protect the kernel itself from invalid execution by tasks and protect data from corruption. It can also protect system peripherals from unintended modification by tasks and guarantee the detection of task stack overflows.

Another point of interest is the security of the application over the network. This security feature checks the authenticity and the integrity of the application packet with cryptographic signatures. A secure crypto-processor, Trusted Platform Module TPM, offers facilities to remote attestation and sealed storage.

A. Installation manual

This section guide the developer to install the framework of REEL on Linux. Unlucky there is not a complete package with all our requirements, so we need to build our framework. The framework of REEL requires the following programs:

- Eclipse development environment and plugin
- Gcc arm tool chain for embedded system
- OpenOCD on-chip debugging
- menu script tool

Figure A.1.: Open Source development software chain

Moreover we need to download the low-level devices and kernel libraries for REEL. The libraries requirements are:

- ST Microelectronics peripheral library
- FreeRTOS operative system

In the following we described the installation procedures for each programs and the download procedure of requirement libraries.

Arm tool-chain

The ARM toolchain consists of a compiler, linker, library and debugger for embedded ARM devices. The main ARM toolchain are:

- GNU ARM toolchain for CygWin, Linux and MacOS : www.gnuarm.com
- Yagarto, yet another GNU Arm toolchain: www.yagarto.de
- Sourcery G++ Lite, Mentor Graphics: www.mentor.com

We test ARM EABI Lite version of the CodeSourcery G++ development tools. The steps to install the tool chain are:

1. Download the IA32 GNU/Linux Installer from: http://www.codesourcery.com/sgpp/lite/arm/portal/subscription3053
2. To launch the graphical installer use this command from the command line: sh arm-2011.03-42-arm-none-eabi.bin
3. Remember to add the bin directory path of the toolchain to the path of the system: export PATH=$PATH:/toolchain_path/bin

OpenOCD

Open On-Chip Debugger, OPENOCD, is a free and open on-chip debugging, in-system programming and boundary-scan testing (tested with 0.5 stable and 0.6 unstable version). The easier way to install OpenOCD is to use a package environment of your Linux distribution, the command is: apt-get install openocd.

We suggest to install the newest version (0.6) but if there is no package, you can compile it from source; no panic in a Linux environment this should be the rule. You can download the source files from the SourceForge project page: http://sourceforge.net/projects/openocd/.

Eclipse

To install Eclipse in your system is as easy as to download and unzip the right distribution. If you don't known witch version to download choose the Eclipse IDE for C/C++ Developers as highlighted in the following picture. The CDT Project provides a fully functional C and C++ Integrated Development Environment based on the Eclipse platform. We tested the system with eclipse Indingo and Juno version that can be download at http://www.eclipse.org. We recommend to download and install the Zylin Embedded CDT plugin for eclipse from http://opensource.zylin.com/zylincdt. Zylin Emebedded CDT pu Eclipse ready to use ARM GNU Debugger.

Libintl

The Libintl is used to compile the setting menu script tool that allow to choose the components of the system. To install the library 'libintl', you can install the package 'gettext-lint' from the repository of your distribution. For Debian/Ubuntu distributions, the command is: apt-get install gettext-lint

ST library

To link the standard ST peripheral library for STM32 board in REEL, the steps are:

1. Download the version of 3.5.0 from http://www.st.com/internet/com/SOFTWARE_RESOURCES/SW_COMPONENT/FIRMWA 0x_stdperiph_lib.zip.
2. Extract the zip file and force lowercase of the files: unzip -L stm32f10x_stdperiph_lib.zip
3. Move and rename the folder: mv –rf stm32f10x_stdperiph_lib_v3.5.0 ./lib/libperi

FreeRTOS

To link the FreeRTOS Operative System in REEL, the steps are:

1. Download the version 7.2.0 of FreeRTOS from http://www.freertos.org
2. Extract the zip file: unzip -L FreeRTOSV7.2.0.zip
3. Move and rename the folder: mv –rf FreeRTOSV7.2.0 ./freertos

B. Eclipse settings

1. Compilation:
   a) Check if the complier (usually arm-none-eabi-gcc) is in the path: Project -> Properties -> C/C++ Build -> Environment
   b) Check the following properties in the: Project -> Properties -> C/C++ Build -> Builder Settings
      i. unCheck: Use default build command
      ii. Build command: make
      iii. uncheck: Generate Makefiles automatically
      iv. Build directory: ${workspace_loc:/ReelFramework_trunk/}
2. OpenOcd: Run OpenOCD To use openocd software in debug server mode go to menu: Run -> External Tools -> External Tools Configurations.
   a) Set the following properties:
      i. Location: /usr/bin/openocd
      ii. Working Directory: ${workspace_loc:/ReelFramework_trunk/arch/openocd}
      iii. Arguments: -f openocd_debug.cfg
3. Debug: To set the debug properties go to the project 'ReelFramework_trunk' in : Run -> Debug configurations -> Zylin Embedded debug (Native).
   a) Set the following properties:
      i. Debugger properties: Debugger
      ii. GDB debugger: arm-none-eabi-gdb
      iii. GDB command file:
      iv. GDB command set: standard
      v. Protocol: mi
   b) Initialization: Commands -> 'Initialize' commands

```
target remote :3333
monitor soft_reset_halt
monitor halt
monitor flash write_image erase unlock ./reelframework/
   sources/total.bin 0x08000000
file ./reelframework/sources/total.out
break
main
continue
```

C. Geting started

We show the steps to build the binary image of REEL:

1. Download REEL middleware: You can download the main trunk sources from svn repository with the subversive plugin in eclipse. Alternatively you can download it in the terminal with the command: SVN CHECKOUT SVN+SSH://SVN.CODE.SF.NET/P/REELFRAMEWORK/CODE/TRUNK REELFRAMEWORK-CODE
2. Import project into Eclipse:
   a) Select the menu item : FILE -> IMPORT... -> EXISTING PROJECT INTO WORKSPACE.
   b) Select the root directory of the project : WORKSPACE/REELFRAMEWORK-CODE
   c) Check the inside: project: REELFRAMEWORK_TRUNK
3. Open you application: in the root directory of the project go to the folder APPS and open the folder of your applications. The name of source folder is the application name. You can edit an existing applications or write a new one.
4. Compilation: the make procedure compile the project and all the applications selected in the make. With the ncurses interface you can select the components of the global system and the applications to build.
5. Run: select the Run button in the upper menu. OpenOCD load the system image to the plugged device. Remind to follow the instruction in Appendix B to set configuration of Zylin plugin.

Bibliography

[1] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.

[2] Athanassios Boulis, Chih-Chieh Han, Roy Shea, and Mani B. Srivastava. Sensorware: Programming sensor networks beyond code update and querying. *Pervasive Mob. Comput.*, 3(4):386–412, August 2007.

[3] Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[4] M. Roveri G. Viscardi C. Alippi, R. Camplani. Netbrick: a high-performance, low-power hardware platform for wireless and hybrid sensor network. *IEEE MASS 2012 (the 9th IEEE International Conference on Mobile Ad hoc and Sensor Systems).*

[5] Inc Crossbow Technology. Mote in network programming user reference. 2003.

[6] David Déharbe, Stephenson Galvão, and Anamaria Martins Moreira. Formal methods: Foundations and applications. chapter Formalizing FreeRTOS: First Steps, pages 101–117. Springer-Verlag, Berlin, Heidelberg, 2009.

[7] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455 – 462, nov. 2004.

[8] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM.

[9] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, May 2003.

[10] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of*

the 3rd international conference on Mobile systems, applications, and services, pages 163–176, New York, NY, USA, 2005. ACM Press.

[11] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 163–176, New York, NY, USA, 2005. ACM.

[12] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[13] Jaein Jeong. Incremental network programming for wireless sensors. In *In Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON*, pages 25–33, 2004.

[14] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGOPS Oper. Syst. Rev.*, 36(5):96–107, October 2002.

[15] Joel Koshy and Raju Pandey. Vmstar: synthesizing scalable runtime environments for sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 243–254, New York, NY, USA, 2005. ACM.

[16] Mark D. Krasniewski, Rajesh Krishna Panta, Saurabh Bagchi, Chin-Lung Yang, and William J. Chappell. Energy-efficient on-demand reprogramming of large-scale sensor networks. *ACM Trans. Sen. Netw.*, 4(1):2:1–2:38, February 2008.

[17] Sandeep S. Kulkarni and Limin Wang. Mnp: Multihop network reprogramming service for sensor networks. In *In Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS*, pages 7–16, 2005.

[18] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.

[19] Philip Levis and David Culler. MatÃ©: A tiny virtual machine for sensor networks, 2002.

[20] Philip Levis, David Gay, and David Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. Technical Report UCB/CSD-04-1343, EECS Department, University of California, Berkeley, 2004.

[21] Philip Levis, David Gay, and David Culler. Active sensor networks. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design*

& Implementation - Volume 2, NSDI'05, pages 343–356, Berkeley, CA, USA, 2005. USENIX Association.

[22] Ting Liu and Margaret Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 107–118, New York, NY, USA, 2003. ACM.

[23] William P. Mccartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Conference On Embedded Networked Sensor Systems*, pages 167–180, 2006.

[24] Christopher Nitta, Raju Pandey, and Yann Ramin. Y-threads: Supporting concurrency in wireless sensor networks. In PhillipB. Gibbons, Tarek Abdelzaher, James Aspnes, and Ramesh Rao, editors, *Distributed Computing in Sensor Systems*, volume 4026 of *Lecture Notes in Computer Science*, pages 169–184. Springer Berlin Heidelberg, 2006.

[25] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Efficient incremental code update for sensor networks. *ACM Trans. Sen. Netw.*, 7(4):30:1–30:32, February 2011.

[26] R.K. Panta, I. Khalil, and S. Bagchi. Stream: Low overhead wireless reprogramming for sensor networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 928 –936, may 2007.

[27] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A.F. Harris, and M. Zorzi. Synapse: A network reprogramming protocol for wireless sensor networks using fountain codes. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, pages 188 –196, june 2008.

[28] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[29] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):8:1–8:29, April 2008.

[30] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[31] Yang Yu, Loren J. Rittle, Vartika Bhandari, and Jason B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 139–152, New York, NY, USA, 2006. ACM.