



NTNU – Trondheim
Norwegian University of
Science and Technology

Internal Wireless Bus for a CubeSat

Jordi Frances Matas

Embedded Computing Systems

Submission date: July 2013

Supervisor: Bjørn B. Larsen, IET

Co-supervisor: Roger Birkeland, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Til de som har vært tålmodige med meg mens jeg jobbet med masteroppgaven min

Abstract

NTNU hosts NUTS (NTNU Test Satellite), that is mainly envisioned as an educational satellite where most tasks are performed by students while supported by university staff.

The scientific payload of the mission is an IR camera that enables the study of gravity waves. But there is a side goal of developing new technical solutions for small satellites.

One of these new solutions is the use of radio modules for the internal communications of the satellite. This master's thesis deals with the design and implementation of a proof of concept for such a wireless network.

There are some advantages to the use of a wireless intra-satellite bus including: lower costs (both economic and in weight) and the possibility to have several transmissions in parallel. The latter could be attained by the use of virtual channels (or similar solutions) that most vendors provide on their radio kits. A proper exploitation of such features would significantly increase throughput while not requiring for additional hardware.

The solution is based on the commercially available nRF24L01 and is envisioned as complementary to the I2C bus that is also present in the satellite. At the same time the project hopes to set the necessary groundwork so that, in a future, the sole use of short range radio transceivers is an option both for future satellites of NTNU or those built elsewhere. Using COTS parts and freely available software and tools enhances this aim of *opening* new possibilities to other projects as well as our own. The implementation is kept as hardware independent as possible, thus deploying it on other satellites should be relatively effortless.

The integration of this new interface with CSP is also considered. Using CSP as a logical layer on top of the radio links.

It is therefore an eminently practical master's thesis but it keeps an spirit to experimenting to discover the possibilities of this novel link

Sammendrag

NTNU huser NUTS (NTNU Test satellite), som hovedsaklig er en utdannings-satellitt. Hvor de fleste oppgavene utføres av studenter, under veiledning og støtte fra ansatte ved universitetet.

Den primære vitenskaplige hensikten med NUTS er bringe et infrarødt-kamera som muliggjør forskning på gravitasjon bølger. Sekundær hensikten er å utvikle nye teknologiske løsninger for små satellitter.

En ny teknologisk løsning er bruken av radio moduler for intern kommunikasjon på satellitten. Denne master oppgaven fokuserer på designe og implementere en prototype for intern trådløs databuss kommunikasjon.

Det er fordeler med bruk av en trådløs kommunikasjon. For det første frigjør det vekt-enheter, som muliggjør en lavere kostnad. For det andre tillater trådløs kommunikasjon utveksling av data mellom modulene i virtuell parallell. Denne parallell kommunikasjonen oppnås ved bruk av virtuelle kanaler, eller lignede, som de fleste leverandører inkluderer i sine radio moduler. Riktig bruk vil gi en merkbar ytelses forbedring ved å øke gjennomstrømningen av data uten å kreve mer maskinvare.

Løsningen er basert på en kommersiell tilgjengelig nRF24L01 og forestilt å komplettere I2C databuss som er nåværende på satellitten. Prosjektet håper å gjøre det nødvendige grunnarbeidet, så for kommende satellitter (både NTNUs og de bygget av andre.) kan bruke det som et utgangspunkt for intern trådløs databuss kommunikasjon. Ved bruk av kommersiell tilgjengelige varer, fritt tilgjengelig programvare og verktøy, styrker det nye muligheter for andre prosjekter så vell som våres. Implementasjonen er gjort så maskinvare uavhengig som mulig, med tanke på å ta det i bruk på andre satellitter skal være så enkelt som mulig.

Integrasjonen av dette nye grensesnittet med CSP er vurdert, med tanke på bruke CSP som et logisk lag på toppen av de trådløse datalink-laget.

Denne masteren er preget av å være utøvende, men utforsker muligheten for en intern trådløs databuss.

Table of Contents

Abstract	i
Sammendrag	ii
Table of Contents	v
List of Figures	vii
Abbreviations	viii
1 Introduction	1
1.1 Foreword	1
1.2 NUTS	2
1.2.1 CubeSat	3
1.2.2 NUTS Hardware and Subsystems	4
1.3 Previous Work	6
1.4 Problem Description	7
1.5 Aims and Goals	7
2 Proposed Solution	9
2.1 Cutting the Cord	9
2.2 Protocols	12
2.2.1 Deciding on a Protocol	12
2.2.2 Two Networks on One Adapter	13
2.3 Real Time	15
2.4 Software Stack	15
2.5 Two Interfaces on the Same Satellite	18
2.6 Summing up	18
3 Technical foundations	21
3.1 Arduino	21

3.1.1	Hardware	22
3.1.2	Software	22
3.2	AVR32	24
3.2.1	Atmel Software Framework	24
3.3	nRF24L01	24
3.3.1	Operating Principles	26
3.4	SPI	28
3.5	FreeRTOS	30
3.5.1	Sempahores	30
3.5.2	Queues	32
3.6	CubeSat Space Protocol	32
4	Implementation	35
4.1	Development Setup	35
4.2	Hello World	36
4.3	Interfacing with the radio transceivers	37
4.3.1	Physical Interface	37
4.3.2	nRF24L01 Driver	39
4.4	First Transmissions	42
4.4.1	Ping	43
4.5	Time for FreeRTOS	43
4.5.1	Bringing FreeRTOS In	43
4.5.2	Using FreeRTOS	44
4.6	Interactive Console	45
4.6.1	Deploying the Console	46
4.6.2	Running the Console	49
4.7	Getting interrupted	51
4.7.1	Listening in	51
4.7.2	Speaking out	53
4.7.3	Receiving a Data Packet	55
4.7.4	Sending a Data Packet	57
4.8	CubeSat Space Protocol	59
4.9	Implementation Outcome	60
5	Integration Guide	61
5.1	Programming Considerations	61
5.1.1	Basic Integration Requirements	61
5.1.2	Skeleton of the Main Program	64
5.1.3	FreeRTOS Tasks' Stack Size	65
5.1.4	Endianness	66
5.1.5	Include Paths	66
5.1.6	Available Functions	66
5.2	Software Design	66
5.3	Software Design Document	66
6	Testing and Results	69

6.1	Testing	69
6.2	Results	70
7	Conclusion	73
7.1	Future Work	73
7.2	Problems Encountered	74
7.3	Learning Outcome	75
	7.3.1 5th European CubeSat Symposium	76
7.4	Personal Experience	76
	Bibliography	77
	Appendix	79

List of Figures

1.1	Artistic rendering of NUTS.	3
1.2	P-POD launch mechanism and mock 1U CubeSat	4
1.3	Current prototype of the OBC board [9]	5
1.4	Current prototype of the backplane [9]	6
2.1	WiFi Module for Arduino, an <i>embedded</i> port of a <i>high end solution</i>	10
2.2	nRF24L01 (one of its variants), a purely <i>embedded</i> solution	11
2.3	Schematic representation of <i>Two networks on one adapter</i>	14
2.4	Software Stack	16
2.5	Communications infrastructure overview	19
3.1	Arduino Duemilanove [1]	22
3.2	Arduino IDE	23
3.3	nRF24L01 State graph [12]	27
3.4	Schematic of an SPI connection [15]	29
3.5	Logical Analysis of an SPI transfer	29
3.6	Representation of a queue	32
3.7	CSP header [14]	33
4.1	Development board connected to the nRF24L01 using the adapter	37
4.2	Left: nRF24L01 (bottom view). Right: built adapter (top view)	38
4.3	Arduino with mounted prototyping shield and nRF24L01	42
4.4	TX operation execution flow	55
4.5	RX operation execution flow.	57

Abbreviations

API	=	Application programming interface
FIFO	=	First In First Out
FPU	=	Floating Point Unit
GPIO	=	General Purpose Input Output
IDE	=	Integrated Development Environment
IP	=	Internet Protocol
IR	=	Infrared
IRQ	=	Interrupt Request
I2C	=	Inter-integrated Circuit
MCU	=	Microcontroller Unit
RX	=	Reception
SPI	=	Serial Peripheral Bus
TCP	=	Transmission Control Protocol
TX	=	Transmission

Introduction

1.1 Foreword

This is a report for a master's thesis. Although this document is meant as a stand alone presentation of the work done, it is important to mention that said work was embedded in a larger NTNU project: NUTS (1.2). Specifically this thesis deals with the development of an internal wireless bus for the communication between different modules of the NUTS satellite. Using an internal wireless bus is a novelty among CubeSats therefore it is desired not only to develop a technical solution but also to study its viability, reliability and performance.

This thesis is eminently a practical work, although the theoretical background has been dully explored. It is done in hopes that it can help further NTNU's CubeSat project's progress and other projects of a similar nature.

This document is divided in chapters. The content of each chapter, ordered as they appear in the document, follows:

Introduction

Presents an overview of this thesis' aims alongside a description of the NUTS project as a whole. Including it's previous status and which improvements are pursued with this work.

Proposed Solution

Describes the solution devised for the intra-satellite communication, on a conceptual level.

Technical Foundations

Contains an overview of the specific technologies that were relied upon to implement the described solution.

Implementation

Exposes the practical implementation of this project.

Porting and integration guide

This chapter is envisioned as a reference guide to help port and integrate this system into the larger NUTS satellite.

Testing and Results

Describes the obtained results.

Conclusion

Presents the conclusions from technical, academic and personal point of view. Including challenges faced during development.

To set things in scope an overview of the NUTS project is here provided. Said overview may, logically, present some overlapping with other reports done within the NUTS project.

1.2 NUTS

The NTNU Test Satellite is a university project that aims to design, build and launch a satellite. It also has a strong educational spirit therefore all tasks are performed by students guided by faculty and staff and, as much as possible, using home-grown solutions developed from scratch in NTNU. The satellite is embedded within the CubeSat project(1.2.1), specifically it is a 2U CubeSat.

From a technical and scientific point of view NUTS also has a dual focus: carrying a payload into space to obtain scientific data and to develop novel solutions in satellite building.

The main payload is an IR camera that will enable the study of atmospheric gravity waves. The study of said waves can provide valuable information regarding the transportation of energy in the upper layers of the atmosphere.

Additionally to the main payload some of the technical approaches that are being used are new to the building of similar spacecraft. Including:

Backplane

The modules of the satellite are interconnected through a backplane, resembling how computer boards are plugged into the motherboard. The backplane provides both power and communication (via an I2C bus) to the subsystems. Many other satellites use a PC104 stacking style to achieve the same effect.

Composite Materials Frame

Whereas most CubeSats use metal for the outer structure and frame, NUTS uses reinforced plastic based composite materials to build a fully custom frame. In hopes of shaving some weight off and thus allowing more weight for crucial systems.

Internal Wireless Bus

Additional the the I2C for intra-satellite communication. NUTS is equipped with an

experimental internal wireless bus. This thesis deals with the development of said bus on a logical and software development level.

Software based memory correction

Instead of using radiation rugged hardware, the satellite will compensate for the hostility of the low earth orbit environment on electronic memories by using a combination of software based solutions [2].

In pursue of this dual spirit NUTS hopes to gather data of scientific value while furthering technical solutions in spacecraft building technique.

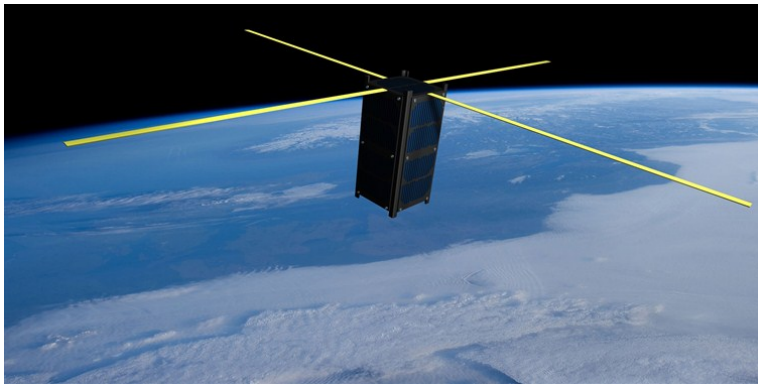


Figure 1.1: Artistic rendering of NUTS.

1.2.1 CubeSat

CubeSat [13] is a type of nano-satellite, that is a very small satellite. The first specification was published by California Polytechnic State University (Cal Poly) and Stanford University as means for universities to have the possibility to run their own space missions on a constrained budget. Over the years the CubeSat community has grown as they gained popularity although there is still a prevalence of academia-related projects.

One of the most recognizable features of said specification is the limited size of the spacecrafts. Originally they were cubical satellites of 10 cm per side. Newer versions of the specifications have multiplied one of the axis' length thus creating satellites shaped like a square prism where the longer side could be either 10 cm (original), 20 cm and 30 cm. The original size is called 1U and following this pattern the bigger ones are 2U and 3U respectively.

One of the advantages of adhering to the specification is the compatibility with the deployment mechanism: P-POD (Figure 1.2), thus simplifying enormously interfacing with the launch vehicle. In addition to that, CubeSat are easily interchangeable as they all would comply with the same launch vehicle interface, enabling the use of last-minute openings in launch opportunities.



Figure 1.2: P-POD launch mechanism and mock 1U CubeSat

1.2.2 NUTS Hardware and Subsystems

NUTS [9], as many other satellites, is composed of several different modules and subsystems. They interact with each other to provide the most advanced functionalities of the spacecraft.

Subsystems

This section presents a brief overview of some of these subsystems. There is also an inherent hierarchy among them: namely the On-Board-Computer and the radio module have control over all other modules. Even more they provide a certain degree of redundancy to each other. Being the main modules they head the following listing of subsystems.

On-Board-Computer (OBC)

It holds the main processor on the satellite and performs tasks such as housekeeping or payload data processing. It also checks for the status of other modules and can reset or disable them if they are malfunctioning. The processor used is an AVR32UC3A3256 and the board also provides a reasonably large amount of SRAM (16 MB) and Flash (16 GB).

Radio Module

It is the main (and only) means of the satellite to communicate with the ground station. Therefore all the messages to and from the spacecraft will, at one point or another, be relayed by the radio module. Aside from its main functionalities as a communications interface this module, like the OBC, also has the capability to control other modules. Furthermore it could, in the event of an OBC failure, take part of the OBC's duties and keep the satellite functioning albeit not at full capabilities as it has less storage and RAM. In order to keep this redundancy between radio and OBC modules, the radio is also based on the AVR32UC3A3256. Aside from redundancy, having a relatively powerful MCU on the radio allows for more

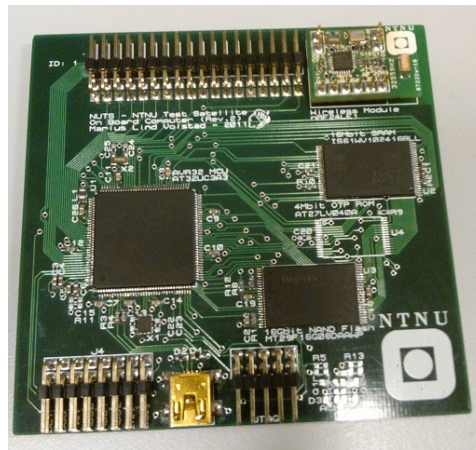


Figure 1.3: Current prototype of the OBC board [9]

autonomous operation in all communications thus the OBC has more resources to perform its tasks.

Both modules are equipped with extra pins on their interface with the backplane. These pins enable to reset or isolate other modules from the power and communication buses if they were being faulty. It is worth mentioning that radio and OBC also check each other for sanity and could reset one another should the need arise. Even though a non-recoverable error in the radio would mean that the satellite is cut off from ground, effectively making the mission impossible to control. Here continues the listing of modules:

Payload - IR Camera

If the OBC is metaphorically the brain of the satellite then the payload would be its heart. The intended payload for NUTS is an IR camera to take images of the upper layers of the atmosphere. These images should enable for the study of gravity waves. The pictures taken by the camera are then transferred to the OBC for processing (or sent as raw data to ground).

Power System

All of the other subsystems require power to work hence there is a specific module devoted to the control, monitoring and distribution of power. The energy is harvested through solar panels and then stored in batteries to be used by the spacecraft.

Attitude Determination and Control System (ADCS)

The satellite requires means to know its orientation and to control it. Also a way to stop the initial, after deployment, tumbling. The ADCS provides this functionalities. It combines magnetometers, gyroscope and the solar panels (as sun sensors) to calculate the current attitude of the spacecraft. It can also actuate to change this attitude through the interaction of the magneto torques with the Earth's magnetic field.

Backplane Based Platform

All the modules described in the prior section(1.2.2) are interdependent thus they require communication among them. Additionally they require power to perform their functions. One of the innovative, and uncommon in CubeSats, solutions NUTS is using is the use of a backplane.

The backplane provides several slots: some of them are generic slots and can house most standard modules whereas others are specially tailored to specific boards. The specialized ones allow access to certain control or power lines for enhanced functionalities such as switching off or resetting other boards as is the case of the OBC and the radio.

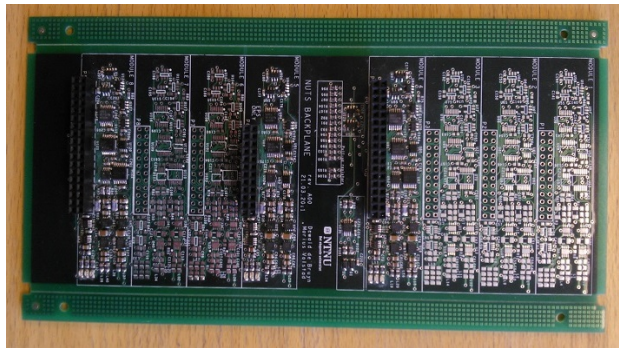


Figure 1.4: Current prototype of the backplane [9]

1.3 Previous Work

Naturally being part of the NUTS project this thesis is grounded on a variety of works also related to NUTS. Two of those works must be specially highlighted because given their nature have been especially used as references.

Some work done previously in NUTS [8] deals with the more purely electrical engineering details of the use of small radio transceivers as means of intra-satellite communication. Paying attention to the hostile environment (due to the presence of many electronic components) that the interior of NUTS is for such small transceivers. His work allows the present master's thesis to deal with the logical aspects of the internal wireless bus and their software implementation.

Also as part of the NUTS project [4], the use of the CubeSat Space Protocol on the backplane's I2C bus was explored. In that work the idea of using two different protocols on the same interface is introduced. A similar approach has been attempted in this thesis.

1.4 Problem Description

NUTS is composed by several modules(1.2.2) and those need to communicate. The original design of the satellite the main interface of internal communication was the I2C bus. It is a very common bus and it is widely used in other CubeSat projects and generally across a multitude of embedded systems. Using I2C clearly has advantages: it is a *de facto* standard and it's reliability has been proven before. On the downside: it has limited bandwidth (400 kB/s) and, obviously, requires wiring across all modules.

While those speeds are enough for most control operations such as the OBC instructing a given module to perform an operation. They could be limited for moving big blocks of data. Since the intended payload of NUTS is an IR camera it is easily foreseeable that big amounts of data will need to be transferred throughout the satellite. For example: to bring the image from the camera board to the OBC for processing.

As I2C was intended as the main interface for intra-satellite communication, all the requirements regarding it's wiring were by allocated in the backplane (lines, repeaters...), all of the requirements regarding space and power have been considered. But it would be interesting to see if alternative solutions can curtail them.

In order to tackle both these limitations the idea of using radio transceivers as a secondary communication interface is introduced.

The use of a radio interface would increase the bandwidth, reduce the area footprint required. And, at the same time, make it more flexible since the radio transponder can be placed anywhere on each of the modules.

Aside from bandwidth and area there is also a desire to experiment. As stated previously (1.2) NUTS aims to the development of novel techniques or approaches in satellite construction. Keeping that in mind, it would be interesting to equip our system with radio transponders to see how well they fare under space conditions. And thus determine if they would be a viable option in future missions. Or even in other settings further from space technology.

The desire to overcome the technical limitations of the current solutions while at the same time exploring other options, drive this master thesis.

1.5 Aims and Goals

This thesis aims at the production of a *proof-of-concept* implementation of all the software required for the use of commercially available nRF24L01 as nodes in an intra-satellite communication solution.

Additionally the developed solution needs not only to be easily integrable in the final NUTS system but also provide a helpful and clear documentation to do so. This document is the central piece of that documentation.

Finally, and not so tightly bound with the specific thesis topic, the author also wishes to be part of NUTS as bigger project than this thesis. Both out of interest for space technology and seeking a real application to an otherwise maybe purely academic exercise.

Proposed Solution

This chapter presents an overview of the solution implemented during this thesis, it does not deal with the details of its implementation as that is done later in the document (4).

The main idea is to *cut the cord*, that is: to provide a wireless interface for intra satellite communication. To solve to the limitations imposed by the I2C wired bus. While keeping a *clean* software interface, so the communication operations are as transparent as possible to the the developers working on other NUTS subsystems.

Naturally, as this is an experimental solution, it would not replace the I2C on NUTS but rather complement it. Therefore all satellite operations should be viable in case only I2C is operational (albeit they might operate more slowly due to bandwidth constraints).

2.1 Cutting the Cord

The first step into developing a wireless interface was to decide which wireless hardware to use. Despite there are plenty of solutions for cable-free transmission in the market, not all of them were viable. For example an infra-red base solution (or any other kind of light based solution) was automatically discarded as they require clear line of sight between transceivers. And that requisite is clearly not met in NUTS, as the way the boards are set on board would prevent such unobstructed line of sight. So it was clear that the solution had to be a radio-based one.

Within radio solutions two different approaches can be found: *embedded* solutions and *high-end* solutions.

High-end Solutions

Generally coming from the PC world they provide very high data bandwidth (in

the order of tens of Mbps) yet they often require interfaces that are uncommon in embedded systems, and use protocols and software stacks that are demanding both in program memory, RAM and computing resources. Some examples are Bluetooth and 802.11b/g (common Wifi)

Despite that was not their original intent it is possible to find these solutions *ported* to the embedded world. Such adaptations usually follow a common pattern: There is a specific processor that handles the costly operations and complex inter facing and at the same time takes commands and configuration through a commonly available interface in embedded systems.

An example of such a *ported* system is the 802.11b/g shield for Arduino (3.1) (Figure 2.1). Where a dedicated microprocessor handles the wireless interface while interacting with it's host system via SPI.

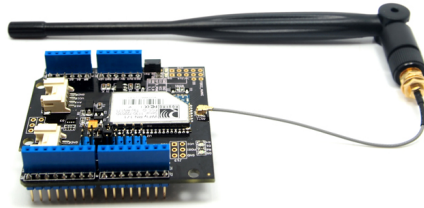


Figure 2.1: WiFi Module for Arduino, an *embedded* port of a *high end* solution.

Embedded Solutions

This branch of solutions are purely embedded, they were designed keeping the embedded world in mind. As such they usually provide less bandwidth or lack advanced features (such as supporting the TCP/IP stack natively) yet, in exchange, they are less demanding to manage both in memory and computing resources. Examples of this approach would be the nRF24L01.

Having considered both possibilities the most logical outcome seems to be the use of an *embedded* solution. First, and foremost, NUTS is an embedded system. Granted that it is a fairly complex one that will be set in orbit, but it is still purely an embedded system. The overhead in size and power to use a *high-end* approach seems too high a burden for our satellite. And the features of a simpler technology suffice for NUTS' needs.

Once settled on the category of radio transceivers to be used for this project, a specific model needed to be picked. Some of the solutions are entire MCUs which radio capabilities whereas others are just radio transceivers with a control unit meant to be interfaced with a host MCU. Since it is intended to integrate the internal wireless bus into the existing NUTS platform it was decided to go for the *hosted* mode.



Figure 2.2: nRF24L01 (one of its variants), a purely *embedded* solution

Taking all of the above into account, our *scissors to cut the cord* are an nRF24L01 (details on 3.3) because it fulfils the requirements. Additionally it also provides some features useful for the implementation of a wireless bus, such as:

Up to 32 bytes Payload

Each transmitted packet entails a certain amount of overhead. The MCU needs to communicate with the radio module and upload the payload to be sent. Then finally give the command to actually send it. By having relatively large payloads (32 bytes) this overhead can be lowered, as large blocks of data can be transmitted with less packets. Additionally the payload size is not fixed, therefore if there is a need to send lesser amounts of data (smaller than 32 bytes), it is possible to resize the packet accordingly, eliminating the need for padding and sending unnecessary data.

SPI Interface

It's interface (SPI) is readily available in most MCUs. Regardless of them being 8-bit or 32-bit. Practically all MCUs have, at least, one SPI unit therefore this transceiver can be interfaced with diverse hardware solutions. Although most of NUTS is based on AVR32. It cannot be discarded that some subsystem or module could be based on a different architecture. But it would, most certainly, have an SPI unit therefore it would still have access to the wireless internal bus.

Auto-ACK

The transceiver can autonomously notify another unit that it successfully received a packet, thus loading off from the MCU the cost of checking for successful transmissions through other mechanisms.

Up to 2 Mbps

It has several bitrates for on-air transmission. They are generally a trade-off between power and bandwidth. The fastest being 2 Mbps, significantly faster than the wired I2C (400 Kbps). This feature allows to solve the bottleneck that the wired bus imposes on the system.

It might seem counter intuitive to use SPI (4 lines) as an alternative to I2C (2 lines) when precisely trying to eliminate the cables to implement a wireless solution. But the SPI

connections are shorter, and stay always within each submodule, they are not extended throughout the whole system. Thus, keeping a flexible and affordable (in terms of footprint) profile.

Once decided on which transceivers are to be used for the interface it is important to set a protocol to enable the communication.

2.2 Protocols

A Protocol could be roughly defined as: *how* should the information look like. In order for any communication to work all nodes need to stick to the set of rules concerning syntax, semantics and synchronization defined by a protocol.

It is important to clarify that the nRF24L01 already has it's own low level protocol. It handles timing, collisions, acknowledging reception. . . Therefore while on-air the data will always be under this radio protocol. Therefore the discussion about protocol options on this section concerns the information up to the point when it is broadcasted to air and, later, right after the message has been received and handed to the target MCU.

An analogy to this duality nRF24L01 internal protocol and user-define protocols on top of it can be found on the internet. Where TCP/IP (mostly) takes care of the logical levels of the communication whereas the lower levels are handled by several different protocols depending on which technology (ADSL, Cable, WiFi...) is being used.

2.2.1 Deciding on a Protocol

The usual trade-off on protocols is overhead against features. Functionality tends to grow alongside resources (both memory footprint and computational requirements). Therefore we decided to explore both extremes of this trade-off:

CubeSat Space Protocol

The CubeSat Space Protocol (more info at [5]) is, naturally, a communications protocol aimed at CubeSat. Despite it was originally intended for use on CAN interfaces it has later been ported to other interfaces such as USART or I2C. Specifically the I2C port was done previously [4] as part of the NUTS project. The protocol offers most of the advanced functionalities of modern day protocols: Socket API, routing, connection oriented and connection-less modes...

It's advantages are the advanced functionality, leaving most of the effort for routing and other control operations to the CSP libraries instead of the applications itself. Resulting in simpler software development.

Against it's use there are arguments regarding simplicity. Is it really necessary to have a full blown TCP/IP -like protocol for NUTS, that will not have post deployment modifications. Therefore all the dynamic routing features might not be needed. This overcomplexity entails , aside from greater resource usage, a major overhead.

RAW

Raw is the name picked to illustrate a very basic protocol, almost lack of one. Yet it is not any *official* naming. The data is sent precisely raw, without processing. Therefore the responsibility to route it (decide to which radio transceiver it is sent) is entirely the responsibility of the applications.

Granted that it lacks the advanced functions of CSP and increases the complexity of applications but on the other hand it has a lower RAM consumption and a negligible overhead.

When the project was first envisioned, I thought that these two solutions would be mutually exclusive and therefore one would have to be picked over the other. Yet during the development I realized that both could be used simultaneously.

2.2.2 Two Networks on One Adapter

One of the features of the nRF24L01 is the possibility to set up multiple receive addresses in one single transceiver (3.3).

By making use of this it is possible to turn each physical adapter in two (even up to six) virtual adapters. Then, each module's nRF24L01 can have 2 addresses: one for RAW and one for CSP traffic. When transmitting the process is not altered at all, simply the packet is sent to a given address depending on the protocol intended for use and the targeted submodule. Then on reception, the receiving MCU would store the packet in one buffer or another depending on which of it's addresses the packet was sent to. Finally the receiving applications would get it from the appropriate buffer depending on what protocol they were using.

That way both protocols are kept, giving greater flexibility to the system.

Figure 2.3 depicts this scheme.

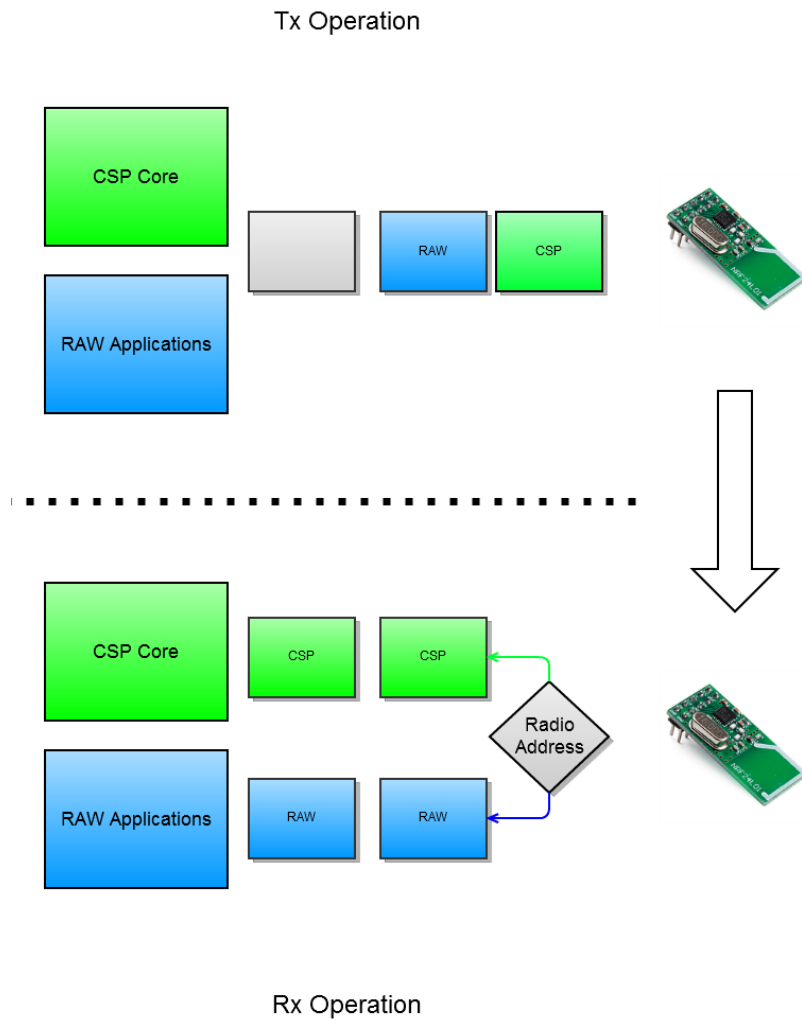


Figure 2.3: Schematic representation of *Two networks on one adapter*

The color coding on the figure has the following meaning:

Blue:

RAW packets, or RAW-specific software components.

Green:

CSP packets, or CSP-specific software components.

Grey:

Empty positions in the messaging queues.

2.3 Real Time

NUTS, regardless of which communication interface is in use, performs several tasks at the same time. All those tasks need to meet their deadlines while balancing the available resources and their respective priorities. In such an environment the communication stack cannot be blind to such requirements.

It is therefore very important to guarantee that all operations using the internal wireless bus is as fast as possible. But it must also be non-blocking and as detached from the applications as feasible. Since the beginning it was clear that the software supporting the Internal Wireless Bus would have to be integrated with FreeRTOS (3.5). Despite during the early phases of the design contemplated such integration simply due to FreeRTOS being already set as part of the NUTS software, it late became clear that some of the synchronization and data structures that FreeRTOS provides were very useful to the used implementation (4).

From a practical point of view this implied the use of interrupt-driven (4.7) operations and queued messages (3.5.2). Therefore the applications interact with these queues (both to transmit and receive data) through the FreeRTOS API. This made accessing the internal wireless bus a simple memory access (from the applications' perspective) both for sending and receiving data:

Sending data (TX):

The application fills a data structure and simply queues it on the TX queue. Then there is a task (blocked when the queues are empty) that will forward such message to the radio and give the command to send it.

Receiving data (RX):

The radio transceivers are, when not actively transmitting, listening for possible incoming data packets. If a packet is received then the radio transceiver requests an interrupt to it's host MCU. When the MCU is servicing the interrupt it will get the received packet and queue it on the RX queue. Therefore the application that requested said data can simply read it from the queue. If, at the time the application requests the read, there are no newly received packets (empty queue) said application can be blocked so it does not hoard resources that could be used for other tasks.

Said scheme would work both on CSP and RAW traffic, as it deals only with packets regardless of their content. The difference is that in the case of CSP the one queueing and reading from queues would be the CSP engine and not the application directly.

2.4 Software Stack

In order to develop a software complying with all the properties described in previous sections yet keeping it modular enough to make development and maintenance more comfortable, a stack-like architecture was envisioned. As a matter of fact such a pattern is

quite usual of communication software. Each software component can usually communicate with those that are adjacent to it. Figure 2.4 depicts the used scheme.

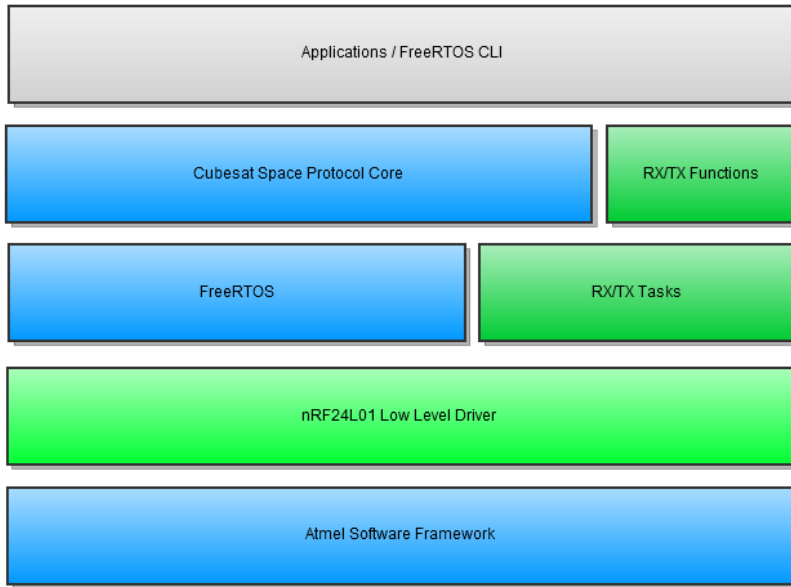


Figure 2.4: Software Stack

The color coding on the figure has the following meaning:

Blue:

Software developed by third parties, the modifications done to it are minimal or non-existent, and only to the extent to make it fit with the rest of the system.

Green:

Software developed from scratch for this project or extensively modified (ported from a different platform) to fit within NUTS. It represents the bulk of the developed code base.

Grey:

Software that uses this stack, yet it is not part of it. Namely: the rest of the NUTS software, performing the tasks that actually need to communicate with one another. During this project, as that software was not readily available (maybe not even developed yet). It was substituted by a console (4.6) (via serial port and built using FreeRTOS CLI) that enabled for interactive use and testing in order to speed up the prototyping process.

Each of the software blocks has its own functionality, a relation of said functionalities and how they fit into the greater system follows:

Applications/FreeRTOS CLI:

Those are the *users* of the communication stack. They are very varied pieces of

software that interact with the stack via the CSP API or the RX/TX functions (*de facto* RAW API). Currently there is an interactive console that has commands for transmitting and receiving as a placeholder for the applications.

CubeSat Space Protocol Core:

Main CSP libraries, they can be seen as a blackbox with a clearly defined API. The developer needs to provide functions bind it to the specific interface (currently the nRF24L01's). The applications can interact with it via the CSP API and it, in turn, interacts with the RX/TX functions to queue the packets for sending.

RX/TX Functions:

They are a pair of functions: one to send and one to receive. Both are built on the FreeRTOS queue system, therefore if the an application requests for data that is not readily available, that application will be suspended by the scheduler until that data is available. Technically they interact solely with the queues and perform only in RAM operations. The RX function can be seen as a consumer of the RX queue whereas the TX is the producer for the TX queue.

RX/TX Tasks:

They are counterparts to the RX/TX Functions as they consume from the TX queue and produce on the RX queue. This tasks are generally dormant unless there are new packets in the queue waiting to be sent or newly arrived packets at the nRF24L01 that require to be queued. Both for transmitting and receiving data it interacts with the low level driver.

FreeRTOS:

It is the operating system of the NUTS project, as such all other software is built on top of it or at least around it. To the specific purposes of this thesis it provides a useful queuing system and the necessary mechanisms for scheduling and synchronizing different tasks.

nRF24L01 Low Level Driver:

Provides functions that act as wrappers for the control commands to be sent to the radio transceivers. The foundations of this module where the nRF24L01 library for Arduino. Although that library was limited in some aspects it was enhanced during the porting to meet the requirements. The main enhancements were done to make it interrupt driven and not as MCU dependant since on the original Arduino code there are active wait loops and polling. Despite it could be directly used by the applications that would break the idea of this stack. It is, thus, mostly used by the RX/TX Tasks. To operate the hardware units on the AVR32 (namely the SPI) it relies on the Atmel Software Framework.

Atmel Software Framework (ASF):

Vendor procured set of libraries to interact with the software units of the processor. It provides an abstraction layer so that during the development it is not necessary to go down to register level configuration of the MCU. ASF code interacts directly with the hardware.

That concludes the overview on the software architecture for this project yet different uses

of the stack are possible, in special how to mesh it together with the pre-existing I2C bus.

2.5 Two Interfaces on the Same Satellite

As pointed out earlier the internal wireless bus will be deployed alongside the existing I2C solution, resulting in two communication solutions on the same satellite. There are two main possibilities regarding how to manage this duality:

Fallback:

This scheme contemplates that only one of the interfaces is operating at any given moment. The system would resort to the other in case the primary one failed. Taking into account that the Internal Wireless Bus is an experimental approach whereas the I2C is a tested and *de facto* industry standard, the internal wireless bus would be the primary interface, because it has more bandwidth than the I2C and in case of failure the I2C would still be there to recover. This approach simplifies software since only one interface is in use at any given but it is wasteful when it comes to resources as it does not use one of the interfaces despite the system supports it.

Cooperation:

Both interfaces run in parallel, even some traffic specialization can be enacted. For example the I2C could deliver small yet time-sensitive messages (such as commands or sensor readings) while the internal wireless bus could take care of bigger data packets (such as those originating on the payload camera or when dumping logs to the satellite-to-ground link). This approach entails greater software complexity as both interfaces are kept running and a given application could use both. For example: the camera could get the command to transmit the last picture by I2C, acknowledge it by I2C and finally send the requested data through the wireless interface. Additionally if one of the interfaces was to fail, the remaining one can still be a *fallback* interface, thus keeping the advantages of the fallback approach.

When considering that the payload will generate big amounts of data we finally settled for a cooperation scheme, additionally it holds the upper hand by keeping the advantages of it's alternative.

2.6 Summing up

To close up and clarify how everything in this chapter fits together, this section presents a general graphical overview (Figure 2.5) of the intended communication scheme:

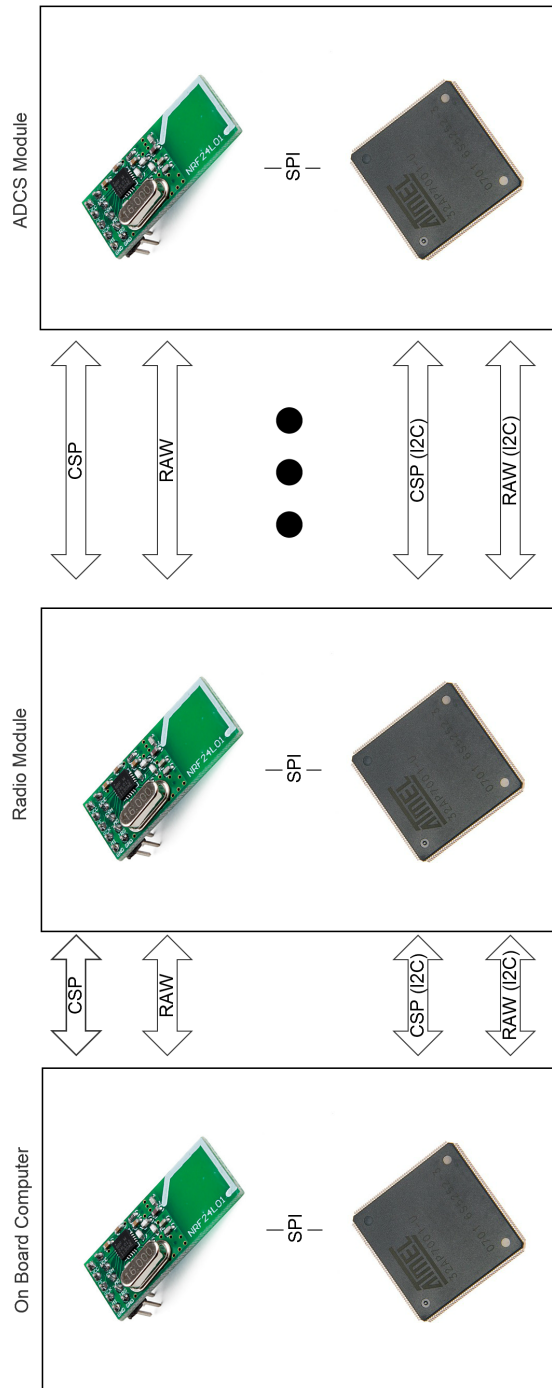


Figure 2.5: Communications infrastructure overview

Basically each module has access to both communication interfaces: I2C and the Internal Wireless Bus. I2C is natively present in the AVR32 (or other MCUs that could be used) whereas the access to the Wireless Bus is done through nRF24L01 transceivers that are, in turn, interfaced via SPI with their corresponding MCU.

Chapter 3

Technical foundations

This chapter includes information on the technologies and techniques that were used during the development of the project. It aims at bridging the gap between Chapter 2 where the solution is presented and Chapter 4 that describes the process of implementing it. Wording differently: here the tools used to go from *what* to *how* are presented. The aim is to provide an overview to better understand the implementation and is never intended to replace the documentation for any of the technologies here described.

3.1 Arduino

Arduino [1] is an open source embedded development platform. It is open source in a wide sense: both software and hardware are open to anyone to use, modify or develop on. It is almost a *de facto* standard for hobbyist embedded systems developers. Such popularity pushes forth the development of many libraries that are freely available and a sizeable on-line community to provide support. These factors made it a choice for fast prototyping in the earliest stages of implementation (4.4).

Figure 3.1 shows the Duemilanove version of the Arduino board, it is the one that was used during this project. There have been several revisions of the boards but they generally keep a common form factor and pinout to ensure compatibility.

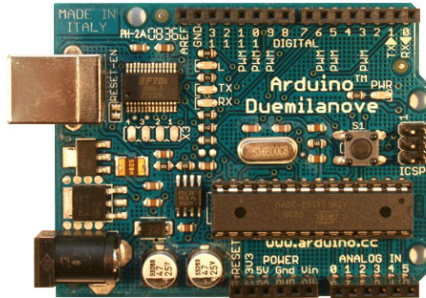


Figure 3.1: Arduino Duemilanove [1]

The Arduino project provides not only the hardware platform but also the accompanying software for it.

3.1.1 Hardware

There is a certain degree of variability among the different board revisions but in general the boards are minimalistic: mounting an MCU and the essentials to have it running (power supply, crystal...). The older boards have an FTDI chip enabling for UART-on-USB functionalities whereas newer revisions have a secondary MCU that can be configured to act as different USB peripherals. The Duemilanove used in this project belongs to the former therefore it was limited to UART-on-USB. This limitation did not hamper development in any way as this was the preferred form of communication between Arduino and PC.

Regarding the MCUs most Arduinos are built 8 bit AVR. Recently the higher end models based on ARM cores have been launched but those are not yet widely used. Specifically the version used in this project relies on an ATmega328 clocked at 16 Mhz. As for memories: RAM: 2 KB and program Flash: 32 KB. It may seem limited but it sufficed for the uses of the board in this project.

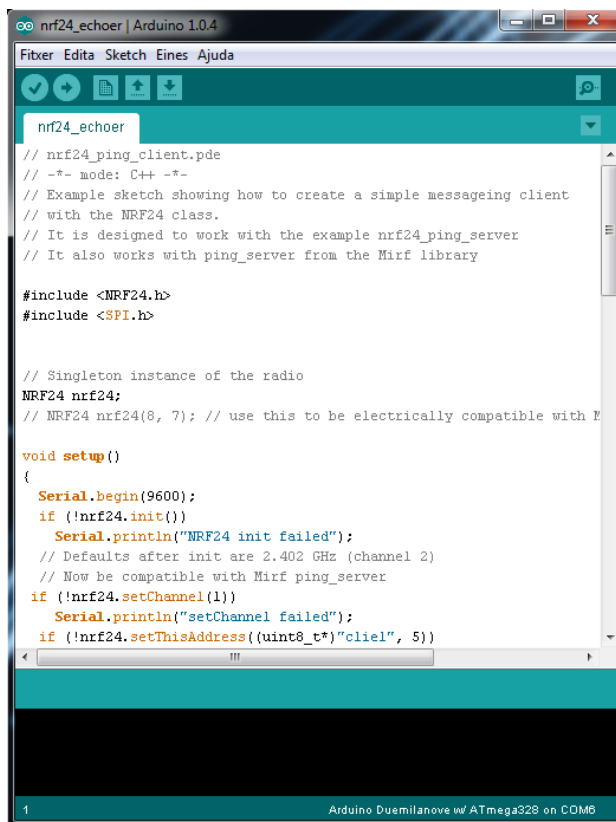
3.1.2 Software

In a sense the Software is the true core of the Arduino project, because the boards are simply a mounted MCU with some of it's pins broken out for easy access. But this does not set it apart from the many other development kits available in the market. The true difference is in the software. It is provided as a pairing of libraries and development environment.

The libraries abstract most of the MCU peripherals and provide clean and easy to use functions that allow for fast development of code. Additionally the libraries homogenize the

diverse boards increasing cross compatibility across Arduino products (always respecting hardware limitations). Additionally to those libraries bundled with the project there is a multitude of libraries made by third parties that can be used as well. One of those libraries is the NRF24 [7], used to manage nRF24L01 transceivers, that has been used in the project. The native version on the Arduino and a ported version on the AVR32.

The development environment (figure 3.2) is tightly integrated with the aforementioned libraries. And integrates a text editor to write the source code with a compiler. Additionally all boards are preloaded with a bootloader that enables users to upload code without a programmer or any specialized hardware, all is done through the USB connection (UART mode). All development is done in *Arduino Programming Language* which is a C/C++ derivative. Essentially it is C++ enhanced with some implicit dynamic memory management. An Arduino program (*Sketch* in their own terms) does not have a `main()` per se but rather an initial function that will be called once: `setup()` and the `loop()` function that is called repeatedly after the initial configuration is done.



```
nrf24_echoer | Arduino 1.0.4
Fitxer Edita Sketch Eines Ajuda
nrf24_echoer
// nrf24_ping_client.pde
// -*- mode: C++ -*-
// Example sketch showing how to create a simple messageing client
// with the NRF24 class.
// It is designed to work with the example nrf24_ping_server
// It also works with ping_server from the Mirf library

#include <NRF24.h>
#include <SPI.h>

// Singleton instance of the radio
NRF24 nrf24;
// NRF24 nrf24(8, 7); // use this to be electrically compatible with M

void setup()
{
  Serial.begin(9600);
  if (!nrf24.init())
    Serial.println("NRF24 init failed");
  // Defaults after init are 2.402 GHz (channel 2)
  // Now be compatible with Mirf ping_server
  if (!nrf24.setChannel(1))
    Serial.println("setChannel failed");
  if (!nrf24.setThisAddress((uint8_t*)"cliel", 5))
    Serial.println("setThisAddress failed");
}

1 Arduino Duemilanove w/ ATmega328 on COM6
```

Figure 3.2: Arduino IDE

On the downside: the Arduino IDE is very limited compared to other IDEs. The performance of an Arduino is significantly lower than when using the MCU plainly due to the

overhead added by the libraries. Both shortcomings of the platform were deemed acceptable for this thesis.

3.2 AVR32

In contrast to the Arduino that holds an interest purely for prototyping, Atmel's AVR32 is the architecture currently used on several (virtually all) NUTS subsystems. That makes it the main target of this project and all developed software is for AVR32.

Specifically on the project an AT32UC3A3256 was used. It belongs to the a subfamily of AVR32 namely the UC3, that include some features to increase their performance. Such as dedicated hardware FPU. Although none of these specific features were used. First, and foremost, they were not used because they were not needed but not doing so also helped to develop a more portable solution.

Development, regarding software, was done relying on the tools provided by Atmel. All those tools were integrated in Atmel Studio that is their IDE and bundles project management, source code edition, compilation, debugging and flashing tools. Additionally the Atmel Software Framework is also provided.

3.2.1 Atmel Software Framework

Configuring peripherals on an MCU *by hand*, that is by filling all of the control registers manually can be laborious. Furthermore it is an error prone task. It is common nowadays for MCU manufacturers to provide some libraries containing drivers, and sometimes advanced functionality on those drivers, for their peripherals.

The Atmel Software Framework is Atmel's approach at such a solution. The provided drivers and libraries integrate easily on projects. In addition the ASF comes with examples for many Atmel development kits that can be used as ready made solutions or to learn how to use a given feature.

On the downside: documentation is scarce and usually outdated. It is common to have to explore the header files to know the functions a library provides. Some not documented requirements or even bugs were detected during the development of this project. In both cases they have been dully documented in their respective sections in Chapter 4.

Despite it's weaknesses the ASF was helpful during the development of this master's thesis.

3.3 nRF24L01

Both the Arduino and an AVR32 MCU have been used as cores for the Internal Wireless Bus' nodes. But the central component to the Internal Wireless Bus is the nRF24L01 radio

transceiver. An alternative wording could be: considering the MCUs to be the *brains* of the system makes the nRF24L01 its *heart*.

The nRF24L01 [12] is a 2.4 Ghz radio with an SPI interface that can be integrated into other systems. As a radio it is an *all-in-one* solution as it requires no external components to operate. This made it an ideal pick for the project. It is configured and operated via the SPI interface.

It can function quite autonomously from the MCU, handling all high-speed link layer operations on its own. On practical terms that means that once configured the MCUs simply upload the data to be sent and give the command to send. Then all timing, coding and modulation operations are done on the transceiver itself. In a symmetrical manner all operations required for a reception are done autonomously and the MCU can then access a buffer holding digital data instead of having to process the RF band to extract information. This autonomy is one of the main reasons why this module was selected as it permitted to focus on *giving uses* to the radio rather than putting a big effort into *getting it to work*.

Some of its features, and how they can be exploited, are:

126 RF Channels

Having multitude of channels could be useful in noisy environments. Although it is not planned to use a multitude of channels in NUTS it is good to have flexibility for possible future changes.

1 and 2 Mbps Air Data Rate

One of the goals of the project is to improve the bandwidth provided by the I2C. Certainly this data rates help in that. 2 Mbps is slightly more power consuming but it is the mode that is used in the project. In any case it is easy to switch between the two.

Programmable output levels: 0, -6, -12 or -18 dBm

Again the decision is a compromise between range and power consumption. On the project the 0 mode has been used but since NUTS is not large maybe a lower mode could be used. Not unlike the air rate it is easily configurable. And lower transmit power implies lower energy consumption.

Dynamic Payload Size

Under normal operation both sender and receiver have to be configured to use the same payload size. The sender one will be adjusted when sending but the receiving end will only match (and thus consider it valid data) if the sizes are alike. There is, although, this option to have the receiving nRF24L01 accept any packet going to its address regardless of size, and then calculate the size based on the received data. The implemented software supports it and experiments were done to verify it works as it could be useful in some situations. But it is not currently in use in the produced prototype. Therefore it is not detailed in Chapter 4.

Auto packet transaction handling

As mentioned earlier all the low-level link operations are performed by the nRF24L01

modules autonomously. This simplifies software greatly and frees computing resources on the MCU (because it no longer needs to perform such operations).

6 Data pipe *MultiCeiver*

This feature allows to configure each nRF24L01 with up to 6 addresses. It is also possible to distinguish which of the different addresses was the packet originally sent to. This enables the implementation of the *two networks on one adapter* (2.2.2) approach.

All of the features and settings of the nRF24L01 are accessed via the registers of the module which in turn are accessed via it's SPI interface.

3.3.1 Operating Principles

Internally the nRF24L01 works as a finite state machine. Figure 3.3 depicts it.

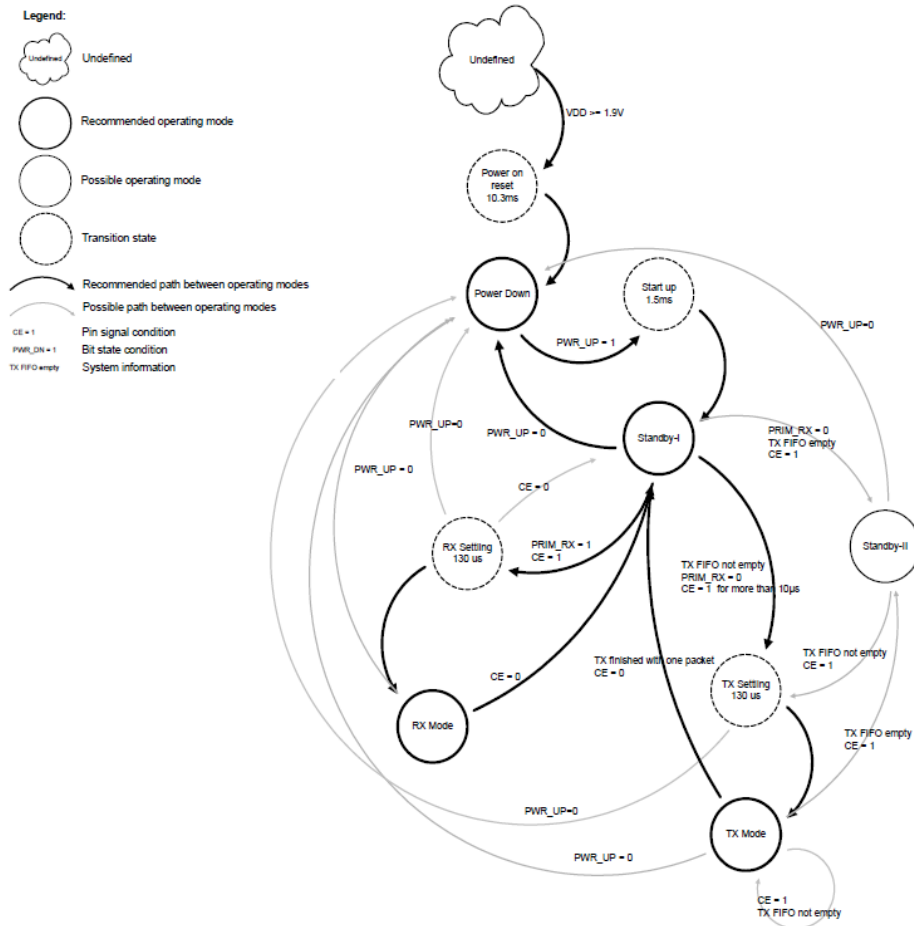


Figure 3.3: nRF24L01 State graph [12]

Many of the depicted modes are transitory, configuration or different levels of stand-by but there are three main modes of operation:

Standby

The radio interface is off but the configuration registers and the SPI interface are operational. After each transmission or reception the system defaults back to this mode. In the implementation the radio is forced to go back to RX mode, as the desired event driven approach has to be able to receive data at any moment.

RX Mode

In this mode the radio expects to receive data, the RF interface is on and constantly decoding. In case of match it decodes the packet and stores it in the RX FIFO. In the implemented design all nRF24L01 spend most time in this mode. In the original

Arduino implementation the system stays in stand by and not in RX therefore it is not possible to receive a data packet unless it is being actively awaited.

TX Mode

The module will transmit all packets awaiting to be sent in it's TX FIFO buffer. It is worth mentioning that in order to have functional Auto-ACK this mode switches quickly to RX mode to await for the acknowledgement. This switching is done automatically by the integrated controller in the transceiver and requires no operation from the main MCU.

It is important to keep the radio in the desired mode of operation, specially distinguishing between the different Standby modes and RX.

3.4 SPI

Up to this point MCUs and radio transceivers have been covered but there is a crucial part in the system: the link between the two. That link uses SPI.

SPI [15] stands for Serial Peripheral Interface and it is a *de facto* standard for embedded systems. It is a master/slave protocol therefore the connected devices will have different roles. The roles are not interchangeable even if both ends are capable of operating in both manners, although this is an uncommon setting. Usually the peripheral (in this case the nRF24L01) can only operate as a slave.

It is a serial interface so the bits are sent *one after another* through the same line. Additionally it is synchronous, meaning that there is a common clock signal that rules over both devices during the transference.

The signals required for an SPI interface are:

MOSI

Master Out Slave In: This line carries the bits from the master to the slave, one bit per clock tick.

MISO

Master In Slave Out: Analogous to the former but in the opposite sense.

CLK

Clock line to regulate the data flow, it is generated by the master.

SS

Slave Select: This line is used to select the slave to talk to, it can be seen as a request for attention from that slave to listen to it's data and clock lines.

Figure 3.4 shows a typical SPI setup, as it is used in the project.

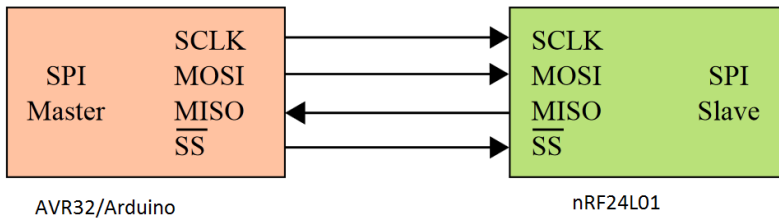


Figure 3.4: Schematic of an SPI connection [15]

SPI is a Full Duplex interface, furthermore an *enforced* Full Duplex interface. All transmissions are bidirectional. If either side has nothing to communicate it is necessary to use some dummy data. The nRF24L01 transmits it's status register value whereas the AVR32 transmits 0x55. The dummy value 0x55 was chosen because it's characteristic waveform makes it recognizable during debugging.

Figure 3.5 illustrates a transference, including the use of dummy values when necessary.

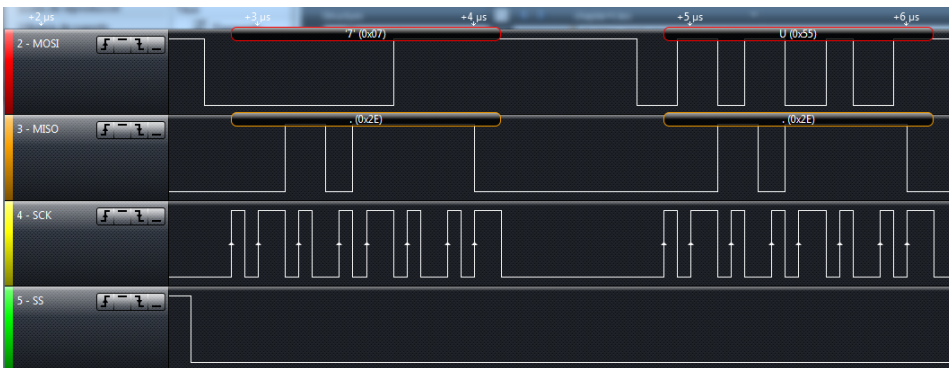


Figure 3.5: Logical Analysis of an SPI transfer

Aside from the SPI lines the nRF24L01 also uses two additional lines:

CE

Chip Enable: Used to toggle between different modes on the transceiver, as presented in figure 3.3.

IRQ

Interrupt Request Line: Used to notify the host MCU (be it Arduino or AVR32) that an event requiring processing has occurred. It's use is detailed in section 4.7.

Although this two additional lines are not part of the SPI connection, they are required for a proper operation of the transceiver. The CE line is managed as a GPIO line whereas the IRQ line is managed by the External Interrupt Controller.

3.5 FreeRTOS

A satellite has to perform a multitude of different tasks. That requires scheduling. Using a real time operating system can provide the scheduling features (among others) that are needed. NUTS uses FreeRTOS [10] for this purpose.

FreeRTOS is an open source real time operating system for embedded systems. It has been ported to multiple architectures, AVR32 among them. It is designed to be portable and to have a low memory footprint. For this project FreeRTOS is used (keeping the development platform as similar to NUTS as possible), naturally during the development FreeRTOS was devoted entirely to the scheduling of operations that dealt with the Internal Wireless Bus. But in the integrated NUTS system it will schedule other tasks as well. In other words: many different software components will share a single instance of FreeRTOS after integration.

Scheduling could be considered the main feature of any real time operating system. FreeRTOS' take on it is built around the idea of Task. The naming empowers the one-to-one association that each of the different tasks the system should perform can be implemented as a FreeRTOS Task. A Task is the unit FreeRTOS uses for scheduling (akin to a process on a PC operating system) and it has a set of resources associated to it (stack, local variables. . .). When there are more than one task in a system (any real world application has more than one task) the system scheduler decides which task is run and when. For convenience the capitalize word *Task* refers to a FreeRTOS Task construct whereas *task* means the general concept of a job to be performed.

There are many possible scheduling algorithms to determine how tasks should run. The one used in this project is Round Robin with priorities. In plain words: Tasks take turns, yet those with higher priorities can *cut the queue* in front of those with lower.

The most common way of operating with FreeRTOS is: configuring the hardware, creating tasks and then launching scheduler. When doing the latter stage FreeRTOS takes control of the execution flow and will alternate execution, following it's policies, among all tasks that are ready to be run. Any given task could not be ready different reasons such as being blocked (waiting for some event or resource) or being stopped.

Having all tasks isolated from one another (having their own local variables and stack) can be a problem when trying to communicate between tasks. And the use of global variables is inadvisable in any multitask system because the variable used for communication generate inconsistencies if it is accessed or modified at critical points when other Tasks switch into execution. The system provides mechanisms that allow safe communication between tasks. The scheduler is aware of this mechanisms which avoids the risk of inconsistencies. Among such mechanisms there are semaphores and queues.

3.5.1 Sempahores

A simplistic view of a semaphore is: a counter. It accounts for the availability of any given resource. The resource can be of different nature: hardware unit, system event...

And is implemented in such a way that while one task is accessing it (either reading or writing) no other task can have access. In this manner problems deriving from a multitask environment can be avoided.

The most common implementation of semaphores (and FreeRTOS adheres to it) has three operations for a semaphore, the nomenclature can differ thus the FreeRTOS naming scheme has been used:

Initialization

The semaphore is created and the current and maximum values are fixed. For example: a maximum value of 1 and a current of 0 would be assigned if the semaphore has to control access to a UART module that is not in use by anyone right now. The FreeRTOS header is:

```
xSemaphoreHandle xSemaphoreCreateCounting
(
    unsigned portBASE_TYPE uxMaxCount,
    unsigned portBASE_TYPE uxInitialCount
)
```

Give

Once the managed resource (the UART in our example) is no longer needed, the Task that used it must *give it*. Effectively this will increase in one the counter of the semaphore (never above maximum value) and if there are other Tasks suspended due to the semaphore, the first one will be resumed. The FreeRTOS header is:

```
xSemaphoreGive( xSemaphoreHandle xSemaphore )
```

Take

Complementary to the give operation, this one is used when the resource is requested for use. It checks the semaphore counter if the counter is bigger than zero (therefore the resource is still available) it decreases the counter and continues execution. If the counter is below zero it suspends the current task that will wait to be resumed by a give call by one of the current users of the resource. The FreeRTOS header is:

```
xSemaphoreTake(
    xSemaphoreHandle xSemaphore,
    portTickType xBlockTime )
```

As it can be seen in the header FreeRTOS allows the configuration of a timeout, if the resource is not obtained after said period the task will be resumed anyway. Based on the returned value it is possible to distinguish in execution time whether the resource was obtained or not, and act accordingly after resuming execution.

Additionally it is worth mentioning that there are special versions of the semaphore operations to be used within ISRs, that is done to avoid a blocking operation inside an ISR. In this variations there is no wait time: either the resource is obtained or the call returns with the *failed to obtain* value and execution can carry on. Semaphores are useful for signalling and synchronization but they do not handle a payload, that is where queues come in.

3.5.2 Queues

Queues are data structures where information can be added or read from. The information is kept orderer and has fixed points of insertion (*enqueueing* on the back and *dequeueing* on the front. As shown in figure 3.6.

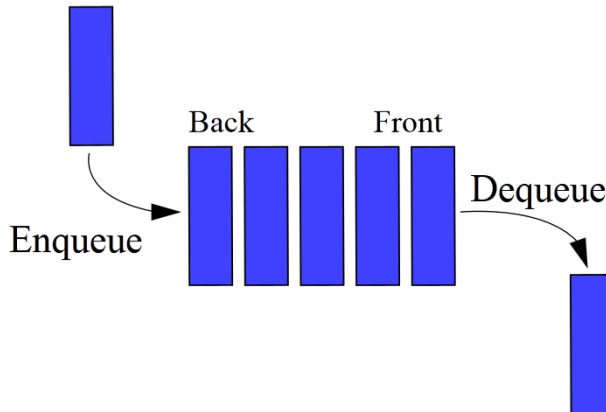


Figure 3.6: Representation of a queue

Using queues to share information between different tasks has some advantages. It is an asynchronous process, that means that the Task producing the data can leave it in the queue regardless of when the consuming task will read it. Additionally the system makes use of the scheduler: if a Task tries to read information from a queue and that queue is empty, the Task will be suspended and kept in that state until there is data available or it reaches timeout.

FreeRTOS provides queues and the API to operate with them. And they have been used in the implementation of this project(4).

3.6 CubeSat Space Protocol

The CubeSat Space Protocol [5] was developed in 2008 by the Aalborg University and has been used in some of their satellites, starting with AAUSAT3. The team that developed it eventually spun off as a commercial the commercial company: GomSpace.

It is a network-layer protocol, similar to the Internet Protocol on common networks. And it borrows many of it's ideas. The information is transferred in packages and said packages are preceded by a 32 bit header (Figure 3.7). Also similar to the internet the packet is routed through several different interfaces.

It provides many features that are common in common computer networks, including: loopback traffic and control packets. Additionally packets can be *on-the-fly*. The latter is

Bit offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Priority		Source				Destination				Destination Port				Source Port				Reserved				H	X	R	C						
32	Data (0 – 65535 bytes)																															

Figure 3.7: CSP header [14]

specially useful when using it for ground to satellite communications although it might not be necessary for intra-satellite communication.

It is designed to be deployed on several different operating systems, including FreeRTOS.

Albeit the core of the system is provided as an open sourced software component, the drivers binding it to physical interfaces need to be developed.

Chapter 4

Implementation

This chapter describes the process followed while developing the designed solution (2). A *bottom-up* coding strategy was chosen. That is: small and specific functionalities are developed first and the rest of the system is built gradually on those *building blocks*. I favour this strategy for projects that, like this master's thesis, have a strong prototyping component since these *building blocks* can be tested before moving on to the next level. Exemplifying for the case at hand: initially the radio transceivers driver is developed and only once the transceivers behave as desired one moves forward into making said driver asynchronous.

The development setup follows:

4.1 Development Setup

The solution has been implemented on AT32UC3A3256 MCUs, manufactured by Atmel. The choice of the MCU was not mine to make but it was rather established by the fact that most of the NUTS hardware is already based on this architecture and even on this specific MCU. Far from being a drawback, this will make the eventual porting of this project onto the satellite platform easier.

As a development platform the UC3-A3 Xplained was used. It is a rather simple development board, it can actually be considered closer to a demo kit than a development board. It is, mainly, a base to operate the MCU (power supply, crystals...) with some of the I/O pins broken out and a few other peripherals (such as a capacitive touch slider, an external RAM). Additionally it is also equipped with a mini-USB connector that proved very useful during development as it was used as a serial port. Albeit being a simple kit it covered the needs of this project. It even exceeded them, seeing how some of the peripherals and components were not used.

From a purely software point of view: the code was written in C on Atmel Studio which in turn relies on AVR-GCC. Again this was more of a fixed constraint than an actual decision. Atmel Studio is provided by Atmel at no cost to develop for their products and it provides a good integration with their debuggers/programmers. GCC is often regarded as one of the best C compilers. Despite I had little freedom to decide on the software to use, I would most have made an identical choice. C is the authors language of choice and it allows *low level* operations that are vital to embedded software development. Flashing and debugging operations was done using an AVR Dragon.

4.2 Hello World

The first step was, although it might seem redundant, to make sure that the the development board was running the code that I was actually developing. The simplest way to to that was to run a *blinker* code, that is a simple snippet of code that simply blinks LEDs on the board. Once that worked one can move forward with the implementation.

Despite there was an AVR Dragon constantly hooked up to the board, and one could have debugged using it, it was thought interesting to have serial port (UART) communication with the PC. Most modern PC no longer have an RS232 connector although it is possible to emulate such a link through a USB device. The usual (and actually used) solution is to do the UART on USB emulation on the MCU side, and the AVR32 has specific hardware to this purpose. A simple explanation of what is actually done is: the MCU's USB modules are configured to enumerate as a serial port. Then the PC would recognise them as such and all software designed to work with serial ports can use that interface. On the MCU side then all UART operations are done on this newly created *virtual* UART and not on the specific hardware units.

On practical grounds, the ASF (3.2.1) provides an example project where the UART-on-USB is already configured. The initial premise was to extract the necessary function calls and configuration files to replicate the emulation behaviour on a blank project. The need to replicate it on a blank project rose from the author's preference for starting from scratch, thus having absolute control on the code. But for reasons still unknown, the behaviour could never be replicated. Despite the same configuration files, function calls and program structure were used, it did not work as expected. After software assisted comparisons (by using diff) of both source codes gave no meaningful differences it was decided to develop the rest of the system on the template of the given example. Thus keeping the UART-on-USB functionality and having a line of communication between the PC and the micro-controller. Following tradition a *Hello World* string was used as a first trial of the link.

That concluded the preparations of the platform and opened way for developing the actual system components.

4.3 Interfacing with the radio transceivers

The interface between the nRF24L01 and the MCU can be seen in two separate layers: the physical interface and the software driver that operates on said interface.

4.3.1 Physical Interface

Connecting the MCU with the transceivers physically was the first step. The connection to operate the nRF24L01 (3.3) is SPI (3.4) plus one GPIO pin, one IRQ line and power supply. One of the connectors on the development boards had all the necessary pins conveniently broken out. But the pinouts on both sides do not match, it was therefore necessary to build an adapter. It is a small board with a 10-pin socket (lodging the nRF24L01) and breaking it's pins out in a convenient manner for use on a breadboard.

The pin matching between the AVR32 board and the nRF24L01 is:

PB10	=	Master Out Slave In (MOSI)
PB08	=	Master In Slave Out (MISO)
PB07	=	Clock (SCK)
PB09	=	Slave Select (SS)
PX57*	=	Chip Enable (CE)
PA23*	=	Interrupt Request (IRQ)

Those pins marked with an asterisk (*) do not belong to the SPI interface. The SPI unit used was SPI1 because all the pins were broken out at connector J1. All pins except PA23 are located in the J1 connector of the Xplained board. PA23 is located at J2 and single cable bridges the connection.

The following figures (4.1 and 4.2) illustrate the setup:

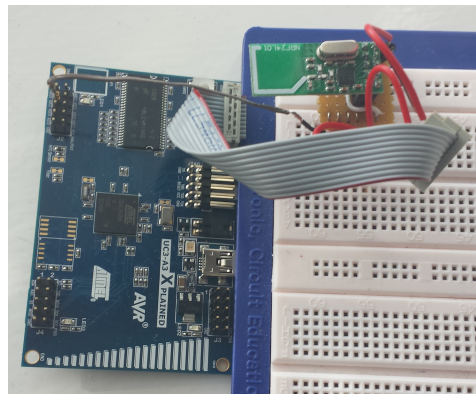


Figure 4.1: Development board connected to the nRF24L01 using the adapter

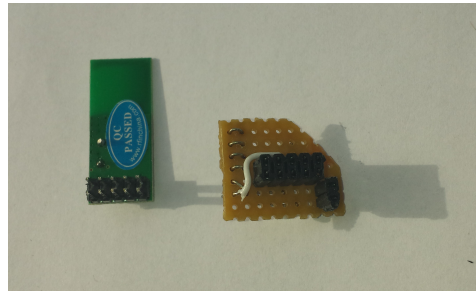


Figure 4.2: Left: nRF24L01 (bottom view). Right: built adapter (top view)

Once all the connections were in place, the SPI module on the AVR32 could be activated. Again the ASF was used as it provided an SPI module driver. Generally speaking all AVR32 modules have a similar setup process: enabling the clock, setting up the pins, configuring the uni and finally initializing the module. The first two steps are necessary as the clock tree is usually kept off to save power unless it is needed and the control of the pins is by default granted to the GPIO. To configure a module before initializing it is a good practice, as it avoids running the unit with an unknown configuration. Once those requirements are met, the module can be activated. The code on listing 4.1 depicts these phases, the 3 visible statement groups match the steps taken:

Listing 4.1: SPI module Initialization

```
1 static void spi_init_module()
  {
    sysclk_enable_peripheral_clock(SPI_RADIO_1);
5   spi_pin_setup();

    spi_initMaster(SPI_RADIO_1, &spi_options_radio);
    spi_selectionMode(SPI_RADIO_1, 0, 0, 0);
9   spi_setupChipReg(SPI_RADIO_1, &spi_options_radio, FOSC0);

    spi_enable(SPI_RADIO_1);

13 }
```

To test that the module had been properly initialized a logic analyser was used to perform naive test: sending a random byte just to see that all the signals were being generated properly. And with the test it was found out the the clock signal on the SPI (SCK) was not being generated, it was a flat 0. At first, as it was on the SCK line, an error on the SPI module clock configuration was suspected, but after inspecting and stepping through the code it was discarded. The next step was to toggle the affected line with the GPIO driver. The hypothesis was: if the line is under control of the SPI, toggling via the GPIO module should have no effect. But it toggled, meaning that it was still under GPIO control. While refactoring the code of `spi_pin_setup` (4.2) it was discovered that the SCK line had to be the last one mapped to the SPI unit (as shown in listing 4.2). This *requirement* is

not documented and by inspecting the code no cause can be detected so it was deemed a hardware bug and a bug report was dully filled with the manufacturer.

Listing 4.2: SPI pin setup

```

static void spi_pin_setup()
2 {

    static const gpio_map_t spil_gpio_map = {
        {AVR32_SPI1_MISO_0_0_PIN, AVR32_SPI1_MISO_0_0_FUNCTION}, //MISO
6        {AVR32_SPI1_MOSI_0_0_PIN, AVR32_SPI1_MOSI_0_0_FUNCTION}, //MOSI
        {AVR32_SPI1_NPCS_0_0_PIN, AVR32_SPI1_NPCS_0_0_FUNCTION}, //SS0
        {AVR32_SPI1_SCK_0_0_PIN, AVR32_SPI1_SCK_0_0_FUNCTION} //SCK
    };
10

    gpio_enable_module(spil_gpio_map, sizeof(spil_gpio_map)/sizeof(
        spil_gpio_map[0]));

}

```

After the function (4.1) is executed, the SPI is ready to transmit and receive data. But still a driver needs to be implemented to manage exactly what meaningful data (and not random bytes) must be sent.

4.3.2 nRF24L01 Driver

The driver itself is based on an Arduino library for the nRF24L01 (3.3) called NRF24 [7]. The library, like most Arduino software, is open source. That made it a good candidate as a foundation for the AVR32 driver. Additionally the author had experience using that specific Android library in a previous project. Logically the original code was based in Arduino's own library to manage it's peripherals and *translating* those calls to ASF was required. Additionally the NRF24 library, written in Arduino's own *enhanced C++*, used some features of object oriented programming that are not available in C. Some adjustments were made to address these issues, such as:

Turning members into constants

The original object oriented had two member variables holding pin mapping information, specifically which GPIO pins match SS and CE. By holding this information as members the Arduino could have handle more than one radio simultaneously, granted that CE and SS were different in each of the instances. Knowing that NUTS was not planning to have more than one radio transceiver per module such flexibility was no longer necessary, thus both pins were fixed as constants. With the use of `#define` constructs enables us to conveniently change the pins in a consistent and clean manner, although such adjustment could only be made in compilation time.

Renaming of functions

Initially all the functions were kept as they were in the original library, in an attempt to keep maximum similarity between the original and the ported version. Hoping that such similarity would help during development since all documentation was

for the Arduino version. Far from the original intent, keeping the names unchanged induced confusion. The original code being object oriented would have its functions called in this fashion: `nrf24.spiRead(parameter)` assuming `nrf24` was the name of the given instance of the object being used. But as C has no objects the same call was then: `spiRead(parameter)`, making it unclear to distinguish if it was an nRF24L01 specific operation or rather a generic SPI call. This issue was solved trivially by prefixing all functions with `nrf24_` rendering a more readable `nrf24_spiRead(parameter)`.

Translation of API calls

Most of the original NRF24 library consisted of functions that simply *prepared* the bytes to be transmitted to the module, so that they would be conforming to the nRF24L01 command format. Later those functions relied on four lower level ones to actually use the Arduino SPI library. Therefore the majority of functions were trivial to port and these latter 4 functions would require a more significant change. Exemplifying:

Listing 4.3: Arduino NRF24::spiBurstRead(...)

```
void NRF24::spiBurstRead(uint8_t command, uint8_t* dest, uint8_t
    len)
2 {
    digitalWrite(_chipSelectPin, LOW);
    SPI.transfer(command);
    while (len--)
6     *dest++ = SPI.transfer(0);
    digitalWrite(_chipSelectPin, HIGH);
}

```

Listing 4.4: AVR32 nrf24_spiBurstRead(...)

```
void nrf24_spiBurstRead(uint8_t command, uint8_t* dest, uint8_t
    len)
2 {
    uint16_t temp;
    spi_selectChip(SPI_RADIO_1, 0);
    spi_write(SPI_RADIO_1, command);
6     while (len--)
        {
            spi_write(SPI_RADIO_1, 0x55);
            spi_read(SPI_RADIO_1, &temp);
10         *dest++ = (uint8_t) temp;
        }

    spi_unselectChip(SPI_RADIO_1, 0);
14 }

```

Listing 4.3 is the Arduino version and listing 4.4 is its ported counterpart. It can be easily seen that both share a common structure yet some differences are worth mentioning:

Arduino relinquishes the control of the SS line to be done by it's GPIO libraries:

```
digitalWrite(_chipSelectPin, LOW)
```

and

```
digitalWrite(_chipSelectPin, HIGH)
```

whereas on AVR32 those lines are tied to the SPI unit. On AVR32 even if the developer preferred to use a GPIO line as SS the calls to

```
spi_selectChip(SPI_RADIO_1, 0) and spi_unselectChip(SPI_RADIO_1, 0)
```

are still required. If those calls are not used the SPI unit is rendered unresponsive.

Arduino has a simple call `SPI.transfer(command)` that writes out it's parameter and returns whatever was read from the MISO line during the transaction. It is worth remembering that SPI is always bidirectional and thus every write implies a read (that can be useless data). The ASF has two functions one to clock the data out `spi_write(SPI_RADIO_1, command)` and one to return the buffer of what was read during the last transfer `spi_read(SPI_RADIO_1, &temp)`. Thus both are necessary to replicate the `SPI.transfer(command)` functionality.

The original NRF24 library relied on busy waits for reception of data, instead of interrupts. Although to avoid blocking the MCU completely waiting for data that may never arrive it introduced the possibility to wait for receiving data until a certain timeout counter run out. Naturally this timeout was controlled using Arduino's timekeeping API, so porting this function also required some additional changes, specifically shown in listing 4.5 (Arduino) and 4.6 (AVR32).

Listing 4.5: Arduino NRF24::arduinoWaitAvailableTimeout(...)

```
1 bool NRF24::waitAvailableTimeout(uint16_t timeout)
  {
    powerUpRx();
    unsigned long endtime = millis() + timeout;
5   while (millis() < endtime)
      if (available())
          return true;
          return false;
9  }
```

Listing 4.6: AVR32 nrf24_waitAvailableTimeout(...)

```
uint8_t nrf24_waitAvailableTimeout(uint16_t timeout)
2  {
    nrf24_powerUpRx();
    t_cpu_time to;
    cpu_set_timeout(cpu_ms_2_cy(timeout, sysclk_get_cpu_hz()), &to);
6
    while (!cpu_is_timeout(&to)){
        if (nrf24_available()){
10     cpu_stop_timeout(&to);
            return true;
```

```
    }  
  }  
14  cpu_stop_timeout(&to);  
    return false;  
  }  
}
```

Again both have a similar structure but while the Arduino one relies on it's specific call `millis()` which returns the uptime of the system in milliseconds, the AVR32 code is based on `cpu_set_timeout(...)` based on the ASF timer libraries. The latter libraries were introduced into the project solely for the implementation of this function.

Once the whole of the driver code was ported it was time for a first trial at transferring information via the nRF24L01 modules. Such a trial would be limited as the driver at this point was only a port of the Arduino code with all it's limitations, such as using *busy wait* instead of interrupts.

4.4 First Transmissions

One of the problems of testing communication interfaces is that usually the result of the test is boolean in nature; either everything is set and configured correctly or some error (or errors) hamper the communication. Therefore eliminating as many unknowns as possible would increase the chances of success. With that in mind the decision to use the AVR32 as one node and an Arduino as the other was taken. The Arduino would be using already tested code, thus any error would either be on the physical interface (either side) or on the AVR32 software. Both development boards had UART-on-USB so they could print on a serial console helpful messages to debug and fine tune the software.

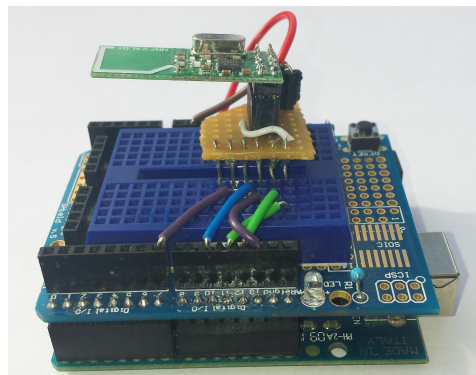


Figure 4.3: Arduino with mounted prototyping shield and nRF24L01

Right after booting up the system and the SPI module is ready, the nRF24L01 needs to undergo a configuration process where some details of it's workings are set. Such as:

data rate, transmit power, address. These settings need to match in all the stations that form the internal wireless network. The developed driver provides functions to set such settings.

4.4.1 Ping

Resorting again to the Arduino library, specifically to the examples it bundled. A useful ping application was found there. Actually the ping example was split in two halves:

Client

It initializes the nRF24L01 and then loops through the following procedure: send current time through the radio transceivers and await for an answer. When the answer, that is just the same data echoed back, is received measure the difference between the current time and the received answer. That difference is the round trip time and is printed via the UART-on-USB. The whole looped is scheduled to run once a second.

Server

It, naturally, acts as counterpart. So, after initializing the radio transceiver it simply awaits for data packages and echoes them back.

This simple application is an ideal test bench because both client and server imply transmitting as well as receiving data. Both boards were used for both roles in order to increase the covered cases.

Fortunately both roles worked on both boards with relative ease. Thus confirming that the interfaces, on their lowest level, worked. The ported driver was functioning correctly. Despite the partial success it still lacked many desired properties (interrupt driven, message queueing, multitasking...). Steps were taken to mend this issues.

4.5 Time for FreeRTOS

Due to the many different tasks that the NUTS system needs to perform a real-time operating system was chosen: FreeRTOS (3.5). Since the aim of the software is to integrate into the NUTS system it seemed a good fit to develop on a system as similar to NUTS as possible which implied the use of FreeRTOS. Initially the integration of FreeRTOS into this master's thesis was done solely to keep that similarity, and to benefit from some of it's scheduling capabilities. But later other functionalities of the system emerged as useful, even required, for the solution presented on this document.

4.5.1 Bringing FreeRTOS In

Using past versions (such as 2.1.1) of the ASF Atmel one could use the provided version of FreeRTOS that could be imported through the ASF Wizard on Atmel Studio (similarly to

how it is done for the SPI or the timer libraries), that is no longer the case with the current version (3.5.1 at time of writing this document). But the port of FreeRTOS to AVR32 is, albeit hidden, still available. It is possible to get it from the example projects provided. The chose example was `FreeRTOS Basic Example - EVK 1104` because that board is based in the same MCU that is used for this project, and also on the NUTS satellite. The FreeRTOS files can then be moved from the example project folder tree onto the project being worked on and included in Atmel Studio.

4.5.2 Using FreeRTOS

FreeRTOS has a multitude of features, most can be included or not on the build depending on that specific project's needs, this adjustments can be done through a the configuration header file: `FreeRTOSConfig.h`. This functions were not used at first and the first take of contact was to replicate the earlier-described (4.4.1) ping functionality.

Generally speaking a FreeRTOS program does not differ in behaviour from an OS-less program until the scheduler is called. Usually the first phase is used to configure the system and create the FreeRTOS tasks and other data structures that might be needed during execution. Once the scheduler is launched it will start running and pre-empting the tasks based of scheduling policies, priorities, synchronization between tasks...

These two-phase scheme was applied to the ping (as seen on listing 4.7). Initially the system was configured: initializing the nRF24L01 module and creating a task that would be run once a second. Three other tasks were created, each blinking a LED at a different rate. This tasks allowed for a fast way to verify that the multitasking capabilities of the system were operating properly. Then launching the scheduler provided the expected results: the LEDs were blinking and the system was successfully pinging the other board, which at this point in time was still the Arduino. Task *Dave* (commented on the listing 4.7) printed the string *Hello Dave* [6] to the serial console. It was intended as means of trying the UART-on-USB from FreeRTOS and it functioned as expected.

For this project FreeRTOS' default scheduling policy was used. That is Round robin with priorities. Therefore Tasks that include potentially blocking operations should be given lower priorities than those that do not, else that blockage would propagate to all other Tasks in the system. That can be seen on lines 25-26, again in listing 4.7.

Listing 4.7: Ping client on FreeRTOS

```
int main (void)
{
3   sysclk_init();
   board_init();

   irq_initialize_vectors();
7   cpu_irq_enable();

   stdio_usb_init();

11  nrf24_initRadioSpi();
```

```

    if (!nrf24_setChannel(1)) printf("setChannel failed \n");
    if (!nrf24_setThisAddress((uint8_t*)THIS_ADD, 5)) printf("
        setThisAddress failed \n");
    if (!nrf24_setPayloadSize(PAYLOAD_SIZE)) printf("setPayloadSize
        failed \n");
15
    NRF24DataRate dr = NRF24DataRate2Mbps;
    NRF24TransmitPower pr = NRF24TransmitPower0dBm;

19    if (!nrf24_setRF(dr, pr)) printf("setRF failed");

    xTaskCreate(&vLedTask1, (const signed portCHAR *)"Led Task 1",
        configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+3, NULL);
    xTaskCreate(&vLedTask2, (const signed portCHAR *)"Led Task 2",
        configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+3, NULL);
23    xTaskCreate(&vLedTask3, (const signed portCHAR *)"Led Task 3",
        configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+3, NULL);
    xTaskCreate(&vServerTask, (const signed portCHAR *)"Server Task",
        configMINIMAL_STACK_SIZE+512, NULL, tskIDLE_PRIORITY+1, NULL);

        //xTaskCreate(&vDave, (const signed portCHAR *)"Dave Task",
            configMINIMAL_STACK_SIZE+8192, NULL, tskIDLE_PRIORITY+2, NULL)
        ;

27    vTaskStartScheduler();

    return 42;

31
}

```

From this this point onwards the system was running on FreeRTOS, but so far it was only looping at scheduled times. It was necessary to have a closer approach to the real use on NUTS an *on-command* scheme . In other words: in real life NUTS could get a command from ground asking to dump the last image taken by the camera. Then the camera module (or the OBC if the image had been already processed) could transfer said image to the radio module that links NUTS with ground by using the internal wireless bus. Thus the whole operation would have been done *by request* and not as an scheduled event. It was thus necessary to create a system that could imitate such requests.

4.6 Interactive Console

An interactive console seemed a good option to create the desired *event driven* system. Occurrences that on the satellite would be commands from ground, or internal satellite events would in this thesis be simulated by console commands. Such commands would be read and it's output written to the serial port, making the system operable from any computer.

A console is, in a simplistic overview, a piece of software that takes an input, parses it and summons other software components based on that input. It can, very often, also take

parameters that can be forwarded to it's called functions. And once those end the console would show the output.

The advantages it has over other solutions (e.g, LCD + buttons interface) are it's lower use of hardware resources (quite often only the communication interface it uses) and it's flexibility. One can add new commands with relative ease. Downsides: consoles can have higher processing demands (mostly due to parsing) and might require relatively big buffers, thus increasing RAM consumption.

4.6.1 Deploying the Console

The earliest idea was to develop the console from scratch, but it was soon discarded as there are packages that can be integrated and tailored to specific needs. One such packages is FreeRTOS Command Line Interface (FreeRTOS CLI). As the name implies it is provided as a component for FreeRTOS and by the same team. Using this option smoothened the integration.

The first command that was implemented had little to do with the nRF24L01 and was implemented as a proof of concept. It allowed to toggle the state of a LED that was passed as a parameter. It was a suitable candidate because the action is simple enough to minimize failure yet requires the parsing of a parameter, allowing to explore the FreeRTOS facilities. In order to add each new command to the FreeRTOS CLI there are four steps to be followed. Here they are detailed by using this first `toggle n` as an example:

Providing a function that implements the command behaviour

The console parses and matches to a command, and then launches the function it has associated. Therefore the function is at the core of each and every command. These functions need to follow this prototype:

```
portBASE_TYPE xFunctionName(  
    int8_t *pcWriteBuffer,  
    size_t xWriteBufferLen,  
    const int8_t *pcCommandString )
```

Where `pcWriteBuffer` is the output buffer, that will be printed on serial port once the function returns. Additionally `xWriteBufferLen` is the size (in bytes) of `pcWriteBuffer`, necessary for the console to control how many characters to print. Finally `pcCommandString` allows access to the whole command that summoned the function. Therefore the parameters are contained in this command string and must be parsed and converted depending on the uses intended. Specifically to the case of `toggle n`:

Listing 4.8: Toggle LED Implementation

```
portBASE_TYPE prvToggleLed(int8_t *pcWriteBuffer, size_t  
    xWriteBufferLen, const int8_t *pcCommandString ){  
    uint8_t* param;  
    uint8_t* pz;
```

```

4   param = FreeRTOS_CLIGetParameter(pcCommandString, 1, pz);
   switch(*param){
       case '0':
           gpio_toggle_pin(LED0_GPIO);
8       break;
       case '1':
           gpio_toggle_pin(LED1_GPIO);
           break;
12      case '2':
           gpio_toggle_pin(LED2_GPIO);
           break;
       case '3':
16      gpio_toggle_pin(LED3_GPIO);
           break;
       default:
           sprintf(pcWriteBuffer, "%s\r\n", "n must be 0 - 3");
20      xWriteBufferLen = strlen(pcWriteBuffer);
           return pdFALSE;
           break;
   }

```

In snippet 4.8 it can be appreciated how the parameter is parsed by using `FreeRTOS_CLIGetParameter(pcCommandString, 1, pz)` .

that returns a pointer to the beginning of the desired parameter. In this case the first one. It also returns it's length.

FreeRTOS will consider that a command matches an action if the the command name and the number of parameters match. But to parse the content of said parameters falls under the summoned function, that is why there is a default case warning of an error if the LED to be toggled does not exist.

Mapping the command to the function

FreeRTOS CLI requires all commands to be stored in a `struct` as the one in listing 4.9:

Listing 4.9: struct xCommandLineInput

```

1  typedef struct xCOMMAND_LINE_INPUT
   {
       const int8_t * const pcCommand;
       const int8_t * const pcHelpString;
5   const pdCOMMAND_LINE_CALLBACK pxCommandInterpreter;
       int8_t cExpectedNumberOfParameters;
   } xCommandLineInput;

```

It contains information about what exact input will trigger this command. Both the command name and the number of arguments it accepts. Logically it also holds a pointer to the function to be called. FreeRTOS CLI provides a built-in `help` command, that prints the relation of available commands and their respective help strings.

It is a simplistic set of commands but it allows interactive sending and receiving of messages.

Registering the command with FreeRTOS CLI

All commands registered in the previous step must be *registered* with the FreeRTOS CLI. The console framework provides a function for it:

```
portBASE_TYPE FreeRTOS_CLIRegisterCommand(  
    xCommandLineInput *pxCommandToRegister )
```

What this function actually does is filling FreeRTOS' internal data structures with the provided command definition. During parsing the entered string will be compared against the data in these structures and if a match occurs the function is launched.

The registering has to be done for each command that is meant to be used. In a vast majority of cases a console will have more than one possible command, then it is possible to have an array of *command defining* structures, where each position holds one command. This idea was applied in this project (listing 4.10).

Listing 4.10: Command registration loop

```
1 void registerCommands() {  
    int i;  
    for(i = 0; i < sizeof(cmdTable)/sizeof(cmdTable[0]);i++)  
    {  
5     FreeRTOS_CLIRegisterCommand(&cmdTable[i]);  
    }  
}
```

Currently, and including the built-in help, there are five commands defined:

```
toggle n:    toggles the n-th LED where n = [0..3]  
receive_raw: prints received message  
send_raw n payload: transmits payload of size n  
conf_rf:    changes radio transceiver configuration  
help:      prints information about available commands
```

Although registering the commands in execution time may be seem as waste of computing resources, it has an advantage: dynamism. New commands could be created (filling the appropriate struct) and registered while the system is running. Obviously the behaviour to be executed would be limited by the functions that are *console callable* (that follow the specified prototype), but aliases could be created. These aliases could, for example, be existing functions but fixing it's parameters. For example: an SOS command could be created. This hypothetical SOS command could be defined as accepting zero arguments and calling the function to transmit (`send_raw`). Once inside the function the outcome could be decided by which command was used to summon the function: SOS would send a message, fixed

in size and content, requesting assistance. Whereas `send_raw` would parse the arguments and operate normally. Both functions differ only in how the arguments are obtained: either hard-coded or parsed.

Said dynamic capabilities were not used in this thesis.

Run the command interpreter

Once all the pieces were in place the interpreter can be run. The interpreter (or shell) need to be run on repeated occasions thus it is common, and recommended, to make it a FreeRTOS task. Briefly the shell takes the input, one character at a time, and when it detects an *end of command* character (most often ' \n ') it provides the newly created string to:

```
FreeRTOS_CLIProcessCommand(  
    pcInputString,  
    pcOutputString,  
    MAX_OUTPUT_LENGTH)
```

The function will then try to match the input string to FreeRTOS' internal data structures by comparing the string and the detected number of arguments. If there is no match an error message is displayed and the shell is ready to start accepting input again. If there is a match the requested function is called and once it returns the console will print it's output and be reading again to take the next input.

For this thesis the serial port was used as interface, but for different projects other interfaces could be used the only changes to the shell are the functions it uses to get and put the characters.

Having followed these steps the console is now ready to be used, so it is added as a task before calling the scheduler. This task will, by definition, make use of I/O operations so it is recommended to give it a lower priority than other periodic tasks else it may block those tasks while waiting.

4.6.2 Running the Console

As earlier mentioned(2.4) adding the console was not a goal of the project *per se* but rather a tool to experiment with the wireless bus interface acting *in lieu* of the applications that will be running on the real NUTS platform.

The first test that was done consisted in sending data packets from the AVR32 board to the Arduino. Upon receiving the data the Arduino would print it on a serial console of it's own. The data being sent were ASCII strings. This test allowed to test the `send_raw` command. The results were satisfactory as the string was printed on the Arduino side.

The second test was meant to assess the `receive_raw` command. The software used in the previous test was modified. On the AVR32 side it no longer sent an ASCII chain but a 32-bit unsigned integer, and right after sending the integer it would expect to receive

an integer back to print it on the serial port. The modifications for the Arduino made it interpret the received data as a 32-bit integer, print it, and then increment it by 1 and send it back to the AVR32. If the test had worked correctly both boards would have printed out different integers (n and n+1) but several problems arose:

Endianness

It was observed that when the AVR32 transmitted a 1 the Arduino interpreted it as 16777216. The initial diagnosis was that some noise was being read or a coding error, but the repeatability of the results eliminated both possibilities. While debugging it the Arduino code was changed to print the received integer in binary, that rendered: 0b000000010000000000000000000000. It was then properly diagnosed as an endianness mismatch. The Arduino is based on an 8bit AVR therefore it's endianness depends on how the types larger than the byte are implemented in software. The AVR32 is natively a Big endian MCU.

Blocking calls

Even with the endianness problems the transmission/reception should have endured, albeit wrong values would be printed out. That was not the case. The blocking nature of the receive functions heavily impaired proper operation. Since the actual sending was done from the console task (any function launched from a task belongs to said task) the task console would block waiting for packets to be received. Since the system would only send a packet when told to by the shell and it was locked, no packets were being sent to the Arduino. Therefore the Arduino was not answering, thus the AVR32 was stuck waiting for packets.

The Endianness problem could have been solved by *bithacking* the data on either board to make it fit it's data representation scheme. The blocking calls problem could have been partially solved by using the functions that await incoming packets for a given time, else they timeout. This would have been a partial solution as it still implied busy waits that could have derived in a system that waits most of the time and seldom (after timeout and before next blocking operation) is responsive.

One of the original requirements was to make the driver interrupt driven, and with this results it was clear that it was a moment to pursue that goal. Since making the Arduino version of the driver interrupt driven was not a goal of this project at this point the second UC3-A3 Xplained board was introduced into the system aiming at improving the AVR32 driver on both boards. This change in the experimental setup also solved, as a side effect, the endianness issues. Although if in a future a LittleEndian device is connected to the Internal Wireless Bus it will be required to *recode* the data.

Naturally the second AVR32 had an identical setup as the first, as depicted in figure 4.1.

4.7 Getting interrupted

To develop an interrupt driven driver was one of the goals of this project since the beginning but why is it so important?

At a first glance: having an interrupt driven driver would eliminate the need for actively waiting and polling the radio checking if the internal wireless bus interface has received anything. This constant checking is not only unnecessarily expensive (computationally speaking) but is also disruptive to the system (4.6.2).

Additionally the use of interrupts grants a *more natural* scheme of communication. Looking at it from a human communication point of view: we do not periodically (and often) check our phones to see if there is an incoming call but rather we wait until the phone *interrupts us* (by sound or vibration) that there is an event that requires our attention. Furthermore if we were to constantly check our phones most of our other tasks would suffer from that behaviour. Thus the analogy of the telephone proves how important the interrupt driven approach is and how damaging being constantly checking (ie, blocking calls) can be.

The nRF24L01 has mechanisms to notify the MCU that it's attention is required. The AVR32, and virtually all other processors, have mechanisms to listen to such attention requests (interrupts). Both were used in this master's thesis.

4.7.1 Listening in

The AVR32 is capable of *paying attention* to interrupts from several different sources. Some of them from it's own peripherals and others that are external to the chip. Naturally for the purposes of the driver the latter were used.

Generally the procedure to use an external interrupt entails three steps, here they are detail by using as an example their implementation on the project:

Preparing the line

The AVR32 has several I/O lines that are accessible through it's physical pins. These lines are under control of the GPIO module unless they have been granted to another peripheral[(4.3.1). The lines that are going to be used to *listen in* to external interrupt requests need to be configured as inputs, and to have GPIO relinquish over to the External Interrupt Controller. Snippet 4.11 shows how it was done during development.

Listing 4.11: Configuring pin as input

```

1 void extint_pin_setup()
  {

    static const gpio_map_t extint_gpio_map = {{
      AVR32_EIC_EXTINT_2_PIN, AVR32_EIC_EXTINT_2_FUNCTION}};
5   gpio_enable_pin_pull_up(AVR32_EIC_EXTINT_2_PIN);

```

```
gpio_enable_module(extint_gpio_map, sizeof(extint_gpio_map) /  
    sizeof(extint_gpio_map[0]));
```

```
9 }
```

The line is configured as an input with a pull-up resistor, such a setting was adopted as the nRF24L01 *clears* the line when it requests for attention. The ASF functions were relied upon when the configuration of the pin was done.

Unfortunately none of the pins broken out in connector J1 of the Xplained development board can be used to receive external interrupts, as they are not wired to the External Interrupt Controller. Thus a pin on J2 had to be used, requiring for an additional single cable. The single cable that is not bundled in the ribbon in figure 4.1 is the IRQ line.

Binding an ISR

An interrupt demands an action to be taken to address it's request. Back to the analogy of the phone call, a ringing phone needs to be answered. It is therefore necessary to define an action to be taken when the interrupt event is detected and that action needs to be bound to the event. The action to be taken is known as an Interrupt Service Routine (ISR). Once more on the phone analogy: *when* the phone rings (event) *answer* (ISR). Listing 4.12 presents both: the action defined and the binding.

Listing 4.12: ISR and binding

```
1  
ISR(radio_irq_handler, 1, AVR32_INTC_INTLEV_INT0) {  
    xSemaphoreGiveFromISR(semaphoreRadioEvent, NULL);  
    gstatus = nrf24_statusRead();  
5    eic_clear_interrupt_line(EIC_RADIO_1, EXT_INT2);  
}  
  
void init_ext()  
9 {  
    extint_pin_setup();  
  
    INTC_register_interrupt (&radio_irq_handler, AVR32_EIC_IRQ_2,  
        AVR32_INTC_INTLEV_INT0);  
13    eic_init (EIC_RADIO_1, &nrf24_irq_eic, 1);  
    eic_enable_interrupt_line(EIC_RADIO_1, EXT_INT2);  
    eic_enable_line (EIC_RADIO_1, EXT_INT2);  
17 }  
}
```

The ISR will be called, it was defined using the compiler directive for ISR's. The compiler needs to be aware of the fact that it is not a common function as some special instructions are required when compiling it. Most notably the use of `RETI` instead of `RET` to return to the previous context. The contents of the ISR are discussed further in subsection 4.7.2.

Function `init_ext()` includes a call to `extint_pin_setup()` presented in listing 4.11, but additionally registers the ISR (`radio_irq_handler` to the specific line being used (EXTIRQ2) and gives it a priority (INT0).

This left the action defined and bound to which event should trigger it. The last three lines on 4.12 are detailed on the next bullet point.

Enabling interrupts

Although interrupts are practical and allow for event-driven software, they can at times be undesirable. For example on a time critical zone of code it might be inadvisable to switch context and waste time attending an interrupt. Or interrupts being triggered before the system is properly configured (both the one that generates it or the one that processes it) as they could result in undefined behaviour.

To solve these issue most platforms allow to activate and deactivate the acceptance of interrupts. AVR32 offers such solution and accompanying ASF functions to use it, as seen on the last three lines in listing 4.12.

Generally there is also a general interrupt switch, in our case it was already set to accept incoming interrupts as it was required for the UART-on-USB functionality. It is an ASF call (`cpu_irq_enable()`) in the main function.

These three steps prepared the AVR32 to receive interrupts. Before tying it to the nRF24L01 it was tested with a simple cable, forcing it to high and low levels. Such level switched would produce interrupts and those bound to an ISR that would toggle a LED allowed for a visual verification that the MCU was acceptin and attending incoming interrupts.

4.7.2 Speaking out

It is obviously not enough to have the AVR32 ready to take interrupts, it is also necessary to set up the nRF24L01 3.3 to generate interrupts. The radio transceivers can generate an interrupt for three different events: a packet has been received, a packet has been successfully sent (it has been acknowledged by it's target) and a packet transmission failed (it exhausted retrial attempts and still got no acknowledgement from it's target).

In 4.4 the initialization process of the radio transceivers was introduced. During that process the nRF24L01 is configured to notify through it's IRQ pin none, some or all of those events. Namely in the configuration register `NRF24_REG_00_CONFIG` there are possible flags: `NRF24_MASK_RX_DR`, `NRF24_MASK_MAX_RT` and `NRF24_MASK_TX_DS` that activate the data received, maximum retrials and data successfully sent interrupts respectively.

So, at this point, we had the AVR32 ready to react to interrupt requests and the nRF24L01 prepared to make those same requests. In other words our caller was ready to make the phone call and the callee was ready to answer to it. But what do they talk about? There are three possible events that could trigger an interrupt, but there is only one interrupt line therefore the same interrupt will be triggered for all three options. Then within the ISR the system distinguishes which event occurred and addresses it. The ISR, introduced earlier,

in listing 4.12 is very simple: as it performs only three commands: note that a radio event has occurred, get the current status of the radio transceiver (as it will be processed later, outside the ISR), and let the nRF24L01 know that it's interrupt has been processed.

Interrupts must be as brief as possible that is why it was decided that the events were to be notified to the system and then processed outside of the ISR. It might seem counter-intuitive to increase the complexity of the software. Why not simply process the event in the ISR ? All of this is done to make sure the ISR is short, both in code and execution time. This derives from the fact that while on an interrupt other interrupts are generally inhibited therefore anything that is interrupt driven works anomalously. That includes: timer (scheduling) which is critical in a real time system. It is also advisable to avoid all kinds of I/O operations while in interrupt context because they are slow. But it could not be avoided because the nRF24L01 clears the interrupt source once the IRQ line has been cleared. So a compromise was made: the ISR reads and stores the kind of event that triggered the interrupt and then notifies the system that this event needs to be processed, the event is then processed in normal execution context. Keeping the I/O operation in ISR short (only one status byte is transmitted).

This notification is done by means of a semaphore (detailed in 3.5.1), that allows to have a Task suspended, thus not consuming resources, and waking it up when the event has happened. A *software based* interrupt of sorts. Such a Task (listing 4.13) awaits for the semaphore `semaphoreRadioEvent` to be set and then dispatches the event to the specific task, again via semaphores, depending on the status of the radio. These tasks will then operate the queue structures (described in 3.5.2: one for received messages and one for messages to be sent, respectively `rxQueue` and `txQueue`). Applications will then operate with them via the queue API.

Listing 4.13: vRadioEvent Task

```
1 void vRadioEvent(void *pvParameters)
  {
    nrf_frame_t qm;
5   while(1) {
      if(xSemaphoreTake(semaphoreRadioEvent, portMAX_DELAY)) {
          if( gstatus & NRF24_MASK_RX_DR) {
9             xSemaphoreGive(semaphoreRX);
          }
          if( gstatus & NRF24_MASK_TX_DS) {
              xSemaphoreGive(semaphoreTX);
13          }
          if( gstatus & NRF24_MASK_MAX_RT) {
              xSemaphoreGive(semaphoreMRT);
          }
17      } else {
          /*SemaphoreRadioEvent could not be acquired (SHOULD NEVER END
              UP HERE)*/
          }
      }
21 }
```

Figure 4.4 depicts this the usual execution flow of sending a data packet.

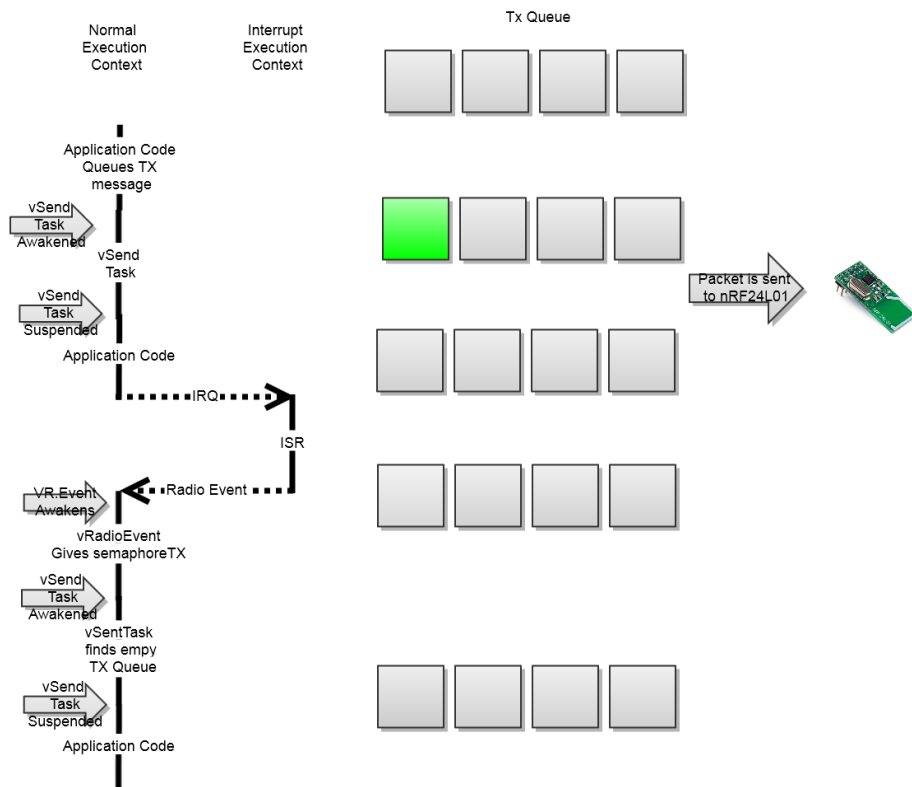


Figure 4.4: TX operation execution flow

At this point the flow of the program diverges depending on which of the event triggered the original interrupt, the following subsections explore the cases.

4.7.3 Receiving a Data Packet

The first situation that was tackled was the reception of a packet. Two facts made it the best option to start. In the first place: receiving a packet is the most asynchronous of the events. As the local system has no way of knowing when the next incoming packet is going to come in. The second fact is: a reception *can't go wrong*, meaning that the nRF24L01 only notifies of a new reception when the new packet is already in its buffers waiting to be read by the MCU. The idea is simple then: once a packet is received the software should collect it from the buffers on the transceiver and place it in the AVR32's RAM to make it available for the applications.

Subsection 4.7.2 presented how the events were dispatched to different tasks. There is

then a specific task (also suspended by the semaphore mechanism unless there are pending messages to be treated). Listing 4.14 presents it:

Listing 4.14: vReceive Task

```
void vReceive(void *pvParameters){
2
    nrf_frame_t qm;

    while(1) {
6        if (xSemaphoreTake(semaphoreRX,portMAX_DELAY){
            nrf24_recv((uint8_t *) &qm.content ,&qm.size);
            xQueueSendToBack(rxQueue,&qm,0);
        } else {
10         //Unobtained Semaphore (Should not happen)
        }
    }
}
```

The task `vReceive` gets via the SPI, by using the low level driver, the packet and queues it in the `rxQueue` for applications to *collect* it. The whole *montage* brings two benefits to the system: One of them being the fact that an application can request to read a packet at any point by using `xQueueReceive(rxQueue, . . .)`. If the packet is ready it will be served to the application. If it is not, the application will be suspended until said packet becomes available. The second advantage is it's symmetrical a packets are received as sent to RAM regardless of whether they had been requested or not, as there is no longer need to actively wait for them.

The current implementation is limited in the sense that there is only one queue for received messages thus an application could be awakened by a message that was not meant for it. The application could then simply requeue the message and be again suspended. Another option would be to have separate reception queues for each application and task `vReceive` could deliver to the pertaining one.

Figure 4.5 depicts this the usual execution flow of sending a data packet.

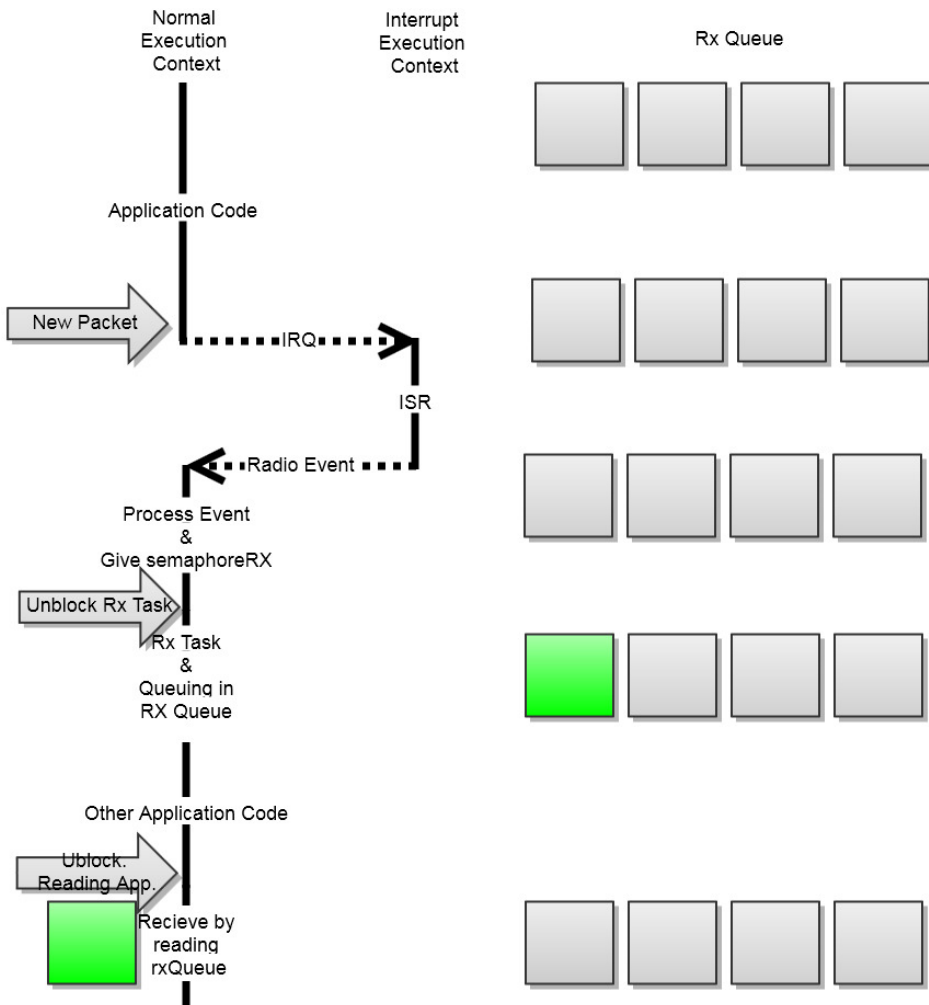


Figure 4.5: RX operation execution flow.

4.7.4 Sending a Data Packet

The internal wireless bus is designed as half duplex, because a transceiver is either sending or receiving. It could be argued that transmitting does not need to be interrupt driven, as an application is aware of the moment it starts sending something and then the transceiver could simply be polled until the packet is acknowledged or a delivery error is detected. While it is true that avoid the transmit phase active waits is not as paramount as those of reception (because the active checking there would be always on going), it is also true that those MCU cycles used checking are wasted and could be put to a much better use. The principle here is symmetrical to the one used during sending. Thus it starts at the local ap-

plication and progresses up to the transceiver. The first step is the queuing of the packet by the sending application through the use of the `xQueueSendToBack(txQueue, ...)`. That will awaken task `vSend()` (see listing 4.15) that is usually suspended unless there are pending packets to send.

Listing 4.15: `vSend` Task

```
void vSend(void *pvParameters){
2   while(1){
       nrf_frame_t qm;
       if (xQueueReceive(txQueue, &qm, portMAX_DELAY)){
           nrf24_setTransmitAddress((uint8_t*)TARGET_ADD, 5);
6       nrf24_send((uint8_t*) &qm.content, qm.size , false);
           nrf24_waitPacketSentSemaphores();
           nrf24_powerUpRx();
       } else {
10      /*NO MSG TO SEND*/
       }
   }
}
```

Task `vSend()` sets the target address of the sending to be done, then uploads the payload and gives the command to send it. `nrf24_waitPacketSent()` is the original function provided on the Arduino library that would poll the status register to monitor the current sending. In an attempt to avoid the use of active polling a function based on events (signaled by semaphores) was developed: `nrf24_waitPacketSentSemaphores()` Its use is exemplified in listing 4.16.

Listing 4.16: `nrf24_waitPacketSentSemaphores()` Function

```
uint8_t nrf24_waitPacketSentSemaphores()
{
3   if (xSemaphoreTake(semaphoreTX, 50/portTICK_RATE_MS)) {
       nrf24_spiWriteRegister(NRF24_REG_07_STATUS, NRF24_TX_DS |
           NRF24_MAX_RT);
       return true;
   }
7   if (xSemaphoreTake(semaphoreMRT, 50/portTICK_RATE_MS)) {
       nrf24_flushTx();
       nrf24_spiWriteRegister(NRF24_REG_07_STATUS, NRF24_TX_DS |
           NRF24_MAX_RT);
       return false;
11  }
   return false;
}
```

The function `nrf24_waitPacketSentSemaphores()` relies on the semaphores: `semaphoreTX` and `semaphoreMRT` that are in turn set by the interrupts and Task `vRadioEvent()` (4.13). In this manner it can return the status of the sending (be it successful or not) while avoiding constant polling, as the semaphores suspend it unless there is a change. The timeouts (maximum amount of time it waits for a semaphore) are necessary as there are two semaphores and both need to be checked. Else it would be possible that the

function waits for `semaphoreTX` and it never wakes up because transmission failed and actually `semaphoreMRT` was the one given. In the event of a sending failure the outgoing buffers are flushed to allocate resources for the next transmission. Although the current implementation does not have this feature, it might prove useful in the future to requeue in `txQueue` the outgoing message if `nrf24_waitPacketSentSemaphores()` as the system knows then that the transmission failed.

On `vSend()` (4.15) the call to `nrf24_powerUpRx()` had to be added. On the original Arduino code (not interrupt driven) the transceivers were set to RX mode when a packet was being actively awaited, but on the new driver it was necessary to return to RX mode after each transmission. This return to RX is not automatic as the state diagram on 3.3 shows. The implications of not returning are that after doing a transmission that node could no longer receive any incoming packet (as it was not in RX mode). Observing this behaviour pushed the addition of `nrf24_powerUpRx()` to be ran after each transmission.

Combining both the interrupt-based reception and sending the driver and stack are fully operational in RAW mode(2.2.1). Albeit only two complete nodes were used during the development (both on AVR32), and an additional *mock* node on the Arduino, the system is scalable theoretically up to filling the nRF24L01 address space.

4.8 CubeSat Space Protocol

Integrating the Internal Wireless Bus with the CubeSat Space Protocol would increase even more the flexibility and transparency of the implementation. That is, an application that operates through CSP does not even need to know which interface will eventually carry it's payload.

CSP is bundled as an external library that can be compiled and then linked into the project. It's developers provide a script based in WAF to manage the build process. WAF in turn is based on Python. Although to build it using the provided scripts is the usual path to take it is not the one that was taken in this project.

NUTS has already a compiled CSP library [4] to be integrated with the system, as it is also used in the wired I2C connection. To guarantee compatibility it was decided to link the project against that compiled version. The commonality in the architecture and development tools allow for the *drop in* placement.

CSP is designed in a modular fashion where there is the core engine of the protocol and then different interfaces can be added, granted they provide a common interface to the mentioned core: One function to send packets so that the core could rely on it to deliver a new packet to the interface. And a second function to get packets from the interface and feed them to the CSP core.

Despite the original intent to have a fully working CSP integration with the internal wireless bus, this goal could not be met. No technical obstacles were detected during the

attempts and the failure at implementing these features was due to lack of time. More information provided on section 7.2.

4.9 Implementation Outcome

Despite the failure to produce a working integration with CSP there have been usable results. The driver is usable and could be easily integrated and deployed with the final NUTS system.

Taking advantage of the fact that most NUTS submodules are already running on FreeRTOS the RAM memory footprint of the driver is reduced to the two queues (`txQueue` and `rxQueue`). Naturally the driver functions and the FreeRTOS tasks associated with the nRF24L01 have RAM usage but only while on execution therefore they are not a constant drainage of resources. As far as program memory is concerned : most of the data structures and algorithms to manage them are based on FreeRTOS and therefore the code is *shared* with the rest of the system. The only *purely* nRF24L01 code cost are the ISR, the `vSend()`, `vRecieve()` and `vRadioEvent()` tasks and the low level driver itself. And only the latter is of a significant code size.

Additionally the command shell, which has been used during this project, despite it is a good tool during development for interactive testing it will not be included in the final system. As it will be replaced by the real application in NUTS. Although some of it's mechanisms used for parsing and registering commands could well be used when processing the commands sent from ground to the satellite. But in a general view the memory and code size footprint of the console are not a source of concern as they will not, *per se*, be in the final integrated system.

The whole design structure aimed at make the driver interrupt driver and non-blocking can benefit a real-time system such as NUTS.

Chapter 5

Integration Guide

The end goal of the solution designed and developed for this master's thesis is to be integrated and used in the NUTS satellite. This chapter provides the necessary information to ease the integration process. The chapter itself is split in two parts: the first with the necessary programming considerations and the second presents the *Software Design Document* of this implementation

5.1 Programming Considerations

This section contains several points that have to be considered when integrating the Internal Wireless Bus into other systems (mostly aimed at the NUTS Satellite). Although they all have relevance they are not necessarily linked to one another.

5.1.1 Basic Integration Requirements

The most basic integration requires to, naturally, move the code base to the new project. The following subsection present a relation of the elements to port.

Elements to Port

Maybe the most important part of this master's thesis is to easily integrate it into the bigger NUTS system. The code base has been developed in a manner that enables said integration. The following list of files (both headers and source code) that must be present in any project that integrates the Internal Wireless Bus:

nrf24l01.c and nrf24l01.h

These files contain all the low level operations necessary to interact with the nRF24L01 modules. Additionally the header file defines the register names for the transceiver thus it allows for a much clearer code and debugging. The operations are based on the ASF therefore if the wireless bus were ported to another architecture changes to nrf24l01.c to match the new SPI routines would be necessary. The SPI unit that is in use and the GPIO pin used as Chip Enable for the radio modules is also defined there and both would require adjustment in case of a hardware change.

user_interrupts.c and user_interrupts.h

The ISR for the external interrupt is defined there. Also the necessary functions to initialize the AVR32 hardware to receive those interrupts. The specific pin used is defined there and needs to be kept coherent with any changes done to the physical connections.

msgQueue.c and msgQueue.h

Both reception and transmission queues are based on these files. Also the functions to create and initialize them. The structure that is enqueued is defined in the header file, so that is where modifications must be made in case new fields are thought necessary. Since the queues are purely on RAM structures only logical changes affect them and no modification is necessary due to hardware changes.

stdio_usb_example.c

It is the file containing the `main()` function, naturally not all the file must be used when integrating into other projects, but some portions need to be extracted and moved to the new project. The list follows:

void vSend(...)

This Task monitors the txQueue and in case there is data pending to be sent, sends it. It is obviously a very important part of the system as it binds the queue system (purely software) with the nRF24L01 modules. As the function is right now all packets are sent to the address: `brd01`, obviously in a system with multiple nodes this would require a change. The function `nrf24_setTransmitAddress(...)` allows that change in execution time. A possible implementation would be to have a *target* field in the enqueued frame and then when dequeuing it for sending the intended address could be set.

This Task also checks (using `nrf24_waitPacketSentSemaphores(...)`) that the transmission was completed successfully.

void vRecieve(...)

It is the complimentary to the prior Task. This one is activated (as it usually lies suspended) when new data is available in the nRF24L01 modules, then reads the data from the transceivers via SPI and enqueues them in the reception queue. The current implementation has a single reception queue but it would be possible to enqueue it on different structures depending on, for example, priorities or the targeted task.

void vRadioEvent(...)

As described in 4.7.2 there are three possible events on the nRF24L01 that interrupt the AVR32 and request it's attention. This Task distributes this requests for attention the the proper Tasks or functions for processing. It could be been as a *software switchboard*. Unless the structure of the semaphore-based synchronization is changed (and that is inadvisable) no changes should be made to this Task.

uint8_t nrf24_waitPacketSentSemaphores()

This function catches semaphores relating to the events of successful and failed packet transfer and notifies the calling task of the outcome additionally it flushes the nRF24L01 buffers if necessary. No modification is advised.

Those are the elements that should be present in any deployment of this solution. It is important to mention that many of them are based on ASF or FreeRTOS. The following subsection describes their use. These source code files can be found in the annex (7.4).

Additional Software Requirements

The ASF and FreeRTOS are integral parts of the system yet they have not been considered as a segment necessary *to port* because it is assumed that they will present in any future software for NUTS where the Internal Wireless Bus is integrated. Considerations about their configuration follow:

Atmel Software Framework

The ASF is split in modules. They are all accessible via an integrated ASF Wizard in Atmel Studio. The ASF version used during the development is 3.5.1. There are significant differences between versions, specially between major versions, but generally they are enhancements. Therefore, it is very likely, that any version above 3.5.1 will keep the functions used. It is the header inclusion `#include <asf.h>` must naturally be present in all files that used the ASF but it is also required to use the ASF Wizard to include the desired features into ASF. The list of used features (and thus required):

```
EIC - External Interrupt Controller (driver)
GPIO - General purpose Input/Output (driver)
SPI - Serial Peripheral Interface (driver)
Delay routines (service)
Generic Board Support (driver)
USB Device (service configured as cdc_stdio)
Interrupt Management (Common API) (driver)
```

Replicating this configuration should enable all the features necessary to operate the the nRF24L01 through the software stack designed and developed during this master's thesis..

FreeRTOS

To comply with the multitasking nature of NUTS FreeRTOS will be certainly deployed in any future development. Some of the optional features of FreeRTOS have been used to implement this project, therefore it is required for them to be enabled. The common way to configure the real time operating system is through one of its header files: `FreeRTOSConfig.h`. In that file there must be the line:
`#define configUSE_COUNTING_SEMAPHORES 1`

That line enables semaphore functionality in FreeRTOS.

This concludes the requirements in software that must be ported to the a NUTS software for integration.

5.1.2 Skeleton of the Main Program

It does not suffice to bring all the necessary pieces of code. It is also necessary to organize it in a specific manner and to initialize all required structures. Listing 5.1 presents a minimal structure that is required for the system to function. It is assumed that the required steps will be done in the main function although it would also be possible to place them elsewhere as long as the execution order is kept. Inline comments have been added to help the reader.

Listing 5.1: Skeleton of a main function

```
void main()
3 {
    ...

    /*Initiallize and stabilize clocks*/
7   sysclk_init();

    /*Initialize and enable interrupts*/
    irq_initialize_vectors();
11  init_ext();
    cpu_irq_enable();

    /*Declare two auxilliary variables requires for nRF24L01
       configuration*/
15  NRF24DataRate dr = NRF24DataRate2Mbps;
    NRF24TransmitPower pr = NRF24TransmitPower0dBm;

    /*Initiallize the SPI interface and the nRF24L01*/
19  nrf24_initRadioSpi();

    /*Configure nRF24L01*/
    nrf24_setChannel(1);
23  nrf24_setThisAddress((uint8_t*)THIS_ADD, 5);
    nrf24_setPayloadSize(PAYLOAD_SIZE);
    nrf24_setRF(dr, pr);

27  /*Initialize RX and TX queues both with size 8*/
```

```
    initMsgQueues(8,8);

    /*Initialize Semaphores*/
31 semaphoreRadioEvent = xSemaphoreCreateCounting(1,0);
    semaphoreRX = xSemaphoreCreateCounting(3,0);
    semaphoreTX = xSemaphoreCreateCounting(1,0);
    semaphoreMRT = xSemaphoreCreateCounting(1,0);
35
    /*Create the three necessary tasks*/
    xTaskCreate(&vRecieve, (const signed portCHAR *)"Recieve Task",
               configMINIMAL_STACK_SIZE, NULL,
               tskIDLE_PRIORITY+3, NULL);
39 xTaskCreate(&vRadioEvent, (const signed portCHAR *)"Radio Event
    Processing Task",
               configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+3,
               NULL);
    xTaskCreate(&vSend, (const signed portCHAR *)"Send Task",
               configMINIMAL_STACK_SIZE+8192, NULL,
               tskIDLE_PRIORITY+3, NULL);
43
    /*Launch FreeRTOS scheduler*/
    vTaskStartScheduler();

47    ...
}

```

It must be kept in mind that the presented snippet of code (5.1) is meant only as an example and it omits the declaration of required variables and of the required functions. It is understood that any future developer will take the limitations of this example in mind. Regarding the priorities associated to the Tasks: they are designed to operate properly when having the same priority (although they may work as well under different priorities), and it must be higher than any task that could block the system for indeterminate amount of time.

5.1.3 FreeRTOS Tasks' Stack Size

When the function `xTaskCreate(...)` is used to add a Task to the scheduler (for example in listing 5.1), a size for the stack is assigned. Generally the minimal stack size suffices unless very memory demanding functions (such as `printf(...)`) is used from within the task. An undersized memory does not present a single distinguishable failure but rather erratic behaviour from that task. Such errors can be hard to diagnose. It is advised to any developers to check for the proper stack size to use by trial and error. Generally stepping through the Task once with the debugger suffices.

5.1.4 Endianness

As discussed in 4.6.2 it is possible to have nodes using different endianness schemes. That depends on which hardware are those nodes based on. It is advised to adapt all platforms other than the AVR32 to the Big endian scheme because it will be the most commonly used in the NUTS subsystems.

5.1.5 Include Paths

Due to some problem with Atmel Studio installed on the development computer it was necessary to hardcode the path to some header files in their respective `#include` directives. Even configuring the IDE to seek header files in the directories hosting them there were still problems to locate the files during compilation. There is no reason to believe that this erroneous behaviour would manifest in other development settings but it is necessary to remember that it might be necessary to switch those paths back to *relative* instead of *absolute*.

5.1.6 Available Functions

The system is constructed in such a way that there should be no need to access the driver level functions. In case that they would be required the documentation of the original Arduino library is still widely usable as the behaviour and the return values have been kept during the port. And behave as described in [7]. Those functions that are new or that underwent significant change are detailed in chapter 4.

5.2 Software Design

The nature of the NUTS project implies rapidly changing engineering teams. This enhances, even further, the need for a clear and standardized documentation. Such a document is not meant as a substitute for the present master's thesis but rather as a quick reference guide to get an overview of the system. Within the NUTS project a standard was developed [11]. One of this thesis' aims is to produce helpful documentation therefore the description of the software following said standard follows.

5.3 Software Design Document

Project Name/ID

NUTS - 1.1.3 Internal Wireless Bus

Creation Date

03/07/2013

Author(s)

Jordi Francès Matas

Change history

Initial version

Context

The Internal Wireless Bus provides a high bandwidth link to the NUTS satellite for inter-module communication. It aims both at solving the limitations of the I2C communication scheme and to test the viability of using small radio transceivers (nRF24L01) in intra-satellite communication.

Composition

Two main layers can be distinguished: a logical layer presenting a non-blocking interface for applications to use the communication interface. And a driver layer that interacts directly with the hardware. There are intermediate components to ensure the binding between the two layers.

Implementation

The logical layer is based on FreeRTOS queues. Specifically there is one queue for sending packets where the applications enqueue outgoing messages. And complimentary there is a queue for receiving packets where the system enqueues the incoming messages so that the applications can access them in an orderly fashion.

The driver layer interacts directly with the nRF24L01 transceiver via the SPI bus, managing all the low level transactions necessary to send and receive the messages.

The intermediate components (based mostly on interrupts and semaphore-signalling) ensure timely communication between the driver and logical layer.

Dependencies and interfaces

The top level (logical) depends on FreeRTOS' queue system.

The driver layer is based on the ASF to operate the SPI unit and the GPIO lines.

The interface exposed to users of the internal wireless bus is the FreeRTOS Queue API as all interaction is eventually simplified to queuing and dequeuing packets.

Glossary

I2C: Inter-integrated Circuit Bus.

nRF24L01: Nordic Semiconductor's integrated 2.4Ghz radio transceiver.

SPI: Serial Peripheral Interface.

ASF: Atmel Software Framework.

Testing and Results

This chapter summarizes the outcome of the work done throughout the thesis and the methods used to gather such information.

6.1 Testing

Admittedly the testing portion of this master's thesis is the most lacking one. No actual systematic testing could be performed due to time constraints (see 7.2). Nevertheless each step of the implementation was tested, it could be seen as an iterated unit testing at each implementation version. As a matter of fact many of the decisions and adjustments that were done during the development were made upon seeing the result of these *unit tests*.

The interactive console 4.6 was very helpful in this informal testing process. Allowing to make fast runtime adjustments or to reproduce *bug suspicious* behaviour. This advantages justify the development and use of an interactive shell despite it was not meant to be integrated in the final product.

Over the many iterations of development this limited testing gave an impression of viability and robustness of the system. But it is inadvisable to rely on such preliminary results, especially for a system such as NUTS where reliability is key. It is of paramount importance to have the system undergo a more rigorous test scheme (see 7.1).

Regardless of the amount of testing done, it is strictly necessary to have the system undergo reliability tests while it is being integrated in the final NUTS satellite. This has been listed as a high priority future work in 7.1

Despite range testing was not a concern of this project since purely radio concerns were addressed by previous work [8]. Informal trials were done by transmitting at ranges of up

to 25 meters in a clear line-of-sight environment. All ranging trials were successful.

6.2 Results

As aforementioned it is not possible to provide quantitative results due to limited testing but some assessment can be done on a purely qualitative basis.

First and foremost: the system works. Data packets were transferred back and forth between two AVR32-based nodes using a FreeRTOS friendly programming model that provides non-blocking transferences. And, in the author's opinion, it could be easily integrated into the satellite to provide a secondary internal communications bus.

From a software point of view one of the objectives was not met: integration with the CubeSat Space Protocol. Despite attempts to attain so.

Looking back at the initial goals:

Bandwidth

The Internal Wireless Bus exchanges information at a maximum speed 2 Mbps, it is possible to configure it at 1 Mbps to ease energy concerns. In both cases it is well over the 400 Kbps provided by the I2C bus on the backplane. The improvement in bandwidth can help to move big blocks of data such as those produced by the IR Camera mounted as a payload.

Flexibility

Among the downsides of the I2C bus is the requirement to have it's lines run throughout the entire satellite, or at least reaching all those points were access to the bus is requires. This imposes a footprint on the backplane. The wireless nature of the solution explored in this document eliminates, by definition, the need for such long lines. Instead it requires much shorter lines, never going across modules, that connect the processors of the different modules to their respective nRF24L01. Doing so increases flexibility in the sense that the required communication wiring is moves from the backplane to the submodules. And this allows for a more independent design of the submodules and even a freer placing of them on the mounted satellite. Obviously this flexibility can not be exploited by NUTS because the I2C bus is still present. But it may be exploited in newer designs.

Experimenting

The goal of exploring whether such a system could be viable in a satellite is still on-going. And will stay on-going until the system is deployed and actually used in space. But the first steps have been taken and they contribute to the desire of using NUTS also as a platform to develop new technologies.

Finally, and from a software point of view: it is objectively good that the solution could be achieved purely by developing software for the MCU. That is: the radio transceivers are used without any modification to their internal firmware. Such modifications are not always possible due to the lack of development tools or documentation and when done

they may lead to performance or reliability problems. Therefore it is safe to conclude that it is positive that such modifications were not required.

Conclusion

This section concludes this master's thesis report with the author's personal opinions and considerations of the work done. It also identifies the problems encountered, what could and should have been done differently. There is also a listing of possible paths to continue the work.

7.1 Future Work

There are two very clear axis around which high-priority future actions should be targeted:

Integration with CSP

Maybe the most notable shortcoming of the thesis has been the failure to integrate the system with CSP. The system would gain greater usability with a working integration with an established protocol such as CSP. Furthermore it would make the solution more attractive to similar projects that might find the idea of a high throughput wireless bus attractive.

The integration with CSP would also enable the development of the *two networks on one adapter* idea as described in 2.2.2.

Testing

As discussed in 6.1 it is a lacking portion of this work. Despite it would not add features *per se* it is a necessary job. Taking into account the the this solution aims at being deployed in a critical environment (such as a satellite) one must be certain of it's reliability.

The two aforementioned items should have, in the author's opinion, priority but there are also some other development paths that are worth exploring.

There might be some energy consumption concerns regarding the use of a radio. During this work such concerns have not been tended because it was understood as a proof-of-concept development from an almost exclusively software point of view. But the limitations in the scope of this project do not, by any means, imply that power consumption can be neglected. If it was found out that the power consumption is too high there are two suggestions to keep it at bay.

A possible approach would be to have the wireless bus in stand by (and not defaulting to RX status) and bring it up only when a transfer needs to be done. This naturally requires for both nodes, sending and receiving, to be aware that an exchange is about to take place. The I2C bus could be used to coordinate both nodes when it is necessary to bring the transceivers out of stand by. This approach requires for the duality of I2C and radio transceivers to be present, while it is a valid solution for NUTS it would not be for a hypothetical future system where the I2C has been stripped. That leads to time slicing.

A time slicing scheme would imply that the transceivers are brought out of standby only for a fraction of time at regular intervals and all communication would happen during that time. It would increase communication latency but reduce energy consumption. It might be problematic to keep all the boards coordinated as there would be no other communication scheme present.

Another possible line of work would be to port the system to an ARM based processor. They are quite popular in the embedded world in general and in the CubeSat community in particular therefore such a port might bring interest from other projects.

7.2 Problems Encountered

Obviously some problems have been encountered during the realization of this project. Some of them could be either solved or circumvented while others implied cuts to the attainment of thesis goals.

The biggest problem faces has been time. The development of the working system (as it it presented in this document) consumed more a much longer time than it was planned. This pushed other tasks aside up to the point where some of them had to be abandoned and left as possible future work. The two main items that suffered from this were the integration with the CubeSat Space Protocol and testing.

The most obvious cause for a lack of time was most an overly optimistic planning. Yet some other factors could have weighted in. During the first stages of development significant effort was made to get the UART-on-USB working on a project from scratch. That was not part of the *core* of the project and maybe it should have been detoured by basing the development on an example where that functionality was already present, as it was eventually done.

A second issue, related with the first, was the limited and sometimes inaccurate ASF documentation. While the library itself is a good piece of software the bundled documentation was found to be quite lacking. Often referring to older and deprecated versions of the

library where functions or even the whole logical structure was changed. This forced to proceed blindly or by example. Yet often times no useful examples were found as they were based on older versions of the ASF as well. Additionally some libraries were found to hold *bugs* or *undocumented requirements* as details in 4.3.1. These factors impacted greatly on the earliest stages of the development while trying to get the SPI communication to work.

A third problem was the assumptions of similarity between PIC and AVR32 architectures made by the author. Given the author's familiarity with the PIC architecture many of the early stages were developed as if everything had an inherent similarity between the two architectures. While both architectures, like many other computer architectures, share some common traits it was an undesirable thought bias to overestimate the influence of said similarities.

All problems could maybe have been avoided, or palliated, with a more careful planning. A planning that takes into account the available documentation and the differences between architectures. Nevertheless the major issue was, and hence it is mentioned first, an overly optimistic planning. And while efforts will be made in a future to correct that, it is an error that drew mainly from the author's inexperience in larger projects. And that will hopefully correct itself over time.

7.3 Learning Outcome

Aside from an engineering project this master's thesis is also an academic work, as such it is conceived as a learning experience. From a purely technical point of view, experience was gained in some areas. Such as:

I had no experience developing embedded software on a real time operating system. I had developed embedded software before but it was logically designed as state machines of diverse complexity, yet only state machines. On the other hand I had taken theoretical courses regarding real timeliness but never had *hands on* experience on one. I consider this the biggest learning outcome of the thesis as it is not technology dependant therefore it will certainly benefit future projects.

The use of radio transceivers was also new to me. Nowadays there are many communication interfaces and in previous projects I had experimented with others but this was a first contact with radios.

It was not the first approach at using an AVR32 but it was by far the most extensive to date. This implied familiarizing with the tools for development and debugging.

On a more general note and taking into account that the project is encased in the NUTS initiative. It was certainly educational to have my own project within a much larger one. Although the systems developed by others were, not yet, interdependent there was not so much a coordination but rather a reporting and bringing others up to date with the current development of the project. Despite the systems were incompatible there was the possibility of collaboration, specially empowered by the fact that most software is being

developed in similar platforms. It was also interesting to partake in NUTS affairs that were not strictly bound to the thesis, such as finding ideas to recruit new students in the future.

An unexpected, yet very welcome, opportunity was the chance to participate, as a speaker [3], in the 5th European CubeSat Symposium.

7.3.1 5th European CubeSat Symposium

The idea of using wireless for the internal communication of a small satellite is rather new. As such it was an interesting opportunity to present it to the CubeSat community. The Symposium was held in Brussels where this work was exhibited through an oral presentation. The used slides can be found in the Appendix (7.4).

The idea was met, in my opinion, warmly by other members of the CubeSat community. Using wireless for NUTS is an experiment but during the conferences we learnt that it would solve some technical issues for similar projects. For example one of the projects were considering drilling through a black boxed hardware module to pass through communication lines. The risk of damaging the perforated module could be avoided by using wireless Technology. That increased my confidence in the potential future use of this technology. Maybe even beyond CubeSats. Devices with moving parts (making wiring troublesome) could also benefit from it.

Personally, it was the first time I presented in such an event. Despite the obvious nervousness I was happy in doing so and found the experience to be enriching.

To attend the symposium I applied to the European Space Agency for sponsoring and obtained it.

7.4 Personal Experience

On a purely personal point of view: I liked this thesis. I was interested in the topic and in the technologies that I got to explore. I was already involved with the NUTS project and wished to do my thesis within it, but the specific topic was chosen out of interest for development using radio transceivers and real time systems.

I also consider the experience of being a member of NUTS a very positive one. The existence of a team and a clear objective makes it easier to keep working towards it. It is different from working on an isolated project, and I clearly liked being part of NUTS.

On the negative aspects: it was sometimes frustrating. Specially in the beginning when there was visible results. And the lack of testing and integration with CSP leaves a bitter taste of incompleteness.

As a final thought: it was an overall positive experience and the produced system is a working prototype and I really hope it will be used and set in orbit when NUTS launches.

Bibliography

- [1] Arduino Community. Arduino development platform, 2013. <http://www.arduino.cc/> [Accessed 14/07/2013].
- [2] K. A. Ødegaard. Error Detection and Correction for Low-Cost Nano Satellites. Master's thesis, Norges teknisk-naturvitenskapelige universitet, Trondheim, 2013.
- [3] J. Frances. CSP on an RF Link. In *Book of Abstracts. 5th European CubeSat Symposium*, Brussels, Belgium, 2013.
- [4] A. Giskeødegård. Implementing CSP over I2C on NTNU Test Satellite. Project Report, 2013.
- [5] GomSpace. CubeSat Space Protocol, 2013. <https://github.com/GomSpace/libcsp> [Accessed 14/07/2013].
- [6] S. Kubrick. 2001: A Space Odyssey, 1968. Film.
- [7] M. McCauley. NRF24 library for Arduino , 209. <http://airspayce.com/mikem/arduino/NRF24/> [Accessed 14/07/2013].
- [8] E. Meland. Internal wireless bus in student satellite experiment. Project Report, 2011.
- [9] NUTS. NUTS Project Website, 2013. <http://nuts.cubesat.no/> [Accessed 14/07/2013].
- [10] Real Time Engineers Ltd. FreeRTOS, 2013. <http://www.freertos.org/> [Accessed 14/07/2013].
- [11] T. A. Thomassen. Software quality assurance on a student satellite. Master's thesis, Norges teknisk-naturvitenskapelige universitet, Trondheim, 2013.
- [12] Various. *nRF24L01 Product Specification*. Nordic Semiconductors, Trondheim, 2007. http://www.nordicsemi.com/eng/content/download/2730/34105/file/nRF24L01_Product_Specification_v2_0.pdf [Accessed 14/07/2013].

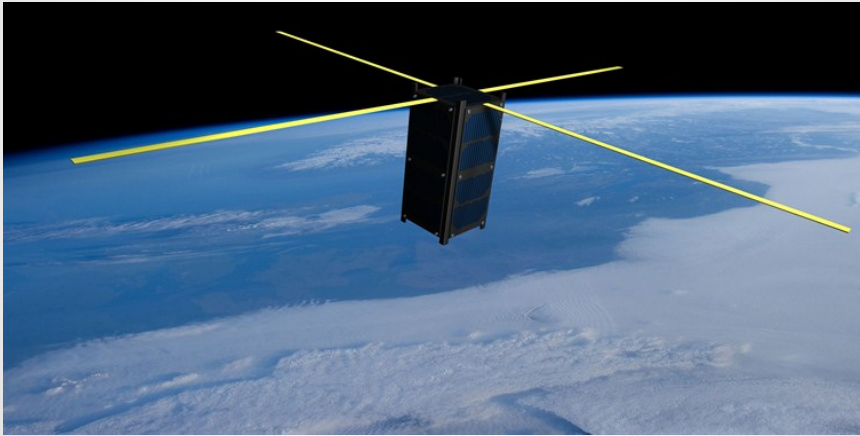
-
- [13] Various. *CubeSat Design Specification*. California Polytechnic State University, San Luis Obispo, California, 2009. Rev. 12.
- [14] Wikipedia. CubeSat Space Protocol — Wikipedia, the free encyclopedia, 2013. www.wikipedia.com/wiki/Cubesat_Space_Protocol [Accessed 14/07/2013].
- [15] Wikipedia. Serial Peripheral Interface — Wikipedia, the free encyclopedia, 2013. www.wikipedia.com/wiki/Serial_Peripheral_Interface [Accessed 14/07/2013].

Appendix

Presentation used in the 5th European CubeSat Symposium

The slides that were used to support the presentation of 15 minutes given in Brussels follow:

NUTS – CSP on an RF Link



Jordi Frances

May 31, 2013

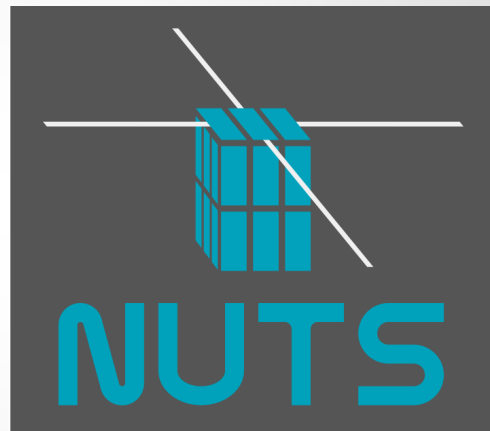


We are NUTS

NUTS

The Norwegian University of Science and Technology (NTNU) is currently engaged in the CubeSat project: **NUTS** – NTNU Test Satellite. Our main payload is an IR-camera that will enable the study of gravity waves. Additionally:

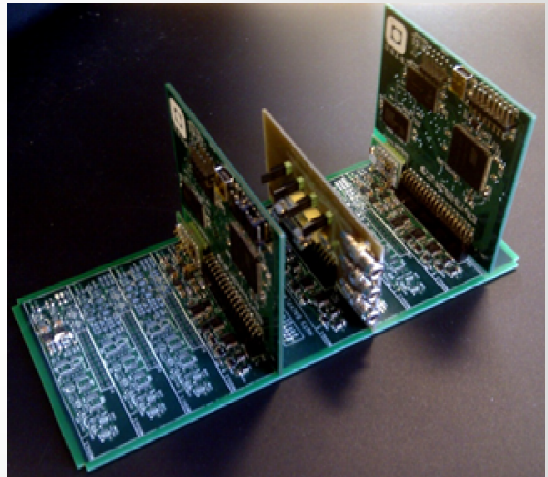
- Backplane
- Reinforced Plastic Based Composite Materials in a Custom Built Frame
- Wireless Internal Communication Bus



NUTS is composed of several modules:

- On Board Computer (OBC)
- Radio module
- ADCS
- Payload
- Power and temperature sensors

All modules are fitted on the backplane, providing power and I2C communication.



Cut the cord

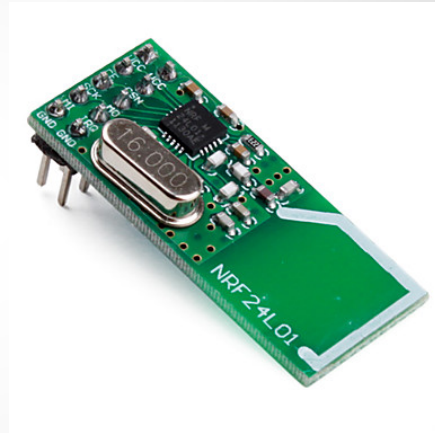
Buses are generally wired, but what if they were not:

- NUTS
 - Designed experiment
- Hardware
 - Reduces area requirements
 - Reduces weight requirements
 - Slightly increases in complexity
- Software
 - Slightly increases in complexity



There are several COTS solutions to provide a radio interface to an MCU. We are using the nRF24Lo1. Some of its features are:

- 2.4 Ghz Band
- Payloads of up to 32 bytes
- SPI interface with the MCU
- Auto-ACK
- Auto-sized Payload (RX)
- Up to 2 Mbps



Protocols

CubeSat Space Protocol:
Specifically designed for CubeSats. Provides a TCP/IP-like scheme. Some of its features:

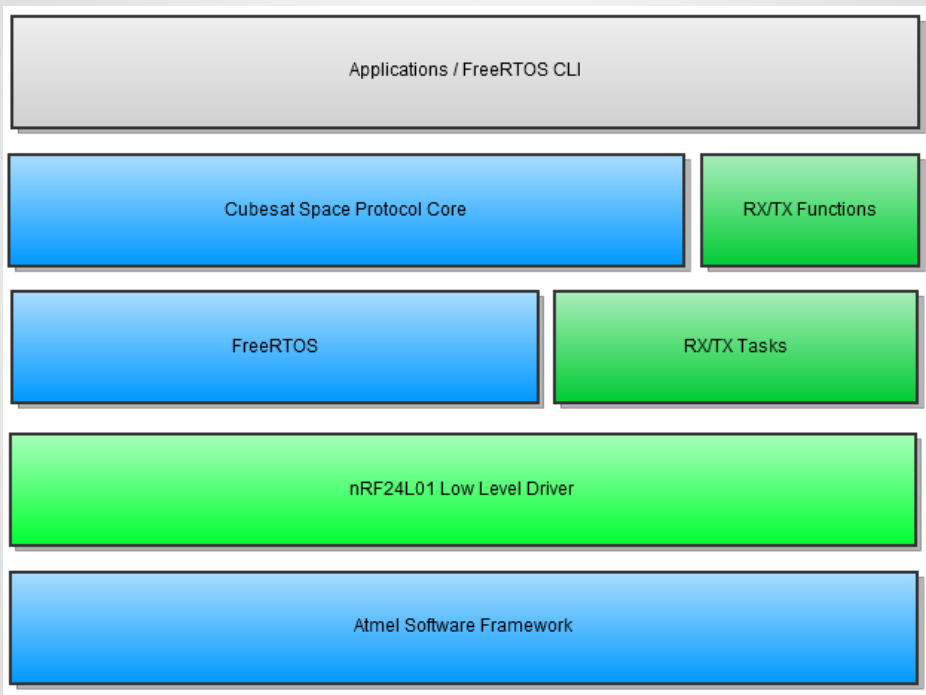
- Socket API.
- Connection oriented and connectionless modes
- Low level functions: ping, buffer status query...
- Supports several physical interfaces

Advanced functionality yet some overhead.

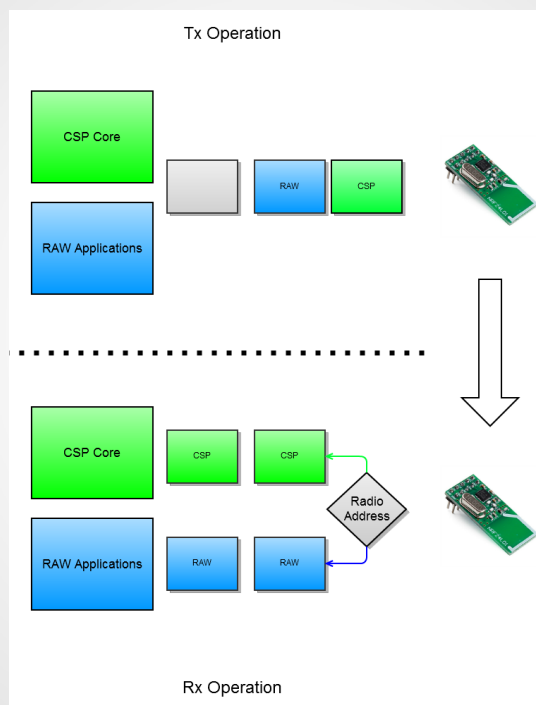
RAW/Ad-Hoc: Simply sending data packets through the radio interface with a minimal header or protocol.

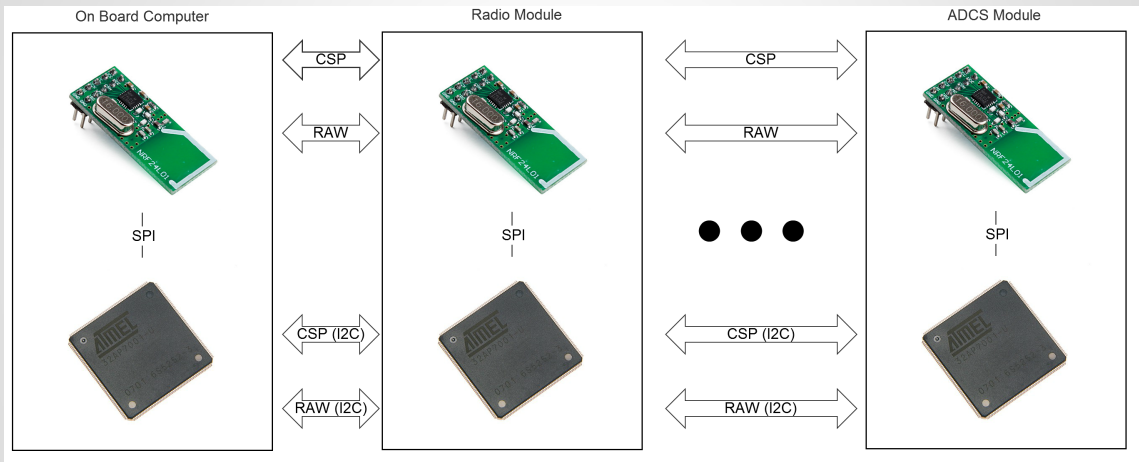
- Quasi-0 overhead
- Lower RAM consumption

Very low level, and lacking advanced functionalities. Increases the software complexity of its client apps.



2 Networks on 1 adapter





Fallback or cooperation ?



Fallback:

The secondary interface (usually wireless bus) would only work if the primary one failed.

- Software simplicity
- Misuse of resources

Cooperation:

Take advantage of both interfaces at the same time

- Higher throughput (Payload data can use the wireless link)
- Fallback is still possible
- Higher software complexity and power consumption



The nRF24L01 has several modes of operation, each with it's own power consumption:

- Power down: 900 nA
- Standby: 22-320 μ A
- TX (-18dBm): 7 mA
- RX (Low Current): 11.5 mA

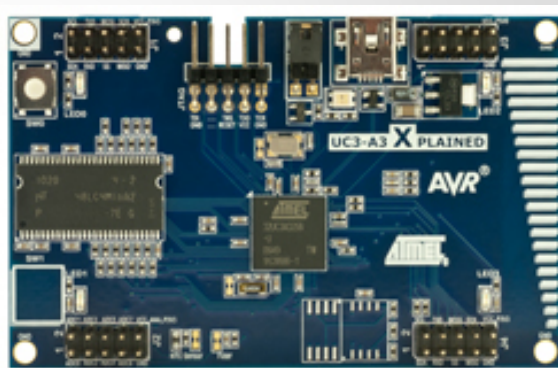


Time slicing could be used to reduce the power consumption, although it would increase software complexity.

Development setup and progress report

Development Board

- AVR UC3-A3 Xplained
 - AT32UC3A3256 (AVR32)
 - SPI-interfaced with an nRF24L01
- Progress
 - RX/TX of dynamically varying sized payloads works interrupt driven and asynchronous
 - Integration with CSP is currently being worked on



Source Code

Here there are some of the source code files that were developed. In the authors opinion they are the most relevant due to playing an important role in the finished system or as stepping stones to develop the project. The header files have not been included for brevity. These files along with the rest of code has been handed in digitally with this thesis although some differences may be found between the digital files and this copies. The changes are not altering functionality and are only removals of dead code to increase readability.

stdio_usb_example.c

```
/*
  Main file, based on the USB-STDIO example provided by Atmel
3
*/

7 #include <asf.h>
  #include <board.h>
  #include <sysclk.h>
  #include <stdio_usb.h>
11 #include <nrf24l01.h>
  #include <user_interrupts.h>
  #include <shell.h>

15 #include "projdefs.h"
  #include "FreeRTOS.h"
  #include "task.h"
  #include "semphr.h"
19 #include "queue.h"
  #include "msgQueue.h"

23

  /*Defines for the addresses, it is the only difference
  between the binaries used in both AVR32s*/
27
  //#define THIS_ADD "brd01"
  //#define TARGET_ADD "brd02"

31 #define THIS_ADD "brd02"
  #define TARGET_ADD "brd01"

  /*defines to decide server or client during early stages (PING test) ->
  NOW DEPRECATED*/
35 #define SERVER
  //#define CLIENT

39 xSemaphoreHandle semaphoreRadioEvent;
  xSemaphoreHandle semaphoreTX;
```

```

xSemaphoreHandle semaphoreRX;
xSemaphoreHandle semaphoreMRT;
43  uint8_t nrf24_waitPacketSentSemaphores()
    {
47      // Wait for either the Data Sent or Max Retries flag, signalling the
        // end of transmission

        if (xSemaphoreTake(semaphoreTX, 50/portTICK_RATE_MS)) {
51          nrf24_spiWriteRegister(NRF24_REG_07_STATUS, NRF24_TX_DS |
            NRF24_MAX_RT);
            gpio_toggle_pin(LED1_GPIO);
            return true;
55        }

59        if (xSemaphoreTake(semaphoreMRT, 50/portTICK_RATE_MS)) {
            nrf24_flushTx();
            nrf24_spiWriteRegister(NRF24_REG_07_STATUS, NRF24_TX_DS |
                NRF24_MAX_RT);
            gpio_toggle_pin(LED2_GPIO);
63            return false;
        }

        gpio_toggle_pin(LED3_GPIO);
67        return false;
    }

71    void vLedTask0(void *pvParameters){
        while(1) {
            gpio_toggle_pin(LED0_GPIO);
75            //xSemaphoreGive(semaphoreRadioEvent);
            vTaskDelay(100/portTICK_RATE_MS);
        }
    }

79    void vLedTask1(void *pvParameters){
        while(1) {
            //xSemaphoreTake(semaphoreRadioEvent, portMAX_DELAY);
83            gpio_toggle_pin(LED1_GPIO);
            vTaskDelay(100/portTICK_RATE_MS);
        }
    }

87    void vLedTask2(void *pvParameters){
        while(1) {
            gpio_toggle_pin(LED2_GPIO);
91            vTaskDelay(50/portTICK_RATE_MS);
        }
    }

95    void vLedTask3(void *pvParameters){

```

```

    while(1) {
        gpio_toggle_pin(LED3_GPIO);
        vTaskDelay(1000/portTICK_RATE_MS);
99     }
    }

    char s[16];
103

    void vRadioEvent(void *pvParameters){

107     queuedMessage qm;

        while(1) {
            //if(semaphoreRadioEvent != NULL) {
111         if(xSemaphoreTake(semaphoreRadioEvent,portMAX_DELAY)) {
                if( gstatus & NRF24_MASK_RX_DR) {

                    xSemaphoreGive(semaphoreRX);
115                 }
                if( gstatus & NRF24_MASK_TX_DS) {
                    xSemaphoreGive(semaphoreTX);
                }
119                 if( gstatus & NRF24_MASK_MAX_RT) {

                    xSemaphoreGive(semaphoreMRT);
                }
123             } else {
                //Semaphore not acquired
            }
        }
127     }

    void vRecieve(void *pvParameters){

131     queuedMessage qm;

        while(1) {
135         if (xSemaphoreTake(semaphoreRX,portMAX_DELAY)){
                //Obtained
                nrf24_recv((uint8_t *) &qm.content ,&qm.size);
                xQueueSendToBack(rxQueue,&qm,0);
139             } else {
                //Unobtained
            }
        }
143     }

    void vSend(void *pvParameters){
        while(1){
147         queuedMessage qm;
            if (xQueueReceive(txQueue,&qm,portMAX_DELAY)){

                if (!nrf24_setTransmitAddress((uint8_t*)TARGET_ADD, 5)) ;//
                    gpio_clr_gpio_pin(LED1_GPIO);
151                 if (!nrf24_send((uint8_t*) &qm.content, qm.size , false)) ;

```

```

        nrf24_waitPacketSentSemaphores();
        nrf24_powerUpRx();

155     } else {
            //NO MSG TO SEND

        }
159     }
}

void vServerTask(void *pvParameters){
163     while(1) {

        nrf24_waitAvailable();

167     unsigned long data;
        uint8_t len = sizeof(data);
        if (!nrf24_recv((uint8_t*)&data, &len)) printf("read failed
            \r\n");
        // Now send the same data back
171     // Need to set the address of the detination each time,
            since auto-ack changes the TX address

        if (!nrf24_setTransmitAddress((uint8_t*)"cliel", 5)) printf
            ("setTransmitAddress failed \r\n");

175     // Send the same data back
        if (! nrf24_send((uint8_t*) &data, sizeof(data) , false))
            printf("send failed \r\n");

        if (!nrf24_waitPacketSent()) printf("waitPacketSent failed
            \r\n");

179     }
}

183 #ifdef SERVER

/**
187  * \brief main function
  */
int main (void)
{
191     /* Initialize basic board support features.
        * - Initialize system clock sources according to device-specific
        * configuration parameters supplied in a conf_clock.h file.
        * - Set up GPIO and board-specific features using additional
        configuration
195     * parameters, if any, specified in a conf_board.h file.
        */
    sysclk_init();
    board_init();

199     // Initialize interrupt vector table support.
    irq_initialize_vectors();

```

```

203  init_ext();

        // Enable interrupts
        cpu_irq_enable();

207  /* Call a local utility routine to initialize C-Library Standard I/
        O over
        * a USB CDC protocol. Tunable parameters in a conf_usb.h file must
        be
        * supplied to configure the USB device correctly.
211  */

        stdio_usb_init();

215  //delay_s(5);

        nrf24_initRadioSpi();
219  if (!nrf24_setChannel(1)) printf("setChannel failed \n");
        if (!nrf24_setThisAddress((uint8_t*)THIS_ADD, 5)) printf("
            setThisAddress failed \n");
        if (!nrf24_setPayloadSize(16)) printf("setPayloadSize failed \n");

223  NRF24DataRate dr = NRF24DataRate2Mbps;
        NRF24TransmitPower pr = NRF24TransmitPower0dBm;

        if (!nrf24_setRF(dr, pr)) printf("setRF failed");

227  registerCommands();
        initMsgQueues(8,8);

231  semaphoreRadioEvent = xSemaphoreCreateCounting(1,0);
        semaphoreRX = xSemaphoreCreateCounting(3,0);
        semaphoreTX = xSemaphoreCreateCounting(1,0);
        semaphoreMRT = xSemaphoreCreateCounting(1,0);

235  xTaskCreate(&vLedTask0, (const signed portCHAR *)"Led Task 0",
            configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+4, NULL);
        xTaskCreate(&vRecieve, (const signed portCHAR *)"Recieve Task",
            configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+3, NULL);
        xTaskCreate(&vRadioEvent, (const signed portCHAR *)"Radio Event
            Processing Task", configMINIMAL_STACK_SIZE, NULL,
            tskIDLE_PRIORITY+3, NULL);
239  xTaskCreate(&vSend, (const signed portCHAR *)"Send Task",
            configMINIMAL_STACK_SIZE+8192, NULL, tskIDLE_PRIORITY+3, NULL);
        xTaskCreate(&vCommandConsoleTask, (const signed portCHAR *)"Shell
            Task", configMINIMAL_STACK_SIZE+1024, NULL, tskIDLE_PRIORITY+2,
            NULL);

        vTaskStartScheduler();

243  return 42;

        }

247  #endif // SERVER

```

```

/*CLIENT IS DEPRECATED, LEFT ONLY AS AN EXAMPLE*/
251 #ifndef CLIENT

int main (void)
255 {
    /* Initialize basic board support features.
     * - Initialize system clock sources according to device-specific
     * configuration parameters supplied in a conf_clock.h file.
259     * - Set up GPIO and board-specific features using additional
     * configuration
     * parameters, if any, specified in a conf_board.h file.
     */

263     sysclk_init();
     board_init();

    //eic_init(EIC_RADIO_1,&nrf24_irq_eic,1);
267 //eic_enable_line (EIC_RADIO_1,1);

    // Initialize interrupt vector table support.
271 irq_initialize_vectors();

    // Enable interrupts
    cpu_irq_enable();

275 /* Call a local utility routine to initialize C-Library Standard I/
     * O over
     * a USB CDC protocol. Tunable parameters in a conf_usb.h file must
     * be
     * supplied to configure the USB device correctly.
279     */
    stdio_usb_init();

    delay_s(5);

283 printf("Gooooood morning sunshine...I'm a client\n");

    nrf24_initRadioSpi();

287 if (!nrf24_setChannel(1)) printf("setChannel failed \n\r");
    if (!nrf24_setThisAddress((uint8_t*)"xplan", 5)) printf("
        setThisAddress failed \r\n");
    //if (!nrf24_setPayloadSize(sizeof(unsigned long))) printf("
        setPayloadSize failed \r\n");
291 if (!nrf24_setPayloadSize(4)) printf("setPayloadSize failed \r\n");

    NRF24DataRate dr = NRF24DataRate2Mbps;
    NRF24TransmitPower pr = NRF24TransmitPower0dBm;

295 if (!nrf24_setRF(dr, pr)) printf("setRF failed");

    //printf("\n New Regs \n");

299 //nrf24_printRegisters();

```

```

printf("\n Hello Dave \n");
303 while (true)
    {
        //printf("send \r\n");
307         // Send some data to the server
        if (!nrf24_setTransmitAddress((uint8_t*)"ardui", 5)) printf
            ("setTransmitAddress failed \r\n");

311         uint16_t time = cpu_cy_2_ms(Get_sys_count(), sysclk_get_cpu_hz())
            ;

        if(!nrf24_send((uint8_t*) time, sizeof(time), false)) printf("
            send failed \r\n");

315         if(!nrf24_waitPacketSent()) //printf("waitPacketSent failed \r\n
            ");
        //printf("Packet Aknownledged: %d\r\n", time);

319         //gpio_toggle_pin(LED3_GPIO);

        //nrf24_waitAvailable();
        if( nrf24_waitAvailableTimeout(1000)){
323             //printf("New Packet Recieved \r\n");

            uint16_t data;
327             uint16_t len = sizeof(data);

            if(!nrf24_recv((uint8_t*) &data,&len)) printf("read failed \r\n
                ");

331             printf("Ping: %d \r\n",cpu_cy_2_ms(Get_sys_count(),
                sysclk_get_cpu_hz())-time);
            } else {

                //printf("TIMEOUT \r\n");
335            }

            gpio_toggle_pin(LED0_GPIO);
339             delay_ms(10);

            }
        }
343

#endif // CLIENT

```

msgQueue.c

```

/*
2  * msgQueue.c

```

```

    *
    * Created: 13/05/2013 17:28:28
    * Author: Jordi
6   */

#include "msgQueue.h"

10  xQueueHandle rxQueue = NULL;
    xQueueHandle txQueue = NULL;

14  char initMsgQueues(uint8_t txSize, uint8_t rxSize){
        txQueue = xQueueCreate(txSize, sizeof(queuedMessage));
        rxQueue = xQueueCreate(rxSize, sizeof(queuedMessage));

18  return (txQueue != NULL && rxQueue != NULL );

    }

```

nrf24l01.c

```

/*
 * nrf24l01.c
 *
4  * Created: 25/03/2013 15:59:49
 * Author: Jordi
 */

8

#include <nrf24l01.h>
#include <asf.h>

12 spi_options_t spi_options_radio =
    {
        .baudrate = 8000000, //1000000,
        .bits = 8,
16  .modfdis = 1,
        .reg = 0,
        .spck_delay = 0,
        .spi_mode = SPI_MODE_0,
20  .stay_act = 1, //TODO: Dubtable
        .trans_delay = 0
    };

24

28 static void spi_pin_setup()
    {

        static const gpio_map_t spil_gpio_map = {{AVR32_SPI1_MISO_0_0_PIN,
            AVR32_SPI1_MISO_0_0_FUNCTION}, //MISO
32  {AVR32_SPI1_MOSI_0_0_PIN, AVR32_SPI1_MOSI_0_0_FUNCTION}, //MOSI
        {AVR32_SPI1_NPCS_0_0_PIN, AVR32_SPI1_NPCS_0_0_FUNCTION}, //SS0
        {AVR32_SPI1_SCK_0_0_PIN, AVR32_SPI1_SCK_0_0_FUNCTION} //SCK

```

```

    };
36     gpio_enable_module(spil_gpio_map, sizeof(spil_gpio_map)/sizeof(
        spil_gpio_map[0]));
    }
40     static void spi_init_module()
    {
44         sysclk_enable_peripheral_clock(SPI_RADIO_1);
        spi_pin_setup();

        spi_initMaster(SPI_RADIO_1, &spi_options_radio);
48         spi_selectionMode(SPI_RADIO_1, 0, 0, 0);

        spi_setupChipReg(SPI_RADIO_1, &spi_options_radio, FOSC0);
52         spi_enable(SPI_RADIO_1);
    }

    uint8_t nrf24_initRadioSpi()
56     {
        spi_init_module();

        delay_ms(100); //Wait for NRF24L01 POR TODO
60         // Clear interrupts
        nrf24_spiWriteRegister(NRF24_REG_07_STATUS, NRF24_RX_DR |
            NRF24_TX_DS | NRF24_MAX_RT);

64         // Make sure we are powered down
        nrf24_powerDown();

        // Flush FIFOs
68         nrf24_flushTx();
        nrf24_flushRx();

        //nrf24_spiWriteRegister(NRF24_REG_00_CONFIG, NRF24_MASK_RX_DR |
            NRF24_MASK_MAX_RT | NRF24_DEFAULT_CONFIGURATION); //Enable only
            TX interrupt
72         //Activate dynamic payload size
        //nrf24_activate();
        //nrf24_spiWriteRegister(NRF24_REG_1D_FEATURE, NRF24_EN_DPL);
76         //nrf24_spiWriteRegister(NRF24_REG_1C_DYNPD, NRF24_DPL_P0 |
            NRF24_DPL_P1);

        nrf24_powerUpRx();

80         return true;
    }

84     uint8_t nrf24_spiCommand(uint8_t command)
    {

```

```

    uint16_t status;
88     spi_selectChip(SPI_RADIO_1,0);
        spi_write(SPI_RADIO_1,command);
        spi_read(SPI_RADIO_1,&status);
        spi_unselectChip(SPI_RADIO_1,0);
92     return (uint8_t) status;
    }

    // Read and write commands
96     uint8_t nrf24_spiRead(uint8_t command)
    {
        uint16_t val;
        spi_selectChip(SPI_RADIO_1,0);
100     spi_write(SPI_RADIO_1,command); // Send the address, discard status
        spi_write(SPI_RADIO_1,0x55); //Send a dummy to clock the read out
        spi_read(SPI_RADIO_1,&val); // The MOSI value is ignored, value is
            read
104     spi_unselectChip(SPI_RADIO_1,0);
        return (uint8_t) val;
    }

108     uint8_t nrf24_spiWrite(uint8_t command, uint8_t val)
    {
        uint16_t status;
        spi_selectChip(SPI_RADIO_1,0);
112     spi_write(SPI_RADIO_1,command);
        spi_read(SPI_RADIO_1,&status);
        spi_write(SPI_RADIO_1,val);
        spi_unselectChip(SPI_RADIO_1,0);
116     return (uint8_t) status;
    }

    void nrf24_spiBurstRead(uint8_t command, uint8_t* dest, uint8_t len)
120 {
        uint16_t temp;
        spi_selectChip(SPI_RADIO_1,0);
        spi_write(SPI_RADIO_1,command); // Send the start address, discard
            status
124     while (len--)
        {
            spi_write(SPI_RADIO_1,0x55); //Send Dummy to clk out
            spi_read(SPI_RADIO_1, &temp);
128     *(dest++) = (uint8_t) temp; //Assign to array and then
                increment, note post incr
                //dest++ ;//= ((uint8_t) dest) + 1;
        }

132     spi_unselectChip(SPI_RADIO_1,0);

        // 300 microseconds for 32 octet payload
    }

136     uint8_t nrf24_spiBurstWrite(uint8_t command, uint8_t* src, uint8_t len)
    {
        uint16_t status ;
140     spi_selectChip(SPI_RADIO_1,0);

```

```

    spi_write(SPI_RADIO_1,command);
    spi_read(SPI_RADIO_1,&status);
144     while (len--)
        {
            spi_write(SPI_RADIO_1,*src++);
148         }

    spi_unselectChip(SPI_RADIO_1,0);
    return (uint8_t) status;
152 }

// Use the register commands to read and write the registers
156 uint8_t nrf24_spiReadRegister(uint8_t reg)
    {
        return nrf24_spiRead((reg & NRF24_REGISTER_MASK) |
            NRF24_COMMAND_R_REGISTER);
    }
160
uint8_t nrf24_spiWriteRegister(uint8_t reg, uint8_t val)
    {
        return nrf24_spiWrite((reg & NRF24_REGISTER_MASK) |
            NRF24_COMMAND_W_REGISTER, val);
164 }

void nrf24_spiBurstReadRegister(uint8_t reg, uint8_t* dest, uint8_t len
    )
    {
168     return nrf24_spiBurstRead((reg & NRF24_REGISTER_MASK) |
        NRF24_COMMAND_R_REGISTER, dest, len);
    }

uint8_t nrf24_spiBurstWriteRegister(uint8_t reg, uint8_t* src, uint8_t
    len)
172 {
    return nrf24_spiBurstWrite((reg & NRF24_REGISTER_MASK) |
        NRF24_COMMAND_W_REGISTER, src, len);
    }

176 uint8_t nrf24_statusRead()
    {
        return nrf24_spiReadRegister(NRF24_REG_07_STATUS);
        // return nrf24_spiCommand(NRF24_COMMAND_NOP); // Side effect is
        to read status
180 }

uint8_t nrf24_flushTx()
    {
184     return nrf24_spiCommand(NRF24_COMMAND_FLUSH_TX);
    }

uint8_t nrf24_flushRx()
188 {
    return nrf24_spiCommand(NRF24_COMMAND_FLUSH_RX);
    }

```

```

192 uint8_t nrf24_setChannel(uint8_t channel)
    {
        nrf24_spiWriteRegister(NRF24_REG_05_RF_CH, channel & NRF24_RF_CH);
        return true;
196 }

uint8_t nrf24_setThisAddress(uint8_t* address, uint8_t len)
    {
200     // Set RX_ADDR_P1 for this address
        nrf24_spiBurstWriteRegister(NRF24_REG_0B_RX_ADDR_P1, address, len);
        // RX_ADDR_P2 is set to RX_ADDR_P1 with the LSbyte set to 0xff, for
            use as a broadcast address
        return true;
204 }

uint8_t nrf24_setTransmitAddress(uint8_t* address, uint8_t len)
    {
208     // Set both TX_ADDR and RX_ADDR_P0 for auto-ack with Enhanced
            shockwave
        nrf24_spiBurstWriteRegister(NRF24_REG_0A_RX_ADDR_P0, address, len);
        nrf24_spiBurstWriteRegister(NRF24_REG_10_TX_ADDR, address, len);
        return true;
212 }

uint8_t nrf24_setPayloadSize(uint8_t size)
    {
216     nrf24_spiWriteRegister(NRF24_REG_11_RX_PW_P0, size);
        nrf24_spiWriteRegister(NRF24_REG_12_RX_PW_P1, size);
        return true;
    }
220

uint8_t nrf24_setRF(uint8_t data_rate, uint8_t power)
    {
        uint8_t value = (power << 1) & NRF24_PWR;
224     // Ugly mapping of data rates to noncontiguous 2 bits:
        if (data_rate == NRF24DataRate250kbps)
            value |= NRF24_RF_DR_LOW;
        else if (data_rate == NRF24DataRate2Mbps)
228     value |= NRF24_RF_DR_HIGH;
        // else NRF24DataRate1Mbps, 00
        nrf24_spiWriteRegister(NRF24_REG_06_RF_SETUP, value);
        return true;
232 }

uint8_t nrf24_powerDown()
    {
236     nrf24_spiWriteRegister(NRF24_REG_00_CONFIG,
            NRF24_DEFAULT_CONFIGURATION);
        gpio_clr_gpio_pin(CE_PIN);
        return true;
    }
240

uint8_t nrf24_powerUpRx()
    {
        nrf24_spiWriteRegister(NRF24_REG_00_CONFIG,
            NRF24_DEFAULT_CONFIGURATION | NRF24_PWR_UP | NRF24_PRIM_RX);

```

```

244     gpio_set_gpio_pin(CE_PIN);
        return true;
    }

248 uint8_t nrf24_powerUpTx()
    {
        // Its the pulse high that puts us into TX mode
        gpio_clr_gpio_pin(CE_PIN);
252     nrf24_spiWriteRegister(NRF24_REG_00_CONFIG,
            NRF24_DEFAULT_CONFIGURATION | NRF24_PWR_UP);
        gpio_set_gpio_pin(CE_PIN);
        return true;
    }

256 uint8_t nrf24_send(uint8_t* data, uint8_t len, uint8_t noack)
    {
        nrf24_powerUpTx();
260     nrf24_spiBurstWrite(noack ? NRF24_COMMAND_W_TX_PAYLOAD_NOACK :
            NRF24_COMMAND_W_TX_PAYLOAD, data, len);
        // Radio will return to Standby II mode after transmission is
            complete
        //delay_ms(10); //TODO: remove delay make it irq driven
        return true;
264 }

    uint8_t nrf24_waitPacketSent()
    {
268     // If we are currently in receive mode, then there is no packet to
        wait for
        if (nrf24_spiReadRegister(NRF24_REG_00_CONFIG) & NRF24_PRIM_RX)
            {
                return false;
272            }

        // Wait for either the Data Sent or Max ReTries flag, signalling
            the
        // end of transmission
276     uint8_t status;
        while (!(status = nrf24_statusRead()) & (NRF24_TX_DS |
            NRF24_MAX_RT)) {}

280     // Must clear NRF24_MAX_RT if it is set, else no further comm
        nrf24_spiWriteRegister(NRF24_REG_07_STATUS, NRF24_TX_DS |
            NRF24_MAX_RT);
        if (status & NRF24_MAX_RT)
            {
284             nrf24_flushTx();

            }

288     // Return true if data sent, false if MAX_RT
        return status & NRF24_TX_DS;
        //return true; //TODO: fer maco
    }

292 uint8_t nrf24_isSending()

```

```

    {
        return !(nrf24_spiReadRegister(NRF24_REG_00_CONFIG) & NRF24_PRIM_RX
            ) && !(nrf24_statusRead() & (NRF24_TX_DS | NRF24_MAX_RT));
296    }

uint8_t nrf24_printRegisters()
{
300    uint8_t registers[] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0
        x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0d, 0x0f, 0x10, 0
        x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x1c, 0x1d};

    uint8_t i;
    for (i = 0; i < sizeof(registers); i++)
304        {
            printf("Register: 0x%X: ", registers[i]);
            printf("0x%x \n",nrf24_spiReadRegister(i));
        }
308    return true;
}

uint8_t nrf24_available()
312 {
    if (nrf24_spiReadRegister(NRF24_REG_17_FIFO_STATUS) &
        NRF24_RX_EMPTY){
        return false;

316    }
    // Manual says that messages > 32 octets should be discarded
    if (nrf24_spiRead(NRF24_COMMAND_R_RX_PL_WID) > 32)
        {
320            nrf24_flushRx();
            return false;
        }
    return true;
324 }

void nrf24_waitAvailable()
{
328    nrf24_powerUpRx();
    while (!nrf24_available());
}

332 // Blocks until a valid message is received or timeout expires
// Return true if there is a message available
uint8_t nrf24_waitAvailableTimeout(uint16_t timeout)
{
336    nrf24_powerUpRx();
    t_cpu_time to;
    cpu_set_timeout(cpu_ms_2_cy(timeout, sysclk_get_cpu_hz()), &to);

340    while (!cpu_is_timeout (&to)){
        if (nrf24_available()){
            cpu_stop_timeout(&to);
344            return true;
        }
    }
}

```

```

    cpu_stop_timeout(&to);
348 return false;
}

uint8_t nrf24_rcv(uint8_t* buf, uint8_t* len)
352 {
    // Clear read interrupt
    nrf24_spiWriteRegister(NRF24_REG_07_STATUS, NRF24_RX_DR);

356 // 0 microseconds @ 8MHz SPI clock
    if (!nrf24_available())
        return false;
    // 32 microseconds (if immediately available)
360 *len = nrf24_spiRead(NRF24_COMMAND_R_RX_PL_WID);
    // 44 microseconds
    nrf24_spiBurstRead(NRF24_COMMAND_R_RX_PAYLOAD, buf, *len);
    // 140 microseconds (32 octet payload)
364 return true;
}

368 void nrf24_activate(){
    nrf24_spiWrite(NRF24_COMMAND_ACTIVATE, 0x73);
}

```

shell.c

```

#include <shell.h>
2 #include "msgQueue.h"

#define MAX_INPUT_LENGTH    50
#define MAX_OUTPUT_LENGTH  100

6
static const int8_t * const pcWelcomeMessage =
    "FreeRTOS command server.\r\nType Help to view a list of registered
    commands.\r\n";

10 /*static xCommandLineInput cmdToggleLed = {
    .pcCommand = "toggle",
    .pcHelpString = "toggle n: Toggle the LED specified as a parameter
        n [0 - 3].\r\n",
    .pxCommandInterpreter = prvToggleLed,
14 .cExpectedNumberOfParameters = 1
};*/

static xCommandLineInput cmdTable[] = {{
18     .pcCommand = "toggle",
    .pcHelpString = "toggle n: Toggle the LED
        specified as a parameter n [0 - 3].\r\n",
    .pxCommandInterpreter = prvToggleLed,
    .cExpectedNumberOfParameters = 1
22 }, {
    .pcCommand = "config_rf",
    .pcHelpString = "config_rf [channel] [datarate] [
        payload size] [power] \r\n",
    .pxCommandInterpreter = prvConfigRF,

```

```

26         .cExpectedNumberOfParameters = 4
           }, {
           .pcCommand = "send_raw",
           .pcHelpString = "send_raw: send_raw [size] [
30             payload]\r\n",
           .pxCommandInterpreter = prvSendRaw,
           .cExpectedNumberOfParameters = 2
           }, {
           .pcCommand = "recieve_raw",
34           .pcHelpString = "recieve_raw: Gets one message
               from the queue and prints it\r\n",
           .pxCommandInterpreter = prvRecRaw,
           .cExpectedNumberOfParameters = 0
           }
       });

38

void registerCommands(){
42     int i;
       for(i = 0; i < sizeof(cmdTable)/sizeof(cmdTable[0]);i++){
           //for(i = 0; i < 4;i++){
               FreeRTOS_CLIRegisterCommand(&cmdTable[i]);
46         }
       }

50     uint8_t z;

       portBASE_TYPE prvConfigRF(int8_t *pcWriteBuffer, size_t xWriteBufferLen
           , const int8_t *pcCommandString ){
54         //"config_rf: Sets RF params for the radio. Use: config_rf [channel]
           [datarate] [payload size] [power].\r\n",
           volatile portCHAR* param;

           uint8_t channel, datarate, payload_size, power;

58         //gpio_toggle_pin(LED3_GPIO);

           param = FreeRTOS_CLIGetParameter(pcCommandString,1,&z);
62         //This function kills it all
           channel = atoi(param);

           param = FreeRTOS_CLIGetParameter(pcCommandString,2,&z);
66         //This function kills it all
           datarate = atoi(param);

           param = FreeRTOS_CLIGetParameter(pcCommandString,3,&z);
70         //This function kills it all
           payload_size = atoi(param);

           param = FreeRTOS_CLIGetParameter(pcCommandString,4,&z);
74         //This function kills it all
           power = atoi(param);

           nrf24_setChannel(channel);
78         //nrf24_setRF(datarate,power);

```

```

nrf24_setPayloadSize(payload_size);

uint8_t readChannel = nrf24_spiReadRegister(NRF24_REG_05_RF_CH);
82 uint8_t readRF = nrf24_spiReadRegister(NRF24_REG_06_RF_SETUP);
uint8_t readPayloadSize = nrf24_spiReadRegister(NRF24_REG_11_RX_PW_P0
    );

86 //rawPrint(param,4);
//channel = atoi(param);

90 //param = FreeRTOS_CLIGetParameter(pcCommandString,2,&pz);
//datarate = atoi(param);

//param = FreeRTOS_CLIGetParameter(pcCommandString,3,&pz);
94 //payload_size = atoi(param);

//param = FreeRTOS_CLIGetParameter(pcCommandString,4,&pz);
//power = atoi(param);

98 volatile int8_t tz = z;

102 //sprintf(pcWriteBuffer,"%d %d %d %d \r\n",power,payload_size,
    datarate,channel);
//sprintf(pcWriteBuffer,"%d %d %d %d \r\n",42,43,44,45);
//sprintf(pcWriteBuffer,"Command: %s Param 1: %c%c Size: %d, Add of z
    : %p\r\n",pcCommandString,*param,*(param+1),tz,&tz);
106 //sprintf(pcWriteBuffer,"Command: %s END\r\n",pcCommandString);
    sprintf(pcWriteBuffer,"Channel: %d Datarate: %d Payload Size: %d
        Power: %d END\r\n",readChannel,readRF,readPayloadSize,readRF);
xWriteBufferLen = strlen(pcWriteBuffer);

110 return pdFALSE;
}

114 portBASE_TYPE prvToggleLed(int8_t *pcWriteBuffer, size_t
    xWriteBufferLen, const int8_t *pcCommandString){
uint8_t* param;
uint8_t* pz;
param = FreeRTOS_CLIGetParameter(pcCommandString,1,pz);
118 switch(*param){
    case '0':
        gpio_toggle_pin(LED0_GPIO);
        break;
122 case '1':
        gpio_toggle_pin(LED1_GPIO);
        break;
    case '2':
126 gpio_toggle_pin(LED2_GPIO);
        break;
    case '3':
130 gpio_toggle_pin(LED3_GPIO);
        break;

```

```

        default: //Wrong parameters
            sprintf(pcWriteBuffer,"%s\r\n","n must be 0 - 3");
            xWriteBufferLen = strlen(pcWriteBuffer);
134         return pdFALSE;
            break;
        }

138     sprintf(pcWriteBuffer,"%s: %c\r\n","OK",*param);
    xWriteBufferLen = strlen(pcWriteBuffer);

    return pdFALSE;
142 }

portBASE_TYPE prvRecRaw(int8_t *pcWriteBuffer, size_t xWriteBufferLen,
    const int8_t *pcCommandString ){
    queuedMessage qm;
146     if (xQueueReceive(rxQueue,&qm,0)){
        //Message available
            sprintf(pcWriteBuffer,"MSG: %s SIZE: %d\r\n",qm.content,qm.size);

150     } else {
        //message unavaliable
            sprintf(pcWriteBuffer,"NO MESSAGES \r\n");

154     }

        xWriteBufferLen = strlen(pcWriteBuffer);
        return pdFALSE;
158 }

162 portBASE_TYPE prvSendRaw(int8_t *pcWriteBuffer, size_t xWriteBufferLen,
    const int8_t *pcCommandString ){
    uint8_t* param;
    uint8_t* pz;
    queuedMessage qm;

166     param = FreeRTOS_CLIGetParameter(pcCommandString,1,pz);
    uint8_t size = atoi(param);

170     param = FreeRTOS_CLIGetParameter(pcCommandString,2,pz);
    param[size-1] = NULL; //Termination

174     qm.size = size;
    memcpy_ram2ram(&qm.content,param,size);

    if( xQueueSendToBack(txQueue,&qm,0)){
178         sprintf(pcWriteBuffer,"Queued in Tx\r\n");

    } else {
        sprintf(pcWriteBuffer,"ERROR: No available spots in Tx\r\n");
182     }

    /*

```

```

186     if (!nrf24_setTransmitAddress((uint8_t*)"cliel", 5)) sprintf(
        pcWriteBuffer,"setTransmitAddress failed \r\n");
        else if (!nrf24_send((uint8_t*) param, size , false)) sprintf(
            pcWriteBuffer,"send failed \r\n");
        else if (!nrf24_waitPacketSent()) sprintf(pcWriteBuffer,"
            waitPacketSent failed size: %d message: %s \r\n",size,param);
        else sprintf(pcWriteBuffer,"SENT !\r\n");
190     */

        xWriteBufferLen = strlen(pcWriteBuffer);

194     return pdFALSE;
    }

    void rawPrint(char* s,uint16_t size){
198         int i;
            for(i = 0; i < size; i++) {
                putchar(s[i]);
                //TODO: mutex ??
202         }

    }

206 void vCommandConsoleTask( void *pvParameters )
    {
        //Peripheral_Descriptor_t xConsole;
        int8_t cRxdChar, cInputIndex = 0;
210 portBASE_TYPE xMoreDataToFollow;
        /* The input and output buffers are declared static to keep them off
            the stack. */
        static int8_t pcOutputString[ MAX_OUTPUT_LENGTH ], pcInputString[
            MAX_INPUT_LENGTH ];

214         /* This code assumes the peripheral being used as the console has
            already
            been opened and configured, and is passed into the task as the task
            parameter. Cast the task parameter to the correct type. */
            //xConsole = ( Peripheral_Descriptor_t ) pvParameters;

218         /* Send a welcome message to the user knows they are connected. */
            //FreeRTOS_write( xConsole, pcWelcomeMessage, strlen(
                pcWelcomeMessage ) );
            //printf("%s",pcWelcomeMessage);
222 //xSemaphoreTake(mutexUART,portMAX_DELAY);
            rawPrint(pcWelcomeMessage,strlen(pcWelcomeMessage));
            //xSemaphoreGive(mutexUART);
            for( ;; )
226         {
                /* This implementation reads a single character at a time.
                    Wait in the
                    Blocked state until a character is received. */
                    //FreeRTOS_read( xConsole, &cRxdChar, sizeof( cRxdChar ) );
230 cRxdChar = getchar();
                    if( cRxdChar == '\n' )
                    {
                        /* A newline character was received, so the input command
                            string is

```

```

234         complete and can be processed.  Transmit a line separator,
           just to
           make the output easier to read. */
           //FreeRTOS_write( xConsole, "\r\n", strlen( "\r\n" );
//xSemaphoreTake(mutexUART,portMAX_DELAY);
238 rawPrint(pcInputString,strlen(pcInputString));
putchar('\r');
putchar('\n');
//xSemaphoreGive(mutexUART);
242     /* The command interpreter is called repeatedly until it
           returns
           pdFALSE.  See the "Implementing a command" documentation
           for an
           explanation of why this is. */
           do
246     {
           /* Send the command string to the command interpreter.
           Any
           output generated by the command interpreter will be
           placed in the
           pcOutputString buffer. */
           xMoreDataToFollow = FreeRTOS_CLIPProcessCommand
250             (
                 pcInputString, /* The command
                               string.*/
                 pcOutputString, /* The output buffer
                               . */
254                 MAX_OUTPUT_LENGTH/* The size of the
                               output buffer. */
             );

           /* Write the output generated by the command
           interpreter to the
258 console. */
           //FreeRTOS_write( xConsole, pcOutputString, strlen(
                 pcOutputString ) );
           //xSemaphoreTake(mutexUART,portMAX_DELAY);
           rawPrint(pcOutputString,strlen(pcOutputString));
262 //xSemaphoreGive(mutexUART);
           //printf("%s",pcOutputString);
           } while( xMoreDataToFollow != pdFALSE );

266     /* All the strings generated by the input command have been
           sent.
           Processing of the command is complete.  Clear the input
           string ready
           to receive the next command. */
           cInputIndex = 0;
270     memset( pcInputString, 0x00, MAX_INPUT_LENGTH );
           }
           else
           {
274     /* The if() clause performs the processing after a newline
           character
           is received.  This else clause performs the processing if
           any other
           character is received. */

```

```

278         if( cRxedChar == '\r' )
            {
                /* Ignore carriage returns. */
            }
282     else if( cRxedChar == '\b' )
            {
                /* Backspace was pressed. Erase the last character in
                the input
                buffer - if there are any. */
286         if( cInputIndex > 0 )
            {
                cInputIndex--;
                pcInputString[ cInputIndex ] = '\0';
290            }
            }
            else
            {
294         /* A character was entered. It was not a new line,
                backspace
                or carriage return, so it is accepted as part of the
                input and
                placed into the input buffer. When a \n is entered the
                complete
                string will be passed to the command interpreter. */
298         if( cInputIndex < MAX_INPUT_LENGTH )
            {
                pcInputString[ cInputIndex ] = cRxedChar;
                cInputIndex++;
302            }
            }
        }
306 }

```

user_interrupts.c

```

/*
2  * user_interrupts.c
   *
   * Created: 04/04/2013 14:32:48
   * Author: Jordi
6  */

//#include "user_interrupts.h"

10 #include "FreeRTOS.h"
   #include "semphr.h"
   #include <asf.h>
   #include <nrf24l01.h>
14 #define EIC_RADIO_1 (&AVR32_EIC)

   xSemaphoreHandle semaphoreRadioEvent = NULL;
18 //xSemaphoreHandle mutexUART = NULL;
   uint8_t gstatus;

```

```

    eic_options_t nrf24_irq_eic = {.eic_async = EIC_ASYNC_MODE,
22         .eic_edge = EIC_EDGE_FALLING_EDGE,
            .eic_filter = EIC_FILTER_ENABLED,
            .eic_level = EIC_LEVEL_LOW_LEVEL,
            .eic_line = EXT_INT2,
26         .eic_mode = EIC_MODE_EDGE_TRIGGERED};

ISR(radio_irq_handler, 1, AVR32_INTC_INTLEV_INT0) {
    //gpio_toggle_pin(LED3_GPIO);
30    //gpio_toggle_pin(LED0_GPIO);
    xSemaphoreGiveFromISR(semaphoreRadioEvent, NULL);
    gstatus = nrf24_statusRead();
    eic_clear_interrupt_line(EIC_RADIO_1, EXT_INT2);
34 }

void extint_pin_setup() {

38 static const gpio_map_t extint_gpio_map = {{AVR32_EIC_EXTINT_2_PIN,
    AVR32_EIC_EXTINT_2_FUNCTION}};

    gpio_enable_pin_pull_up(AVR32_EIC_EXTINT_2_PIN);
    gpio_enable_module(extint_gpio_map, sizeof(extint_gpio_map)/sizeof(
42         extint_gpio_map[0]));
}

void init_ext() {
46     //sysclk_enable_peripheral_clock(EIC_RADIO_1);

    extint_pin_setup();
    INTC_register_interrupt (&radio_irq_handler, AVR32_EIC_IRQ_2,
50         AVR32_INTC_INTLEV_INT0);

    eic_init (EIC_RADIO_1, &nrf24_irq_eic, 1);
    eic_enable_interrupt_line(EIC_RADIO_1, EXT_INT2);
    eic_enable_line (EIC_RADIO_1, EXT_INT2);
54 }
}

```
