Radio Communication for PIE -
a Tiny Timber Interface

Master's Thesis in Computer Systems Engineering

Mohsen Amerion

School of Information Science, Computer and Electrical Engineering
Halmstad University
Radio Communication for PIE -
a Tiny Timber Interface

Master’s thesis in Computer Systems Engineering

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

October 2009
Description of cover page picture: Left, Image taken of a PIE. Right, Image of the nRF24L01 Antenna and SPI Interface.
Preface

This report is being presented at the end of the one academic year master program in computer systems engineering at IDE department of Halmstad University. I would like to thank the staff at IDE helpdesk for providing me with equipment and their valuable support, without which it would have been difficult to complete the project successfully.

I would also like to thank Dr. Veronica Gaspes, Prof. Tony Larsson, and Mr. Tommy Solomonsson, and all faculties from IDE department, for their guidance throughout the course which has helped me a lot in standing up to the challenges that I faced during the whole course.

Finally, I would offer my appreciation to my parents who have always been supportive of my ambitions.

Mohsen Amerion
Halmstad University, October 2009
Abstract

Tiny Timber is a very lightweight, real-time kernel for small, event-driven embedded systems. It implements reactive objects as a way of structuring embedded software. These objects are components that react to incoming events by updating their internal state and issuing an outgoing event. Implementing a control system based on reactive objects enhances the reactivity of the system and it helps to optimize the energy consumption, as the system goes to sleep mode when there is nothing to react to. It is also a programming technique that facilitates the organization of the program.

In this project, we develop a device driver for radio communication with a Tiny Timber interface. This is an extension to the Platform for Intelligent Embedded (PIE) system which is a robot assembled and used at the IDE department of Halmstad University as a platform for developing new applications as well as for educational purposes in the area of embedded and intelligent systems. The interface is programmed in C and consists of a collection of device drivers for different components of PIE in the form of reactive objects.
Contents

PREFACE .......................................................................................................................................................... II
ABSTRACT ......................................................................................................................................................... III
CONTENTS .......................................................................................................................................................... IV

CHAPTER 1: EMBEDDED/REAL-TIME SYSTEMS ....................................................................................... 1
  OVERVIEW ............................................................................................................................................... 1
  RESEARCH AND DEVELOPMENT METHODOLOGY .................................................................................... 1
  PART I – EMBEDDED/REAL-TIME SYSTEMS ......................................................................................... 2
  WHAT IS AN EMBEDDED/REAL-TIME SYSTEM? .................................................................................... 2
  GENERAL REQUIREMENTS OF EMBEDDED SYSTEMS ........................................................................... 2
    Reliability ........................................................................................................................................... 2
    Cost-effectiveness ............................................................................................................................... 3
    Low power consumption ..................................................................................................................... 3
  PART II – TINY TIMBER .............................................................................................................................. 3
  INTRODUCTION ....................................................................................................................................... 3
  METHODS ................................................................................................................................................ 4
  REACTIVE OBJECTS ................................................................................................................................ 4
  MESSAGE PASSING .................................................................................................................................. 5
  CONCURRENCY ........................................................................................................................................ 5

CHAPTER 2: THE TINY TIMBER DEVICE DRIVER FOR THE RADIO INTERFACE ............................. 6
  PART I – RADIO MODULE ........................................................................................................................... 7
  DESCRIPTION OF NRF24L01 AND SPI ..................................................................................................... 7
    SPI and nRF24L01 related pins ................................................................................................................ 8
  HOW TO INTERFACE WITH NRF24L01 VIA SPI .................................................................................. 9
    Description of SPI Commands ............................................................................................................. 9
  DESCRIPTION OF THE ANTENNA’S FUNCTIONALITY ......................................................................... 11
  PACKET DESCRIPTION FOR NRF24L01 ................................................................................................. 13
  DATA TRANSMISSION ............................................................................................................................... 13
  PART II – THE FIRST PHASE IMPLEMENTATIONS ............................................................................... 15
    Initialization functions .......................................................................................................................... 15
    Addressing functions .............................................................................................................................. 16
    Interrupt check/clear functions .............................................................................................................. 16
    The payload related functions .............................................................................................................. 17
    RX/TX mode setting functions ............................................................................................................. 17
    power-up, power-down functions ......................................................................................................... 18
    Channel frequency function ............................................................................................................... 18
    Debugging function ............................................................................................................................... 18
  PART III – THE TINY TIMBER INTERFACE .......................................................................................... 18
    nRF24L01 and TINY TIMBER ................................................................................................................ 18
    nRF24L01 as a reactive object ............................................................................................................... 19
  SELF-APPLICATION METHODS FOR NRF24L01 .................................................................................. 19
  SELF-APPLICATION METHODS FOR THE PIE ...................................................................................... 20

CHAPTER 4: APPLICATION SAMPLES AND CONCLUSIONS ..................................................................... 22
  APPLICATION SAMPLES ............................................................................................................................ 23
  CONCLUSIONS .......................................................................................................................................... 26

REFERENCES ............................................................................................................................................... 28

APPENDIX A .................................................................................................................................................. 30
  INTRODUCTION OF PIE .......................................................................................................................... 30
PIE COMPONENTS ................................................................. 30
CHAPTER 1

Embedded/Real-Time Systems
Overview

The engineering part of this thesis consists of the implementation of a wireless communication device driver for the radio module of a robot known as PIE which has been designed and assembled at IDE department of Halmstad University (see Appendix A for PIE). The project is suggested by IDE department as it is used in a course, called “Design of Embedded and Intelligent Systems”, to remotely control the movement of the PIEs [19]. Another issue that is addressed in some courses in the IDE department of Halmstad University is to use reactive objects as an approach for programming embedded systems and the goal in this project is to have a radio communication device driver based on a real-time kernel, called “Tiny Timber”, which is developed at Luleå University of Technology. The Tiny Timber interface is used to support real-time reactivity and timing constraints by taking advantage of reactive objects. Reactive objects are the components of a system that react to the incoming events by updating their internal state and issuing outgoing events when necessary. They do rest if there is no incoming event, making it possible to turn the processor into sleep mode. Reactive objects are discussed more in detail later on this report.

The component for which the device driver is developed is a nRF24L01 radio module, manufactured by Olimex [2]. The antenna is interfaced with an ARM microcontroller via Serial Peripheral Interface (SPI) produced by ATMEL [1]. This microcontroller is mounted on SAM7P256 board which is also manufactured by Olimex [3].

In this chapter, we discuss our research methodology, and then we provide an overview on embedded/real-time systems as well as the Tiny Timber real-time kernel. In the next chapter, we discuss the hardware characteristics of the nRF24L01 transceiver and its SPI interface; we explain the development of the device driver and then we show how our radio module is considered as a reactive object. In chapter three, the functionality of the device driver is evaluated and two applications are introduced that our project is used for and, finally, our report is finalized with the conclusion.

Research and Development Methodology

In order to establish a wireless communication between PIEs, we did the requirement analysis for our project and we found out about the hardware and software requirements of the system. Also, we have found out about the tools and environments that are necessary for development of the project.

The overall work was divided into two phases. The purpose of the first phase was to test the functionality of the hardware components such as SPI interface and radio module, so we wrote simple C functions and programmed the microcontroller via Joint Test Action Group (JTAG) interface in C/C++ Eclipse environment [20]. The JTAG standard provides the hardware interface and software that allows the testing of chips and boards within a system. In second phase of the
project we used winAVR tool integrated with Tiny Timber kernel and developed the device driver based on the Tiny Timber RTOS interfaces.

Also, to understand the functionality of the hardware components we used the manuals and corresponding datasheets published by the manufacturers of the products. For the programming part, we considered publications related to the Timber language and the Tiny Timber real-time kernel.

Part I – Embedded/Real-Time Systems

What is an Embedded/Real-Time System?

The desktop computers which most of us are familiar with, mainframes as well as workstations, are “general purpose” systems. In contrast, an embedded/real-time system performs a single, well-defined task and we find such systems everywhere—in our homes, in our offices, in cars, in hospitals, in aircraft and so on. Embedded/real-time systems can be defined as computing devices that perform a specific, focused job within specific time constraints and their prime function is not to process information; however they need to process the information in order to perform the task that they are specifically designed for. As an example, a microprocessor-controlled washing machine that is used to wash the clothes performs different modes of washing by executing different washing programs. Applications such as office automation devices, digital audio/video players, video game players, digital cameras, biomedical systems, protocol converters, encryption devices, global positioning system (GPS) receivers, mobile phones Bluetooth devices etc. are also examples of embedded/real-time systems [10, 17].

The main classes of embedded/real-time systems are identified as “soft real-time” and “hard real-time”. In soft real-time systems, the deadlines might be missed occasionally but the system does not halt and continues its functionality, like a packet delay in a video conferencing resulting in a short interrupt. However in hard real time systems, if a dead line is missed it can lead to a great loss or a threat to human life, for example any problem in an embedded medical system in ICU of a hospital that controls the patient’s health condition would cause serious problems [17, 18].

General Requirements of Embedded Systems

Reliability

The more society leaves its works to the computer systems, the more important it becomes that those computers do not fail. They should be designed an implemented in such a way that, even if they fail, their failure occurs in a controlled way [17]. As far as our project is concerned, we provide our robots with the ability of wireless communication in order to automatically control
their movement. We send control packets each and every half second until the robot reaches its destination, but wireless communication is unreliable in its nature and always there is a chance of losing a packet. So an alternative approach for controlling such failure is to provide the system with a method that causes the PIE to stop if it does not receive the control packet within a specific time interval and, in this way, there will not be any collision for our PIEs. Hence, we implemented a method that would send a stop command to the controller of the motors if our PIE did not receive the control packet from base station within 1 second.

Cost-effectiveness
The cost is always an important issue for users of embedded systems. Designers should choose a proper processor, memory and peripheral devices in such a way that the system meets the customer expectations while it is still cheap and affordable. However, for embedded systems that are used in safety applications of a nuclear plant or in a spacecraft, cost may not be very important but, in consumer electronics or office automations, the major importance is the cost [10]. In our system, the components are selected an assembled in a feasible manner that meets the IDE department expectations.

Low power consumption
Most of embedded systems work with a battery and this is an important issue that must addressed in order to reduce the power consumption of the system to avoid draining of the battery. For example, the processor can be put on "sleep mode" when there is no function to perform [10]. Here we implement our system based on reactive objects that are in resting mode if there is no incoming event and, in this way, we save energy.

Part II – Tiny Timber

Introduction
The operating systems that are used for general purpose personal computers are rather heavy for small embedded systems, so some other operating systems having limited functionalities have emerged, especially for embedded/real-time systems. Tiny Timber is amongst them and it is a light-weight, real-time kernel developed in C language that guarantees the state integrity of real time systems, i.e. it preserves the real time system’s consistency, and scheduling of its events are based on the timing constraints using the semantics of the Timber programming language. TIMBER (TIme, eMBEdded, Reactive) is a functional, reactive, object oriented, real-time and concurrent programming language which was developed as a joint project by Lulea University of Technology, Chalmers University of Technology, and Oregon Health and Science University. The main task of Tiny Timber is to manage the message queues and schedule them onto threads of execution in such a way that all of them (if possible) can meet their deadlines. Tiny Timber is event driven, i.e. it reacts to stimuli from environment, therefore the system goes to idle state if there is no any incoming message to be scheduled. In Timber, the idle state is used to do some tasks in the background, such as garbage collection, or putting the system in the low power state for energy efficiency [15].
**Reactive Objects**

Tiny Timber is based on reactive object programming model. A reactive object is a component that reacts to the incoming events by updating its internal state and emitting outgoing messages if needed. Every object is a computing structure that is seen as a “black box” that does nothing unless stimulated by external events. It is an advantage in terms of energy consumption as the processor can be put to sleep mode until an event occurs. If a method of an object is invoked then the state of the object might change [12].

Reactive objects in Tiny Timer are extended from object class. The object class implements a lock and an ordinary method that is invoked on an object executes under mutual exclusion. Once a method is invoked on an object it means a message is sent to it. Here, Tiny Timber ensures the state integrity of the object by blocking on the objects lock if another method is already invoked on that object, so there will not be any conflict between concurrent threads of execution and, on other hand, the consistency of the internal status of the reactive object is guaranteed.

**Methods**

Objects in Tiny Timber maintain a state and they have method(s). Each method has two input parameters. The first parameter of a method is an object pointer that points to the address of the reactive object on whose state the method operates, and the second parameter is an integer pointer.

A method in Tiny Timber is invoked in one of two ways, either synchronously (SYNC) or asynchronously (ASYNC). Synchronous invocation of a method waits until the execution of the method is completed and then it returns a value. Invoking methods with SYNC does not allow concurrent execution of any other method. Asynchronous invocation of a method does not return any value; it causes an event to happen and continues with the execution of the next instruction. It is an ASYNC call that introduces concurrency [13].

**Task scheduling**

As far as the scheduling of the tasks is concerned, Tiny Timber follows this rule: “each method which is invoked by sending a message has to be executed within base-line and dead-line”. Tiny Timber takes in consideration two levels of scheduling. One is for those methods which are invoked within an object (SYNC) and the other is for the methods between objects (ASYNC). Scheduling within an object is non-pre-emptive, as the state integrity of the system should be guaranteed. Scheduling between objects is pre-emptive and, in run-time system, it is maintained by earliest deadline first (EDF) algorithm in such a way that the current deadline of an object is equivalent to the execution of its most urgent message [12].
Message passing

The message passing mechanism in this run-time system is facilitated by queues of messages. Each object maintains a message queue which is sorted based on EDF algorithm. A message send command inserts that message in the execution queue of the recipient object and, when it is in the front of the queue, its correspondent method is invoked. These queues are local and, in addition to such queues, a global queue is maintained for those messages which have a specific base-line. This queue is sorted by earliest base-line first algorithm [12].

Concurrency

For concurrency between objects, each object should have its own execution context and it should be able to refer to its executing point in the program code. Context switching is done by storing the code and stack pointer in a jump buffer which is attached to each object. There are three cases due to which the context switch in a run-time system is allowed: 1) when a method is finished i.e. the object returns back the control to the scheduler which, in turn, it does the proper context switch. 2) When an interrupt has happened and the corresponding message is sent. In both cases, context switching may happen, but not necessarily. 3) when a synchronous request has happened. In order for an object to be able to receive the requested value, the caller object should allow the calling object to execute the method and, therefore, a forceful context switch takes place [13].
CHAPTER 2

The Tiny Timber device driver for the radio interface
Part I – Radio Module

Description of nRF24L01 and SPI

nRF24L01 (we refer to it in short as nRF) is a radio transceiver designed by Nordic Semiconductor which operates at worldwide 2.4 GHz ISM band. It is programmable via SPI interface and its current consumption is very low. Some features of this antenna are listed as following [7]:

- Auto acknowledgment and retransmit
- On the air data 1 or 2 Mbps
- Digital interface (SPI) 0 – 8 Mbps
- Fully RF compatible with nRF24XX
- Power supply range 1.9 to 3.6 V
- Complete OSI link layer in hardware

And some applications of the antenna are as following [1]:

- Wireless data communication
- Alarm and security systems
- Automotive
- Intelligent sports equipments
- Industrial sensors
- Toys

In order to have a wireless communication between two transceivers, we deal with 8 pins, VDD, GND, IRQ, CE, CSN, MISO, MOSI, and SCK, which the last four pins are related to the SPI interface. Figure 1 shows the block diagram of the nRF.

![Block diagram of the nRF24L01 REFMOD](image)
In the above picture, J1 is a connector through which all digital signals are routed. Figure 2 describes the J1 connector:

<table>
<thead>
<tr>
<th>Pin #</th>
<th>Signal name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GND</td>
</tr>
<tr>
<td>2</td>
<td>VDD (1.9V – 3.6V)</td>
</tr>
<tr>
<td>3</td>
<td>CE</td>
</tr>
<tr>
<td>4</td>
<td>CSN</td>
</tr>
<tr>
<td>5</td>
<td>SCK</td>
</tr>
<tr>
<td>6</td>
<td>MOSI</td>
</tr>
<tr>
<td>7</td>
<td>MISO</td>
</tr>
<tr>
<td>8</td>
<td>IRQ</td>
</tr>
</tbody>
</table>

Fig. 2. nRF24L01 REFMOD J1 pin out [16]

**SPI and nRF24L01 related pins**

To describe the pins, we start from the first two pins. The ground pin (GND) is connected to the same pin of the host microcontroller. VDD pin is connected to an input power regulator which allows nRF to operate at 3.3V.

CE pin is always considered as input with respect to the nRF. This pin has different meanings based on the mode that the antenna operates in. If the device is a receiver and CE is high, then it allows the antenna to listen to air for receiving the packets. If nRF is a transmitter, then CE is always kept low, unless a SPI command is sent to the device or it receives data from SPI which, in turn, are sent from microcontroller. We connect this pin of nRF to the TWCK pin of AT91SAM7S256, which is the pin number PA4.

CSN stands for Chip Select Not which is the enable pin for SPI bus and it is active-low. This pin is always kept high unless the device is sent a SPI command or it receives the data over SPI from microcontroller. We connect this pin of nRF to the NPCS1 pin of AT91SAM7S256 which is the pin number PA31.

SCK is the clock for SPI bus. When initializing the SPI, it should be kept low and, in order for clock to sample the data the values of CPOL and NCPHA, bits in SPI’s chip select register (CSR) are 0 and 1 respectively. We connect this pin of nRF to the SPCK pin of AT91SAM7S256 which is the pin number PA14.

MOSI stands for Master-Out-Slave-In. The host microcontroller is always master and nRF is always slave because nRF never sends the data unless it is requested by the microcontroller. Essentially, this pin is part of the bus through which the master (host microcontroller) sends the data to the slave (nRF). We connect this pin of nRF to the MOSI pin of AT91SAM7S256 which is the pin number PA13.
MISO stands for Master-In-Slave-Out and it is like MOSI but in the reverse order. This pin is part of the bus through which the slave (antenna) sends the data to the master (host microcontroller). As we see, SPI is a full-duplex bus through which we can send and receive the data in both directions simultaneously but on different lines. We connect this pin of nRF to the MISO pin of the microcontroller which is the pin number PA12.

The last pin is IRQ which is the interrupt pin to signal the microcontroller that something is happening. Interrupts can be considered for any combination of events such as data is received, data is transmitted, and the maximum number of retransmission is reached. If this pin is not used, then the STATUS register of the antenna can be polled to find out if an interrupt has happened. This pin is also active-low and usually it is kept high, but when an interrupt happens, it goes low. We connect this pin of nRF to the TWD pin of AT91SAM7S256 which is the pin number PA3 [7].

**How to interface with nRF24L01 via SPI**

The SPI interface allows us to read and write the registers within nRF, transmit data, receive data, and do the other stuff related to the nRF. As it is mentioned before, in order to have a full-duplex connection via SPI, the NCPHA and CPOL bits of the SPI-CSR register in our board should be set as 1 and 0 respectively. We set the SPI for transmission of 8 bit data and, finally, it is set as master. In order to make sure that we read the same data that we send to SPI, we take the advantage of using loop-back functionality by logically connecting the MISO to the MOSI pin and test the value of the data to find out if they are identical. This test is necessary but not enough: to make sure that the whole SPI works properly, we do need to check the functionality of both CE and CSN pins as well and, if they work as we want, then we can assume that we have our SPI working [7].

**Description of SPI Commands**

The detailed information related to the SPI Instructions are to be found in table 8-page 19 of nRF’s datasheet. In order to send or receive data to/from SPI, the following tasks should be done:

The CSN pin of nRF must be high to start with transmission. Then CSN pin should be taken to low to alert the nRF that it is about to receive something and it will remain low during whole transmission. Each new instruction is started by changing CSN from high to low. In parallel to the SPI commands which are applied on the MOSI pin, the STATUS register is also serially shifted on the MISO pin. Figure 3 from datasheet shows the SPI instructions.
Fig. 3. Instruction set for the nRF24L01 SPI interface [7]

When the required data are transmitted or all of them are received, the CSN should be set again. Finally, this data can be processed in microcontroller as per our wish. As an example of using such instructions, suppose we want to execute the R_REGISTER command on TX_ADDR register in order to read the content of the TX address register. The TX_ADDR has 5 bytes wide and we also use 5 bytes addressing. First, we take the CSN low and send the “00010000” command byte to the nRF. This command byte instructs the antenna to read the register at 0x10, which is the TX_ADDR register, and returns its content. Then the CSN is taken back to high. When we send any kind of command byte, the antenna returns the STATUS register.

The first instruction is the R_REGISTER that allows us to read the content of any register within nRF. Its binary format is “000AAAAA” where “AAAAA” is the address for the register whose content is going to be read. This address can be in range of 0x00 to 0x17 and we shall be able to send 1 to 5 bytes using this command (for the address registers 1 to 5 bytes can be sent but the rest have only 1 byte).

The second command is W_REGISTER that allows us to write any register within nRF. This command is very similar to R_REGISTER and its binary format is “001AAAAA” where “AAAAA” is the address of the register which is going to be written.

The next one is R_RX_PAYLOAD that allows us to read the content of RX FIFO when the antenna is in receiving mode. Here, the RX_DR interrupt occurs and CE pin is also involved in such a way that while we are receiving a packet CE is held high, once the packet is received CE
goes low in order to disable the receiving mode. Then, nRF deletes the content of which we have read from the top of FIFO. Then the RX_DR interrupt must be cleared and CE should go high to be able for monitoring the packets on air and, finally, the received data will be processed.

The last one is the W_TX_PAYLOAD instruction, which is commonly used when the device is in its transmitting mode. Here, first the “0xA0” command byte and then the payload is sent. It is important to consider that the length of payload on the transmitter must match the corresponding one on the receiver. In order to transmit the packet, the CE pin must be held high at least for 10 micro-seconds. Once the transmission is completed, the TX_DS interrupt must be cleared [9].

**Description of the Antenna’s Functionality**

When the device is initialized, it will work in one of the transmitting or receiving modes. If the antenna is in receiving mode and CE pin is set then, after 130 US, the device will listen to the air for packets. Here, the microcontroller has two options: either it can use interrupts or it can detect and receive the packets by polling. In case of using interrupt, the proper function that uses the R_RX_PAYLOAD SPI command should be delivered to the interrupt service routine. If polling is used, first the IRQ pin has to be checked to find if it is active and then the RX_DR (Receive Data Ready) bit should be checked for the possible data to be received and, finally, the R_RX_PAYLOAD command is executed. When the packet has been received, both IRQ and RX_DS interrupts must be cleared.

If the device is in transmitting mode and there is a packet to be sent, the W_TX_PAYLOAD SPI command should be executed. The next packet can not be sent until the TX_DS (Transmit Data Sent) interrupt occurs.

It is to be noticed that, before R_RX_PAYLOAD is executed, CE must be low and then after it should back to high and, therefore, in order for the device to go to the active mode for monitoring the air for packets, it will need 130 US of time. There is a risk of packet loss if this delay is not considered by transmitter.

The details of the modes of operation for nRF are to be found in the figure 4 and figure 5 tables of its datasheet as following:
When the device is to change its mode of operation from transmitter to receiver, and vice versa, the content of its Config register should be read and then, in order to be a receiver, the value of PRIM_RX should be set which, in this case, if CE is high, the device will be active and it listens to the air for the packets and, if the CE is low, the device will be in standby mode and it does not monitor the air. In order to put the device in transmitter mode, the value of PRIM_RX of Config register should be cleared.

If the device needs to consume less power, then the PWR_UP of Config register should be cleared and then CE should be low so that all registers will remain in their current state. If the device is to do something and wants to be active again, the PWR_UP of Config register should be set. In this case, if the device is in receiving mode and it should be active, then CE should be low and, if it wants to be in standby mode, the CE should be high [9].
Packet Description for nRF24L01

The radio module used in our project supports complete OSI data link layer in hardware beside other useful features and it also performs address and CRC computation. It is possible to send up to 32 bytes of data and also to use 3, 4, or 5 bytes addressing.

A shock burst packet format and description of its fields is illustrated in following figures.

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Address 3-5 byte</th>
<th>Payload 1 - 32 byte</th>
<th>CRC 0/1/2 byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 6.</td>
<td>A Shock Burst Packet Format [7]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Preamble | • Preamble is used to detect 0 and 1 levels. It is stripped off (RX) and added (TX) by nRF24L01. |
| Address  | • The address field contains the receiver address.  
|          | • The address can be 3, 4 or 5 bytes wide  
|          | • The address fields can be individually configured for all RX channels and the TX channel  
|          | • Address is automatically removed from received packets.\(^{16}\)  
| Payload  | • 1 - 32 bytes wide.  
| CRC      | • The CRC is optional.  
|          | • 0-2 bytes wide CRC  
|          | • The polynomial for 8 bits CRC check is \(X^8 + X^2 + X + 1\)  
|          | • The polynomial for 16 bits CRC check is \(X^{16} + X^{12} + X^5 + 1\).  
| Fig. 7.  | Data Packet Description [7] |

Data Transmission

Various scenarios can be defined for transmission of data between nRF radio modules using the functions that are defined in our device driver. For example, we can have different sizes of payload for transmission but here the point that has to be taken in consideration is that the payload must be defined of the same width in both parties. Another example is in case where there are more than two radio modules communicating with each other, so we should be able to send our data with different frequencies in order to avoid the interference between signals, to address that, we defined an interface that allows us to set the antenna operating on different frequency channels for transmission.

Now we will envisage a scenario whereby a character is to be sent to the other party and then the same character is received back as acknowledgement. The following sequence diagram shows the flow of data between different components in our project:
The TinyTimber device driver for the radio interface

The Radio Module on Base station is initialized as Transmitter

1. Initialization
2. Transmitter

The Radio Module on Remote PIE is initialized as Receiver

3. Initialization

User is asked to Enter a Character from Keyboard

4. Press key
5. Character

USART gets the character from keyboard

6. Get from keyboard
7. character

The character is radiated over Radio Channel

8. character
9. character
10. character

nRF24L01(1) switches on Receiving Mode to receive back the acknowledgement

11. Receiving Mode
12. character
13. character
14. character

nRF24L01(2) switches on Transmitting Mode to send the Acknowledgement back

15. Transmitting Mode
16. Transmitter
17. character
18. character
19. ACK

Acknowledgement is radiated over Radio Channel

20. character
21. character
22. character

nRF24L01(2) switches on Receiving Mode to receive the next character
PART II – The first phase implementations

There are many functions that are defined for different components of the PIE such as USART, timers, interrupts, etc. In this section, there is a brief description of the C functions that are developed to test the functionality of the radio module.

Initialization functions

```c
void InitSpi( void );
```

In order to be able to read and write from/to the nRF’s registers, this function should be called, so that the SPI interface will be initialized and the microcontroller can communicate with the radio module.
unsigned char spi_send_read_byte(<list of parameter(s)> );

Whenever a value is written or read from the nRF, it returns the STATUS register to the microcontroller. So, a full-duplex connection between AT91SAM7S256 and nRF is required which is established by calling this function.

void nrf_initialize(<list of parameter(s)>);

We call this function for writing the initialize values into the registers of nRF.

void nrf_initialize_debug(<list of parameter(s)>);

This is called for establishing a simple connection by determining the mode of operation as well as the length of the payload. The radio module can be initialized either in transmitting or receiving mode and the payload can be in range of 1 – 32 bytes.

**Addressing functions**

void nrf_set_rx_addr(<list of parameter(s)>);

This function sets the values for RX_ADDR register. We can determine the actual address that we want to use, the length of the address and the pipe number (from 0 to 5) on the receiver. If the pipe number is not in range of 0 to 5 then the function does nothing.

void nrf_set_tx_addr(<list of parameter(s)>);

This function sets the values for the TX_ADDR register. We can determine the actual address that we want to use as well as the length of the address.

**Interrupt check/clear functions**

bool nrf_irq_pin_active();

The IRQ pin is active low, so this function returns true if the IRQ is low, otherwise it returns false. When the antenna is about to receive a packet, or it has transmitted a packet, this pin is triggered.

bool nrf_irq_rx_dr_active();

This function returns true if the RX_DR interrupt is active i.e. the data is received otherwise it returns false.

bool nrf_irq_tx_ds_active();

This function returns true if the TX_DS interrupt is active i.e. the data is sent otherwise it returns false.
bool nrf_irq_max_rt_active();

This function returns true if the MAX_RT interrupt is active i.e. the number of maximum re-transmissions is reached otherwise it returns false.

void nrf_irq_clear_all();

This function clears all the interrupts in the STATUS register.

The payload related functions

unsigned char nrf_read_rx_payload(<list of parameter(s)>);

When the initialization function is invoked, the device will be set either as a transmitter or a receiver. If the device is in receiving mode and the CE pin is high then after 130 US the air will be monitored for the packets. To receive a packet, first we call the nrf_irq_pin_active( ) function and then we call the nrf_irq_rx_dr_active( ) function to check if there is something to receive and, if so, we call the nrf_read_rx_payload( ) function to receive the packet. When the packet is received we should clear the interrupts. So, first we call the nrf_irq_clear_rx_dr( ) function to clear the RX_DR interrupt and then we call the nrf_irq_clear_all( ) function to clear all interrupts in the STATUS register.

unsigned char nrf_write_tx_payload(<list of parameter(s)>);

If the device is in transmitting mode and it wants to send a packet, then this function is called. When the payload is sent, we wait until the TX_DS interrupt is activated, which is almost the same as the RX_DR interrupt in the receiving mode. This function writes the payload into the TX FIFO and, when the payload is transmitted, the TX FIFO is automatically deleted. If the rate of the transmission is high, the TX FIFO should be checked to find if it is not full before writing the next packet into it. If the enhanced shock-burst feature is used, the MAX_RT interrupt also should be checked for the packets which never reach to their destinations. Using this feature, the payload will not be deleted from the TX FIFO after transmission until it is acknowledged, so that it is possible to retransmit the same data.

RX/TX mode setting functions

void nrf_set_as_rx( );

and

void nrf_set_as_tx( );

While the radio device is operating as transceiver, it will need to switch from the transmitting mode to the receiving mode and it can take place by invoking these functions. It has to be taken into consideration with respect to the transmitter that when the nrf_read_rx_payload( ) function is invoked, the device will need 130 US to be able to monitor the packets on the air and, without this delay, there would be the possibility of losing packets. We can put the required delay by calling the DelayUS(unsigned int microseconds) function.
**power-up, power-down functions**

void nrf_power_up();
and
void nrf_power_down();

Most of devices need to save the power and consume less energy. We can save energy by keeping the content of all of registers in their current value except the PER_UP bit in the CONFIG register which can be done by invoking these functions.

**Channel frequency function**

void nrf_set_rf_ch(unsigned char channel);

here, “unsigned char channel” is the channel number that we want to use.

**Debugging function**

void nrf_get_all_registers();

This is a very helpful function which can be used for debugging purposes to observe the value of the registers within nRF.

---

**Part III – The Tiny Timber Interface**

**nRF24L01 and Tiny Timber**

As explained in previous sections, the foundation of Tiny Timber is based on reactive objects. A reactive object is a component that reacts to the incoming events by updating its internal status or by dispatching its outgoing events and it rests if there is no any other incoming message.

According to this point of view, a hardware component like serial port is a reactive object that, in all cases, the content of its internal registers is the current status of the object. The microcontroller itself also is a reactive object that reacts to the internal events and interrupts by updating the content of its registers and memory [14].

In order to encode the concepts of the object oriented paradigm in C language, we use a combination of `typedef` and `struct` to define and declare the variable structure of classes. Here, the standard technique for invoking the methods is called “self-application”. The self-application method involves invoking a method by applying that method to the address of the receiving object (that is to the `self` of the receiver). Thus, a method that belongs to the class X has to take a pointer to an object X as the first argument [14].
**nRF24L01 as a reactive object**

Hence, nRF also can be defined as a reactive object as following data structure in the “nrf.h” file:

```c
typedef struct{
    Object super;
    } nRF;
```

And then we look at the PIE as a reactive object in the file “pieRadio.h”:

```c
typedef struct{
    Object super;
    nRF *nrf;
    } PIE;
```

this means that the PIE is an object which is based on another existing object.

**Self-Application methods for nRF24L01**

The same concept is used for the nRF object on the PIE and the methods in the “nrf.h” file are defined as following:

```c
int initRadioDeviceAsTransmitter(nRF *self, int *arg);
```

When we want to initialize our radio module as transmitter, we invoke this method on the nrf component of PIE.

```c
int initRadioDeviceAsReceiver(nRF *self, int *arg);
```

When we want to initialize our antenna as receiver, we invoke this method on the nrf component of PIE.

```c
int local(nRF *self, int *arg);
```

We apply this method on the transmitter which sends a packet to another antenna that is in the receiving mode. When the packet is sent, it switches to the RX mode to receive the acknowledgment. Then, it switches back on the TX mode to send another packet.
int remote(nRF *self, int *arg);

We apply this method on the receiver which listens to the air for a packet that is sent from another antenna. When the packet is received, it switches to the TX mode to send the acknowledgment. Then it switches back on the RX mode for receiving the other packets.

Self-Application methods for the PIE

In the same way, the applications based on the Tiny Timber interface are defined for PIE, which we are looking at as a reactive object. It has a component, known as nRF, as another reactive object. The header definitions for these methods are defined in the “pieRadio.h” file as:

```c
void PIEInitTransmitter( PIE *self , int *arg ) ;
void PIEInitReceiver( PIE *self , int *arg ) ;
void PIELocal( PIE *self , int *arg ) ;
void PIERemote( PIE *self , int *arg ) ;
```

The functionality of these methods is as same as the methods that are defined for the antenna. The only difference is that here the antenna is a part of PIE, so those methods are invoked and applied on the nRF which now is acting as a component of PIE. The definition of these methods in “pieRadio.c” file, are as following:

```c
void PIEInitTransmitter( PIE *self , int *arg ){
    ASYNC( self -> nrf , initRadioDeviceAsTransmitter , NULL ) ;
}

void PIEInitReceiver( PIE *self , int *arg ){
    ASYNC( self->nrf , initRadioDeviceAsReceiver , NULL ) ;
}

void PIELocal( PIE *self , int *arg ){
    ASYNC( self->nrf , local , NULL ) ;
}
```
void PIERemote( PIE *self, int *arg ){

    ASYNC( self -> nrf, remote, NULL );
}

CHAPTER 4

Application Samples and Conclusions
Application samples

In this project, we have developed a device driver for radio communication with a Tiny Timber interface. This device driver can be used in any application that needs wireless communication. For example, it is used in development of a project known as “partly autonomous vehicles for transports in limited spaces” defined for “Design of Embedded and Intelligent Systems” course offered by IDE department of Halmstad University [19].

As Fig.1 shows [19], the components of this project are camera, system PC, project PC, base station, and PIEs. The idea is to recognise the position and direction of the PIEs by analysing the images taken by the camera and then move PIEs from source to destination and pass them through the passage in such a way that there will be no collision between PIEs and, also, they should not cross the borders of the room. The communication between base stations and PIEs is wireless. So, in order to control the movement of the PIEs, an application based on our device driver is developed which sends the speed of the right and left engines to the remote PIEs.

The other application based on our project is “heart beat”, where a character is sent from the base station to the PIE periodically in each 100ms and the same character is received back. If the character that has been sent, and its correspondent acknowledgement are identical, then the green LED of the board on base station is turned on.
Now we test the functionality of our device driver by developing an application that initializes the antenna either as a receiver or transmitter, so that the wireless communication module, in its transmitting mode, sends a character to the other module, then switches on receiving mode and receives back the same character as a manual acknowledgment. The hardware requirements for running this application are two SAM7P256 boards, which are connected to two nRF antenna modules via SPI interface. Also, a serial connection (RS-232) is required to connect one of the boards to the PC, so that the AT91SAM7S256 microcontroller can receive a character from the PC’s keyboard and deliver it to the nRF via SPI. We initialize this antenna in TX mode and, when it has finished with the transmission of the character, it switches on the RX mode to receive the acknowledgment. The other kit is stand-alone and we initialize it in the receiving mode which listens to the air for a packet that is transmitted from the other party. When the packet is received, it switches on the transmitting mode to send the character back as acknowledgment.

We use the HyperTerminal application in order to display the results on the screen. Using the HyperTerminal, we create a new connection and set it at 9600 baud rate and 8-N-1 without flow control, as following figures show:

![Fig. 9. Creating a new connection in the HyperTerminal](image-url)
Once the connection is established, the application for testing the device driver can be executed.

In order to have one PIE as base-station, it should be initialized by invoking the
PIEInitTransmitter( ) and then PIELocal( ) methods on PIE object. Then, the
STARTUP(app) macro would be called to execute the local application so that the user will be
asked to press a key on the keyboard and this character will be radiated over RF channel. The
following shows the “main.c” file through which we call our methods using Tiny Timber
directives:

```c
#include <TinyTimber.h>
#include "nrf.h"

nRF nrf24101 = initnRF( );
PIE thisPIE = initPie( &nrf24101 );

void app ( ) {

    //Initializing PIE as Transmitter
    WITHIN( MSEC(0), MSEC(5), &thisPIE, PIEInitTransmitter, NULL );

    // getting a character from keyboard and sending over RF
    AFTER( MSEC(10), &thisPIE, PIELocal, NULL );
}

STARTUP( app ) ;
```
The remote PIE, which is stand-alone, is to be initialized by calling the `PIEInitReceiver()` and then `PIERemote()` method should be invoked on PIE. Then, the `STARTUP(app)` macro would be called to execute the remote application so that the antenna will monitor the air for the packets and, after receiving a packet, it changes its mode and sends the same packet back to the local antenna. The following figure shows the result of the execution of the application.

![Figure 11: The result of the execution of the application](image)

It is also possible to produce the characters internally and send them sequentially to the receiver and get back the same characters as acknowledgment.

**Conclusions**

In this work, we developed software implementing a Tiny Timber interface to a radio device placed on PIE. We have shown how a small, embedded system can be modelled and prototyped, based on reactive objects and then how it can be transformed to an executable code based on the programming principles of Timber.
References


[13] Johan Nordlander, Mark P. Jones, Magnus Carlsson, Richard B. Kieburzt, and Andrew Black. “Reactive Objects”, OGI School of Science & Engineering at OHSU, 20000 NW Walker Road, Beaverton, OR 97006


Appendix A

Introduction of PIE

PIE an abbreviation of “Platform for Intelligent Embedded System” is a small robot assembled in IDE department of Halmstad University. It is used for prototyping and developing of different applications as well as educational purposes in the area of embedded and intelligent systems. This robot is used in “Embedded Systems Programming” course at IDE department to practice the concepts of embedded programming and to get familiar with the literacy of real time operating systems. Another project that uses the PIE is “Partly autonomous vehicles for transports in limited spaces” offered in “Design of Embedded and Intelligent Systems” course at the same department [19]. Our radio communication device driver is also effectively involved in DEIS course that is discussed in previous chapter.

PIE components

Different components of PIE are displayed in following picture:
PIE is controlled by “AT91SAM7S256”, an ATMEL ARM 32/bit RISC microcontroller which is programmable in C [1]. Some features of this microcontroller are that it has 256 KB high-speed flash and 64 KB SRAM memory. This microcontroller is mounted on a board called as SAM7-P256 developed by Olimex [3]. The microcontroller is extensively including more peripherals such as two USARTs, a SPI, an USB 2.0 Full Speed Device Port, and etc.

Programming of the microcontroller is done via JTAG port with ARM-JTAG-EW which is also produced by Olimex [4].
PIE is equipped with a complete Robot Drive System, comprising of two EMG30 engines with encoders, two 100mm wheels, and MD25 motor drive module. The MD25 will drive two motors Controlled by I2C Interface. The collection of the two wheels, two engines, and the I2C interface is called RD02 which is developed by Robot-Electronics [5].
The robot also consists of two proximity infrared units which are used as distance sensors. These sensors are developed by Sharp [8].

![Infrared distance sensor](image1)

Fig. 16. Infrared distance sensor

It uses a transceiver called as “nRF24L01” designed by Nordic Semi Conductor which is discussed in chapter 2.

![nRF24L01 and SPI interface](image2)

Fig. 17. nRF24L01 and SPI interface
The robot gains its power from a rectangular 12V battery [6].