Reducing power usage in wireless sensor networks using software abstractions

Paffen, T.F.P.

Award date:
2010

Eindhoven University of Technology
Department of Mathematics and Computer Science

Confidential

Master Thesis

Reducing Power Usage in Wireless Sensor Networks using Software Abstractions

By
T.F.P. Paffen

Author:
Paffen, T.F.P. (s0549987)
t.f.p.paffen@student.tue.nl

Supervisor Technical University Eindhoven:
Hilbers, P.A.J.
p.a.j.hilbers@tue.nl

Supervisor Philips Research Eindhoven:
Stok, P.D.V. van der
Peter.van.der.Stok@philips.com

Eindhoven, July 2008

Summary

Wireless sensor networks (WSN) are a very diverse and relatively new area of research. WSNs consist of sensor nodes able to communicate wirelessly and it is preferred that they operate for a longer period of time. Since many wireless sensor nodes are powered by an independent power source such as a battery, power consumption is an important issue in the WSN research field. In this project, two objectives are set: to reduce power consumption using software abstractions (1) and to create the software abstractions in such a way that the development for the sensor nodes is made easier and natural (2). The hardware that is going to be used is the blue box, a sensor node developed by Philips Research. These nodes contain a CoolFlux DSP [10], a CC2430 radio chip with an 8051 microcontroller (packaged called AG2) [7], three one-axis gyroscopes [6], two two-axis magnetometers [8] and one three-axis accelerometer [9]. The nodes run Flex-OS as the operating system, which is a port of FreeRTOS with in addition peripheral abstraction architecture, time and task synchronization and Media Access Control (MAC) services.

To realize the two objectives, three abstraction layers are devised on top of the peripheral drivers of the OS. On top of the peripheral driver architecture, device drivers are designed for sensors and radios. There are two drivers, one for sensors and one for the radio. It provides the programmer with API functions as create, configure, sample, send, receive, etc. Above that layer, the network schedule layer is placed. This handles the network scheduling to allow the radio to shut down. The necessity of a schedule is due to the fact that a radio needs to know when it needs to be powered to receive a message from a gateway (computer, PDA, etc.) or other node. If the radio does not know when to expect a message, it continuously has to be powered and listen for messages, making it very power consuming. The top layer is the Program Specification Functions (PSFs) layer. These PSFs are four functions with which the node can be programmed. The four PSFs are: sample, send, receive and compute.

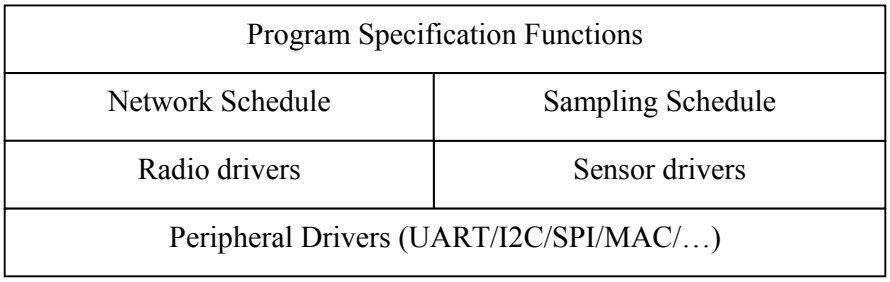

Figure 1: Proposed abstraction layers

The sample function allows the programmer to set the sampling frequency, use the shutdown function of the sensor for a specific sensor. For example, if the accelerometer needs to be sampled periodically at 25 Hz and the sensor can power down between sampling, the program would specify the following PSF: sample(Sensor, frequency, ShutdownBetween, ShutdownInactive, name of the sensor). In the specific case of the accelerometer this would be: sample(Acceleromter, 25, 1, 0, “Accelerometer”). The blue boxes contain three sensors which can have different sampling frequencies in the implementation with PSFs. All sensors are sampled in a separate task when the frequencies are different, but sensors sampling at the same frequency are grouped in one task. The send PSF allows the programmer to specify the send frequency at which the radio needs to send the information to the

sink. If the sampling frequency is higher than the send frequency, several samples are grouped together in one message. For the AG2 radio the following example of the PSF with a send frequency of 25 Hz can be called: send(AG2radio, 25, 1). If the send PSF is called, a task is created (as with the sensors) which will handle the actual sending. In this project it was chosen to use a network schedule which allows asynchronous sending and receiving. The sending can happen according to the send frequency, while the receiving is periodic once every second. The reason this is done is that the specific application which this project focused on did not need to receive anything from other nodes and only occasionally needed to receive data. The receive PSF sets up a task which will periodically power up the radio to receive data. The last PSF, compute, is currently not used. It can be used to do calculations on the node itself before sending data to the host or other nodes. This can be done by manually programming the computations that are done on the data.

All PSFs are connected via queues. The sensor driver allows every sensor to have one queue associated with it in which the samples need to be stored. The radio driver allows two queues to be associated with the radio, namely a send and a receive queue. The former is used to store samples which need to be send by the send PSF and the latter stores packets which are received by the radio.

After implementation of the abstraction layers, measurements were done to see how much overhead was produced by the abstractions (both in time and in memory). There were also measurements done to see the percentage of total time, the devices of the node (radio and sensors) are powered up. In the case of sampling at 100 Hz and sending at 25 Hz (the physiotherapy application), the magnetometer is shut down 87.5% of the total time and the radio 91.4%. The power consumption of the node is reduced by approximately 42% allowing the node to operate for approximately 12 hours (approximately 2.5 times longer than the original application).

Especially for the magnetometer, the accelerometer and the radio a big decrease in usage can be seen. The gyroscope, which is the most power consuming device in the blue box, has a start-up time of 40ms and can therefore only be shut down between samples for frequencies lower than 25 Hz. The additional memory usage of the abstraction layers was almost 10 kilobytes. The data memory of the CoolFlux DSP is 64 kWords (one word is 24 bits). This means the overhead of the abstraction layers is within reasonable limits and the trade-off between memory usage and ease of development and power reduction is worth it. The overhead in execution time is mostly found in the start-up time of the devices. Since the DSP is sequential, the devices have to wait until it is their turn to be powered on to take a sample. Since the actual time to take the sample or send the data is negligible, the main overhead is in the start-up times of the devices. This is also the main reason why the gyroscope (with a start-up time of 40ms) is sampled last of all three sensors to minimize the interference of this start-up time on the other two sensors.

Contents

1. Introduction

Wireless sensor networks (WSN) are a relatively new area of research. There is a diverse range of possible applications for such networks from body sensor networks to large-scale networks such as forest fire monitoring. As the name WSN suggests, the networks consist of sensor nodes able to communicate wirelessly. The sensor nodes often have an independent power source (for example a battery) and their own processor and radio. Body sensor networks already existed before the introduction of WSNs but were not practical due to the cumbersome cabling necessary to connect the nodes to the controller. This setup can of course be unpleasant for the wearer and restricts his/her movement with the length of the wires. Wireless sensor nodes would bring a lot of freedom to this field and this is one of the reasons why wireless sensor networks are an important research topic. In the area of wireless sensor networks the reduction of power consumption is an important field of research. Due to the limitations of batteries (in most cases the power source of the wireless sensor nodes), the amount of time a node can operate is severely limited.

Several types of wireless sensor nodes exist, but our research focuses on the blue boxes [11], nodes developed by Philips Research for body sensor networks. These blue boxes contain the CoolFlux DSP [10], a CC2430 radio chip with an 8051 microcontroller (packaged called AG2) [7], three one-axis gyroscopes [6], two two-axis magnetometers [8] and one three-axis accelerometer [9]. The gyroscope, magnetometer and accelerometer are seen as three axes sensors in the application and no distinction is made between a sensor with three axes and three separate one-axis sensors. They are powered by a lithium polymer battery. Flex-OS [12] is the operating system running on the CoolFlux DSP and contains also a complete peripheral abstraction architecture, time and task synchronization and Media Access Control (MAC) services. Flex-OS is not only used on the blue boxes but also on the SAND nodes. These nodes are also developed by Philips Research and have a similar hardware layout as the blue boxes. The main difference between the nodes is that the SAND nodes use a CC2420 radio (which has no integrated microcontroller) and the blue boxes use an AG2 radio chip (which includes a microcontroller). The MAC services integrated in Flex-OS are only used for the CC2420 which has no dedicated microcontroller that runs MAC services. The MAC services for the CC2430 run on the 8051 microcontroller embedded in the AG2 chip which is connected via UART (Universal Asynchronous Receiver/Transmitter) and GPIO (General Purpose Input Output) to the CoolFlux DSP and the sensors are connected to the DSP via SPI (Serial Peripheral Interface) and GPIO. Figure 1.1 shows a simplified schematic of the blue boxes. A software application that runs on the blue boxes helps with physiotherapy exercises for stroke recovery by providing feedback. This application provides patients, recovering from a stroke, with feedback on exercises they perform to help their rehabilitation process. The sensors are used to measure if the patient correctly performs the physiotherapy exercises and then provide feedback to the patient on the PC. The application uses a PC as the sink to receive all data from the sensors and perform further computations on it. On the node itself, no computations are done. In this application the nodes do not need to communicate with each other, only with the sink, thus creating a star network.

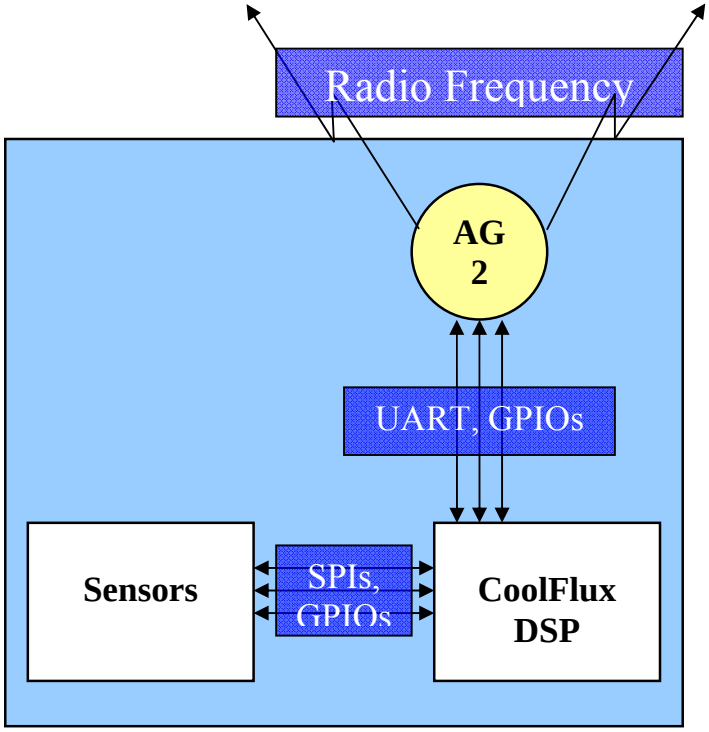

Figure 1.1: Simplified schematic of the blue boxes (base image from [4])

By turning on and off the sensors and radio when they are not active sampling/sending/receiving, power can be saved and therefore the battery life of the node extended. Furthermore, to ease development on these sensor nodes several abstractions are proposed and implemented to provide programmers with a straightforward way of programming applications. The physiotherapy application will be seen as the main application on which this approach will be tested. Therefore, design decisions that need to be made will try to be as generic as possible but are largely motivated by this specific physiotherapy application.

2. Problem description

Programming wireless sensor nodes of the blue box type at this point requires that the programmer needs to be familiar with the hardware layout of the nodes. This familiarization requires time and thus the time to develop the WSN application increases (1).
The second problem is battery live of sensor nodes in wireless sensor networks. In the current software, a programmer has to shut down and start sensors and radio by programming this explicitly. The programmer also has to manage that the radio and/or sensors are started up before taking a sample or sending/receiving a message. This not only complicates the development for a programmer, it can also mean a programmer does not even use the start up and shutdown functionality (2).
This presents the third problem for the programmer. The moment to shut down and start up the radio should be known. Because the node may need to receive information from another node or a gateway/sink, the program needs to know when data is expected from other nodes. In our case, a network schedule can help to determine the intervals at which the radio can be shut down (3).
The fourth and last problem addressed here is the power consumption of wireless sensor nodes. For nodes with active sensors and radios, the power consumption can be high. In [2] an analysis was done on the power consumption of the blue boxes running a specific application. Since nodes are battery powered (for the blue boxes a cell phone battery is used) power consumption is a big issue. The blue boxes with the physiotherapy application currently runs 3.5 hours continuously while weeks are needed (4).

To tackle the first problem (1), a software abstraction is devised to hide the underlying hardware from the programmer while retaining the freedom to set application parameters. These include the sampling frequency, sending frequency and the option of shutting down a sensor between taking samples.
The second problem (2) will be addressed by the software abstractions mentioned above. The abstractions, provided by this project, will include the functionality to turn a radio or sensor off between taking samples or sending/receiving data (2).
This project will use one fixed network schedule (3) which is calculated offline (see section 4.4). For the sensors, the sampling schedule does the same but leaves the programmer with the ability to change the sampling frequency and thus the sleeping periods. Both the radio and the sensor drivers will allow the programmer to specify if and when the shutdown functions should be applied. All improvements will have a reduction of the power consumption as effect by switching off sensors and the radio for a given period of time while maintaining communication.
Because the software abstractions (4) simplify the use of shutdown functions for both the sensors and the radio, the power consumption of these components will reduce significantly. Especially by shutting down power consuming components like a magnetometer and the radio when they are not used, it will help prolong the time a node can run continuously.

Several constraints are placed on this design. First the network schedule is going to be calculated before operation of the nodes, which means that during operation it is predetermined beforehand when a node needs to listen to the wireless medium and when it can send data over the medium. The second constraint is that the project assumes a connected (all nodes can directly communicate with the sink) body sensor network environment. The applications of interest have in general not more than eight

nodes in their network. Therefore the network schedule can be simple. For the nodes, only communication with the gateway/sink is needed, no communication between nodes. The nodes send their data to a sink and must be able to receive data from the sink (to keep the implementation as generic as possible). Furthermore the sensors and radio drivers will be implemented for the blue boxes running Flex-OS. A different connection of sensors or radio might require a different implementation. On the blue boxes, Flex-OS is used. The consequence is that time- and task synchronisation functions are implemented in the Flex-OS environment as well as peripheral drivers (SPI, UART, etc) and a MAC layer implementation. The time synchronisation algorithm runs in a separate task that is not further considered in this project.

Problem formulation

The shutdown of sensors in between taking samples and the shutdown of the radio between sending/receiving realize the two improvements on the power consumption that are gained from these implementations. The shutdown of sensors during inactivity (application on the gateway has not yet started, sensor is still being attached, etc) is not taken into account for the calculation of this improvement since it also requires information from the application running on the gateway. The number of nodes is fixed to a maximum of eight nodes.

The second part of our hypothesis is that the time to develop should be decreased by increasing the usability of the software for the programmer. Therefore drivers and high level abstractions are presented to the programmer.

3. Objectives

Two objectives are set for this project. These objectives are interwoven with each other. The first objective of this project is to develop software abstractions as an extension of the current operating system for wireless sensor nodes. These software abstractions will provide the programmer with a way to develop applications without having to be familiar with the underlying hardware and with low level software. The second objective is to use the software abstractions to control the hardware in such a way that a reduction in energy use is realised.

The software abstractions will be validated on the blue box sensor nodes [11] developed by Philips Research. They contain a CoolFlux DSP [10], a CC2430 radio chip [7] (with an 8051 microprocessor) and three 3-axis sensors (an accelerometer [9], magnetometer [8] and gyroscope [6]). The CoolFlux DSP runs Flex-OS [12], based on the FreeRTOS micro-kernel, which provides the programmer with a complete peripheral abstraction layer, MAC services and time and task synchronisation.

4. Improvement

As described in chapter three, one of the goals of this project is to simplify the programming of wireless sensor nodes, but within the confines of the FreeRTOS environment. The Flex-OS software platform has been designed to be easily portable to new microcontrollers. This exchangeability over different hardware platforms requires a certain level of abstraction over the underlying hardware. On top of that, the reason to increase the level of abstraction in this project is to improve the ease of development for programmers and therefore shortening the time to develop. The current abstractions implemented in Flex-OS provide the programmer with peripheral drivers as UART, SPI, etc, but not yet more generic drivers as sensor and radio drivers (the latter holds for the version of Flex-OS running on the blue boxes since a radio driver analog already exists in the MAC services for the CC2420). These extra abstractions of course cost both program memory as well as a small amount of delay. It is felt however that the ease of programming is for now more important than the possible increase in delay. Only after testing can it be concluded if the amount of delay is acceptable or not. The same holds for the memory usage. After actual testing it can be concluded if the memory usage with the new abstractions is acceptable.
First, a global overview will be given of the abstractions that are envisaged and why these need to be made. After that, a more detailed description will be given for every abstraction.

The highest level of abstraction that is going to be provided is the "Program Specification Functions" (PSFs). These PSFs are four functions which the programmer uses to specify the temporal aspects of the application. An example is the frequency with which data needs to be sampled and the frequency at which the data has to be sent. PSFs use the network schedule and sampling schedule layers to access underlying device drivers like the radio and the sensors. This way the PSFs are completely separate from the underlying radio and sensor implementations. In section 4.1 these PSFs will be discussed in more detail.

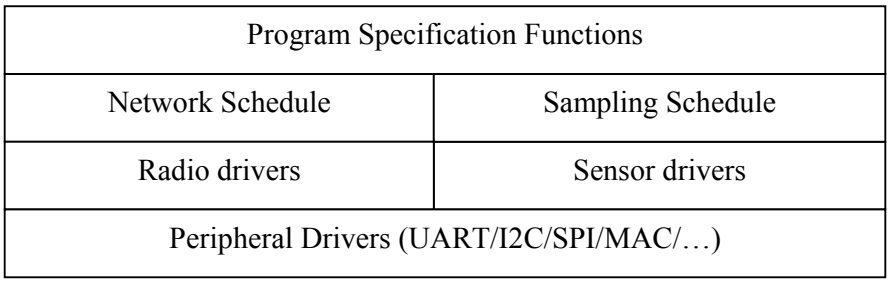

Figure 4.1: Proposed abstraction layers

The second abstraction layer is composed of the schedules. They consist of the network schedule and the sampling schedule. The network schedule regulates when the radio can send data and when it has to receive data. This is to allow the radio to be shut down in between. The main reason to keep the schedule as a separate abstraction layer is that a schedule can vary depending on the application. Therefore by separating this in a separate abstraction layer, the schedule can independently be (ex)changed. While the network schedule is fixed beforehand in this project, separating the network schedule layer will make it easier for programmers to develop different schedules than if the schedule was integrated into the PSFs or radio drivers.

The sampling schedule regulates the moments when the sensors take samples. Every sensor has its own sampling schedule that is set by the PSF (see section 4.1). This facilitates the shutting down and starting up of the sensor in between samples. The programmer does not need to take care of that. A sensor is seen as one entity, so for example a gyroscope is seen as a 3-axis gyroscope and not as 3 separate 1-axis gyroscopes. This makes sure the number of sensors does not become too large and the sensor samples of all 3 axes are kept together. It also keeps the gyroscope as one device, making the use of the device driver easier.

The third abstraction layer consists of two parts: the sensor driver and the radio driver. These will be briefly discussed here and then in greater detail in section 4.4. The sensor driver abstracts from the peripheral drivers (GPIO, SPI, etc). It provides the programmer with functions to access the sensors. This way the peripheral drivers are hidden to the programmer and thus provide an interface to the schedule layers. It will also include the automatic start up and shutdown before and after sampling a sensor. The programmer will have the option to shut down the sensor by the application. By automating this start up and shutdown, reducing power becomes more intuitive for the programmer.
The radio driver abstracts the access to MAC functions used to access the radio. This abstraction provides an interface to the network schedule layer and, as with the sensor driver, facilitates the shutdown and start up of the radio by calling a function. These power reduction functions are incorporated in the sending and receiving functions so the programmer does not need to switch the radio off manually.

The second goal, as stated in chapter two, is to reduce the energy consumption in the wireless sensor nodes. This will be achieved by using the software abstractions specified in the first part of this section. The highest software abstraction, the set of PSFs, gives programmers the option to shutdown both the chosen sensors as well as the radio. The sensor will then shut down in between taking samples. Depending on the sampling frequency and the energy consumption of the sensor, this can already reduce the energy consumption of the node significantly. The network schedule performs a similar function for the radio, allowing it to shut down if it does not need to receive or send information. One of the PSFs (as is discussed in more detail in section 4.1) is going to be used to do calculations on the node before the data is sent. Depending on the calculations that are done on the node, usually less data needs to be sent. For example, the three x-values from the three sensors need to be averaged. Doing it after sending means three x-values need to be sent. If the averaging is done on the node, only one average x-value has to be sent, reducing the amount of energy used by the radio. The CoolFlux DSP is a processor which cannot be shut down due to hardware limitations. Therefore the processor is always on and the increase of computation that has to be done is negligible in comparison to the amount of energy used by sending data via the radio.

4.1 Program Specification Functions

The program specification functions (PSFs) will be used to give the programmer a way to define the desired application at a high level. These functions will use functions in lower level abstraction layers to keep the layers as separate as possible. PSFs will set up tasks for the program to run, so the programmer does not have to do this explicitly. The programmer should be able to create the application by specifying the desired tasks and the way they should be connected (for example as shown in

figure 4.2). The interaction between tasks as seen in figure 4.2 will be done via queues. By calling multiple instances of the sample task for example, multiple sensors will take samples. There are limitations to this approach. The first is that a sample task cannot be directly linked to a receive task since both are producing data. The next limitation is that a send task will have only one send queue, which can be accessed from the compute task or from one of the sample tasks. The connections between blocks can vary depending on the type of application. It is possible to not use the compute task and directly connect the sampling tasks via the send queue to the send task. This has as a consequence that the queue may be accessed from multiple tasks (depending on the number of sample tasks). The data in this queue will be sent. To limit the number of tasks that are running, sample tasks that sample at the same frequency will be merged. Both the receive and compute tasks can be omitted if there is no computation to be done or if no information needs to be received from the sink (the time synchronisation is not considered in this).

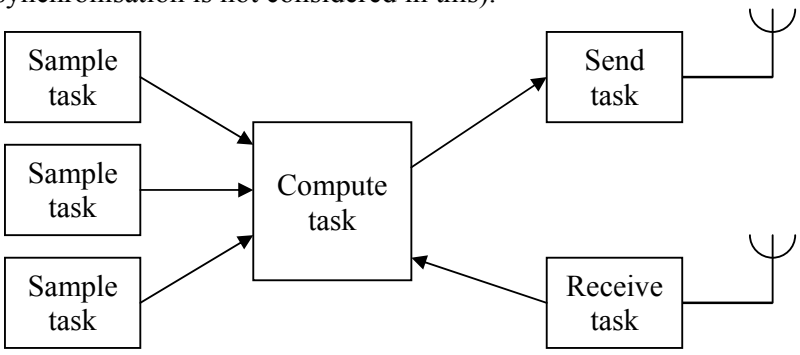

Figure 4.2: Graphical depiction of a possible arrangement of tasks created by PSFs

A main advantage of sensor node applications is that many have similar tasks and requirements. Most sensor nodes are used to sense data, process this data and communicate this data to the outside world. Because of this generally common pattern there are four functions that sensor nodes need to perform: taking samples using sensors, sending information, receiving information and performing different computations on sensor data. This presents an opportunity to help the programmer with creating an application. By only providing functions that achieve these four functions for a sensor node, the programmer does not need to program these manually. The "program specification functions" are thus restricted to four functions, namely the following:

Sample: This function lets the programmer specify the sampling frequency for a certain sensor and indicate the use of the shutdown functions of the sensor (both the "between samples" and "during inactivity" shutdown separately)

Compute: Since the programmer is the only one who exactly knows what computations need to be performed on the data, a generic implementation of a computation function would not be feasible. Therefore the programmer has to manually write part of this function. A pre-created task for the compute function is already there, with access to the radio queues (send and receive) and all sensor queues. The programmer can manually program what computations the compute function actually has to perform on the code. The send queue of the radio can be used by the programmer to store the end result(s) which can later on be sent by the send PSF. This task is automatically created when the compute PSF is invoked.

Send: This function lets the programmer specify the data to send, the radio to be used and the frequency the data has to be sent. The use of the shutdown function in between sending can also be enabled or disabled.
Receive: The programmer can specify the radio that has to receive data and if the radio should shut down. What should be done with the received data can also be specified by implementing a custom function developed by the programmer (as with the compute function). If the data is used in the compute function than the receive queue can be accessed in the compute function that a programmer has to implement manually and the receive function should not be used. Since the send part of the radio already knows if the radio should shut down in between, this does not have to be specified in this PSF again. This is to avoid conflicting shutdown parameters if they are specified in both the send and receive PSF. One option is to let it be specified in both but give priority to one of the two functions, so that if they both have a different parameter the send PSF always overwrites the one from the receive PSF or the other way around. This can lead to confusion as to which PSF has precedence and therefore it is decided to only let the shutdown of the radio be specified in the send PSF.

Restricting the number of functions the sensor node can perform is of course a trade off between ease of development and what a programmer can specify. Certain nodes may still need to perform additional tasks or programmers may wish to expand the number of functions the node can perform, but this will also again increase the complexity for the programmer to create his application. Due to the layered nature of the software abstraction, additional PSFs can later always be written if the desired functions of sensor nodes change or the set of functions is enlarged to encompass other tasks. This will make this approach more flexible even if other features are needed.
Increasing the number of computations on the node is actually a good practice since in most cases it reduces the number of messages or the length of messages that need to be sent. Especially at this time trading sending energy for computation energy is more efficient since the processor of the blue boxes cannot be turned off due to hardware limitations but the radio can. In general this is already an efficient trade off since in most cases the power consumption of the processor is much lower than the power consumption of the radio. Also, the use of the PSFs will make it easier for programmers to use the start up and shutdown functions of the sensors and radio without having to devote time on when and how start up and shutdown functions need to be evoked.

The sampling and sending frequencies can be specified in the PSFs. Therefore, to verify that the sampling and sending frequencies are valid, the following three conditions are tested:

- Sampling frequency ≥ Send frequency (avoid sending empty packets)
- Sampling frequency / Send frequency ≤ 20 (avoid overflowing the queue which stores data samples, it has 60 places for samples (1 sample is 3 integers), 20 for each sensor)
- 1/sampling frequency > Start-time sensor (only applicable if the shutdown function between samples is used, to make sure the sensor has enough time to start)

Example

To illustrate the use of these PSFs, an example will be given. A programmer wants the sensor nodes to sample two sensors (sensor A and B) at a frequency of 100 Hz. Both sensors provide him with x, y and z values each. The programmer wants to add the x-values, the y-values and z-values and then send these triplets at a frequency of 50 Hz. The way the programmer would do this using the PSFs would be as follows:

1. First use the sample function to set up both sensors with the desired frequency and shutdown parameters: *Sample(sensor A, 100Hz, shutdown, …)* and *Sample(sensor B, 100Hz, shutdown, …)*
2. Then the pre-made function is implemented to take the three parameters of the sensors and add them. The result of this computation is stored in the send queue. Now he invokes the compute function: *Compute(…)*
3. Now he needs to specify the sending function to concatenate two processed samples in one packet and send it. The two samples are concatenated because the send frequency is half of the send frequency. This gives the following function: *Send(Radio A, 50Hz,…).* This function will retrieve the items in the send queue and sent the data at the indicated frequency.

4.2 Sampling Schedule

The sampling schedule is used to determine at which points in time the sensor needs to take a sample. The sensor can be shut down in between these samples, a parameter set by the programmer. As stated in [2] shutting off the sensor between samples can corrupt the sampled data. An example is aliasing, which can occur to accelerometer data if there is no continuous sampling. Therefore turning the accelerometer off at random times should not be done. Each sensor can have a different schedule. The schedule is made by providing the sampling frequency in the PSF. The sampling schedule is not a separately implemented schedule like the network schedule. The schedule is enforced by the sample PSF.

The reason to choose for an integration of the sampling schedule in the sample PSF is threefold. First, the sampling schedule is not as complicated as a network schedule. Since sensors only sample data at a certain frequency and no outside influences require a sensor to be powered up (like a radio that needs to receive) the sampling schedule can be kept simple. A second reason is that all data is available at the level of the PSF like frequency and shutdown boolean. Placing this outside the PSF would complicate the implementation and later also the understanding of the sample PSF. A third consideration is that the only real setting for the sample schedule is the frequency at which the sensor needs to sample. With that information, the schedule knows when the sensor can sleep and when to wake up. Since the frequency can directly be set in the PSF, the schedule can be changed by the programmer on that level.

4.3 Network Schedule

To be able to shut down the radio when there is no data to be sent or received, a network schedule is made. If there is no schedule, the application does not know when the radio has to wake up. Since many factors play a role in determining which network schedule works best in a specific case, the network schedule layer has been separated from both the radio driver layer and the program specification layer. The network schedule is fixed during operation of this network. If a different schedule is needed a new schedule should be made and replace the current network schedule.

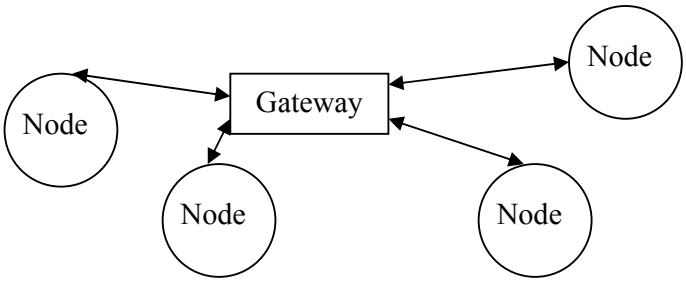

Figure 4.3: Layout of a star network supported by this implementation

In the application where these abstractions will be validated, a star topology is applied. End-nodes do not need to communicate with each other; they communicate their data to the sink. The sink is in this case a PC which does all calculations on the data sampled by the sensors. The sink only needs to communicate time synchronisation messages to the nodes. There will be no multi-hop topology applied, since it is not necessary in the example body network environment. As a result the schedule that will be used can be simple. The schedule is adhered to by both the sink and all the nodes. The use of such a topology places limitations on what can be done in the network. The first limitation is that no inter-node communication is possible. Secondly, nodes that are outside the range of the sink cannot communicate at all with the sink, since there is no multi-hop supported. There are several options in which this star topology (without multi-hop) can be represented in a schedule. Three options will be discussed further in the coming sections and advantages and disadvantages will be noted to propose a selection.

4.3.1 Schedule option one: Timeframe and slot division

The first option for a network schedule is to divide a timeframe into slots. The length of both the frame and slots can be set with a function. The first eight slots are used for sending information and the last two for receiving. In these last two timeslots, the radios of all nodes need to be powered up. If a node does not need to send anything, the radio will not be started and remains in "power down mode". The number of slots is chosen to be ten because there needs to be two receiving slots (one for time synchronisation and one for additional information from the gateway/sink) and eight sending slots since in the current setup there will be no more than eight nodes, making it possible to give each node its own slot later on if necessary.

Send	Send	Send	Send	Send	Send	Send	Send	Receive (TS)	Receive

Figure 4.4: Division of slots in one timeframe for schedule option one

The API that this network layer would offer to the PSFs consists of three functions:

- *sendASAP(…):* sends the data in the first send slot that is available
- *received(…):* provides the received information to the higher layers
- *configure(framelength, slotlenght, …):* configures the network schedule

The main advantage of this schedule is that every node can have its own sending slot and therefore the chance of having to retransmit due to collisions is greatly reduced. It also allows the programmer to shuffle the slots (including framelength and slotlength) around to the desired need of the application. A big drawback of this schedule is the

larger restriction on sending information than necessary. However as a generic approach, this is much more usable when for example the receiving gateway/sink has to shut the radio down as well. Since the receiving radio of the sink is always on, adhering to a network schedule that restricts sending to slots creates unnecessary delays in receiving the samples at the sink. This is a disadvantage that of course only holds for this application specification.

4.3.2 Schedule option two: Only schedule for receiving

Exploiting the fact that the physiotherapy application does not need communication between nodes, a strict schedule for sending data to the sink computer is not needed. The application needs to receive information from the gateway/sink like time synchronisation messages. Thus a schedule is needed for system messages but a schedule for sending application messages over the network is not. This option for a network schedule will therefore only schedule the receiving part and not the sending part. The way the current task synchronisation works is that only one task (which according to [4] has to have the highest task priority for the scheduler of the OS) can synchronise with identical tasks on different nodes. To ensure that all nodes turn on their radio at the same time, task synchronisation will be used in the receive task. In pseudo-code the receive schedule would look as follows:

```
//Period that the radio has to wake up in ms
const int period = 1024;
//Synchronise tasks on all nodes using synchronise function from task synchronisation
int xLastWakeTime = synchronizeTask(period);
for ( ; ;)
{
        //Wait till next receive moment
        vTaskDelayUntil (&xLastWakeTime, period);
        //Turn on the radio
        vStartUpRadio(xRadio);
        //Receive data using the receive function from the radio driver
        vReceiveData(xRadio);
        //Turn the radio off
        vShutDownRadio(xRadio);
}
```

The sending of messages to the sink becomes completely asynchronous from receiving messages and is not scheduled in the network schedule. Sending messages to the radio happens at the frequency determined by the send frequency stated by the programmer in the send PSF.

One of the main advantages is that sending data is not constrained by a schedule. The receiving task will always have the highest priority due to the restrictions of task synchronisation so sending will not interfere with receiving. The drawback of this schedule is that collision detection and avoidance is completely delegated to the MAC services running on the AG2 and a programmer has no direct control over it with this schedule.

4.3.3 Schedule option three: TICOSS schedule for multi-hop

One of the schedules developed for extensive power reduction tests on nodes was V-Scheduling created by Antonio Ruzzelli from University College Dublin visiting

Philips Research. It is called TICOSS (TImezones COordinate Sleeping Scheduling) and was created for multi-hop networks. A network was divided in time zones (figure 4.5). Only adjacent timezones can communicate with each other and there are fixed slots for sending and receiving. The schedule works from one node (designated the Personal Area Network (PAN) coordinator) outwards and is set up so that if nodes in one zone are sending, the nodes in the zone one further away are listening. This is the first stage of the schedule. The second stage is the other way around. The nodes in the outermost zone can send and the nodes in the zone that is one closer to the PAN coordinator are listening and so back to the lowest zone and PAN coordinator. After that, the schedule has one local broadcast in which all radios have to be on. This is used for time synchronisation (see figure 4.6 for a representation of the schedule).

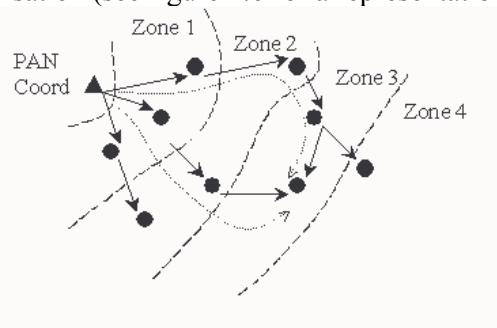

Figure 4.5: Time zone division in TICOSS

The power reduction that could be achieved by implementing this schedule was extensively tested and was 56.4% (see [1] for more information on the schedule and the testing). The versatility of the schedule is the main advantage. It can easily be used by a diverse variety of applications, both single and multi-hop. A drawback is actually the same as for the first schedule: it restricts the sending of information and therefore creates delays as to when the data is received by the gateway/sink (at least for an application which does not need inter-node communication). It is also more complex to implement.

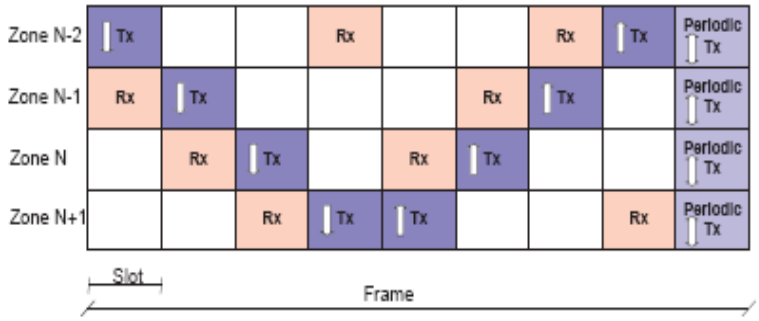

Figure 4.6: Representation of the schedule with period local broadcast

4.3.4 Chosen schedule for this project

The second schedule (section 4.3.2) is chosen (only receiving schedule while sending when ready) to fit well with the application that is tested in this project (the physiotherapy application as described in chapter one). The main reasons are its simplicity and the fact that data is immediately sent when it needs to (in accordance with the send frequency). The use of schedule three was rejected mainly because of its unnecessary complexity. The implementation that was developed for [1] was for a

different sensor node platform. The first schedule introduces delays while not adding functionality that was needed for the application.

4.3.5 Changing the network schedule

If the network schedule needs to undergo major changes (like supporting send slots, supporting multi hop networks, supporting complex network schemes, etc), the network schedule should be rewritten manually. It is possible to have configuration functions for an existing schedule which changes parameters on a limited scale. Another possibility is to have several implementations of schedules of which the programmer can chose one. These improvements fall outside the scope of this project. The lower levels provide the network schedule with the radio driver to access the radio and also the send and receive queues containing the data that needs to be sent and that has been received respectively. The send and receive functions of the radio driver will insert and extract the data into messages respectively. The difficulty of programming a new schedule depends heavily on the requirements for it and the extensiveness of the services it should be able to support (multi hop, reserved slots, etc). Since the two functions called by the PSFs using the radio (the send and receive PSFs) are from the network schedule layer (sendASAP and received), if the implementation of these functions is changed by for example implementing network schedule one (section 4.3.1) the PSFs do not need to be changed.

4.3.6 Effect of the network schedule on power consumption

Implementation of a network schedule will reduce the power consumption of the node because the radio will have sufficient time to sleep while it does not need to send or receive data. During receive slots however the radio always has to listen. Since these receive slots are fixed in time, the radio will not miss packets it needs to receive, as long as the sending radio also upholds the receive scheme as its sending slots. In this case the sink computer holds the sending radio during receive slots. The reduction in power by using a network schedule as envisioned in option two in combination with start up and shutdown of the radio should be approximately 90 percent with a send frequency of 25 Hz and combining four data samples in one packet. One packet contains nine integers per data sample and packet overhead. It takes the radio approximately 4 ms to send this giving it a power consumption of approximately 8 mW (with 25 packets per second). This is together with receiving a time synchronization packet every second approximately 10 percent of the total time the radio is powered on (see [2] for more information).

4.4 Sensor & Radio Drivers

Two types of drivers will be provided, one for the sensors and one for the radio. The set of functions provided for both sensor and radio is done with a device driver so only the underlying implementation of the functions has to be changed for different hardware configurations.

The sensors and the radio will both have a device driver, but these will be separate from each other. There is no generic device driver for both the sensors and the radio. The set of functions for the sensor handle will at least consist of the following (based on the peripheral handles of FreeRTOS):

- *xSensorCreate:* creates the sensor
- *vInitSensor*: initialize the sensor

- *vConfigSensor*: configures the sensor with correct values for ADC, start-up time, etc.
- *vStartUpSensor:* start the sensor (when it is shut down)
- *vShutDownSensor:* shut the sensor down to conserve energy
- *xSampleData:* retrieve a sample from the sensor

For the radio there is also a minimum set of functions that should be provided:
- *xRadioCreate:* create the radio
- *vInitRadio*(included in the create function): initialize the radio
- *vConfigRadio:* configures the radio with start-up time, queues to be used, etc.
- *vStartUpRadio:* start the radio (when it is shut down)
- *vShutDownRadio:* shut down the radio to conserve energy
- *vSendData:* write data to the radio to be sent
- *vReceiveData:* receive data from the radio

One of the advantages of this separate radio and sensor setup is that functions can be specifically tailored to be used with either the radio or sensor. Parameters for the functions can be specific for that function thus having only one purpose. A disadvantage of this separate API is that a program has two separate sets of functions instead of just one for all devices.

These specific functions were chosen because all actions that are necessary to access the device can be done with these two sets of six functions. They are derived from the physiotherapy application as analysed in [2] and the proposed API for the WASP project (for more information see [5]). The main difference with the WASP API is that the WASP API uses one set of functions for all device drivers (radio, sensors, flash, etc). The main advantage is that all device access is done in the same way, no matter which device is accessed. This main advantage also holds the main drawback and the reason it was not chosen in this project. Due to the generality of the drivers as envisioned in WASP, one function is used to perform many functions (start up, shutdown, initialisation, calibration, etc). Some of these functions are only for the sensors, some only for the radio and some for both devices. This would mean that the function itself has to distinguish which device is calling the function and also the different parameters that need to be conveyed to the function can be different. This would make the parameter list longer and more complicated. Such generality is not necessary for this project and therefore it was chosen to use separate drivers for radio and sensors.

The energy reduction that can be gained from these device drivers is mainly in the start up and shutdown functions. By offering the programmer device drivers, the use of these drivers will be more natural than if the programmer has to type in GPIO commands to start up and shutdown the device. It also offers the higher level PSFs a more fixed implementation of the devices, thus increasing portability of the system and PSFs.

4.5 Order of implementation

The implementation of these proposed improvements should be made in stages to allow for good testing and to allow a good control over the development. It will also help spot errors and faults early on. The sensor driver will be implemented first. This allows for all three sensors that are present in the blue boxes to be implemented using

a sensor driver and then tested to see if it works correctly. After this, the driver for the radio will be created. As with the sensor, the radio of the blue boxes will be implemented using the radio driver and then tested. This concludes the lowest abstraction layer as suggested in section 4.4.
The next implementation will be the Program Specification Functions. These will be tested by first separately testing each PSF and after that by implementing the stroke rehabilitation software in this format. The sampling schedule will be included in the *sample* PSF and thus implemented here.
The last abstraction layer, the network and sampling schedules, are implemented third. The sampling schedule is incorporated in the sample PSF and thus already implemented.

5. Implementation

The implementation order proposed in section 4.5 is followed during the implementation of the abstractions. Indeed, after the construction of the sensor and radio drivers, the PSFs were implemented for an easy testing of these functions without adhering to any network schedule. After this was tested, the network schedule was implemented and the direct calling of radio driver functions was substituted by network schedule functions with similar functionality. Before the sensor driver was implemented, two other parts of code were incorporated into the project, delay functions and functions for the AG2 radio. Timing is an important part of the implementation of the PSFs and the underlying abstractions. Therefore the delay functions mentioned in [3] (section 6.2) are incorporated and tested. After that, functions abstracting the use of the specific AG2 radio were incorporated into the project and tested. All queues mentioned in this project are the standard queues provided by FreeRTOS (Flex-OS is a port of FreeRTOS and the queues are directly ported from their FreeRTOS implementation). It has a “create” function to allocate the memory for the queue and to set it up and "send” and “receive” functions to put data in the queue and to take data out of the queue respectively.

5.1 Sensor Driver

The first implemented abstraction was the sensor driver, a generic template with which to create a sensor handle and API functions to control the sensor. The first section will outline the information and structure of the sensor handle. In the second section the implementation of the API functions that are offered by the sensor driver to manipulate the sensor are discussed and the third section deals with how the actual three sensors (accelerometer, magnetometer and gyroscope) are implemented in the main file. The last section discusses problems that were encountered during implementation of the sensor driver.

5.1.1 Structure of the sensor handle

The sensor handle that is created and used by the various functions is a pointer to a data structure containing information about the sensor. The information stored in the data structure is needed for the API functions that the sensor driver is offering to the programmer. This information includes the following:

- *bDirection:* Input (1) or output (0). In the case of sensors, the direction is always input.
- *bInterruptEnable:* Enabled (1) or disabled (0).
- *bPolarity:* Falling edge (1) or rising edge (0).
- *eGPIOLine:* The GPIO line that is connected to the sensor; is used for shutdown and start up commands.
- *iNumberOfChannels:* The number of channels a sensor needs to read from, for example 3, being x, y, z.
- *aADCChannels[sensorMax_Channels]:* The channels connected to the ADC with the order x, y, z, referenceV, etc.
- *eUsedADC:* The designation of the ADC to which the sensor is connected. Since all sensors in the blue boxes use an ADC and in the sample function the ADC is read, if a sensor would be connected without an ADC, the sensor driver should be changed.
- *xShutdownInactive:* Indicates if the sensor should shut down during inactivity. (this value can be set, but nothing is done with it in this project).

- *xShutdownBetween:* Indicates if the sensor should shut down between samples.
- *xStartupTime:* The start-up time of the sensor in milliseconds (rounded up) after it receives the start-up command. This value is the start-up time as noted in the datasheet of the particular sensor. It is used to wait the appropriate amount of time after starting to assure the sensor is started up properly.
- *xSampleFrequency:* The frequency at which this sensor has to take samples.
- *xSampleQueue:* A pointer to the queue associated with this sensor. The queue itself is a data structure with its own set of functions offered by the OS. The queue is created outside of the sensor driver (in the main file). Only the pointer to the data structure of a queue is associated to a sensor.

5.1.2 API functions of the sensor driver

The functions described in section 4.4 are implemented with the specified functionality. The implementation of the sensor and radio drivers is modelled on the implementation of the peripheral drivers of the Flex-OS environment. The *create* function of the sensor only allocates memory for the data structure and sets all information of the structure to NULL. Therefore the *create* function does not need parameters. After the creation of a sensor handle, the information is inserted into the structure via the *ConfigSensor* function. All other functions have the sensor handle as input and therefore have access to all the information contained in it. This limits the parameters that have to be given to a function and makes it easier to change the implementation without having to modify higher level function. Since queues are used to communicate data between PSFs, none of the functions has a return type. A pointer to the queue is found in the sensor driver structure as well, thus making it possible to access the sample data elsewhere. The functions of the sensor driver have the following parameters, return types and operation:

- *xSensorHandle xSensorCreate(void)*
  This function creates the sensor handle and returns it. The sensor handle is a pointer to a structure containing sensor information, like the used GPIO line, if it need to shutdown between samples, sample frequency, used ADC, and other information.
- *void vInitSensor(xSensorHandle xSensor)*
  The initialisation of the sensor is done with this function. It initializes the GPIO port that is used and shuts down the sensor (if shutdown between samples is desired) or makes sure the sensor is powered up (if shutdown is not desired for this sensor).
- *void vConfigSensor(xSensorHandle xSensor, int Direction, int InterruptEnable, int Polarity, portBASE_TYPE GPIOLine, int ADCChannels[sensorMax_Channels], portBASE_TYPE UsedADC,int NumberOfChannels, int StartupTime, int ShutdownBetween, int ShutdownInactive, int SampleFrequency, xQueueHandle SampleQueue)*
  The configuration function of the sensor is done here. It sets all values in the structure of the sensor.
- *void vStartUpSensor(xSensorHandle xSensor)*
  To start a sensor, this function is used. It uses the GPIO line of the sensor to send a start-up command to the senor.

- *void vShutDownSensor(xSensorHandle xSensor)*
  The shutdown function shuts down the sensor by using the GPIO line connected to the sensor and sending the off command (a 0 in this case).
- *void xSampleData(xSensorHandle xSensor, xQueueHandle xSampleQueue)*
  This function starts up the sensor, waits the amount of time needed for start up (if the shutdown mode of the sensor is used) and then takes a sensor reading on all specified channels (as specified in the structure representing the x,y and z values). These results are put into the queue that is passed to this function and from which the data can later be retrieved. The last action of this function is shutting down the sensor (if the shutdown mode is active).

As stated before, a sensor has a queue associated with it where the samples are stored for use by other PSFs. Queues are implemented in Flex-OS as data structures and have their own set of functions to create the queue, receive a value from the queue and send a value into the queue. When a value is received from the queue, it is deleted from the queue. During creation of the queue, the size of the queue and the type of values that will be stored in the queue has to be specified.

5.1.3 Implementation of the three sensors using the device driver

The physiotherapy application does not require on-node processing. Data can directly be put into the send queue of the radio which is called xAG2SendQueue. This queue is created before the three sensors via *xQueueCreate(60, (unsigned portBASE_TYPE) sizeof(DataSample))*. The DataSample type is an array of three integers (the three values a sensor samples, x, y, z) and memory is allocated for 60 elements in the queue (thus the sample frequency can be maximally 60/3 times higher than the send frequency assuming all three sensors sample at the same frequency). After the three sensor handles are created, they are added to the sensor lists which is a global variable. This makes them easy to access for the *compute* function if that would be necessary. As an example only the accelerometer is shown. The other two are analogous with only different initialisation parameters.

```
//Create Accelerometer Handle
xSensorHandle Accelerometer = xSensorCreate();
SensorADCChannels[0] = 5;
SensorADCChannels[1] = 4;
SensorADCChannels[2] = 6;
vConfigSensor(Accelerometer, 0xffffff, 0, 0, 0x1000, SensorADCChannels,
(GPIOSIGNAL__ADC1GVT | GPIOSIGNAL__FLASH),3, 2, 0, 0, 1,
xAG2SendQueue);
vInitSensor(Accelerometer);
```

//Create Magnetometer Handle

Analogous to accelerometer handle, but with some other parameters in the configuration

//Create Gyroscope Handle

Analogous to accelerometer handle, but with some other parameters in the configuration

//Add the sensors to the sensorlist for easy access

```
vSensorListAdd(Accelerometer);
vSensorListAdd(Magnetometer);
vSensorListAdd(Gyroscope);
```

The actual configuration of the sensors is done in the *vConfigSensor* function and for the accelerometer, the configuration looks as follows:

```
xSensor->bDirection = 0xffffff;
xSensor ->bInterruptEnable = 0;
xSensor ->bPolarity = 0;
xSensor ->eGPIOLine = 0x1000;
xSensor ->aADCChannels[0] = 5;
xSensor ->aADCChannels[1] = 4;
xSensor ->aADCChannels[2] = 6;
xSensor ->eUsedADC = ( GPIOSIGNAL__ADC1GVT | GPIOSIGNAL__FLASH);
xSensor ->xShutdownInactive = 0;
xSensor ->xShutdownBetween = 0;
xSensor ->xSampleFrequency = 1;
xSensor ->iNumberOfChannels = 3;
xSensor ->xStartupTime = 2;
xSensor ->eGPIOLine = 0x1000;
xSensor ->xSampleQueue = xAG2SendQueue;
```

5.1.4 Problems encountered during sensor driver implementation

The major problem encountered while testing the implementation of the radio driver was that the initialisation order done in the configuration function of the sensor mattered. When all values were set before calling the vInitSensor function, several values were set back to zero. No other negative effects seemed to transpire so the values that were set to zero were reset to the proper value after the initialisation function and if the specific value was not needed for the initialisation, it was set after this function had been called. This seemed to solve the problem but did obsolete the configuration function. It also meant that a lot of additional initialisation was done in the main file instead of in the appropriate configuration of the sensor. The problem appeared to be in the memory allocation of the structure and this was eventually solved and all initialisation was moved to the configuration function as shown above.

5.2 Radio driver

The radio driver was implemented after the sensor driver. It provides a generic way of accessing the radio by providing a radio handle to the programmer and offering API functions to access the radio.

5.2.1 Structure of the radio handle

The radio handle is a pointer to a data structure containing information on the radio. The information in the data structure is the following:

- *xShutdown:* Indicates if the radio should be shutdown.
- *xStartupTime:* The start-up time of the radio in milliseconds (rounded up) after receiving the start-up command. This value is the start-up time as noted in the datasheet of the particular radio. It is used to wait the appropriate amount of time after starting to assure the radio is started up properly.

- *xSendFrequency:* The frequency at which the radio needs to send the data samples.
- *xSendQueue:* A pointer to the queue containing the data to be sent. The queue itself is a data structure with its own set of functions offered by the OS. The queue is created outside of the radio driver. Only the pointer to the data structure of a queue is associated to a radio.
- *xReceiveQueue:* A pointer to the queue containing the data that is received by the radio. The queue is a data structure with associated functions. These are offered by the OS. The creation of the queue is not done in the radio driver and only the pointer to the data structure of a queue is associated to a radio.

5.2.2 API functions of the radio driver

For the radio, the functions stated in section 4.4 with its functionality are implemented. As with the sensor driver, the radio driver was modelled after the peripheral drivers of Flex-OS. The sensor and radio drivers were meant to be as identical as possible. The *create* function of the radio allocates the memory for the data structure of the radio and initialises all values. The correct values have to be added with the configuration function after the handle is created. The radio functions also all have the radio handle as input to keep it as generic as possible. The only exception is the *send* function. This also requires the sample period of the samples put into the message (deltatime) and the number of samples per packet (latency). The number of samples per measurement (for example 3: x, y, z) is needed to make sure that the values of one sensor are kept together when they are sent. The ReceiveData and SendData functions both use the locking mechanism of the OS to make sure the radio is not used simultaneously by the receive and send PSFs. The start-up and shutdown functions are used in these two functions and are included in the locking. This is done to prevent that for example after powering up the radio via the send PSF, the receive PSF pre-empts it and again starts up the sensor (which the application will note is started already and will not do this again). After the sensor is started and the data is received, the radio is shut down by the receive PSF. The pre-empted send PSF can continue assuming the radio is powered up and will send its data. To prevent these mistakes the start-up and shutdown are also done in the critical section. The radio driver functions have the following parameters, return types and operation:

- *xRadioHandle xRadioCreate(void)*
  Creating the radio handle is done with this function. It returns the pointer (radio handle) to a structure containing information on the radio.
- *void vInitRadio(xRadioHandle xRadio)*
  The initialisation of the radio is done with this function. It invokes the AG2 radio initialisation function with the correct parameters for baudrate, parity, etc.
- *void vConfigRadio(xRadioHandle xRadio, int StartupTime, int Shutdown, int SendFrequency, xQueueHandle Sendqueue, xQueueHandle ReceiveQueue)*
  Configuring the radio is done with this function. The parameters given in this function are set in the structure of the radio to which the radio handle is a pointer.
- *void vStartUpRadio(xRadioHandle xRadio)*
  This function is used to start the radio after it has been put in sleep mode.
- *void vShutDownRadio(xRadioHandle xRadio)*
  With this function, the radio is put in sleep mode.

- *void vSendData(xRadioHandle xRadio, int SamplesPerMeasurement, int DeltaTime, int Latency)*
  This function adds the data put in xSendQueue into (a) message(s) and adds timestamp, send frequency and sample frequency to the message. The message is then send with the AG2 send function. If the data in the queue contains more data than fits into one packet, as soon as the first packet is sent, the remaining data is put into the next packet and send and so on.
- *void vReceiveData(xRadioHandle xRadio)*
  To receive data, this function is used. Data that is received via the UART connection with the AG2 radio is put into the receive queue and can be retrieved by other functions like the compute function.

5.2.3 Actual radio implemented via radio driver

The AG2 radio was implemented via a radio driver in the same manner as the sensors were. The radio driver needs two queues, a send and a receive queue. The send queue is the one already discussed in section 5.1.3 and the receive queue is created via *xQueueCreate(60, (unsigned portBASE_TYPE) sizeof(char*)).* It has the same number of maximum elements as the send queue but uses a pointer to the character data type, the format in which a packet is offered. The queue can handle a maximum of 60 packets. This was chosen to allow applications that send data between nodes to be able to store for example 20 data samples for each sensor (if the other nodes for example send data samples per sensor to other nodes). This queue is not actually used since the *receive* function is not used in the physiotherapy application. In the physiotherapy application, the *receive* PSF is used to simulate the receiving of time synchronisation packets (since this is not implemented). Therefore the receive function will start up periodically to measure how often the radio is turned on more accurately.

```
//Radio Handle Creation
xRadioHandle AG2Radio = xRadioCreate();
vConfigRadio( AG2Radio, 2, 0, 1, xAG2SendQueue, xAG2ReceiveQueue);
```

In the configuration function of the radio, the following values are set in the data structure of the radio:

```
AG2Radio->xShutdown = 0;
AG2Radio->xStartupTime = 2;
AG2Radio->xSendFrequency = 1;
AG2Radio->xSendQueue = xAG2SendQueue;
AG2Radio->xReceiveQueue = xAG2ReceiveQueue;
```

5.3 Program Specification Functions

The program specification functions (PSFs) perform three functions: setting the values the programmer configures (frequency, shutdown, etc) in the appropriate handle, make sure only one task is created for sensors with the same sample frequency and if the task needs to be created, creating it. There are four PSFs with four corresponding task functions which are started by the appropriate PSF. The functionality of the PSFs is described in section 4.1. The sample PSF has five parameters. The first four are as stated in the specification (the sensor handle, the sampling frequency, the shutdown between taking samples and the shutdown during long inactivity) and the last one (the name of the sensor) is used to give the task that is

created by the PSF a unique name. The shutdown during longer inactivity (for example if the sensor is being attached or the application on the host is not started yet) is included in the PSF but not yet implemented. It is added in such a way that the API of the sample PSF does not need to be changed if this functionality is added afterwards. The compute PSF has as parameter the radio handle. A list of sensors is also available in the compute function but since this is a global list, it is not needed as a parameter. One of the main reasons for this is that only one parameter can be given to the task that is created and since the sensor list was already global, only the radio had to be given as a parameter. The sample and send functions in addition set the frequency and shutdown parameters in the data structure of the sensor handle.

5.3.1 Implementation of the Program PSFs

The PSFs are available to programmers in the main file of the Flex-OS project. Due to the nature of FreeRTOS it is advised to use the already created task to change the PSFs as needed and not directly from the main function. Furthermore, the three sensors that are present in the blue boxes and the AG2 radio (including their respective queues) are already created in this task. The following four PSFs are available:

- void vSample(xSensorHandle Sensor, int SampleFrequency, int ShutDownBetween, int ShutdownInactive, const signed portCHAR * const SensorName)
- void vCompute(xRadioHandle Radio)
- void vReceive(xRadioHandle Radio)
- void vSend(xRadioHandle Radio, int SendFrequency, int Shutdown)

The vSample PSF sets up a sensor at a specific sampling frequency and starts a task (if there are not already other sensors with the same sample frequency). The PSF checks if there is already a sensor that samples at the same frequency. If this is the case, no new task is created. If no sensor with the same frequency is already started, a task is created for the sensor (with idle priority + 3, one higher as the send task) and the frequency is added to an array (xSampleFrequencies). The sensor handle of the sensor that has to gather the sampling data, has to be specified with the name of the sensor (a string which is needed to give the created task a unique name). Furthermore, the programmer can specify the desired sampling frequency of the sensor and the shutdown parameters (shutdown between taking samples and shutdown during inactivity). The PSF, before creating the task, also checks whether the sampling frequency is low enough for the sensor to start up between taking samples (for example sampling at 25 Hz with a sensor with 40 ms start up time means it cannot shutdown between sampling; 1000/40 > 25 has to hold). If this condition is not met, the shutdown for that sensor is disabled. The *sample* PSF which creates the sample task looks as follows:

```
int i = 0;
int foundFrequency = 0;
while (xSampleFrequencies[i] != -1 && foundFrequency == 0)
{
	if ( xSampleFrequencies[i] == SampleFrequency)
	{
		foundFrequency = 1;
	}
	else
	{
		i++;
	}
}
if (foundFrequency == 0)
{
	xSampleFrequencies[i] = SampleFrequency;
}

SampledSensor->xShutdownInactive = ShutdownInactive;
SampledSensor->xShutdownBetween = ShutDownBetween;
SampledSensor->xSampleFrequency = SampleFrequency;

//Check dependency: 1/sampling frequency > Start-time sensor
if ( ((1000/SampledSensor->xSampleFrequency) <= SampledSensor->xStartupTime)
&& (SampledSensor->xShutdownBetween == 1) )
{
	//No time to power down, so do not shut the sensor down
	SampledSensor->xShutdownBetween = 0;
}

//Create the task only if the frequency was not yet found, so not to create multiple
//tasks for sensors with the same sampling frequency
if (foundFrequency == 0)
{
	xTaskCreate( vSampleTask, SensorName, configMINIMAL_STACK_SIZE,
	SampledSensor, tskIDLE_PRIORITY + 2, ( xTaskHandle * ) NULL );
}
```

The compute PSF (vCompute) provides the programmer with a task to do computations or calculations on the collected data. This PSF only creates the task called vComputeTask which is where the programmer can manually implement which computations this task should perform on the data (see section 5.3.2 for more information on the task). The compute task will receive the lowest priority of the tasks created by the PSFs (idle priority + 1). The vReceive PSF is used to receive data. It starts the receive task as which in turn calls the received function of the network schedule layer. The receive task is given the highest priority of all PSF created tasks to make sure it always pre-empts other tasks and can always receive messages in time (idle priority + 3 is its priority).

The PSF to send the data (vSend) that is gathered by the sensors or after computation is done with this function. The frequency at which to send the data and the use of the shutdown function can be set by the programmer. This information is added to the radio handle that is given in the PSF (here called SendingRadio). This handle is also given as a parameter to the task via the task creation function. The priority of the send task is chosen to be one lower than that of the sampling task(s). The send PSF is implemented as shown here:

```
SendingRadio->xShutdown = Shutdown;
SendingRadio->xSendFrequency = SendFrequency;
xTaskCreate( vSendTask, "RadioSend", configMINIMAL_STACK_SIZE,
SendingRadio, tskIDLE_PRIORITY + 1, ( xTaskHandle * ) NULL );
```

5.3.2 Tasks created by the PSFs

The four PSFs create tasks that are implemented beforehand. These tasks all run in a continuous loop. There are four tasks (one for each PSF): vSampleTask, vComputeTask, vReceiveTask and vSendTask.

The sample task uses the sample frequency to periodically take samples. Since sensors with equal sampling frequencies are sampled in one task, the list of sensors is used to check all sensors if they have the same sampling frequency as the sensor which is given as a parameter to the sample task (this is the sensor which created the task; see section 5.3.1 for more information). This task periodically starts up and then samples every sensor with the same frequency. The implementation of the sample task is shown below:

```
period = (1000/pxSensor->xSampleFrequency) - pxSensor->xStartupTime;
portTickType xLastWakeTime = xTaskGetTickCount();
for( ; ; )
{
        vTaskDelayUntil (&xLastWakeTime, period);
        ListOfSensors = xSensorListing();
        while (ListOfSensors != NULL)
        {
                if (ListOfSensors->pxSensor->xSampleFrequency ==
                pxSensor->xSampleFrequency)
                {
                        xSampleData(ListOfSensors->pxSensor,
                        ListOfSensors->pxSensor->xSampleQueue);
                }
                ListOfSensors = ListOfSensors->pxNextSensor;
        }
}
```

The compute task provides the programmer with access to the sensor handles and the radio handle. The programmer has to program the computations he wants to perform on the data manually. Access to the sensors is provided by the sensor list in the same manner it is used in the sample task. The radio handle is provided as a parameter of the task and thus is also available. The handles are available in the compute function so the sample queues and the send and receive queues can be accessed in this function

(depending on what computations need to be done, some or all of the data in these queues may be needed).

The receive task directly calls the received function of the network schedule (see section 5.4.2). The send task sets three values which need to be included in the message as defined for the physiotherapy application in the AG2 implementation. These three values are hard coded at this moment. To relate these to sample frequencies, a coupling has to be made between the sensors and the send task. This is not done here because sensors may have different sampling frequencies while still all data is being placed into one send queue. Another reason is that if computations have to be done on the data, the sampling frequency may not have any meaning anymore if sensors have different sampling frequencies. The SamplesPerMeasurement variable is needed in the send function of the radio driver to know how many samples need to be kept together as one measurement (for example one measurement consists of a sample from the accelerometer, magnetometer and gyroscope taken at the same moment in time). The send task that is created by the *send* PSF is implemented as follows:

```
int SamplesPerMeasurement = 3; //samples per measurement (1 sample is 3 integers)
int DeltaTime = 100; //sample period in ms
int Latency = 4; //samples per packet
period = (1000/pxRadio->xSendFrequency) - pxRadio->xStartupTime;
portTickType xLastWakeTime = xTaskGetTickCount();
for( ; ; )
{
        vTaskDelayUntil (&xLastWakeTime, period);
        vSendASAP(pxRadio, SamplesPerMeasurement, DeltaTime, Latency);
}
```

5.4 Network Schedule

The network schedule consists of two parts, the sending of messages and the receiving of messages. As stated in section 4.3.2 the two parts work asynchronously from each other. The receiving part is periodic with a period of 1024 OS ticks (1 tick is 1 ms). The sending of messages is controlled by the send frequency and not according to a specifically created schedule. The API of the network schedule consists of two functions: sendASAP() and received() for sending and receiving respectively. These are the functions that are used in the PSFs and not the send and receive functions of the radio driver.

5.4.1 Send schedule

The sendASAP function of the network schedule actually only calls the send function of the radio driver. The reason it is called in the send PSF instead of directly calling the send function of the radio driver is to keep the schedule abstraction layer in place atop of the device driver layer. This also makes it easier for programmers to change either the radio driver without changing the network schedule and PSF layers or the network schedule layer without changing the PSF layer.

5.4.2 Receive schedule

The implementation of the receiving part of the network schedule uses the method of implementing task synchronisation as described in [4]. It first sets the last wake time by calling the synchronizeTask function of task synchronisation and then using

vTaskDelayUntil to delay from the last wake time a given period (which must be a power of two, in this case 1024). Due to the problems with time synchronisation, the local time was set by calling the xTaskGetTickCount (the OS timer ticks where one tick is equivalent to approximately 1 ms) from the operating system and delaying from there. This avoided the problems of having to test with two nodes while still simulating the receiving of information every 1024 ticks (approximately 1 second in real time). After waiting for the next period, the radio is powered up and after this, the data is received using the receive function of the radio driver. When this is finished, the radio is shut down and it waits for the next receive period. The following part of code illustrates the implementation described above:

```
const portTickType period = 1024;
portTickType xLastWakeTime = xTaskGetTickCount();
for ( ; ; )
{
        //Wait till next receive slot
        vTaskDelayUntil (&xLastWakeTime, period);
        vReceiveData(xRadio);
}
```

5.5 Physiotherapy application

The specifications for the physiotherapy application are to sample all three sensors at a frequency of 100 MHz and send this data out at a frequency of 25 MHz. No messages need to be received by the node except for the time synchronisation messages and no computations are done on the data gathered with the sensors. All three sensors are used (accelerometer, magnetometer and gyroscopes) and only the magnetometer needs to shut down between taking samples. The accelerometer can also shut down but this would distort the samples so is not done. The start-up time for the gyroscope is too long to shutdown with a frequency of 100 MHz and is therefore also not set to shut down. If it would be set to shutdown, the start up time would be too long and the shutdown for this sensor would be automatically disabled. The radio is the AG2 radio and only needs to be powered up if something needs to be sent or a time synchronisation message needs to be received. The time synchronization messages are handled directly by the AG2 radio and not by the CoolFlux DSP. Due to problems with the time synchronization for the blue boxes it was chosen to simulate the receiving of the time synchronization messages every second to allow for more accurate measurements instead of using time synchronization. It is assumed that the handles for the sensors and the radio (which is hardware that is fixed on the blue boxes) are already available with the following names: Accelerometer, Magnetometer, Gyroscope and AG2Radio (see section 5.1.3 for information on how they are created). The invocation of the PSFs for the physiotherapy application is done as follows:

```
vSample(Accelerometer, 100, 0, 0, ( const signed portCHAR * const ) “Accelerometer");
vSample(Magnetometer, 100, 1, 0, ( const signed portCHAR * const ) "Magnetometer");
vSample(Gyroscope, 100, 0, 0, ( const signed portCHAR * const ) "Gyroscope");
vSend(AG2Radio, 25, 1);
vReceive(AG2Radio);
```

5.6 Testing

To make these proposed changes and frameworks, material is needed. The first is the version of Flex-OS as it was in December of 2007 when this project started. This allows for the integration of the abstraction layers into the latest version (at that moment) of the operating system. The software running on the AG2 is needed to allow the radio to be turned on and off, but this functionality is not yet implemented. For compiling Flex-OS and the application, Target's ChessDE compiler is needed. The last material that is needed is a blue box to test the software abstractions and measure the energy that is used by the node.

The testing of the abstraction layers is done during implementation of the different layers. As each layer is implemented, it is tested to spot and correct errors and faults as early as possible. This also helps to find errors in the design and therefore makes it easier to correct it. To control the evaluation and validation of the implemented abstraction layers, print statements over UART are used. With HyperTerminal, these statements could be printed on the screen.

5.6.1 Testing the device drivers

The sensor driver is the first sub layer that is implemented. It is tested by first adding print statements to the code to check the initialisation of the driver. The accelerometer is implemented first. After the initialisation is tested, the queue is tested to see if all values are put in as they should. First the value is printed to the screen and then put in the queue. They are then taken out and again printed. These two values are then manually compared.

After the implementation of the radio driver and the creation of the AG2 radio handle, the radio is first tested by comparing the values that arrive at the host computer via the radio with the values that are put into the message. After that is tested, the values are put in the send queue and taken out by the radio driver and sent. This way the workings of the queue can be tested as well. It is done the same way by sending to the sink and manually checking if the values are correct with the ones put in. The combination of sensor and radio should be tested next by letting one sensor put data into the send queue and let the radio send it to the sink to be checked manually again.

5.6.2 Testing the Program Specification Functions

The PSFs are tested one by one after their implementation. The sample task is the first. It can use the implementation of the sensor driver and the print functions used to test this. The sample PSF will first be tested if it starts up and if the data is indeed sampled (by printing the sampled values to the screen as for the sensor driver). After this, the merging of calls to the sample PSF with the same sample frequency is tested. By printing to the screen the tasks that are started and the sensors in each task, it can be determined if they are grouped correctly.

The compute PSF is only tested to see if the radio and sensor handles are available in the function (by printing information from both on the screen) and if it starts up the compute task when it is called.

The testing of the send PSF is done by first checking if the task is started and then if the data is sent via the radio (as it is tested for the send function of the radio driver).

The last function to test is the receive PSF. This function actually only starts the receive task of the network schedule. Therefore nothing extra has to be tested for this specific function.

5.6.3 Testing the network schedule layer

The last layer to be implemented and tested is the schedule layer. Since the sampling schedule will be implemented in the sample PSF, it is already tested. The send part of the network schedule is also already tested when the send function of the radio driver was tested (sending does not need to adhere to a schedule). The receiving part of the schedule builds on the task synchronisation to synchronise the receiving part of the node with the sending part of the sink. Since testing has to be done with one node, it has to be determined if the task synchronisation can actually work with only one node and then test if packets are actually received by the node by printing to the screen as soon as a message is received on the node. The task synchronisation uses a function to set the current local time and synchronise this over all nodes. This time is then used to synchronise the tasks. The main problem is that this value, as well as the local time value stored by time synchronisation, is not updated if time synchronisation is not running or if there is no second node to synchronise with. The value will stay zero. One solution which will allow for testing is to use the timer tick count. This is what was actually done, since time synchronisation was not working in the OS for the blue boxes. Due to this, the task synchronization used in the receive schedule could not be tested. The receive function was used to simulate the receiving of a time synchronization packet without being synchronized with other nodes. This was done to have more accurate measurements for the power consumption of the radio.

5.6.4 Testing the whole system and taking measurements

After this, the PSFs should be called with their appropriate frequencies and shutdown parameters (accelerometer and magnetometer shut down and sample at 100 Hz, gyroscope does not shut down and samples at 100 Hz, the radio sends information at 25 Hz and shuts down and receives according to the schedule with intervals of 1024 timer ticks). This will generate three tasks: one for the three sensors combined, one for sending and one for receiving. To measure how long the sensors and the radio are on and off respectively, the current time is marked (at shutdown or start up) and subtracted from the previous change in state (start up after shutdown and shutdown after start up). This time is then added to the total time the radio has been on or off. This is done for all three sensors and the radio and regularly printed out to the screen.

5.7 Measurement Results

As described in section 5.6, the three sensors and the radio were measured as to how long they were on and how long they were off. These were periodically printed to the screen. For the sensors that were shut down (the accelerometer and the magnetometer) the percentage of time they were powered up was almost the same. The gyroscope was always on due to the long start up time and when it was set to shutdown in the PSF, it is automatically detected that this is not possible with a sampling frequency of 100 Hz and the shutdown is turned off. In the eventual application, the accelerometer will not be turned off between samples due to aliasing in the measurements. As stated in section 5.4, the time is measured in OS ticks and not in milliseconds. Since only the percentage of the time a sensor is on is needed, this will give the same results. From the percentage that the device is on, the power consumption per second can be calculated using the datasheets of the various devices ([6], [7], [8], [9]). The percentage of the time the device is on multiplied by the power consumption per second gives the power consumption per second that the device is on and vice versa. Table 5.1 lists the percentage of the total time each of the devices is powered up at

different frequencies. The measurements where done with all devices running at the same frequency (so if the accelerometer samples at 1 Hz, the magnetometer and gyroscope also sample at 1 Hz and the radio sends data at 1 Hz). This was done to ensure all frequencies were measured in the same manner. During all tests, the receive PSF was periodically powering up the radio to simulate the receiving of time synchronization packets. This was done at 1024 OS timer tick intervals (1 OS tick is 1 ms, see section 5.4.2 for an explanation why 1024 ticks were chosen).

Frequency (Hz)	Accelerometer	Magnetometer	Gyroscope	Radio
1	0.2%	0.1%	4.0%	1.1%
5	1.0%	0.5%	20.2%	1.7%
10	2.0%	1.0%	40.8%	3.1%
15	3.1%	1.6%	62.5%	4.7%
25	7.9%	2.7%	100.0%	8.6%
40	13.0%	4.6%	100.0%	13.5%
50	16.6%	5.6%	100.0%	16.9%
75	27.2%	9.1%	100.0%	27.4%
100	37.5%	12.5%	100.0%	37.5%
150	50.0%	25.0%	100.0%	75.0%
200	53.3%	33.3%	100.0%	99.9%
300	53.3%	33.3%	100.0%	99.9%

Table 5.1: The percentage of total time the devices are powered up at different frequencies

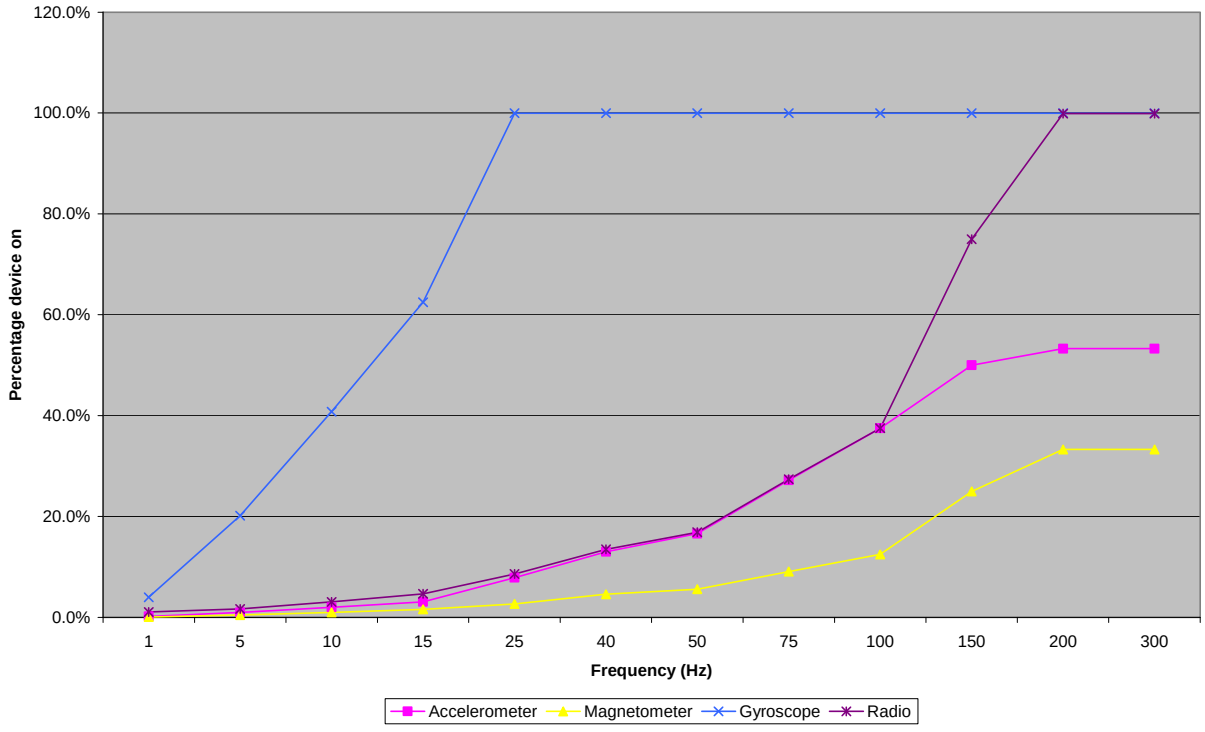

Figure 5.1: The percentage of the total time the four devices are powered on in contrast to the frequency of sampling/sending

The measurements from table 5.1 are plotted in figure 5.1. It shows clearly that the difference in the power down percentage of the three sensors is closely related to the start-up time of the sensors. The gyroscope has a start-up time of 40 OS ticks (40ms),

the start-up time of the magnetometer is 1 OS tick and of the accelerometer it is 2 OS ticks. This explains the differing graphs of the three sensors, showing that the magnetometer is powered on the least of all three. The jump that can be seen from 15 to 25 Hz in the graph of the gyroscope is due to this start-up time of 40ms. Sampling at 25 Hz is not enough for the sensor to start up (40 ms is not smaller than 1/25). Sampling at 24 Hz would give the sensor enough time to complete its start up and therefore is the highest sampling frequency of the gyroscope which reduces the power consumption for that sensor.

The specific sampling frequencies of the physiotherapy application (100Hz for the sensors and 25Hz for sending) give the results shown in table 5.2. This is run as a separate test result to eliminate possible overhead due to the higher sending frequency in the original results as shown in table 5.1. In section 5.5 the accelerometer and gyroscope are noted as not being shut down, but to test the amount of time the accelerometer would be powered up the shutdown parameter in the sample PSF of the accelerometer is set to 1 for this test. As can be seen from table 5.2, the accelerometer is shut down longer in this configuration than the one with a 100 Hz send frequency as depicted in table 5.2. A possible reason for this is that due to the higher send frequency, the receive function pre-empts this sending more and this delays the sampling of the accelerometer. This can happen because sometimes the radio begins its start-up before the accelerometer is shut down but after sampling its data.

Frequency (Hz)	Accelerometer	Magnetometer	Gyroscope	Radio
100	25.0%	12.5%	100.0%	--
25	---	--	--	8.6%

Table 5.2: The percentage of total time the devices are powered up in the physiotherapy setup

Figure 5.2 shows the periods of time when the four devices (accelerometer, magnetometer, gyroscope and radio) are powered up. The execution shown here samples at 1 Hz and also sends data at 1 Hz. The magnetometer is the first sensor to be sampled, followed by the accelerometer and after that the gyroscope. This is not a necessary order although sampling the gyroscope last helps for the frequencies at which the gyroscope does need to power up (frequencies below 25 Hz). This is due to the longer start up time of the gyroscope which will delay the sample taking of the other sensors as well if it was done before other sensors. Therefore sampling the gyroscope last will minimize the effect of its start up time on the other sensors. The time the radio is on to receive data is not shown in this graph since it is asynchronous of the sampling and sending procedure and only takes place once every second. Since it runs in a task with the highest priority, it will always pre-empt the other tasks if necessary. The time axis shown in figure 5.2 is relative to the start of a period (since the sampling and sending frequencies are the same this period is also the same).

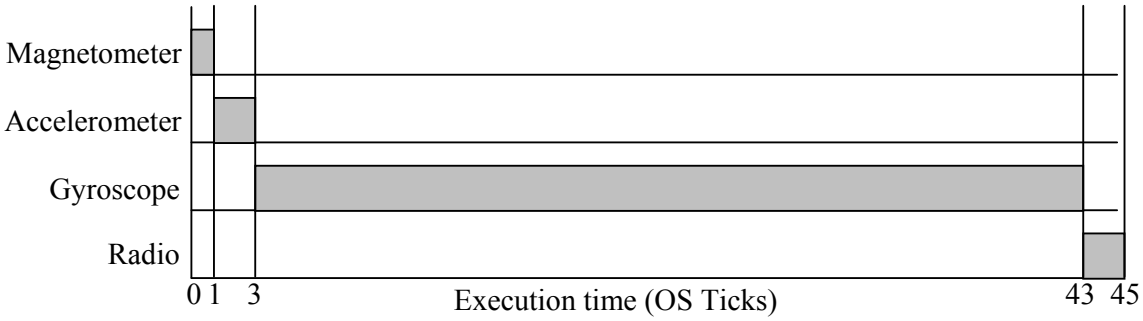

Figure 5.2: Periods of the execution time the devices are powered up (relative to the start of a period)

The memory usage of the application including all abstraction layers is approximately 9942 bytes. The memory of the CoolFlux DSP consists of two times 64 kWord of data memory and one times 64 kWord of program memory. One Word consists of 24 bits. The main components of the software abstractions are individually measured to see which parts have the biggest memory usage. This is shown in table 5.3.

	Number of Instances	Memory Usage per instance (bytes)
Send queue	1	618
Receive queue	1	258
Sensor driver structure	3	192
Radio driver structure	1	39
Tasks	5	1269

Table 5.3: Memory usage of the main components of the software abstraction layers

6. Future Work

This chapter will cover the possible extensions that can be made to these abstractions to make them more usable and further decrease the time to develop an application. The future changes can use the already created abstractions as a basis and would not require a redesign of the abstraction layers. It merely extends them.

6.1 PSFs from flash memory without recompiling code

One of the major drawbacks of programming and using the Flex-OS code on nodes is that for every change, the code has to be recompiled. This is especially noticeable by the developers of the physiotherapy application who are often asked to implement small changes for others using their software but without access to the compiler. Now that the PSFs became the highest level of programming, the number of parameters that the programmer needs to/can set is greatly reduced. The way a programmer would construct an application with the PSFs is by stating which PSFs should be evoked and with what parameters. If this information is stored in the flash memory of the node, the code would not need to be recompiled, but can read the information from its flash memory. One problem with the current implementation of the compute PSF is that it needs to be written by hand, thus requiring a compiler to change this function. A possible solution for this is presented in section 6.3. Another main advantage (besides not recompiling) is that a lot of dependencies (like the sample frequency larger or equal to the send frequency) can be checked beforehand by the tool used to upload and create the code/data that needs to be in the flash memory. This will make the life of the programmer easier since error messages can be generated on the PC instead of from the sensor node.

6.2 Graphical User Interface to create application with PSFs

After the possible improvement from section 6.1 is implemented, a graphical user interface (GUI) can be constructed on top of this. This will facilitate the programmer to write his application on a graphical level by drawing the PSFs as blocks connecting them with lines representing queues. Figure 6.1 is an indication of how such a GUI could look like, with the programmer being able to set the parameters for the PSFs as well.

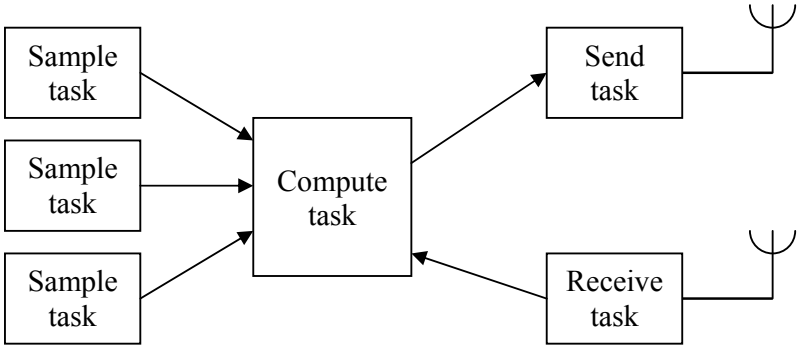

Figure 6.1: Possible GUI for PSFs

The programmer then also needs to set the different parameters of the required functions which can be accomplished with simple dialog boxes containing the parameters that can then be set by the programmer. The tool should then automatically convert this to the format developed for the future addition of section 6.1 and uploaded to the node. By solving the issue of the compute function (see section 6.3) clicking on the compute task in such a GUI could open a new field with

in- and output queues where the different compute functions can be selected and put together.

6.3 Compute function with a PSF-like block structure

To alleviate the problem of having the programmer completely write the compute function himself, a structure can be created in which pre-made operations can be linked together in a dataflow fashion. The queues from the sample tasks and the receive task are seen as input for the entire compute task and the queue from the send task is its output. Since using queues takes time for every access and the possible number of operations that has to be done in the compute task, using queues inside this task is not possible. Therefore performing an operation on the data avoids this problem. The incoming data is consumed and the desired data is produced, not necessarily having the same type. The incoming type needs to match the desired input of the function, so if for example the output of a function is needed as input for the next function, it needs to be checked if the output of the first type matches the input type of the second. A small example in pseudo-code depicted graphically in figure 6.2 would look as follows:
AverageSamples(AddSamples(FilterSamples(SampleQueue 1), FilterSamples(SampleQueue 2))).

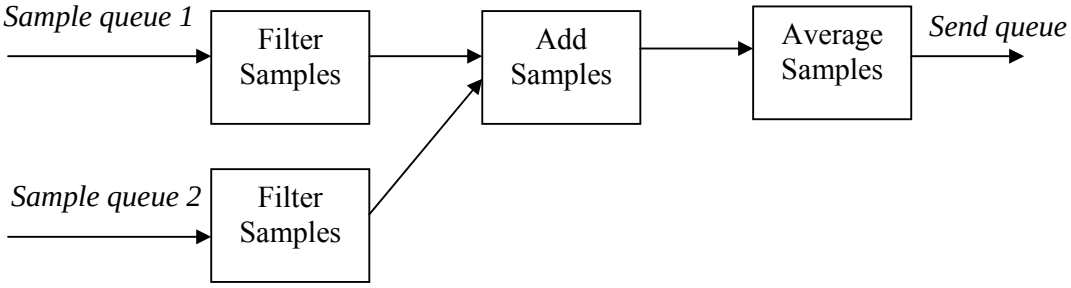

Figure 6.2: Graphical representation of a block-like compute PSF

There are several important issues that need to be addressed if this future addition has to be implemented. First of all, the incoming queues store integer values, so for a specific sample three integers (or more, depending on the sensors) needs to be taken out and given to the first function. This can not be done inside the function since this would mean that certain functions (the outer ones connected to the queues) have a queue as input and certain functions have only one sample (three integer values) as input. Since they need to be interchangeable, it would be a good idea to have a "split" block/function directly after the queue that offer only one sample value at a time. The same holds for the send queue at the end: have a separate "merge" block/function which puts the sample value in the queue for sending.
A second issue deals with how to check whether the functions can be linked or not. If this addition is implemented in such a way that constructing this structure of blocks without the need to recompile (as discussed in section 6.1), then the tool used to create these (either graphical or text-based) can check whether these dependencies hold.

6.4 Implementation of schedule and start up and shutdown on AG2

Currently starting and shutting down the radio is simulated in the code. This is due to the fact that these commands are not implemented in the code that runs on the AG2. This is a separate code from the one running on the CoolFlux DSP where the

operating system is running. Information is exchanged between these two processors via UART and send and receive commands are also done via this UART connection. There is however currently no implementation of a command that can be given to place the radio in sleep mode. This should later on be implemented to give the full functionality and power reductions from starting and shutting down the radio.
The second issue that still has to be implemented on the AG2 is the sending of time synchronisation packets between nodes. This is done in the code running on the AG2 and thus not controlled by the CoolFlux DSP or the operating system. Therefore it will not adhere to a schedule. This schedule should be implemented on the AG2 or, for a more generic approach, should be made configurable by the CoolFlux DSP. This would avoid having to change the code on the AG2 again when the schedule changes.

6.5 Implementation of shutdown during longer periods of inactivity

During the operation of a wireless sensor node, circumstances and surroundings may change depending on where the node is used. For nodes of body sensor networks for example in the physiotherapy application, the time the user is attaching the nodes to the body or the user is starting the application on the host computer, the node does not need to take samples or send information to the host. This is an improvement which later could be made by having the application on the host send a start signal constantly when it is ready to start. The nodes can then periodically (for example once every 10 ms) start the radio and listen for this start message from the host. Before this message is received, all sensors can be shut down until the application is actually started.

6.6 Scalable sampling frequencies for sensors

Related to these changing surroundings of a sensor node, the sampling frequency could also be automatically changed depending on the activity the sensor is monitoring. For example, if the sensor constantly reads the same value during its sampling, it can lower its sampling frequency and therefore save more energy. When the values it samples then start fluctuating more, the sampling frequency can be increased again. This improvement is not suited for every application. Especially sensitive monitoring (for example heart rate of patients) which are inherently stable will not work with this improvement.

7. Conclusions

The test results described in section 5.7 indicate that the power consumption of the blue boxes can indeed be reduced. The amount of power that could be saved as stated in the problem formulation was split into two parts. The first part was the reduction of power consumption during the use of the sensors and the second part the reduction of power consumption during the use of the radio.

Generally speaking, without looking at the specific sampling frequency of the physiotherapy application, the amount of time the three sensors need to be powered up is reduced. The amount of time the sensor is on depends on the frequency at which the sensor needs to sample. Comparing the three sensors shows that the difference is all in the start up time of the sensor. The sensor sampling time is negligible and the dominant factor in the amount of time the devices are powered is the start up time. The magnetometer has the biggest reduction in the amount of time it is powered. Adding this to the large power consumption of the magnetometer (almost 60 mJ/s) the power reduction of the entire sensor node is already drastically reduced. For the frequencies below 25 Hz the gyroscope is also able to power down. Due to its large power consumption (90 mJ/s), shutting it down for even a small amount of time already has an effect. As can be seen from figure 5.2 the highest sample frequency of the three sensors is dependent on the start up time of the three sensors. In the case of the three sensors in the blue boxes this would be 43 ms (1 ms for the magnetometer, 2 ms for the accelerometer and 40 ms for the gyroscope). With a total start up time of 43 ms the maximum sampling frequency at which all three sensors can be shut down is 23 Hz. By starting all sensors at the same time, this can be reduced to the length of the longest start up time (40 ms in this case of the gyroscope). This would require the start-up and shutdown function currently in the sample function of the sensor driver to be moved to the PSF level. This is an improvement which is not yet implemented.

The second part of the power consumption, the power consumption of the radio, has also decreased. The amount of time the radio is powered up scales with the frequency at which it needs to send data. When the frequency becomes larger than 100 Hz the accumulated power-on time of the radio increases significantly per period. The reduction of power for the radio is not completely the same as with the sensors since sensors are completely turned off between taking samples, while the radio goes into power saving mode which still uses energy (approximately 0.7 percent of the sending energy cost in the highest power down mode).

The overall power reduction in the specific case of the physiotherapy application is mainly found in the shutting down of the magnetometer and radio. The magnetometer is shut down 87.5% and the radio 91.4% (see table 5.2). The accelerometer is also shut down 75.0% of the time but the power consumption is not as high as for the other two devices. The gyroscope does not have the time to shut down in the physiotherapy application and still remains a large power drain. The sampling and sending frequencies used in the physiotherapy application (100 Hz for sampling the three sensors and 25 Hz for sending data) can still be achieved with the software abstraction layers without compromising the timing of the application. The power consumption of the radio would thus become approximately 1 mW (the original power consumption was approximately 88 mW) and the power consumption of the magnetometer would be almost 0.8 mW (with an original power consumption of 59.4 mW). The power consumption of the node would be reduced by approximately 42% meaning it would

be able to operate approximately 2.5 times as long as originally (approximately 12 hours) (see [2] for more information about the power consumption of the different devices and components of the node not included in this report).

The other area of focus in the problem formulation dealt with decreasing the time to develop applications on the blue boxes by increasing the usability of the software. By increasing the abstractions in the operating system environment, the programmer is able to program the nodes with only four functions. To determine how programmers of the nodes experience this, further tests with programmers are needed. The future improvements which are mentioned in sections 6.1, 6.2 and 6.3 will increase the usability of the software even further, especially the graphical interface.

8. References

[1] Timezones sleep scheduling over IEEE802.15.4 in multihop environment - Integrating energy saving scheduling and routing; A.G. Ruzelli, Philips Research Eindhoven, 2006

[2] Internship Report: Reducing Power Usage in Wireless Sensor Nodes; T.F.P. Paffen, Philips Research Eindhoven, 2007

[3] FreeRTOS - A Real Time Kernel for Wireless Sensor Network Platforms; Julien Catalano, Philips Research Eindhoven, 2008

[4] Master Thesis: Distributed Task Synchronisation in Wireless Sensor Networks; Marc Aoun, Department of Wireless Networks RWTH Aachen

[5] WASP API proposal; Ramon Serna Oliver,Gerhard Fohler (TUKL), Michael Hauspie,Gilles Grimaud,Sylvain Buisine (INRIA Lille), Carlo Brandolese(CEFRIEL); Information Society Technologies 2007

[6] Datasheet ±300°/s Single Chip Yaw Rate Gyro with Signal Conditioning ADXRS300. Analog Devices, Inc. March 2004.

[7] Datasheet CC2430: A True System-on-Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee®. Chipcon Products from Texas Instruments

[8] Datasheet Integrated Compass Sensor HMC6052. Honeywell International Inc.

[9] Datasheet KXM52 Series Accelerometers and Inclinometers. Kionix Inc. September 9, 2005.

[10] CoolFlux DSP, the embedded ultra low power C-programmable DSP core. Philips Applied Technologies. September 2005.

[11] Leaflet π-node: Wireless tracking system, the easy way to capture motion with high accuracy. Philips Research. May 2007.

[12] FreeRTOS: A real time kernel for wireless sensor network platforms. Julien Catalano, Philips Research Eindhoven. January 2008.