ZACHODNIOPOMORSKI UNIWERSYTET
TECHNOLOGICZNY W SZCZECINIE

Wydział
Informatyki

PRACA DYPLOMOWA

Communication algorithms and principles for a prototype of a wireless mesh network

Autor:
Sergiusz Urbaniak

Opiekun pracy:
dr inż. Remigiusz Olejnik

Szczecin, 2011

Oświadczenie

Oświadczam, że przedkładaną prac magisterską/inżynierską kończącą studia napisałem samodzielnie. Oznacza to, że przy pisaniu pracy poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem rozprawy lub jej części od innych osób. Potwierdzam też zgodność wersji papierowej i elektronicznej złożonej pracy. Mam świadomość, że poświadczenie nieprawdy będzie w tym przypadku skutkowało cofnięciem decyzji o wydaniu dyplomu.

Sergiusz Urbaniak

Contents

List of Figures

List of Tables

Chapter 1

Introduction

cel tesa

1.1. Research overview

Chapter 2

Evaluation

2.1. Existing solution

2.2. Assumptions

2.3. Requirements

Chapter 3

Hardware Design

3.1. RAM

- Harvard architecture
- RAM bus
- Latch

3.2. USB Serial Device

3.3. RFM12B Radio

3.4. Keyboard

Chapter 4

Software Modules

4.1. UART

4.2. SPI

4.3. Watchdog

4.4. Timer

4.5. Shell

4.6. Network Stack

4.7. RFM12 Driver

Chapter 5

Software Algorithms

5.1. Module orchestration

Designing a software system that executes on embedded micro-controllers implies a lot of challenges when many software modules are involved and complexity grows. The conceptually defined modules must be somehow implemented. If the micro-controller lacks an operating system then there is no possibility of using provided abstractions and APIs for module orchestration and execution. Another challenge are limited hardware resources which prevent the deployment of many existing operating system kernels. Basically there are two types of execution models which can be implemented in micro-controllers:

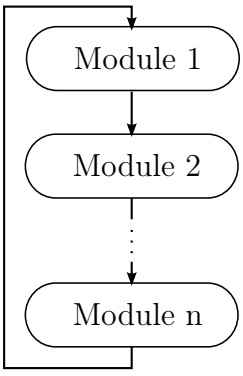

Figure 5.1. Sequential execution model

1. **Sequential execution model:** This type sequentially executes all modules

inside an infinite main loop starting from the first module until the last one. Once the last module ends the execution starts again from the first module.

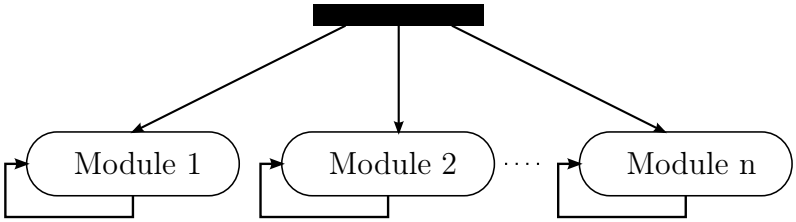

Figure 5.2. Concurrent execution model

2. **Concurrent execution model:** This type executes modules concurrently. Instead of having an infinite main loop that iterates sequentially over all modules the main function only initializes and launches concurrent modules.

5.1.1. Sequential execution

This model does not necessarily needs operating system support or frameworks. It can be simply implemented as a sequence of function calls inside an infinite loop as shown in algorithm 1.

Algorithm 1 Sequential model algorithm

```
while true do
    module_1
    module_2
    ...
    module_n
end while
```

There is one challenge that comes with this type of execution model. That is that only one module can execute at a time due to its sequential nature. If a module i.e. waits for an external resource to provide data it must not block the execution of the main loop until the external resources becomes ready. This would prevent the execution of the other modules. The classic solution to this problem is the introduction of states in modules. Module states can be implemented as classical Finite State Machines ([1]).

If we take the example from above about waiting for external resources a finite state machine for modules can be modeled as shown in figure 5.3.

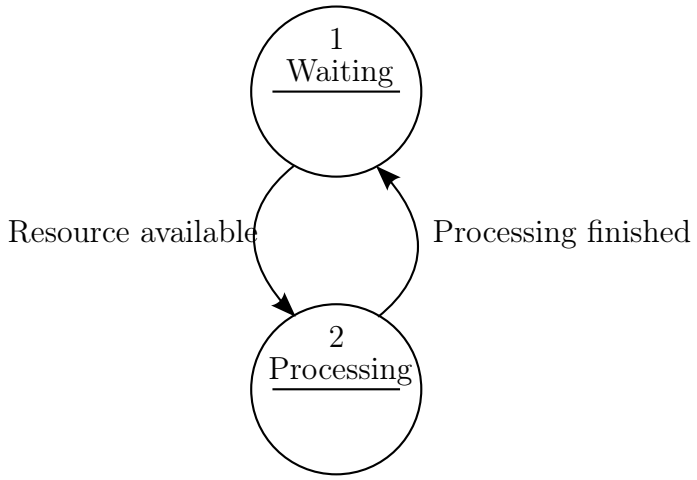

Figure 5.3. State Machine for a module

State machine models can be implemented using `if` or `case` statements which is shown in algorithm 2. The nice side effect of a state machine based implementation is the non-blocking nature of the module execution. Take for instance the execution of state 1 "Waiting" as shown in figure 5.3. The CPU only needs to execute as many instructions as are necessary to check if the awaited resource is available. If the resource is not available the execution returns to the main loop and the next module (together with its state machine) is being executed.

Algorithm 2 State machine algorithm

```
if state is WAITING then
    if resource is available then
        set state to PROCESSING
    else
        exit module
    end if
else if state is PROCESSING then
    process data
    set state to WAITING
end if
```

This implementation emulates a concurrent execution of modules. The context switch between module executions is being done by the modules themselves (using self-interruption) and no external scheduler is involved. This form of concurrent behavior can therefore be described as a non-preemptive or cooperative multi-tasking between modules. The predecessor thesis [2] implementation heavily used the described state machine algorithm although the model theory behind the implementation was not being mentioned in the thesis. Listing 5.1 shows the main function of the predecessor implementation.

Listing 5.1. main loop routine (see [2])

```
382 while(0x01)
383 {
384         if(uartInterrupt == ON) // got a character from RS232
385 +---- 44 lines:
429
430
431         // --- RECEIVE A DATAGRAM ---
432
433         else if((datagramReceived = datagramReceive(...))
                        && netState > 0)
434 +----182 lines:
616
617
618         else if(helloTime) // prepare periodic Hello message
619 +---- 19 lines:
638
639
640
641         // --- SEND A DATAGRAM ---
642
```

```
643          if(datagramReady && netState > 0)
644 +----  8 lines:
652
653 }
```

A couple of problems arise from the existing implementation. First of all listing 5.1 reveals the following modules:

- UART Module
- Datagram Receiver Module
- Hello Message Sender Module
- Datagram Sender Module

Which module is being executed depends on the state of the main module being represented by the main function. The state of the main module on the other hand depends directly from the state of the submodules. The main module therefore acts more like a controller of the submodules and takes away the responsibility of the submodule's state management. Furthermore the main function is very long and complex (271 lines of code). Therefore the following goals were defined by the author:

- Clear separation of responsibilities between the main loop and the concurrently running modules.
- Simplification of the main loop implementation.

Listing 5.2 shows the new implementation of the main loop. The new implementation makes it very clear which modules are being executed sequentially. Furthermore the main function does not act as a controller but rather leaves the state management in the module's responsibility.

Listing 5.2. main function implementation

```
95 while (true) {
```

```
 96    shell();
 97    batman_thread();
 98    rx_thread();
 99    uart_tx_thread();
100    watchdog();
101    timer_thread();
102 }
```

The next question was how to implement the actual concurrently running modules. One possibility was to reuse the predecessor's methodology and use state machine based implementations. There is a problem though in state machine based implementations and that is the rapidly growing complexity. This problem is called "state explosion problem" and has even a exponential behavior as shown in [3]. The equation 5.1 shows that the number of states is dependent on the number of program locations, the number of used variables and their dimensions.

$$\#states = |\#locations| \cdot \prod_{variable\ x} |dom(x)| \tag{5.1}$$

This equation shows that for instance a program having 10 locations and only 3 boolean variables already has 80 different states. Although this equation might not apply exactly to state machine based implementations it underlines the practical experience of big state-machine based implementations. The alternative to state-machine based applications are thread or process based implementations using the concurrent execution model as shown below.

5.1.2. Concurrent execution

This model requires support from an existing operating system. An existing framework or API provides the necessary abstraction to create new concurrent modules. Each module runs in isolation and can have its own main loop or terminate immediately. In terms of operating systems two abstractions are widely used for concurrently running software modules:

- **Processes**: Processes are usually considered as separately concurrently running programs. Usually each process owns its own memory context and communication with other processes happens through abstractions like pipes or shared memory.

- **Threads**: Threads are concurrently running code parts from the same program. The initial program is considered to run in its own "main thread". Other threads can be started from the main thread. Threads also do run in isolation to each other. Each thread has its own stack. Communication with other threads happens through shared memory provided by static data or the heap.

Processes as well as threads are widely known concepts in classical desktop operating systems. In the area of embedded micro-controllers these concepts also are implemented in many different implementations:

1. FreeRTOS (http://www.freertos.org)
2. TinyOS (http://www.tinyos.net)
3. Atomthreads (http://http://atomthreads.com)
4. Nut/OS (http://www.ethernut.de/en/firmware/nutos.html)
5. BeRTOS (http://www.bertos.org)

The above solutions have chosen different names for threads or processes (some call them "tasks") but essentially they all share the same concept of the concurrent execution model and will be referred to as concurrent modules from now on. Algorithm 3 shows the pseudo-code that initializes concurrent modules. One can see that in contrast to the sequential execution model the main loop actually does nothing.

Algorithm 3 Concurrent model initialization

start $module_1$

start $module_2$

...

start $module_n$

while true **do**

// no operation

end while

But how does a context switch happen between concurrent modules? Two methodologies exist:

- **Cooperative**: The concurrent modules by themselves return the control to a scheduler which then delegates the control to a different module. Which concurrent module gets control is often based on priorities which are controlled by the scheduler.
- **Preemptive**: Here the concurrent modules do not have control about how and when they get interrupted. It can happen anytime during the execution. Again the context switch between concurrent modules is often handled using priorities in the scheduler.

Nearly all existing solutions have one feature in common. That is that every thread has its own separate stack memory space. This is necessary in order to be able to run the same block of code (for instance a function) in multiple thread instances. On the other hand threads are being executed in the same memory context so sharing data between threads is possible by using the heap or static memory. All of the above mentioned frameworks provide common abstractions which are needed in thread based implementations:

- Semaphores
- Mutexes

- Yielding

In contrast to state machine based or sequential based concurrency thread based implementation can be expressed in very linear algorithms using the above mentioned abstractions. Take for instance the state-machine based algorithm 2. This could be translated into a linear thread-based algorithm as shown in 4.

Algorithm 4 Thread based algorithm

while true **do**

wait for resource mutex

process data

release resource mutex

end while

One can easily see that the thread-based algorithm 4 is much more expressive than the state-machine based algorithm 2.

Together with the necessity of having a scheduler these solutions can be considered as heavy-weight. The scheduler consumes additional CPU cycles and the separate stack memory space per thread consumes additional memory which is very scarce in embedded micro-controller systems.

Although usually a concurrent execution model must be provided in form of an API or an existing kernel there is one exception in the context embedded micro-controllers and that are ISRs (Interrupt Service Routines). Interrupt service routines behave like *preemptive* concurrent modules with highest priority.

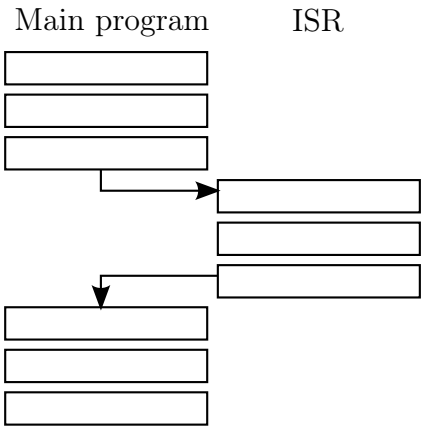

Figure 5.4. Illustration of an Interrupt Service Routine

The ISR interrupts the main program at any time when an external resource triggers an event and executes the service routine. The scheduler in this case is the CPU itself. There is one caveat with ISRs. When one ISR is being executed no other ISR can be triggered. Therefore it is being considered best practice not to perform intense and long running operations in ISRs.

5.1.3. Conclusion

For the implementation of this thesis the following conclusions were drawn:

- Existing solutions supporting the concurrent execution model were considered too heavy-weight for this type of application. Although 32KB of RAM are available the purpose is the support of route storage and network support.
- A sequential execution model was favored instead of the concurrent execution model. On the other hand thread-like linear algorithms are definitely favored instead of state machine based implementations which could lead to a state explosion.

One framework exists which implements the sequential execution model but providing a linear thread-like API being called Protothreads as described in [4]. It is implemented using C macros and expands to switch statements (or to goto statements if the GCC compiler is being used). Instead of consuming a complete stack per thread

the protothread implementation uses only two bytes per (proto)thread. Protothreads actually are stackless and variables initialized on the stack of a protothread function will stay initialized only during the very first call of the protothread.

Algorithm 5 Simple linear algorithm

while true **do**
  wait until timer expired
  process data
end while

Algorithm 5 shows a very simple linear use case where it waits for an external resource. In this case it waits for the expiration of an external timer by merely watching the timer's state. Since this is a read-only operation no explicit mutual exclusion is needed. This algorithm expressed as a protothread implementation is shown in listing 5.3.

Listing 5.3. linear protothread implementation

```
19 PT_THREAD(test(void))
20 {
21   PT_BEGIN(&pt);
22   PT_WAIT_UNTIL(&pt, timer_ready());
23   process_data();
24   PT_END(&pt);
25 }
```

The implementation of the algorithm is self-describing and corresponds to the APIs known from the concurrent execution model. The expanded version of the listing after the preprocessor stage is seen in 5.4.

Listing 5.4. expanded linear protothread implementation

```
char
test(void)
{
```

```
  // PT_BEGIN
  switch((&pt)->lc) {
    case 0:

      // PT_WAIT_UNTIL
      do {
        (&pt)->lc = 22;
    case 22:
        if(!(timer_ready())) {
          return 0;
        }
      } while(0);

      process_data();
  // PT_END
  };
  (&pt)->lc = 0;
  return 3;
}
```

The expanded version after the preprocessor stage of the implementation looks much more like a state machine based implementation from the sequential execution model. It uses a clever trick called loop unrolling ([5]) which breaks ups the while statement using the switch statement. This technique is also known as Duff's device as described in [6]. Unfortunately this implementation has one drawback. One cannot (obviously) use switch statements in protothreads. A slightly more efficient implementation using GCC labels circumvents this. Since the context switch is managed by the concurrent modules themselves the behavior can be classified as *cooperative* multitasking.

Due to the lightweight nature of protothreads and the possibility to express algorithms in a linear thread-like fashion this framework was chosen by the author for the

Listing 5.5. Sender route for the UART module (see [2])

```
void rsSend(uint8_t data)
{
  while( !(UCSRA & (1<<UDRE)));
  UDR = data;
}
```

implementation.

5.2. Ring Buffers

The predecessor thesis [2] used the UART interface in order to communicate with the user and to inform about incoming packets, changes to routes, etc.. As already analyzed in the previous chapter a state machine based sequential concurrent model was used to implement the UART module. There exists one problem with the current implementation.

Listing 5.5 shows that the algorithm examines the UCSRA (USART Control and Status Register A) and blocks infinitely until the UDRE (USART Data Register Empty) bit becomes zero. The execution of all other concurrent modules and the main loop will be blocked until the UART becomes ready to accept data. In this time period no data can be received from the radio. The above mentioned implementation uses the same function for sending strings via the UART interface. For sending the string "hello" via the UART with a speed of 19.2kbps the main loop will be physically blocked for 2.5 milliseconds. In order to improve the implementation the author wanted to accomplish the following goals:

- Refactoring to a non-blocking operation.
- Migration to a concurrent execution model using protothreads.

The Atmega162 micro-processor offers the following ISRs for receiving and sending data via the UART ([7]):

- **SIG_USART_RECV**: Is being invoked, when the UDR register contains a new byte received from the UART.

- **SIG_USART_DATA**: Is being invoked, when the UDR register is ready to be filled with a byte to be transmitted via the UART.

So we have the possibility to send or receive data asynchronously from the main loop in the context of a concurrent execution model by using ISRs. Filling the UDR or reading the UDR in the main loop (and thus blocking it) is actually not necessary at all. The main loop can communicate with the ISRs through a receiving and transmitting queue buffer where it writes data to the transmitting queue and reads data from the receiving queue.

Using this sort of communication is known as the "producer-consumer problem". It can be implemented using a FIFO buffer. The Linux kernel (see [8] chapter 5.7.1) as well as (embedded) DSP micro-controllers (see [9]) use a very elegant FIFO-algorithm by providing a lock-free buffer being called "circular buffer" or "ring buffer".

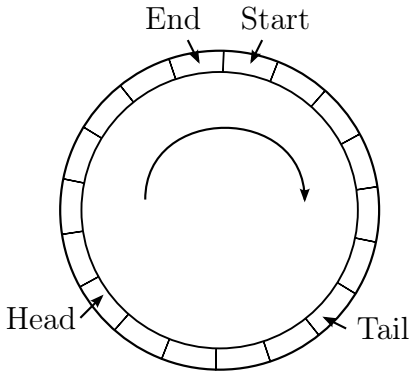

Figure 5.5. Illustration of a ring buffer

Figure 5.5 shows the basic principle of the algorithm. A circular buffer is defined by the following four pointers:

- **Start:** This pointer defines the beginning of the buffer in memory. This pointer is static.
- **End:** This pointer defines the end of the buffer in memory. It can also be expressed as the maximum length of the buffer. This pointer is static.
- **Head:** The head pointer is being changed dynamically by the producer. Whenever the producer wants to write data in the buffer the head pointer is increased

and the corresponding memory filled. If the head points to the same address as the tail pointer the buffer is full or empty.

- **Tail:** The tail pointer is being changed dynamically by the consumer. Whenever the consumer wants to read data from the buffer the tail pointer is increased and the corresponding memory cleared. If the tail points to the same address as the head pointer the buffer is full or empty.

In order to distinguish whether the buffer is full or empty an additional size variable was implemented. The biggest advantage of the presented algorithm is the possibility to write and read data in a lock-free fashion. A consumer thread does not need to wait for a mutual exclusion on the buffer since the consumer thread is the only instance manipulating the tail pointer. The same applies for the producer being the only instance manipulating the head pointer.

The complete listing of the ring-buffer implementation can be seen in appendix A in the file `src/ringbuf.c`. There are two functions provided:

- **ringbuf_add:** This function is being called by the producer. The function immediately returns `true` if a byte could be written to the the buffer or `false` if the buffer is full.

- **ringbuf_remove:** This function is being called by the consumer. The function immediately returns `true` if a byte could read from the buffer or `false` if the buffer is empty.

The important nature of the above mentioned functions is that they are non-blocking because they return immediately. These functions could therefore be called from protothreads. A producer protothread running in the context of the main loop can write data like presented in listing 5.6. The consumer of this data is the SIG_USART_DATA ISR as presented in listing 5.7.

Listing 5.6. Producer writing data

```
PT_THREAD(producer(uint8_t data))
```

```
{
    PT_BEGIN(&pt);
    PT_WAIT_UNTIL(&pt, ringbuf_add(buf, data));
    PT_END(&pt);
}
```

Listing 5.7. Consumer reading data

```
ISR(SIG_USART_DATA)
{
  uint8_t c;
  if (ringbuf_remove(buf, &c)) {
    UDR = c;
  }
}
```

Instead of *physically* blocking the algorithm expressed in listing 5.6 only *logically* blocks the protothread. If the buffer is full a context-switch back to the main loop is performed. The main loop sequentially executes all other concurrent modules and returns to the protothread which then again tries to add data into the ring buffer.

5.3. Half-Duplex Radio Access (Petri Net)

The predecessor implementation used an identical (physically) blocking implementation in order to send or receive data via the RFM12B hardware module. Listing 5.8 shows the algorithm used for sending data.

Listing 5.8. Sender routine for the RFM12B hardware module (see [2])

```
void rfTx(uint8_t data)
{
        while(WAIT_NIRQ_LOW());
        rfCmd(0xB800 + data);
}
```

This implementation physically blocks the main loop the same way as the UART algorithm shown in listing 5.5. In this case the algorithm does not wait for the status of an internal register to send data but rather waits for the external nIRQ pin from the RFM12B hardware module to go low. The official "RF12B programming guide" (see [10]) also proposes a physically blocking algorithm.

The author wanted to improve the algorithm in a similar fashion as the UART algorithm. The nIRQ pin of the RFM12B was connected to the INT0 pin of the ATMega162 micro-processor allowing to execute the SIG_INTERRUPT0 interrupt service routine asynchronously. But it turned out that the implementation could not be reused at all. The RFM12B radio hardware imposes the following algorithmic challenges for the driver implementation:

- **Single interrupt request for multiple events:** The RFM12 radio module uses only one nIRQ pin in order to generate an interrupt for the following events (see [11]):
  - The TX register is ready to receive the next byte (RGIT)
  - The RX FIFO has received the preprogrammed amount of bits (FFIT)

  The state management has to be implemented in software otherwise the current state of operation (sending or receiving) is undefined.

- **Half-Duplex operation:** The RFM12 radio module only allows either to receive or to send data at a time but not simultaneously.

The author abstracted the operation of the RFM12B driver algorithm as a (proto)thread. Interestingly enough the thread has a state modeled as a state machine depending whether it receives or sends data. The following states are valid:

- **RX:** This is the receiving state. The thread (logically) blocks until a complete packet has been received. Whether a packet is complete or not depends on the upper network stack layers.

- **TX:** This is the sending state. The thread (logically) blocks until a complete packet has been sent. Again the upper network stack layers decide whether the transmission is complete or not.

The abstract algorithm is shown in 6. The question is who sets the actual state of the radio driver. Receiving data is *non-deterministic*. A packet can arrive at any time and thus the invocation of the SIG_INTERRUPT0 interrupt service routine. Therefore the algorithm sets the RX state as the *default* state for the radio thread.

Sending data on the other hand is *deterministic*. When a user hits the Enter key via the UART module a packet can be constructed. A 3rd party thread has to request the control over the radio module and occupy it until the packet has been fully transmitted. The author realized that this is a more or less classical concurrency problem between two threads and a single resource:

- **Sender Thread:** The sender thread wants to acquire the control over the radio module until the transmission of a packet is complete.
- **Receiver Thread:** The receiver thread also wants to acquire the control over the radio module until the packet reception is complete.
- **Single resource:** The external resource in this case is the radio module. Only the receiving radio driver thread or the transmitting sender thread can own the radio hardware resource at a time.

Algorithm 6 RFM12B driver thread algorithm

```
while true do
  if state is RX then
    receive data
  else if state is TX then
    send data
  end if
  set state to RX
end while
```

In classical multi-threaded algorithms this problem can be solved using mutual exclusion. Modeling such an algorithm can be done using petri nets as shown in figure 5.6.

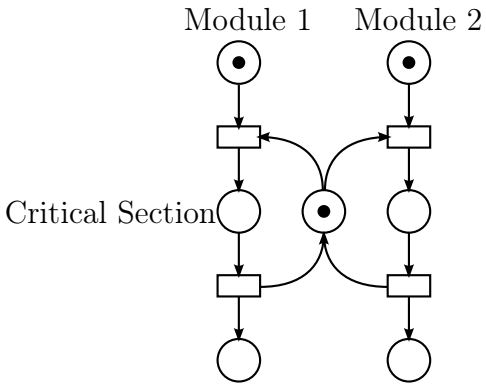

Figure 5.6. Mutual exclusion model using a petri net

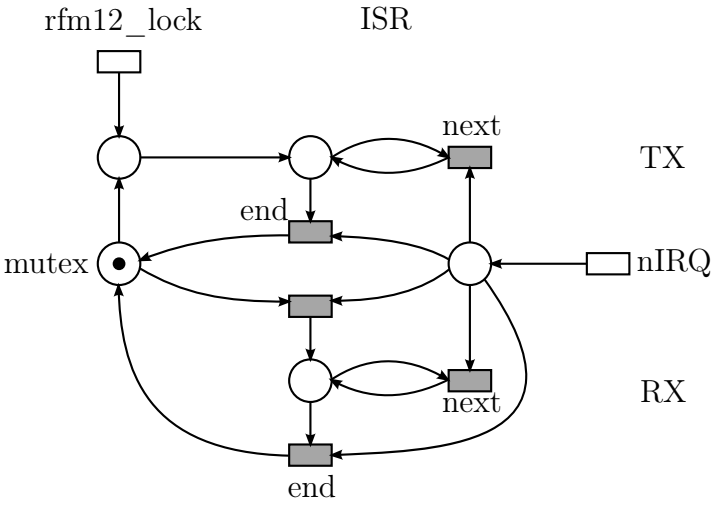

Figure 5.7. Half duplex algorithm modeled as a petri net

Chapter 6

Network Stack

6.1. Layer 2a: MAC Layer

6.2. Layer 2b: Logical Link Control

Hamming Code

6.3. Layer 3: Batman Routing

6.4. Layer 7: Application

Chapter 7

Research

7.1. Simulations

7.1.1. Shell

7.1.2. Routing

7.1.3. Radio Transmission

7.2. Mesh evaluation

7.3. Results

Chapter 8

Conclusion

Appendix A

CD content

1. **src** — The source files for the hopemesh implementation

    - **ringbuf.c** — The ringbuffer implementation

Bibliography

[1] Taylor L. Booth. *Sequential Machines and Automata Theory (1st ed.)*. 1967.

[2] Marek Korniowski. *Projekt odpornej na awarie sieci komputerowej z transmisją danych w pasmach nielicencjonowanych*. 2009.

[3] Joost-Pieter Katoen. The state explosion problem, 2008. URL `http://www-i2.informatik.rwth-aachen.de/i2/fileadmin/user_upload/documents/MC08/mc_lec5a.pdf`.

[4] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. 2006.

[5] Michael Abrash. *Graphics Programming Black Book*. 1997.

[6] Tom Duff. Tom duff on duff's device, 1988. URL `http://www.lysator.liu.se/c/duffs-device.html`.

[7] Atmel. *Datasheet: ATmega 162*. Atmel, 2009.

[8] Jonathan Corbet, Greg Kroah-Hartman, and Alessandro Rubini. *Linux Device Drivers, 3rd Edition*. 2005.

[9] Randy Restle. Circular buffer in second generation dsps. 1992.

[10] HOPERF. *RF12B programming guide*. 2011.

[11] Silicon Labs. *Datasheet: Si4421 Universal ISM Band FSK Transceiver*. Silicon Labs, 2008.