LaTeX 2_ϵ-Vorlage von Matthias Pospiech

Leibniz Universität Hannover

Matthias Pospiech

August 6, 2011

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

<Ort einfügen>, den <Datum einfügen> <Autor einfügen>

Contents

1 Introduction

1.1 Research overview

2 Evaluation

2.1 Existing solution

2.2 Assumptions

2.3 Requirements

3 Hardware Design

3.1 RAM

- Harvard architecture
- RAM bus
- Latch

3.2 USB Serial Device

3.3 RFM12B Radio

3.4 Keyboard

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

5 Software Algorithms

5.1 Protothreads

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:

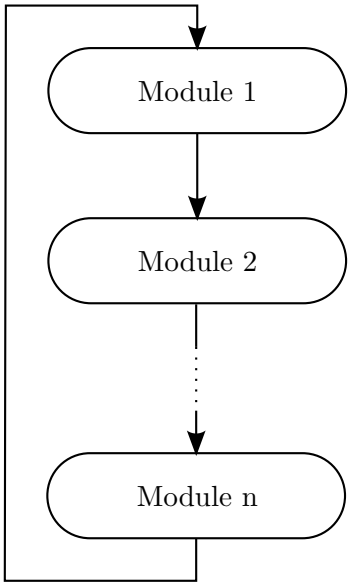

Figure 5.1: Sequential execution model

Sequential execution

This type sequentially executes all modules starting from the first module until the last one. Once the last module ends the execution starts again from the first module. It is a very simple model that does not need any 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 ([Boo67]).

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 5.2.

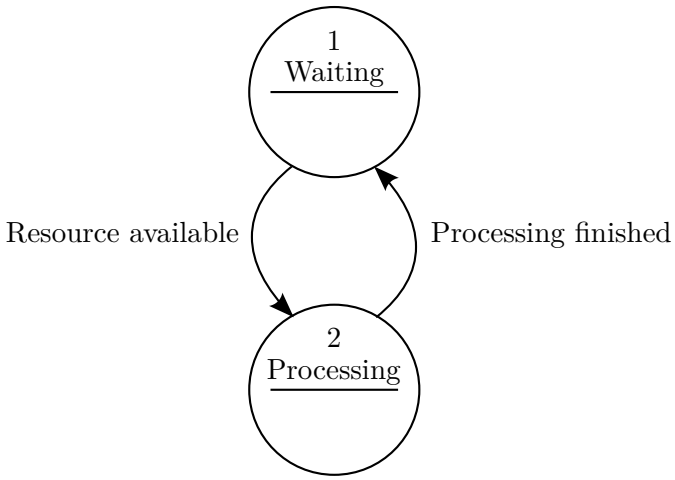

Figure 5.2: State Machine for a module

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 5.2. 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 [Kor09] 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 thesis implementation.

Listing 5.1: main function implementation in [Kor09]

```
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). The lack of a clear separation of module responsibility and conformance to the state machine theory led me to a completely new implementation as show in listing 5.2.

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 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.

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 [Kat08]. 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.

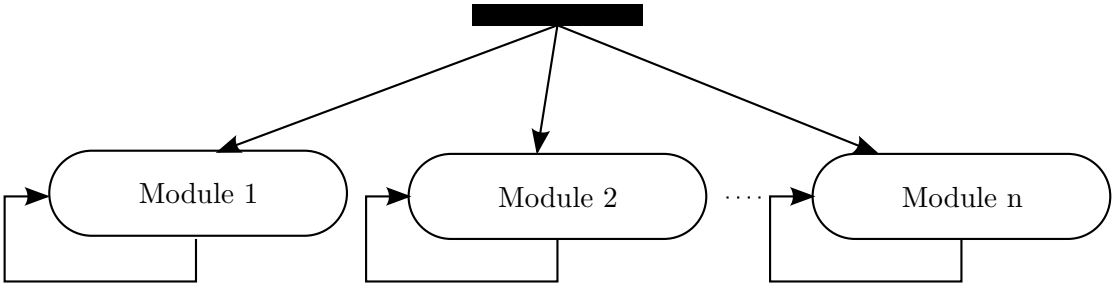

Figure 5.3: Concurrent execution model

Concurrent execution

This execution model executes modules concurrently. Instead of heaving an infinite main loop that iterates sequentially over all modules the main function only initializes and launches concurrent modules. Each module runs in isolation and can have its own main loop or terminate immediately. This model requires support from an existing operating system. An existing framework or API provides the necessary abstraction to create new concurrent modules. In terms of operating systems two abstractions are wide-spread 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 seperate 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 seperate stack memory space per thread consumes additional memory which is very scarce in embedded micro-controller systems.

Although ususally 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 very much like preemptive concurrent modules with highest priority.

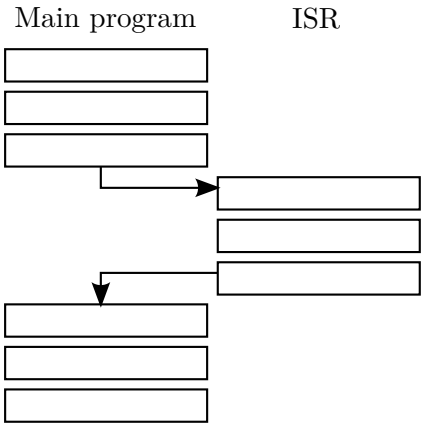

Figure 5.4: Illustration of an Interrupt Service Routine

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.

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 [Dun06]. 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 ([Abr97]) which breaks ups the while statement using the switch statement. This technique is also known as Duff's device as described in [Duf88]. Unfortunately this implementation has one drawback. One cannot (obviously) use switch statements in protothreads. A slightly more efficient implementation using GCC labels circumvents this and was used for the complete implementation of the thesis.

5.2 Ring Buffers

The predecessor thesis [Kor09] 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.

The 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

Listing 5.5: UART module sending function from [Kor09]

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

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 following goals had to be accomplished:

- 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 ([Atm09]):

- **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 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 receiving queue.

As shown in [Cor05]

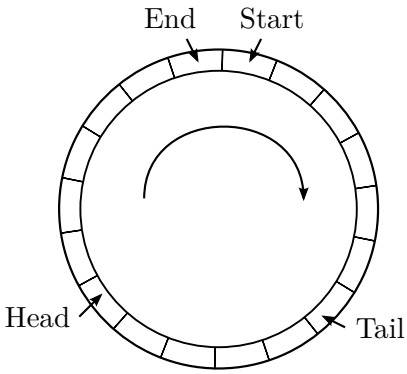

Figure 5.5: Illustration of ring buffer

5.3 Half-Duplex Radio Access (Petri Net)

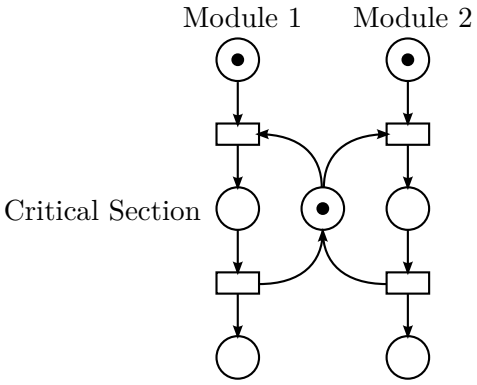

Figure 5.6: Mutual exclusion model using a petri net

6 Network Stack

6.1 Layer 2a: MAC Layer

6.2 Layer 2b: Logical Link Control

6.3 Layer 3: Batman Routing

6.4 Layer 7: Application

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

8 Conclusion

Bibliography

[Abr97] Abrash, Michael: *Graphics Programming Black Book* (1997)

[Atm09] Atmel: *ATmega 162 datasheet* (2009)

[Boo67] Booth, Taylor L.: *Sequential Machines and Automata Theory (1st ed.)* (1967)

[Cor05] Corbet, Jonathan; Kroah-Hartman, Greg und Rubini, Alessandro: *Linux Device Drivers, 3rd Edition* (2005)

[Duf88] Duff, Tom: Tom Duff on Duff's Device (1988), URL `http://www.lysator.liu.se/c/duffs-device.html`

[Dun06] Dunkels, Adam; Schmidt, Oliver; Voigt, Thiemo und Ali, Muneeb: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems (2006)

[Kat08] Katoen, Joost-Pieter: The State Explosion Problem (2008), URL `http://www-i2.informatik.rwth-aachen.de/i2/fileadmin/user_upload/documents/MC08/mc_lec5a.pdf`

[Kor09] Korniowski, Marek: *Projekt odpornej na awarie sieci komputerowej z transmisją danych w pasmach nielicencjonowanych* (2009)

List of Figures

List of Tables

Danksagung