$\LaTeX\,2_\epsilon$-Vorlage von Matthias Pospiech

Leibniz Universität Hannover

Matthias Pospiech

August 3, 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 Personal motivation

This thesis describes the analysis, enhanced design and implementation of an existing microcontroller based mesh solution [Kor09]. The current solution showed.

1.2 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:

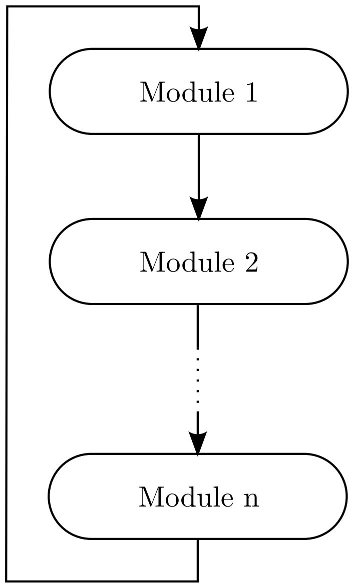

Figure 5.1: Sequential execution model

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.

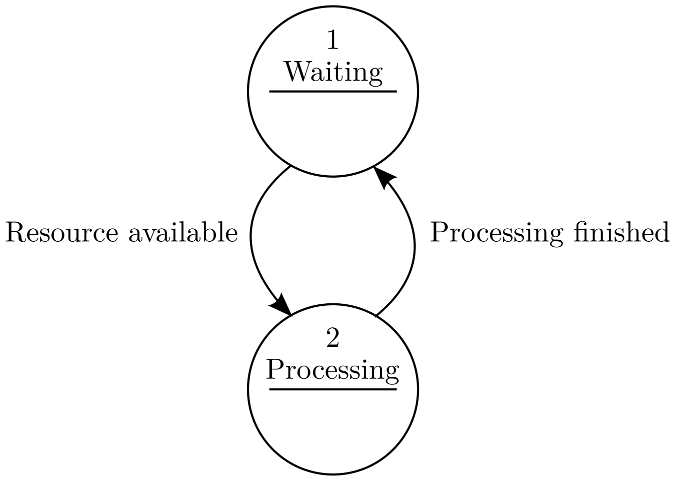

Figure 5.2: State Machine for a module

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
        return
    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]

```
while(0x01)
{
        if(uartInterrupt == ON) // got a character from RS232
+---- 44 lines:


        // --- RECEIVE A DATAGRAM ---

        else if((datagramReceived = datagramReceive(...))
                && netState > 0)
+----182 lines:


        else if(helloTime) // prepare periodic Hello message
+---- 19 lines:



        // --- SEND A DATAGRAM ---

        if(datagramReady && netState > 0)
```

```
+----  8 lines:

}
```

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 the state machine theory led to a completely new implementation as show in listing 5.2.

Listing 5.2: main function implementation

```
while (true) {
  shell();
  batman_thread();
  rx_thread();
  uart_tx_thread();
  watchdog();
  timer_thread();
}
```

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.

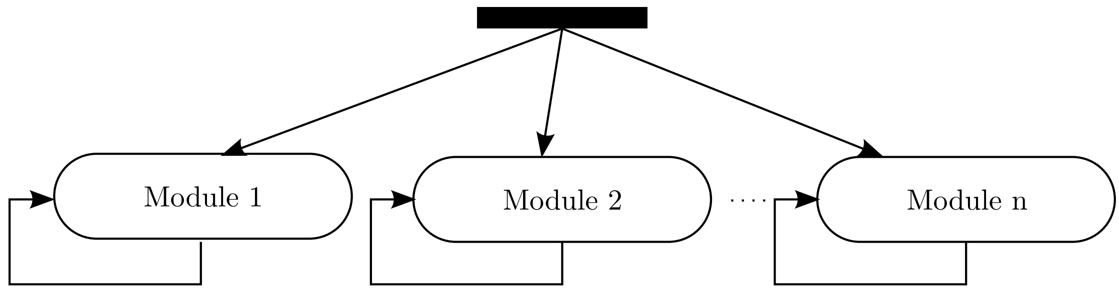

Figure 5.3: Concurrent execution model

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 seperately concurrently running programs. Usually each process owns its own memory space 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.

Processes as well as threads are widely known concepts in classical desktop operating systems. In the area of micro-controllers these concepts also are implemented in very lightweight implementations. Many solutions exist like:

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 choosen different names for threads or processes (some call them "tasks") but essentially they all share the same concept of the concurrent execution model.

Alternatives:
- Heavyweight: Real Operating System. Enumerate them and compare ...
- Lightweight: Thread implementations. Problem: Each thread has its own stack which consumes a lot of memory.
- More Lightweight: Protothreads. Best compromise between classical state machines and real threads.

5.2 Ring Buffers

5.3 Half-Duplex Radio Access (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

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

[Kat08] Katoen, Joost-Pieter: The State Explosion Problem (2008)

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

List of Figures

List of Tables

Danksagung