# Chapter 5
# Software Production

## What are you about to learn?

### Knowledge Objectives

- Understand the structure of a foreground/background system.
- Understand the given coding rules.

### Skill Objectives

- Ability to develop real-time software that runs on a personal computer.
- Ability to create PC replacements for real-time system hardware components.
- Ability to develop test cases for real-time software that run on a personal computer.

## 5.1 Implementing Simple Real-Time Systems

Simple real-time systems are characterized by

- Little or no concurrency
- Small number of event sources
- Event frequencies which are either low or multiples of each other
- Low to moderate computational complexity

Many real-world systems are of this type. Examples: laundry machine, microwave, bicycle computer. This chapter shows how to construct such types of systems.

### Foreground/Background Systems

Basically all processors work like this, until they are turned off:

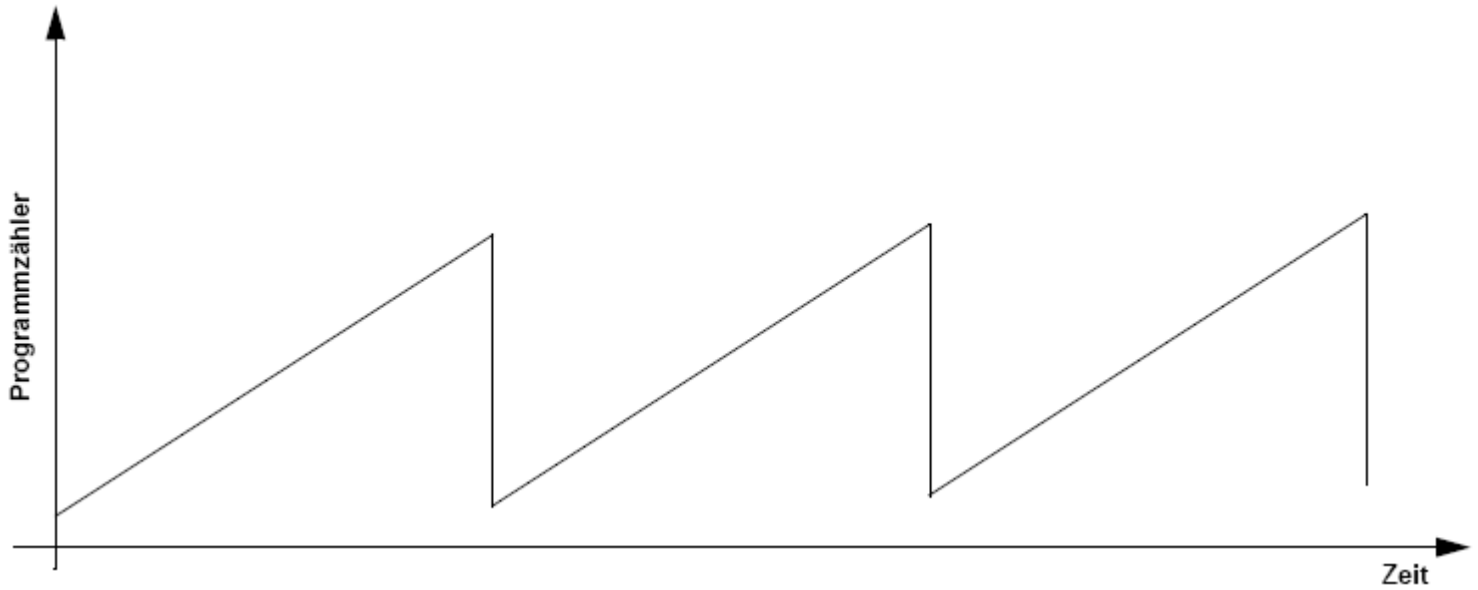
1. Fetch machine opcode from program memory location to which program counter points
2. Advance program counter to next opcode location in program memory
3. Decode opcode
4. Execute opcode

If there were no branches or interrupts, the program counter would wrap around once it has reached its maximum value. Example: for the Freescale 68HCS12 processor the maximum program counter value is $FFFF, thereafter it would wrap back to 0.

In real systems, the programmer controls this wrap-around by using a loop. This leads to the most simple of all real-time programs:

```
1 int main() {
2   /* do here what needs to be done once ... */
3   for (;;) {
4      /* do here what needs to be done all the time...*/
5   }
6 }
```

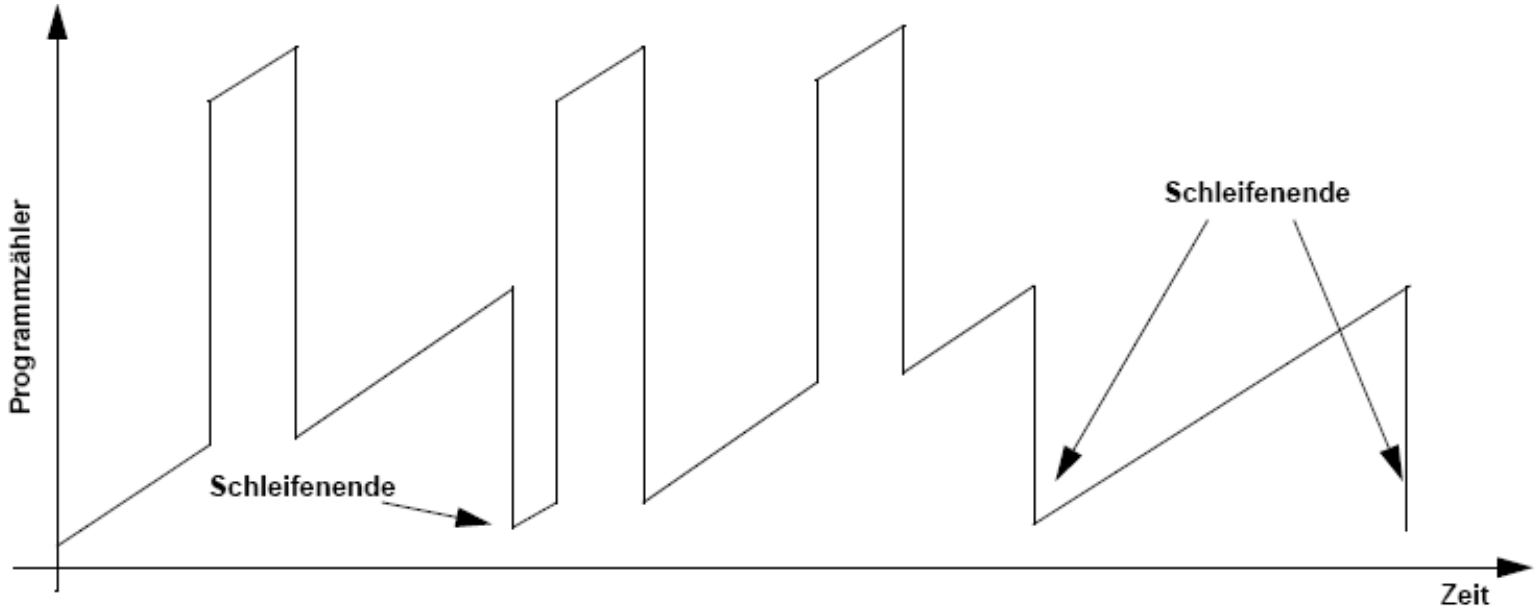
The following picture illustrates the value of the program counter over time for this simple loop.



This would permit us to implement a simple clock, if the time spent inside each loop cycle would be constant. However, this is typically not the case, since depending on the situation different commands would have to be executed (e.g., overflow from seconds to minute).

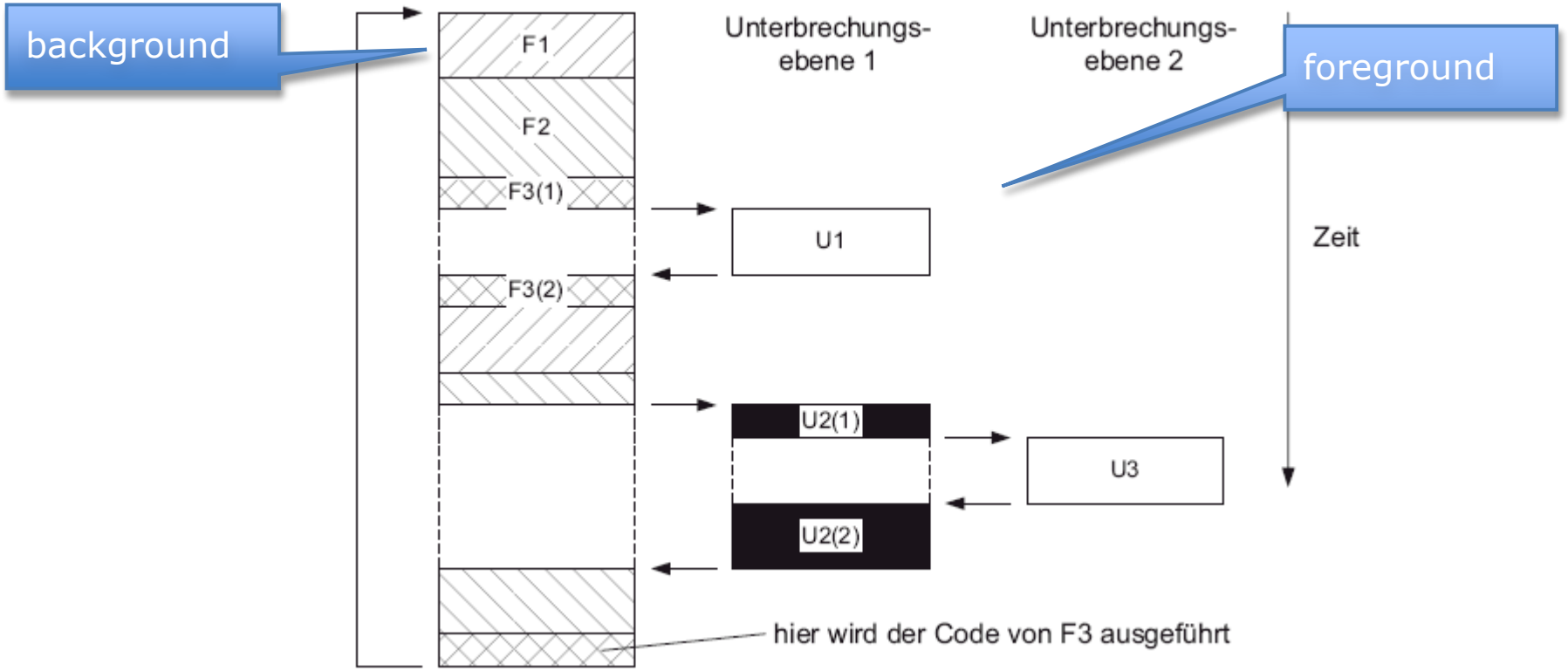
To get precise real-time, we need support from the hardware. This comes in form of a digital hardware counter which interrupts our loop when the hardware counter has reached a predefined value.

The following picture illustrates the value of the program counter over time for the simple loop interrupted by the hardware:




Counting seconds, minutes, and hours could be done inside the interrupt service routine (illustrated by the peeks in the figure above).

We call a software structure consisting of a main program with a loop plus interrupt service routines a foreground/background system.




The foreground/background structure is the most simple practical software structure for an embedded real-time system.

## Exceptions and Simple Interrupt Service Routines

As mentioned above, basically all processors work like this, until they are turned off:

1. Fetch machine opcode from program memory location to which program counter points
2. Advance program counter to next opcode location in program memory
3. Decode opcode
4. Execute opcode

For most processors, the advancement of the program counter can be changed from this scheme by

- Branch commands
- Call of subroutines
- Return from subroutines
- Exceptions

There are three types of exceptions:

- Interrupts
- Resets
- Traps

Interrupts are asynchronous events caused by hardware (e.g., digital counters, interface circuits, A/D converters, timers).

Resets are linked to a specific hardware input on processors (reset pin) and initialize the processor hardware (registers, program counter, etc.).

Traps are events caused by software commands or processor internal hardware:

- Software interrupts
- Hardware related errors (division by 0, misaligned memory access, etc.)

Exceptions interrupt the regular program flow. They are handled in dedicated interrupt service routines. After the interrupt service routine has finished, regular program execution resumes (except for reset).

### Example: Simple Blinkenlight

As an illustration for a simple foreground/background system we use a blinking led program, where the blink frequency is controlled by a hardware interrupt. We use a Dragon12 hardware with an 68HCS12 processor for this example. The background software (main loop) doesn't do anything in our example.

Here is the main program:

```
1 void main(void) {
2 EnableInterrupts; /* We need this for the debugger and real time interrupt */
3 /* Deactivate 7-segment display */
4 DDRP = DDRP | 0xf; /* Data Direction Register Port P */
5 PTP = PTP | 0x0f; /* Turn off all four digits */
6
7 /* Activate LEDs */
8 DDRJ_DDRJ1 = 1; /* Data Direction Register Port J */
9 PTJ_PTJ1 = 0; /* Turn on LED row */
```

```
10
11 /* Set Port B to output */
12 DDRB = 0xFF; /* Data Direction Register Port B */
13
14 /* First switch LED PB0 on */
15 PORTB = 0x01;
16
17 /* Configure hardware divider for real-time interrupt */
18 RTICTL = 0x7f;
19 CRGINT = CRGINT | 0x80; /* switch on real-time interrupt */
20
21 /* And here is the main loop (background)... */
22 for(;;) {
23    /* We don't do anything here, interrupt service routine does */
24    /* all the work ... */
25 }
26 }
```

Configuring the hardware requires knowledge of the processors periphery registers (see lecture on computer architecture).

The software above initializes the hardware, and then enters the main loop. Inside this loop we could check the switches and buttons, or run a state machine, or do anything else.

The LED will blink exactly with the frequency determined by the hardware counter linked to the real-time interrupt (see lecture on computer architecture).

The interrupt service routine will look like this:

```
27 interrupt 7 void rtiISR (void) {
28   CRGFLG = CRGFLG | 0x80; /* reset interrupt flag */
29   PORTB = ~PORTB & 0x01; /* turn LED on/off */
30 }
```

The keyword “interrupt” and the “7” are compiler specific commands (not ISO C). They link this subroutine to the hardware interrupt number 7, which is the real-time interrupt.

Whenever interrupt number 7 is raised, program execution will divert to the interrupt service routine rtiISR. The name of this routine is of no importance.

Please note that this interrupt is not well suited to generate a 10 msec or 1 sec interrupt. We will have to user other timers for that, see chapter 3!

## 5.2 Hints for Programming in the C-Language

### Rules for Programming in the C-Language

The C-language is the most widely used programming language for real-time systems. There are a number of reasons for this:

- If there is any compiler available for a microprocessor at all, it is typically a C compiler
- C very well supports high-speed, low-level input/output operations, essential for many real-time embedded systems
- C compilers are very efficient generating small and fast programs
- C programs can be reasonably well ported to other hardware platforms
- Many modelling packages generate C code (e.g., ASCET, Simulink)

However, C does not have real-time support built into the language. It requires operating support for real-time primitives like concurrency and synchronization.

Every programming language comes with language insecurities:

- Programmers make mistakes (e.g., misspelled variable name, a=b)
- The programmer misunderstands the language (e.g., operator precedence)
- The compiler doesn't do what the programmer expects (e.g., incomplete definition, portability)
- The compiler contains errors (not that rare for embedded compilers)
- Run-time errors (C has poor run-time checking, e.g. pointer errors, stack overflow)

## 5.2 Hints for Programming in the C-Language

Thus, to develop C programs, it is good to follow some rules.

In the automotive industry, MISRA provides a set of rules (http://www.misra.org.uk).

The following slides will give some example rules from the MISRA-C 2004: Guidelines for the use of the C-language in critical systems standard.

All code shall conform to ISO 9899:1990 "Programming languages – C". This precludes the use of C++ comments (//) and other proprietary extensions. Reason: portability.

The following type definitions are to be used (modified from MISRA standard):

- `char` plain 8 bit character
- `uint8` unsigned 8 bit integer
- `uint16` unsigned 16 bit integer
- `uint32` unsigned 32 bit integer
- `int8` signed 8 bit integer
- `int16` signed 16 bit integer
- `int32` signed 32 bit integer

Reason: portability

Assembly language shall be encapsulated and isolated. Reason: portability.

Sections of code should not be "commented out". Use "ifdef" statements. Reason: C does not support nested comments.

Identifiers in an inner scope should not use the same name as identifiers in an outer scope, and therefore hide that identifier.

```
int16_t i;
{
    int16_t i;      /* This is a different variable                    */
                    /* This is not compliant                           */
    i = 3;          /* It could be confusing as to which i this refers */
}
```

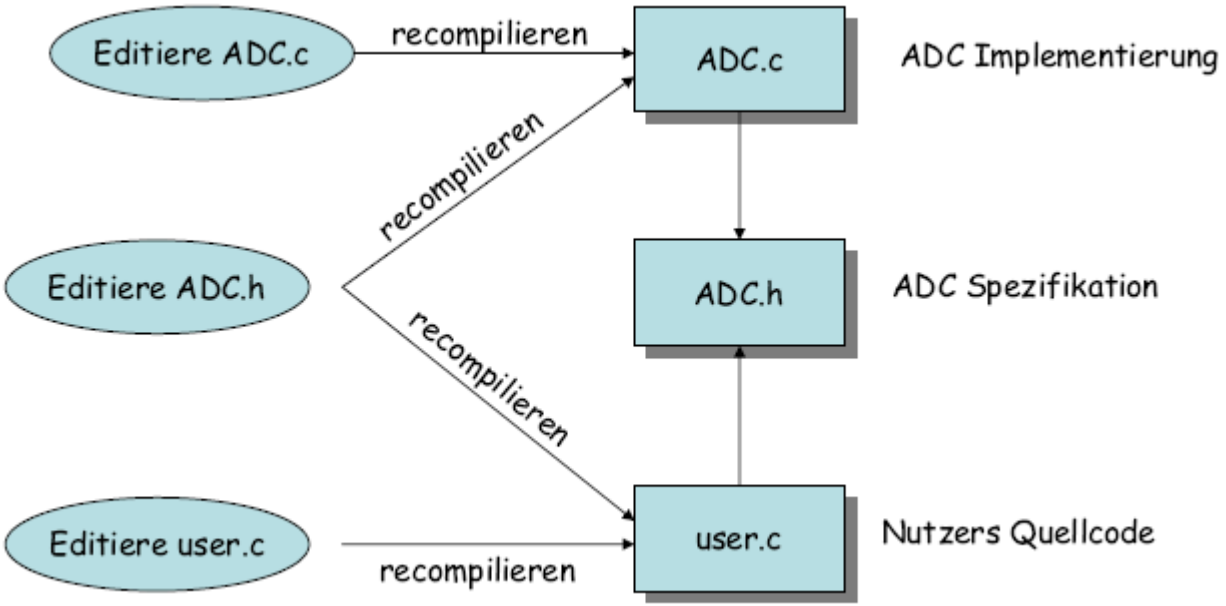
In total, there are about 120 rules in the MISRA standard, some of which are required, and some of which are advisory.

Here are some basic rules for good interface design:

- Interface for file <module.c> is always contained in <module.h>.
- Both files are located in the same directory, except for libraries.
- All functions not declared in <module.h> must be declared "static" in <module.c>.
- Description of functions and data is only in <module.h>.

For the make process, if not automatically handled by the IDE:

- Use Miller make approach (http://miller.emu.id.au/pmiller/books/rmch/)




Use meaningful variable names, it vastly improves readability of your program. Humans don't like to read a name like p_ui_count (pointer to unsigned int variable count). Better:

```
if (refridgeratorEmpty && purseFilled()) {
    GotoGroceryStore();
    beerBottles = BuySomeBeer(12);
    StuffFridge(beerBottles);
}
```

All header files shall have guards. Guards prevent multiple definitions with recursive inclusions and infinite recursions.

x.h

```
#include "y.h"
...
```

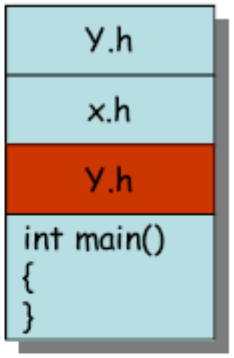
y.h

```
...
...
```

x.c

```
#include "x.h"
#include "y.h"

int main()
{...
}
```




x.h

```
#ifndef XH
#define XH

#include "y.h"
...
#endif
```

y.h

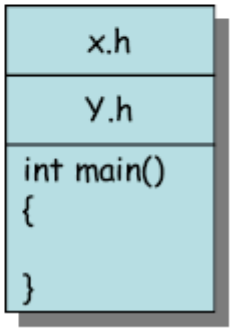
```
#ifndef YH
#define YH

...
#endif
```

x.c

```
#include "x.h"
#include "y.h"

int main()
{...
}
```




In the first line, <y.h> would be included twice, with possibly duplicate definitions.

## 5.2 Hints for Programming in the C-Language

Some more rules:

Don’t use global variables! If you have to, place them all into <global.c> and provide getter and setter methods. Example for <global.c>

```
#include “global.h”
static int16 globalVar;

/* Getter function */
int16 getGlobalVar() {
  return globalVar;
}

/* Setter function */
void setGlobalVar(int16 _var) {
  globalVar = _var;
}
```

Example for <global.h> (without comments to save space):

```
#ifndef __globalh__
#define __globalh__
int16 getGlobalVar()
void setGlobalVar(int16 _var);
#endif
```

Do not use the “extern” keyword.

This is just tranquilizer for the compiler and not necessary at all. It just tells the compiler that this symbol has been defined elsewhere, somehow, don’t worry. If so, than it should be pulled from an included file. If not, this is an error which might be detected at link time (hopefully).

Turn on all warnings of your compiler and linker.

This will give you hints on possible language misunderstandings or dangerous constructs. As a basic rule: use any static checks that you can. Don’t ship code with warnings.

Do not use unions (18.4).

This will prevent problems with endianess, alignment of structures, bit order in bit fields, and padding at the end of unions.

Functions shall not call themselves, either directly or indirectly.

Recursions can exceed the available stack space, and since there is no run-time checking in C, this can cause hard to trace problems.

Do not use the “goto” and “continue” statements.

Usage of these statements defies the good principles of structured programming.

Attempt to make all functions reentrant.

## 5.2 Hints for Programming in the C-Language

Real-time systems often have to support concurrency. This means that a running thread might be interrupted by another thread any time. In many instances this rule can be realized by not using global (module global) variables. Example:

The following function is reentrant (just automatic variables):

```
void myLittleSwapper(int *a, int *b)
{ /* This function is reentrant. */
  int temp = *b;
  *b = *a;
  *a = temp;
}
```

The following function is not reentrant (because of the static variable):

```
static int temp;
void myLittleSwapper(int *a, int *b)
{ /* This function is not reentrant. */
  temp = *b;
  *b = *a;
  *a = temp;
}
```

Problematic scenario: this function is being called by one thread after it has executed the first statement. It is then interrupted, and called again by another thread.

Don’t use macros unless unavoidable.

Macros can have nasty side effects not visible to the programmer. Macros can make source code unreadable.

Examples:

```
#define roundUpDivider (x, y) (x + y - 1) / y
```

use like

```
a = roundUpDivider (b & c, sizeof(int));
```

expands to

```
a = (b & c + sizeof(int) - 1) / sizeof(int);
```

which yields the wrong result because the “+” operator has higher precedence than the “&” operator.

Another problematic macro example:

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))
```

use like

```
next = min (x + y, foo(z));
```

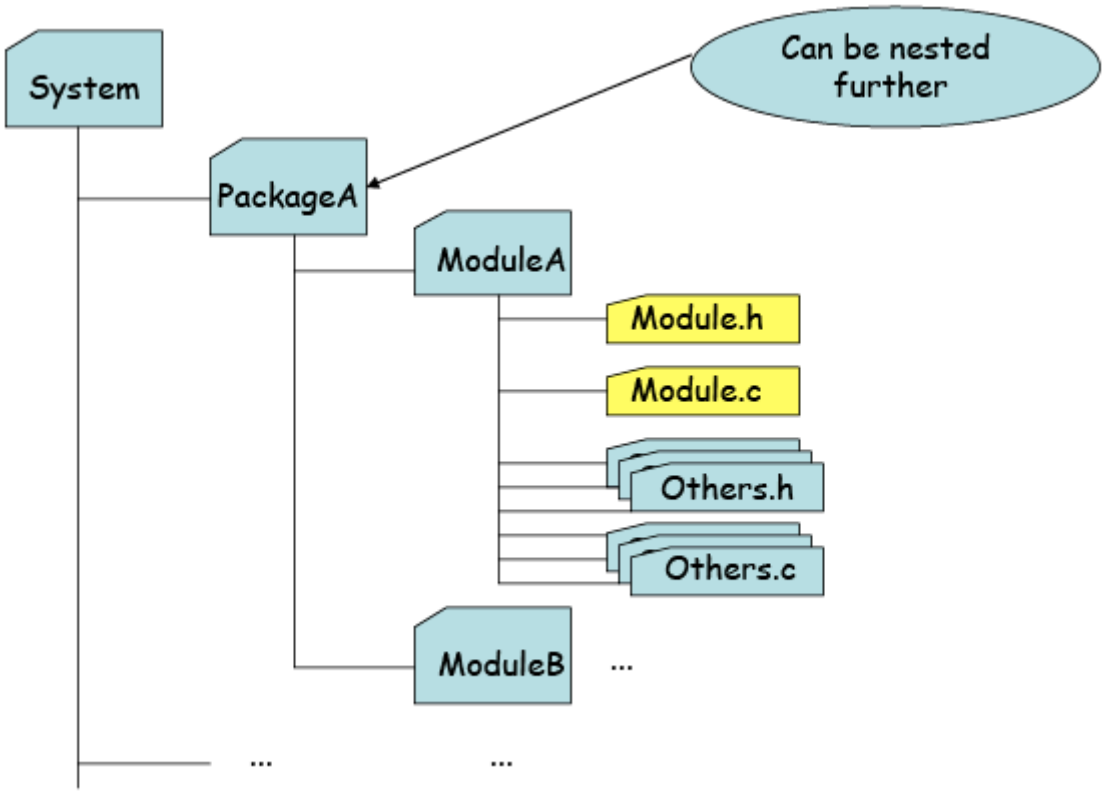
expands to

```
next = ((x + y) < (foo(z)) ? (x + y) : (foo(z));
```

Which means that foo(z) is being called twice, while in the call it looks like it is just being called once. If foo(z) contains a counter, it will have the wrong value.

## Rules for Software Structuring and Directory Layout

The software structure should be reflected in the directory structure.




Do not use paths in include statements. Paths to include files are configured in the IDE or in the make files. Reason: portability of development environment.

## 5.3 Software Development without Target Hardware

It is desirable to develop a large part of the real-time software on a standard workstation. This gives the following advantages:

- Software can be developed and tested without hardware (simultaneous engineering)
- Efficient development environment (no hardware problems, powerful tools)
- High testability (code instrumentation, fault injection, code coverage, memory leaks, controlled execution environment)
- Inexpensive regression testing

However, this approach has some limits:

- Real-time behavior not equivalent to real hardware
- When using a real-time operating system, simulating this on a workstation can be difficult
- Simulation of intelligent interfaces not always easy (complex interface controllers)

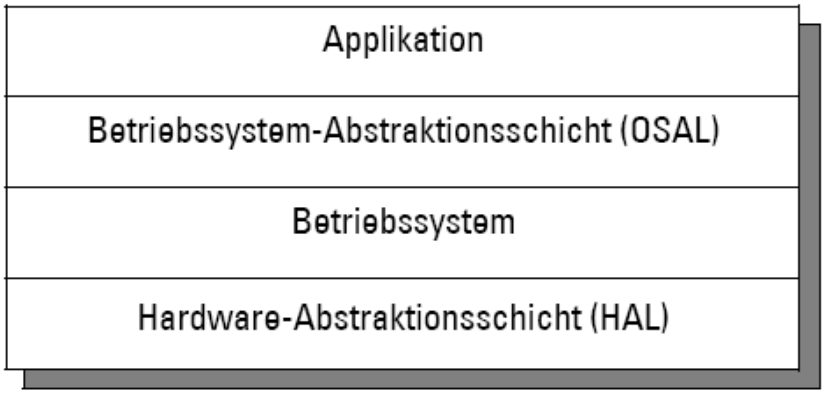
Nonetheless, whenever possible this approach should be taken.

To use this approach, we structure the software in layers. Layering software has some advantages:

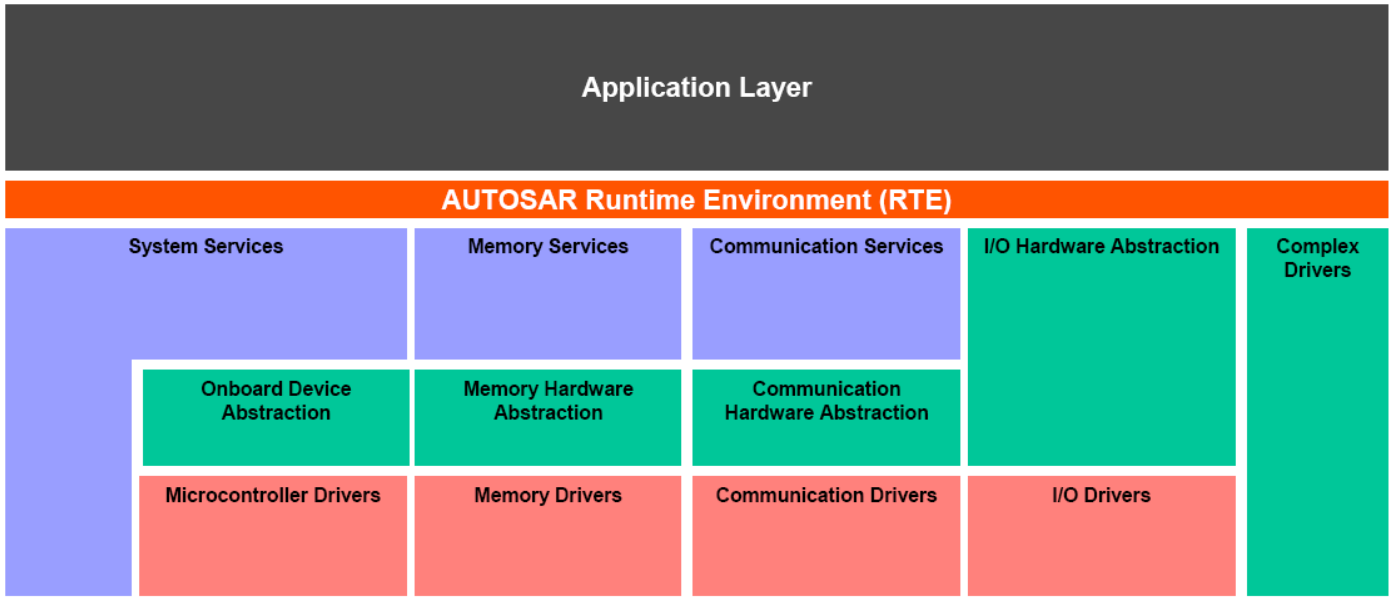
- Reduced complexity
- Improved testability
- Improved portability

## 5.3 Software Development without Target Hardware

The following figure shows a simple layer structure for real-time software:




A more complicated layer structure is given in the AUTOSAR specification:



## 5.3 Software Development without Target Hardware

For simple real-time systems, we do not use an operating system. Thus we have two layers:

- Application layer, containing most of the software
- Hardware abstraction layer (HAL), containing hardware drivers

The development process is thus:

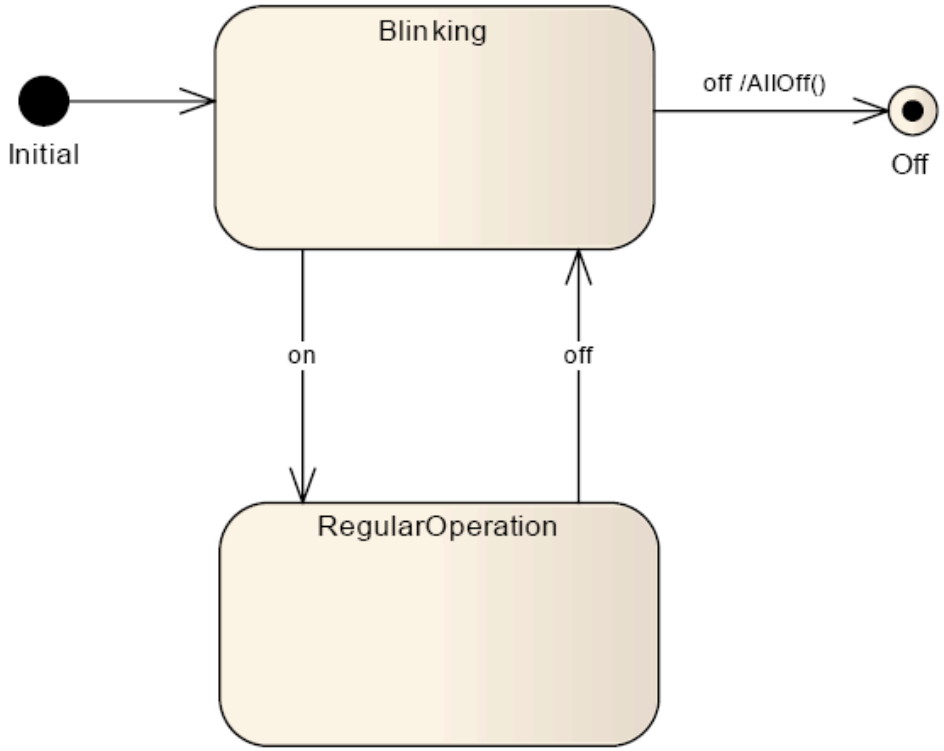
- Develop the application layer on a workstation. Make sure it can later be easily ported to the real hardware.
- Develop the hardware abstraction layer interface, i.e., the header files containing functions and definitions used by the application layer software.
- Develop the hardware abstraction layer software for the workstation, i.e., implement the functions defined in the HAL interface for the workstation environment.
- Develop the hardware abstraction layer software for the target hardware, i.e., implement the functions defined in the HAL interface for the real hardware.
- Integrate the application layer software and the hardware abstraction layer for the target hardware on the real target.

To develop the HAL for the real system we need the hardware.

To develop the application layer, we use the workstation and later port the software to the hardware.

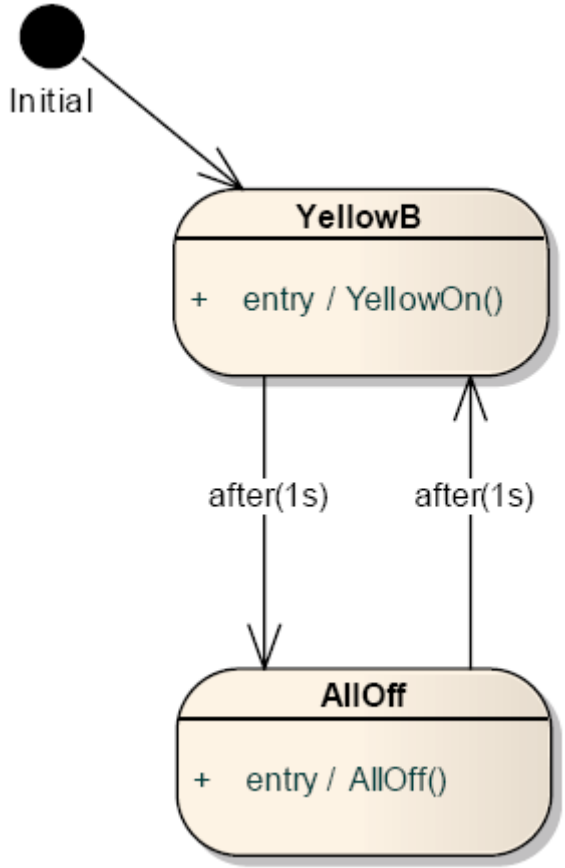
## Example: Traffic Light

Traffic light example from the section on UML state machines shall serve as an example on the software production process for simple real-time systems.



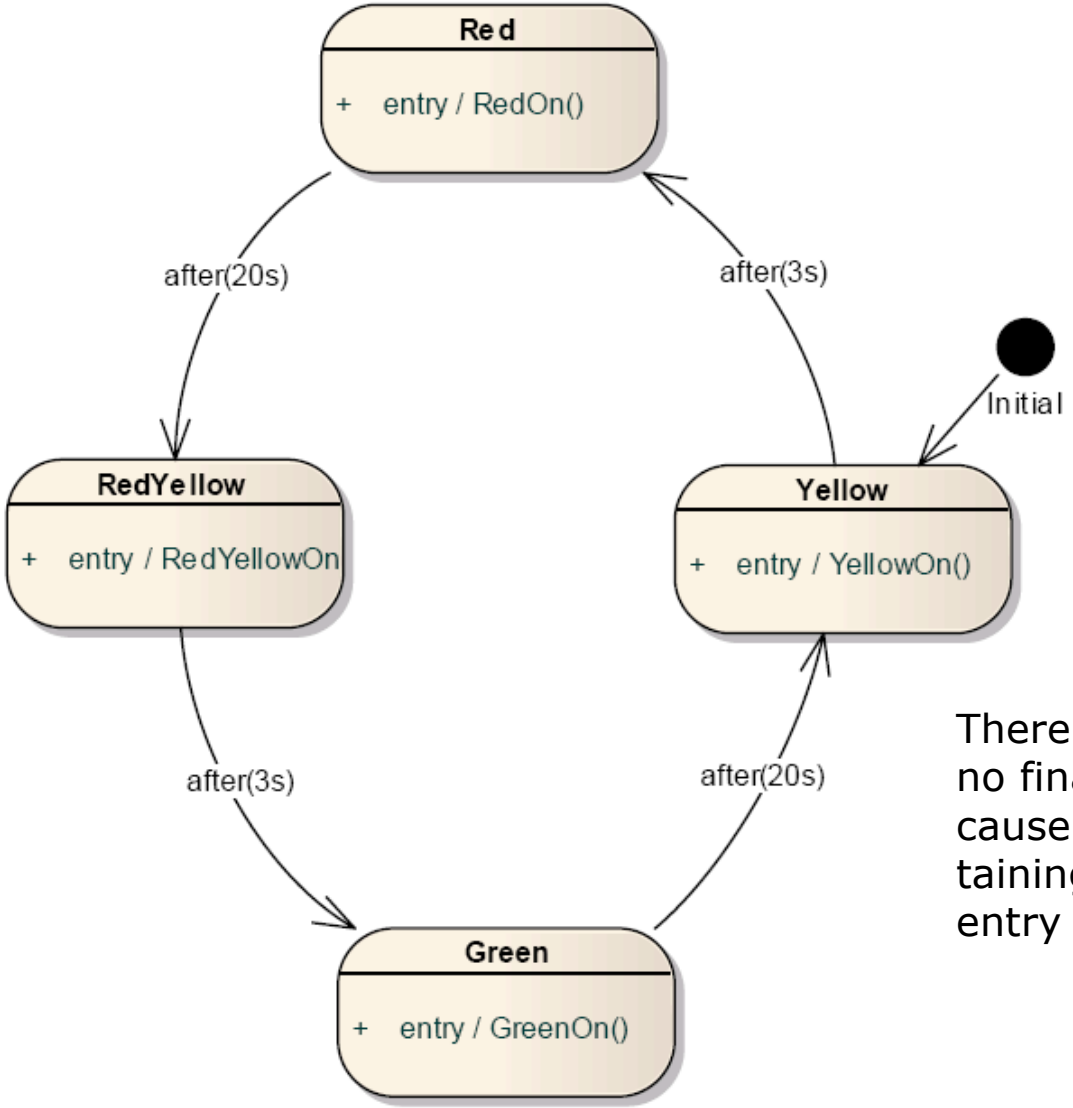

The traffic light shall be implemented on the Dragon12 hardware in the lab. We will use three LEDs and two buttons to simulate the user interface.

The Blinking substate machine looks like this:




There is no specific exit point or final state. This is because the superstate containing this substate defines exits to the RegularOperation state (on button pressed) and the Off state (off button pressed).

The RegularOperation substate machine looks like this:




There are no exit points and no final state. This is because the superstate containing this substate defines entry and exit conditions.

## 5.3 Software Development without Target Hardware

When we develop the software on the host (workstation), we have to simulate the hardware interface. For example, we have to be able to define at what time what event like a button press takes place.

We put these events into a file and during the execution of the software on the host read this file. A sample input file looks like this:

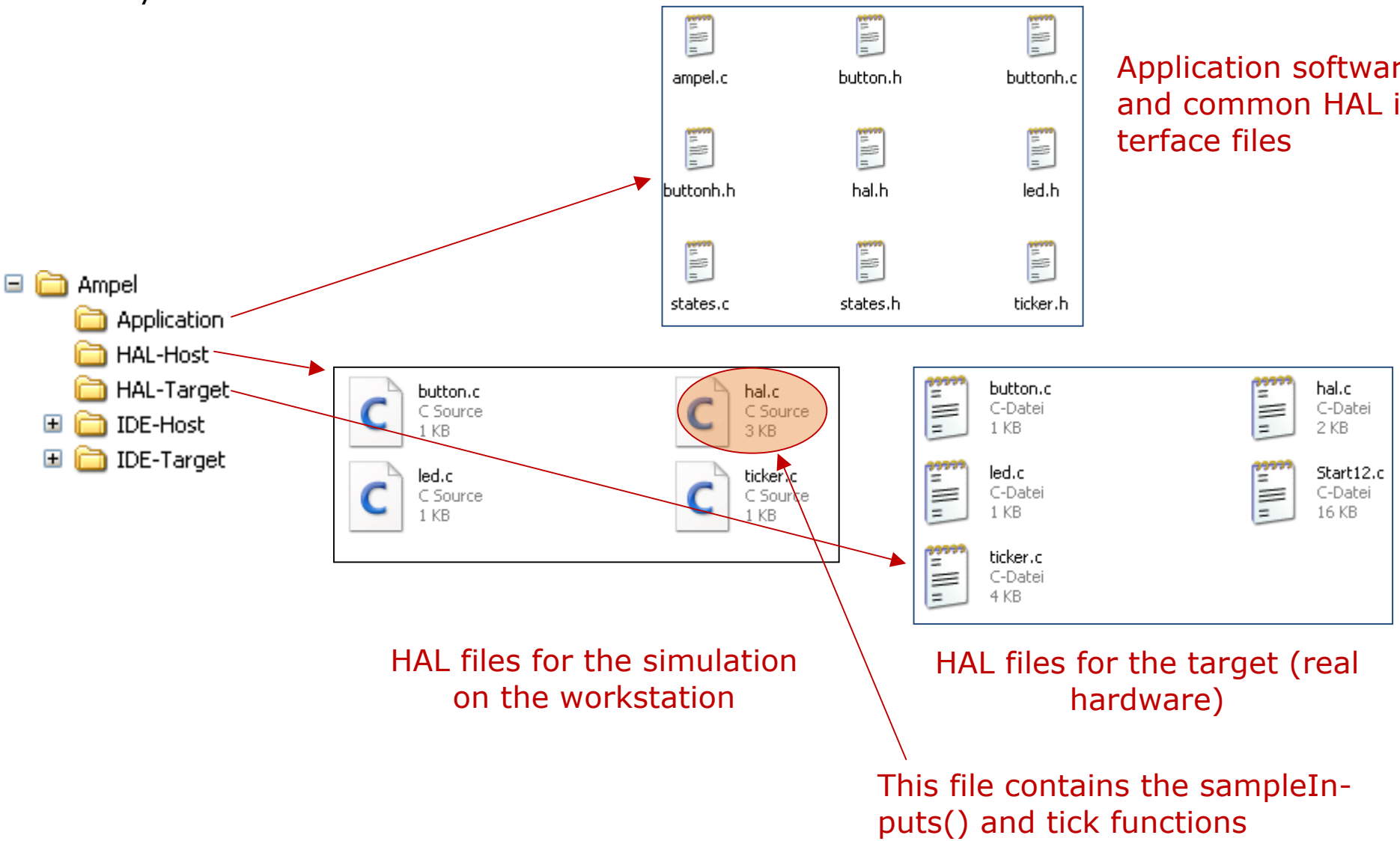
```
# advances ticker by 10 ticks (one tick = 10 msec)
+10
# presses "on" button for one tick (at time == 10)
on
# presses "off" button for one tick (at time == 11)
off
# advances ticker by 5 ticks to 16 (160 msec)
+5
# presses "off" button for 10 ticks (from time 16 until time 26)
off+10
```

The function that gets these inputs we put into the hardware abstraction layer software and we call it sampleInputs(void). It will be called from the main loop of our software.

In the hardware abstraction layer for our real target hardware this function polls the ports that are connected to the buttons.

When sampleInputs() reads "+10", it calls the tick() function 10 times to simulate 10 ticks. In our real system, the ticks would come from a hardware timer interrupt service routine.

Directory and file structure:



The main program (ampel.c):

```
/*
Real-Time Systems
(C) 2011 J. Friedrich
University of Applied Sciences Esslingen
Author: J. Friedrich, March 2011
*/
#include "ticker.h"
#include "button.h"
#include "hal.h"
#include "states.h"
/* A table with function pointers. Each state is represented by a single function */
/* It must be made sure that the order of the functions here correspond to the */
/* order of the state enumeration in the states.h header file. */
void (* stateTable[])()={ Red, RedYellow, Yellow, YellowB, Green, AllOff, Off };

void main( void ) {
   initHardware();
   initStateMachine();
   /* This is the heart of the state machine */
   for {
      sampleInputs(); /* reads input file (host) or polls buttons (target) */
      stateTable[ getCurrentState() ]();
   }
}
```

Hardware abstraction layer interface header file (hal.h):

```
/*
Real-Time Systems
(C) 2011 J. Friedrich
University of Applied Sciences Esslingen
Author: J. Friedrich, March 2011
*/
#ifndef _halh
#define _halh
/* Initialize the real hardware here */
void initHardware(void);

/* This we need to get inputs, either from events delivered by interrupts */
/* or polling the hardware, or by reading the input file during simulation */
void sampleInputs(void);

/* This is the exit for the simulation, and could possibly put the */
/* real system to sleep. */
void turnMachineOff(void);

/* This increments the real-world time on either the hardware or during simulation. */
/* In hardware, this is driven by a hardware timer interrupt. In the simulation this*/
/* is driven by the simulation input file. */
void tick(void);
```

Hardware abstraction layer interface continued:

```
/* As the name says, this is a very low level routine to get the time from the */
/* simulation input file. The real time is to be obtained from the ticker(.h) */
/* You can replace this with a dummy for the real system. */
unsigned long getTimeSimu(void);

/* Again, two routines solely for simulation. They simulate the two buttons, */
/* where the values are read from the simulation input file. For the real */
/* system these would be obtained from button(.h), and you could provide just */
/* dummies for these routines. */
char onDownSimu(void);
char offDownSimu(void);

/* Some more information on the simulation input file supported here:
# (as a first character in a line) makes this line a comment
# all text below has to be left aligned in a line
+10 advances ticker by 10 ticks
on presses "on" button for one tick (at time == 10)
off presses "off" button for one tick (at time == 11)
+5 advances ticker by 5 ticks to 16
off+10 presses "off" button for 10 ticks (from time 16 until time 26)
*/
#endif
```

We need two implementations for the HAL: one for the host and one for the target. Here are excerpts from the host version (hal.c):

```
void sampleInputs() {
   long delta = 0;
   char line[ BUFFERSIZE];
   char * tailptr;
   char * dum;
   if ( remainingTicks > 0) {
      tick();
      return;
   }
   if ( gets_s( line, BUFFERSIZE) == NULL) {
      exit( 0 );
   }
   if ( line[ 0] != '#') {
      /* advance ticker */
      if ( line[ 0] == '+') {
         tailptr = NULL;
         remainingTicks = ( int ) strtol( line, &tailptr, 0);
         return;
   }
   /* switch "on" pressed */
   if ( strstr( line, "on") == line) {
      remainingTicks = 1
      /* keep switch pressed for so many ticks */
      if (( tailptr = strstr( line, "+")) != NULL) {
         remainingTicks = ( int ) strtol( tailptr, &dum, 0);}
```

Here are excerpts from hal.c for the target:

```
#include <hidef.h>
#include <mc9s12dp256.h>
…
#pragma LINK_INFO DERIVATIVE "mc9s12dp256b"
/* Initialize the real hardware here */
void initHardware( void ) {
   initButton();
   initLed();
   initTicker();
   EnableInterrupts;
};
/* This we need to get inputs, either from events delivered by interrupts */
/* or polling the hardware, or by reading the input file during simulation */
void sampleInputs( void ) {
};
/* This is the exit for the simulation, and could possibly put the */
/* real system to sleep. */
void turnMachineOff( void ) {
};
/* This increments the real-world time on either the hardware or during simulation.*/
/* In hardware, this is driven by a hardware timer interrupt. In the simulation this
*/
/* is driven by the simulation input file. */
void tick( void ) {
};
```

This is the header file for the state machine (states.h):

```
/* Real-Time Systems
(C) 2011 J. Friedrich
University of Applied Sciences Esslingen
Author: J. Friedrich, March 2011 */
#ifndef _statesh
#define _statesh
/* All states as an enum */
typedef enum {RED=0, REDYELLOW, YELLOW, YELLOWB, GREEN, ALLOFF, OFF} states;
/* One function for each state */
void Red(void);
void RedYellow(void);
void Yellow(void);
void YellowB(void);
void Green(void);
void AllOff(void);
void Off(void);
/* Setter and getters for the current state */
states getCurrentState(void);
void setCurrentState(enum cs);
/* Setter and getters for the previous state */
/* to detect a state transition */
states getLastState(void);
void setLastStateState(enum cs);
/* Initialization of the state machine, go to initial state */
void initStateMachine(void);
#endif
```

This is the source file for the state machine implementation (states.c):

```
/*
Real-Time Systems
(C) 2011 J. Friedrich
University of Applied Sciences Esslingen
Author: J. Friedrich, March 2011
*/
#include "states.h"
#include "ticker.h"
#include "led.h"
#include "button.h"
#include "buttonh.h"
#include "hal.h"
static states currentState;
static states lastState;
typedef enum {NONE=0, YELLOWB2YELLOW=1, YELLOWB2OFF=2} transition;
static transition transitionActivity;
states getCurrentState() {
   return currentState;
}
void setCurrentState(states cs) {
   currentState = cs;
}
states getLastState() {
   return lastState;
}
```

Source for state machine (states.c) continued (1):

```
void setLastState(states cs) {
   lastState = cs;
}
void initStateMachine() {
   setLastState(OFF); /* force initial transition */
   setCurrentState(YELLOWB); /* next after initial state */
   resetTimer();
}

/* State YELLOWB */
void YellowB() {
/* three events are possible here:
- eventOn
- eventOff
- after(1s)
*/
   if (getCurrentState() != getLastState()) {
      resetTimer();
      setLastState(getCurrentState());
      transitionActivity = NONE;
      YellowOn(); /* onEntry activity */
   }
```

Source for state machine (states.c) continued (2):

```
    if (eventOn()) { /* reaction on event "on" */
    /* here we could place a guard condition */
    /* here we could prepare for a transition activity */
    /* transitionActivity = YELLOWBTOYELLOW; */
       setCurrentState(YELLOW); /* next state */
    }
    else if (eventOff()) {
    /* reaction on event "off" */
    /* transitionActivity = YELLOWBTOYELLOW; */
       setCurrentState(OFF); /* next state */
    }
    else if (getTimerValue() >= 1) {/* reaction of event "after(1s)" */
       setCurrentState(ALLOFF); /* next state */
    }
    if (getCurrentState() != getLastState()) {
    /* Here we could place the onExit activity */
    /* Here we execute the transition activities, if any */
       switch (transitionActivity) {
          case NONE: break;
          case YELLOWB2YELLOW: break;
          default: break;
       }
    }
}
```

Source for state machine (states.c) continued (3):

```
void AllOff() {
   if (getCurrentState() != getLastState()) {
      setLastState(getCurrentState()); /* eat up state transition */
      setAllOff(); /* onEntry activity */
      resetTimer();
   }
   if(getTimerValue() >= 1) {
      setCurrentState(YELLOWB); /* set next state */
   }
   else if (eventOn()) {
      setCurrentState(YELLOW); /* set next state */
   }
   else if (eventOff()) {
      setCurrentState(OFF); /* set next state */
   }
}
```

The other state functions are implemented in the same fashion.

Button handling, interface file (button.h):

```
/*
Real-Time Systems
(C) 2011 J. Friedrich
University of Applied Sciences Esslingen
Author: J. Friedrich, March 2011
*/
#ifndef _buttonh
#define _buttonh

/* Initialize the button hardware */
void initButton( void );

/* Low level routines which detect if the "on" or "off" buttons are pressed. */
/* There is no need for debouncing, this is handled elsewhere. */
char onDown( void );
char offDown( void );
#endif
```

Low level button handling for the real hardware (button.c):

```
#include "button.h"
#include "hal.h"
#include <mc9s12dp256.h>
void initButton() {
   DDRH = 0x00;
}
char onDown() {
   if ( PTH & 0x01) {
   /* if (~PTH & 0x01) { */ /* for real hardware inverted */
      return 1;
   }
   else {
      return 0;
   }
}

char offDown() {
   if ( PTH & 0x02) {
   /* if (~PTH & 0x02) { */ /* for real hardware inverted */
      return 1;
   }
   else {
      return 0;
   }
}
```

Low level button handling for simulation (button.c):

```
/*
Real-Time Systems
(C) 2011 J. Friedrich
University of Applied Sciences Esslingen
Author: J. Friedrich, March 2011
*/
#include "button.h"
#include "hal.h"
void initButton() {
}

char onDown() {
   return onDownSimu();  /* this function is defined in hal.c */
}

char offDown() {
   return offDownSimu(); /* this function is defined in hal.c */
}
```

Output in the simulation is handled by writing to stdout (led.c):

```
#include "led.h"
#include "ticker.h"
#include <stdio.h>
void initLed() {
}

void setAllOff() {
   printf( "%d: All off\n", getTime());
};

void RedOn() {
   printf( "%d: Red on\n", getTime());
};

void RedYellowOn() {
   printf( "%d: RedYellow on\n", getTime());
};

void GreenOn() {
   printf( "%d: Green on\n", getTime());
};

void YellowOn() {
   printf( "%d: Yellow on\n", getTime());
};
```

Output for the target (led.c):

```
#include <mc9s12dp256.h>
#include "led.h"

void initLed() {
   DDRJ_DDRJ1 = 1;
   PTJ_PTJ1 = 0;
   DDRB = 0xff;
   PORTB = 0x00;
}
void setAllOff() {
   PORTB = 0x00
};
void RedOn() {
   PORTB = 0x80;
};
void RedYellowOn() {
   PORTB = 0x80 | 0x40;
};
void GreenOn() {
   PORTB = 0x20;
};
void YellowOn() {
    PORTB = 0x40;
};
```

## Points to Remember