Semester Thesis

Implementing a Display Driver and a Console for TinyOS

Thibaut Britz
britzt@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
SA-2006-29 Summer 2006

Prof. Dr. Roger Wattenhofer
Distributed Computing Group
Advisor: Yves Weber

Contents

1 Introduction **2**

2 Display Driver **3**
2.1 Hardware . . . 3
2.2 Software . . . 3

3 Console **5**
3.1 Features in a Nutshell . . . 6
3.2 The `TerminalC` Component . . . 6
3.2.1 Sending Data . . . 7
3.2.2 Receiving Data . . . 7
3.3 The `ConsoleC` Component . . . 7
3.3.1 Supported Terminal Commands . . . 8
3.3.2 List Function . . . 8
3.3.3 Execute Function . . . 8
3.3.4 Special Commands . . . 8
3.3.5 Passing Parameters . . . 8
3.4 The `Console.pl` Script . . . 9
3.4.1 Supported Types and Functions . . . 10
3.5 Using the Console in your Module . . . 10
3.5.1 Example . . . 11
3.6 Performance and Possible Future Extensions . . . 11

A Overview of Component Layout **13**

B Interface Commands **14**
B.1 The `Terminal` Interface . . . 14
B.2 The `DisplayOutput` Interface . . . 14
B.3 The `DisplayPercentage` Interface . . . 15
B.4 The `DisplayLeds` Interface . . . 15
B.5 The `Console` Interface . . . 15

C TestConsole **17**
C.1 The `TestConsole` Interface . . . 17
C.2 The `TestConsoleC` Component . . . 17
C.3 The updated `TestConsole` Interface . . . 18
C.4 Updated `TestConsoleC` Component after applying the `Console.pl` script . . . 19

Chapter 1

Introduction

While there has been some progress on helping developers create TinyOS applications in the past [6], debugging running applications on the nodes is still a time consuming and error prone process. The average developer hasn't access to a JTAG adapter, so debugging over the JTAG interface is not an option. In addition most hardware platforms TinyOS runs on, only have 3 leds for debugging. There exists however a solution[1] for printing out text over the serial port, but this solution does not work on all platforms and blocks all other asynchronous events.

The first goal of this semester thesis therefore is to create a suitable display driver allowing us to display status information and text on an external display, without having to use any computer. This could for example be used to check the wireless link quality in the field without the need of having a computer around.
The second goal is to create a console which can be used to execute commands on the node, return status data, or simply to display text on the computer, helping the programmer to debug its applications
All two goals should be achieved independently of the platform TinyOS is running on. For the present implementation, all code has been developed and tested on the tinynode platform [2].

[1] It is possible to print out text to the serial interface by including a header file, whose code accesses the serial interface directly

Chapter 2

Display Driver

2.1 Hardware

The easiest way to attach a display to the tinynode is by using the UART interface of the node. After having searched for suitable displays, the decision was taken to order 3 displays (BPP-420L, BPP-440L and TRM-425L) from Seetron [3]. They have the advantage that they are unidirectional (simple protocol) and that they support a standard terminal interface (There is no need to transform the ascii bytes before sending them). According to their specification [5], they would work in combination with a tinynode.

At first I had some problems connecting the display to the tinynode, since the UART signal on the motherboard of the tinynode was inverted to the signal the display expected. (There was a note in the specification [2] that the UART RX and TX pins were directly mapped to the serial port, so I thought the signal at the motherboard would be at the same voltage as at the serial port). What I didn't know was that an inverter reversed the signal for the serial port. But it also turned out that this inverter is only active, when a device supplying power over the serial port is attached to the tinynode[1].
After beeing unsuccesfull with the first attempt of getting the display to work (soldering a resistor on the mainboard, feeding the inverter with power), I bought an inverter to reverse the UART signal on the motherboard, and could finally use the display without having to supply the tinynode with power with my computer.

2.2 Software

The display driver is composed of two components, entirely implemented in NesC, the language provided with TinyOS:

- the `TerminalC` component
- the `DisplayC` component

[1] This was done in order to reduce power consumption

In addition there are two components `DisplayPercentageC` and `DisplayLedsC` on top of the `DisplayC` component which generate a percentage bar and an array of leds on the display.

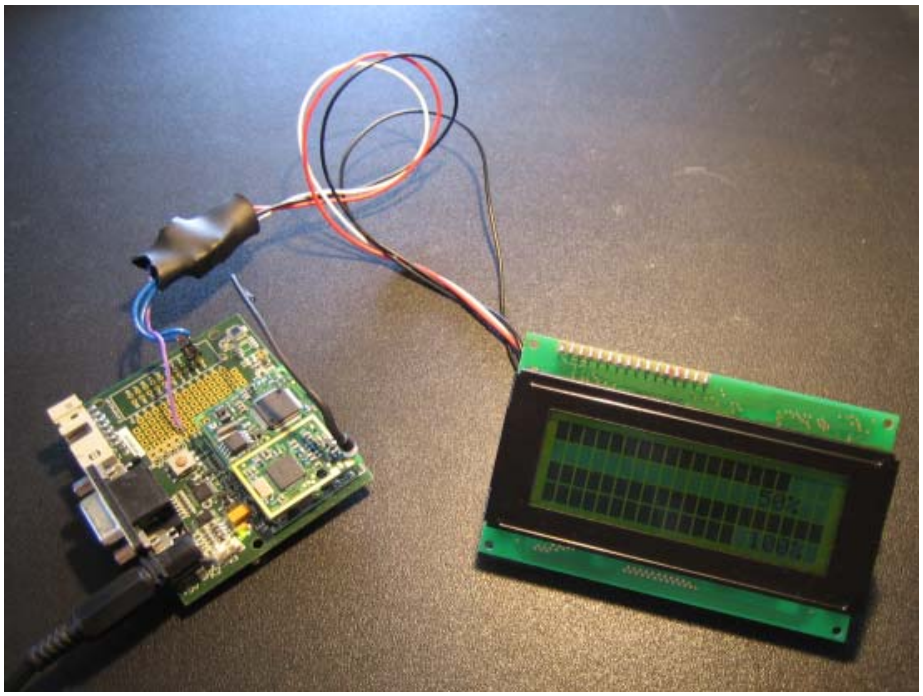

Figure 2.1: A tinynode and the attached display BPP-420L

Chapter 3

Console

The second goal of the semester thesis was to create a console which would be accessible over the serial interface and would allow to execute arbitrary commands on the tinynode, giving the opportunity to debug running applications and display status information.
There are a few alternatives to include commands, so they become executable over the serial interface:

- The calling code for each to be used procedure could be written manually. One advantage would be that the written code would be very small and very optimized, although it would have to be updated on every change made to the command (e.g. when the interface to the module would change). Another advantage is, that the code could be written in NesC only and there would be no need to use any external tools (e.g. Perl).
- Another possiblity would be to use an external script which parses the current module and its used interfaces for functions, and automatically generates calling code for the commands.
- Yet another possibilty would be to halt the make process after the to be compiled C code has been generated. We could parse this generated code and add the calling code for the functions there. We would have access to all the functions in all the modules, but we would need a performant parser to parse the entire code. Also the code generated could change after a version upgrade, and our implementation could break.

I chose the second method, since it's a more elegant solution to the problem and it reduces the amount of code needed to write (and to debug or maintain) to use the console in a module.

The console consists of three components:

- The `TerminalC` component, acting directly with the serial UART interface.
- The `ConsoleC` component, processing the received characters and executing the corresponding commands.
- The `Console.pl` Perl script which generates the calling code and the list code for the commands from the module using the console.

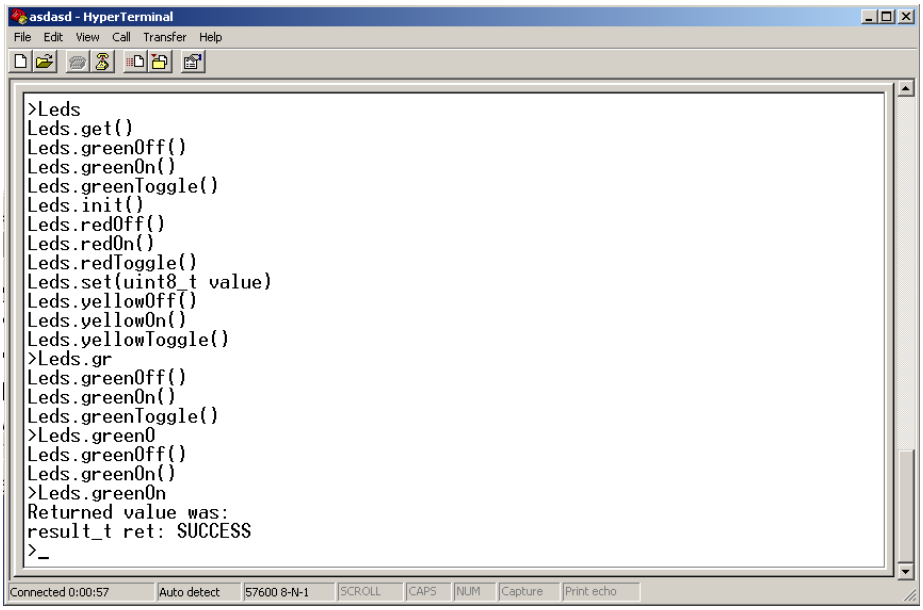

Figure 3.1: Listing the commands available on the tinynode and calling Leds.greenOn(). After calling the function, the return value is displayed

3.1 Features in a Nutshell

A short list of available features:

- All functions (commands, events, and tasks) defined by the module and its used interfaces are automatically included into the console.
- Functions can be explicitely included or excluded to reduce code size.
- Global variables can be displayed.
- There is practically no interference with other asynchronous events. The asynchronous code is kept as small as possible.
- Parameters can be passed to functions if required.
- Commands can be executed by using a VT100 compliant software (e.g. Windows' HyperTerminal).

Figure 3.1 shows the console in action.

3.2 The `TerminalC` Component

The terminal component acts directly with the serial UART interface and contains the functions for printing the standard NesC types. This component can also be used if we were solely interested in outputing text to the serial interface.

3.2.1 Sending Data

At first, I wanted to implement a print function similar to the `printf` function from the standard input/output library. There was however the problem, that the TinyOS language definition (NesC) does not allow to pass a non defined number of parameters (at compile time) through an interface. The number of parameters passed to the function needs to be known at compile time.
Thus, I chose a less elegant solution, implementing a separate print function for each standard type, trying to reuse as much code as possible.
Because the Terminal interface should be non-blocking, all charachters are first written into a local circular buffer. The buffer is then emptied on a byte by byte basis by an asynchronous event which is triggered as soon as a byte has been sent successfully over the serial interface. That way, the asynchronous code is kept as small as possible, so it does not interfere with other asynchronous events.

3.2.2 Receiving Data

Received bytes are forwarded bytewise to an event handler, which has to be implemented by the module using the component's interface.

3.3 The `ConsoleC` Component

The Console component processes the bytes received through the Terminal component. Depending on the byte received, one of the following actions is executed:

- the input buffer (containing the actual entered text in the command) is updated.
- a command is executed.
- matching commands are listed.
- the byte is simply written out.
- the terminal screen is updated (this happens if there has been a change at the non end of the input buffer).

Again, since we want to keep the asynchronous code small, the received bytes are first written into a queue buffer which is then processed bytewise by a task, which then executes the appropriate action. When the queue is full, the byte is simply discarded (this only happens when the node is really busy handling lots of asynchronous events, leaving no time to handle non asynchronous tasks).
Due to the single-threaded design of TinyOS, the main application running on the tinynode has to be divided into short tasks, such that the console task gets a chance to run. If this is not the case, the console will not work.
A workaround would be to place the processing and the calling functions into the asynchronous event. This has however one main disadavantage: While a character is being processed or a command is being invoked and executed, all other interrupts are disabled. This will cause timing issues and the print buffer will never get emptied (so after the buffer is full, all following text would be discarded). Therefore I selected the first approach, dividing interrupt and action.

3.3.1 Supported Terminal Commands

The console implements a VT100 Terminal server [7] and supports the following commands:

- Cursor keys: they move the current position in the input buffer.
- Backspace key: deletes one character from the current position in the input buffer.
- Tab: lists all the matching commands. (Calls the list function)
- Enter: executes a command. (Calls the execute function)

3.3.2 List Function

The list function, as the name suggests, lists all the matching commands and updates the input buffer.
The input buffer is being parsed, and all the matching commands[1] are printed out. In addition the input buffer is also updated with the remaining characters of the command string, if any left. (e.g. if the input buffer consists of `"_sta"` and the only matching command is `"_status"`, the input buffer is updated to `"_status"`).
The list function is automatically generated by the `Console.pl` Perl script and added to the main module.

3.3.3 Execute Function

The execute function executes a command. The `input buffer` is being parsed for input variables (see section 3.3.5), which are then automatically converted into the corresponding types. Then the appropriate command is called, using the freshly parsed variables. If the command delivers a return value, the value is printed out as well. The execute function is automatically generated by the `Console.pl` Perl script and added to the main module.

3.3.4 Special Commands

The command `"_status"` is a special command which lists all global variables in the module. Like the list and the execute function, this function is automatically generated by the `Console.pl` script.

3.3.5 Passing Parameters

Some functions require parameters. Parameters are passed by separating them by spaces and enclosing them into quotation marks (").
If not enough parameters are provided, the values will be undefined[2].

[1] The input string is compared one by one to the available commands. This could be done in a more efficient way by creating a compare tree which would reduce the amount of compares. This approach however has been left out for the sake of simplicity.

[2] Depending on the version of the compiler, the variables are initialized to 0 or just stay undefined. As a consequence, one can not assume them to be 0 since the default behavior of the compiler could change in future versions.

If a struct is being passed to a function, the variables contained in the struct have to be passed in order of their appearance in the struct definition. Structs containing structs are ignored to avoid recursion.
Pointers are always passed as an integer containing the address where the pointer is pointing to. One exception is the pointer to a string, where the actual string is passed as parameter and not the pointer to the string. This can easily be changed if required.

Example:
`Calculator.add "1" "2"` would execute the add-function (expecting two parameters) of the Calculator module with two parameters, `1` and `2`. If one additional parameter is provided, it will simply be discarded, since `Calculator.add` only expects two parameters. If only one parameter is supplied, the second parameter is left undefined, potentially leading to non-deterministic behaviour.

3.4 The `Console.pl` Script

The `Console.pl` script parses the module and generates the necessary calling and listing code for the console. I chose to implement this in Perl, since Perl has a great flexibility in parsing and manipulating text files.

The Perl script consists of these steps:

1. Parse the command line options. Map all files to their position in the file system.
2. Scan the main module for used interfaces.
3. Parse the main module for global variables and generate new types.
4. Get the functions from the used interface files
5. Get the functions from the main module.
6. Generate the calling and list code and insert it into the corresponding files.

The commands and variables are extracted by regular expressions. Another possibility would be to create a parser on basis of the Backus-Naur form of the NesC language definition. But since we are only interested in commands and global variables (which are only found at well-defined positions in the code) and not interested in the rest of the source code, I chose to extract the data by using regular expressions.
Regular expressions are faster to implement and easier to use, but the risk of missing valid code is much bigger (e.g. not having thought of all the valid cases).
At the moment, only the used interface files and the main module are scanned for functions. In addition the main module is also scanned for global variables.
To include all functions used in a module and its submodules, the script could be extended to recursively visit all the used interfaces and either:

- apply the console code to the interfaces and call the functions from there. (If an event handler is implemented by multiple modules, all of the implementations will be called)

- or include all found interfaces in the main module and call the functions from there.

I did however chose not to do this since this would increase the size of the generated code considerably. An interface can still be included manually, if access to a layer below would be required.

3.4.1 Supported Types and Functions

- All standard types (`int`, `char`, ...), `"normal" structs` and `"typedef" structs` are supported. `Unions` however are not supported, but could be added in a later version. As a workaround in the current version, the programmer could simply rewrite them as structs.
- `Events`, `tasks` and `commands` are extracted from the interfaces used. Remapping of interfaces (e.g. `uses Leds as LedsArray;`) is also supported, except for the console interface itself.
- `Commands`, `tasks`, normal `functions` and global `variables` are extracted from the main component. (`events` can be neglected since they are already defined in the interface definition)

One additional requirement is that the main module follows the TinyOS naming convention, i.e. the Main module has to end on *M.nc. Additional types with different input and output behavior can easily be added, if required.

3.5 Using the Console in your Module

First make sure that your programm is devided into multiple tasks, or otherwise the console will not work correctly. Then do the following 4 steps:

Edit the interface file

Add the interface `"ConsoleC"` to the list of used interfaces, and update the module's interface wiring.

Edit the main component

Add the Console to the list of used Interfaces (**no remapping of the Console interface is allowed**) and add `"call Console.init()"` to the initialization code of your module. The serial console will not work if the Console module is not initialized.

Apply the `console.pl` script on your Module

One possibilty is to pass the module name and the library path (where the used interface files reside) on the command line as first and second parameter to the script.
Another possibility is to edit the `console.pl` script (at the top of the script) and add the module name and the path to the included libraries at the top of the script.

You can also explicitely allow or disallow functions that will later be added to the console, thereby reducing code size.
Rerunning the `console.pl` script on the same module will update the generated code from the last run, in the module.

Connect to the node

Open a terminal connection to the node, and press `tab` to see the list of available commands. (e.g. using Hyperterminal in Windows).

3.5.1 Example

You can have a look at the included `TestConsole` module to see a working example of the console. This module is found on the cd in the `/source/ConsoleTest` directory.

3.6 Performance and Possible Future Extensions

The console implementation poses no additional computation overhead if the console is compiled in, but never used. Also all the asynchronous calls are kept very small to not interfere with other asynchronous calls suggestible to timing issues. However, the generated code size by the Perl script is very big if no commands are explicitly excluded. This reduces the code size significantly. From a memory point of view (RAM), the console only needs about 500 bytes at run time (for the input and output buffer).

Possible future extensions include:

- Multi-hop support. The console could be installed on multiple nodes with one node being connected through the serial interface to the computer. Then that node, instead of executing the command, would forward the command to the addressed node which would execute the commandd (e.g. by specifying the node ID in front of the command), and wait for the returned value, which then will be forwared through the wireless interface and the serial interface to the computer. This could be achieved by adding a multi-hop interface layer similar to the serial interface layer.

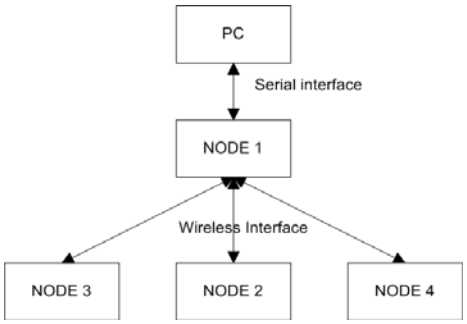

Figure 3.2: Node layout

- Support for unions. Unions are rarely used since the same functionality can be achieved with structs. They could be easily added by modifying the parser in the `Console.pl` perl script.
- Recursively update multiple interfaces, so it is possible to execute every command, even the one on the lowest layer. However, this feature is barely required, since access to lower layers is generally not needed and including lower layers would cause the generated code to become huge.
- Generate an optimized compare tree to find the appropriate list and call functions. The actual implementation uses O(n) time in worst case to find the correct procedure.

Appendix A

Overview of Component Layout

Figure A.1 shows the component layout of the console and the display driver.
`HPLUARTC` is the TinyOS serial interface layer, providing a function to send a byte and a receive byte event, which is triggered when a byte is received.
The `TerminalC` component uses the `HPLUARTC` component and contains the printing functions and forwards the receive byte event to the compononents using its interface.
The `DisplayOutputC` and the `ConsoleC` component both make use of the `TerminalC` component to receive bytes (only the `ConsoleC` component) and to print out text to the console or to the display.

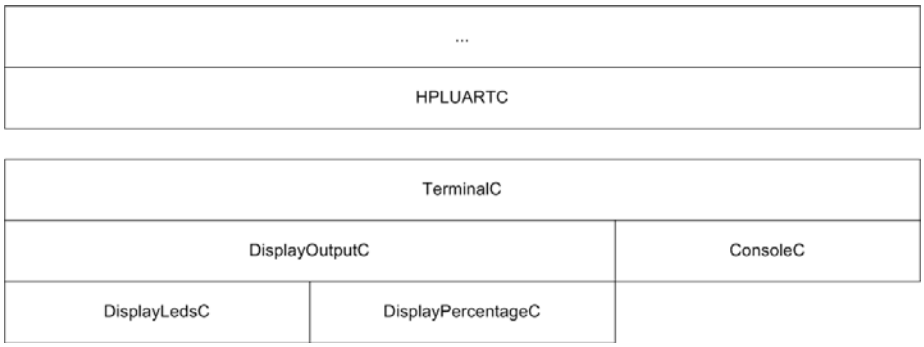

Figure A.1: Overview of components

Appendix B

Interface Commands

This chapter contains a list of all functions provided by the different interfaces. Since the function names are pretty descriptive, no additional details are given here, but can be found in the source code.

B.1 The `Terminal` Interface

```
command result_t init();
command result_t start();
command result_t stop();
command result_t printnewline();
async event result_t get(uint8_t data);
command result_t print(char *msg);
command result_t printchar(char c);
command result_t printbool(bool in);
command result_t printuint16_t(uint16_t i);
command result_t printuint8_t(uint8_t i);
command result_t printuint32_t(uint32_t i);
command result_t printint16_t(int16_t i);
command result_t printint8_t(int8_t i);
command result_t printint32_t(int32_t i);
command result_t printint(int i);
command result_t printresult_t(result_t t);
```

B.2 The `DisplayOutput` Interface

```
command result_t init();
command result_t start();
command result_t stop();
command result_t cleardisplay();
command result_t setPosition(uint8_t data);
command result_t replaceCharAtPosition(uint8_t position, char c);
command result_t clearLine(uint8_t position);
command result_t backLightOff();
command result_t backLightOn();
```

```
command result_t buzzer();
command result_t bigCharsOn();
command result_t bigCharsOff();
command result_t defineCustom(uint8_t pos, uint8_t b1,uint8_t b2,uint8_t b3,...
uint8_t b4,uint8_t b5,uint8_t b6,uint8_t b7,uint8_t b8);
command result_t printCustom(uint8_t pos);
command result_t restoreCustom();

  /*
  * Printing
  */
command result_t print(char *msg);
command result_t printchar(char c);
command result_t printnewline();
command result_t printbool(bool in);
command result_t printuint16_t(uint16_t i);
command result_t printuint8_t(uint8_t i);
command result_t printuint32_t(uint32_t i);
command result_t printint16_t(int16_t i);
command result_t printint8_t(int8_t i);
command result_t printint32_t(int32_t i);
command result_t printint(int i);
command result_t printresult_t(result_t t);
```

B.3 The `DisplayPercentage` Interface

```
command result_t init();
command result_t start();

command result_t stop();
command result_t setstartstop(uint8_t position, uint8_t s, uint8_t t);
command result_t setpercentage(uint8_t position, uint8_t value);
command result_t hide(uint8_t position);
command result_t show(uint8_t position);
```

B.4 The `DisplayLeds` Interface

```
command result_t init();
command result_t start();
command result_t stop();
command result_t Toggle(uint8_t position);
command result_t On(uint8_t position);
command result_t Off(uint8_t position);
```

B.5 The `Console` Interface

```
command result_t init();
command result_t start();
```

```
command result_t stop();
event result_t execute_command(char *buf);
event result_t list_commands(char *buf,uint8_t *stop);

  /*
  * Printing
  */
command result_t print(char *msg);
command result_t printchar(char c);
command result_t printbool(bool in);
command result_t printuint16_t(uint16_t i);
command result_t printuint8_t(uint8_t i);
command result_t printuint32_t(uint32_t i);
command result_t printint16_t(int16_t i);
command result_t printint8_t(int8_t i);
command result_t printint32_t(int32_t i);
command result_t printint(int i);
command result_t printresult_t(result_t t);
command result_t printnewline();
```

Appendix C

TestConsole

This chapter contains an example on how to use the console. `TestConsole` is a simple module using only the `LedsC` component on which the `Console.pl` perl script is applied.

C.1 The `TestConsole` Interface

```
configuration TestConsole {
}
implementation {
components Main, TestConsoleM, LedsC;

TestConsoleM.Leds -> LedsC;

Main.StdControl -> TestConsoleM.StdControl;

}
```

C.2 The `TestConsoleC` Component

```
module TestConsoleM {
provides {
interface StdControl;
}
uses {
    interface Leds;
  }
}
implementation {

struct test {
int a, *b;
int **c;
int d;
struct test *pointertostruct;
```

```
};
struct test struct_a;
int x;

command result_t StdControl.init() {
call Leds.init();
struct_a.a = 10;
struct_a.b = &struct_a.a;
struct_a.c = &struct_a.b;
struct_a.pointertostruct = &struct_a;
struct_a.d = 20;
x = 30;
return SUCCESS;
}
command result_t StdControl.start() {

    return SUCCESS;
  }
command result_t StdControl.stop() {
    return SUCCESS;
  }

void turnallon()
   {
   call Leds.redOn();
   call Leds.yellowOn();
   call Leds.greenOn();
   }

/*
Simple function to demonstrate the parameter passing of structs
*/
struct test* dostruct(int k, struct test instruct)
   {
   struct_a.a = k;
   struct_a.d = instruct.a;
   return &struct_a;
   }


}
```

C.3 The updated `TestConsole` Interface

The wiring of the interfaces and the used componentes have been updated by hand.

```
configuration TestConsole {
}
implementation {
```

```
components Main, TestConsoleM, ConsoleC, TerminalC, HPLUARTC, LedsC;

TestConsoleM.Console -> ConsoleC;
TestConsoleM.Leds -> LedsC;

Main.StdControl -> TestConsoleM.StdControl;

ConsoleC.Terminal -> TerminalC;
TerminalC.HPLUART -> HPLUARTC;

}
```

C.4 Updated `TestConsoleC` Component after applying the `Console.pl` script

Have a look at section 3.5 for more details on what steps have been executed. The generated module contains about 1600 lines of code and is therefor too long to be listed here. It is located in the directory `/source/ConsoleTest` on the cd.

Bibliography

[1] Distributed Computing Group
http://www.dcg.ethz.ch

[2] Tinynode specification sheet
http://www.tinynode.com/uploads/media/TinyNode_Users_Manual_rev11.pdf

[3] Seetron
http://www.seetron.com/slcds.htm

[4] Bpp 420 Display
http://www.seetron.com/bpp420_1.htm

[5] RS232 Input signal of the display
http://www.seetron.com/ser_an1.htm

[6] TinyOS Plugin for Eclipse
http://www.dcg.ethz.ch/ rschuler/

[7] ANSI/VT100 Terminal control
http://www.termsys.demon.co.uk/vtansi.htm