Assignment week 1.3: remote login

Requirements

- A laptop computer
- MPLAB 8.8x
- C32 C-Compiler
- The Cerebot MX4cK board
- A terminal emulator, such as putty (available via N@Tschool)
- A servo motor
- Optionally: an extra USB A to micro USB B cable
- The comDriver from the software folder in our N@Tschool entry

Goals

In this week's assignment we are going to expand the assignment of week 1.2. We will use the UART/USB connection option (enabled by the FT232 IC) of the Cerebot board to control the servo via a terminal application on a PC. We will not change the system of last week, except for the servocontrol block, and add new blocks, *uart* and *shell*[1], for communicating with the terminal. We end up with the following requirements:

1. The system works as specified in assignment 1.2
2. Additionally, a connection with a terminal can be established through a UART connection
3. The settings of the UART connection are 9600 baud, 8 data bits, 2 stop bits, and 1 parity bit (even)
4. After startup, the microcontroller prints the following text to the UART connection:

```
embc shell
==========
embc>
```

   where the last line is a command prompt.
5. The user can enter one simple commands (where <value> is an integer value between -45 and 45):

```
setangle <value>
getangle
```

6. The first command will position the servo at the desired angle. After the servo has been set, the shell responds with the following text:

```
servo set to <value> degrees
```

[1] "A shell in computing provides a user interface for access to an operating system's kernel services." http://en.wikipedia.org/wiki/Shell_(computing)

7. The second command will show the current angle of the servo. It will be shown as follows:

   ```
   current angle: <value> degrees
   ```

8. Any other command will result in the following response from the shell:

   ```
   syntax error
   ```

9. After a command has been handled, the shell will transmit a new prompt:

   ```
   embc>
   ```

The software design is given in Figure 1. We will have to create or modify the grey blocks ourselves. The top-level block contains the main function and controls the servo, the EEPROM storage and the terminal communication.

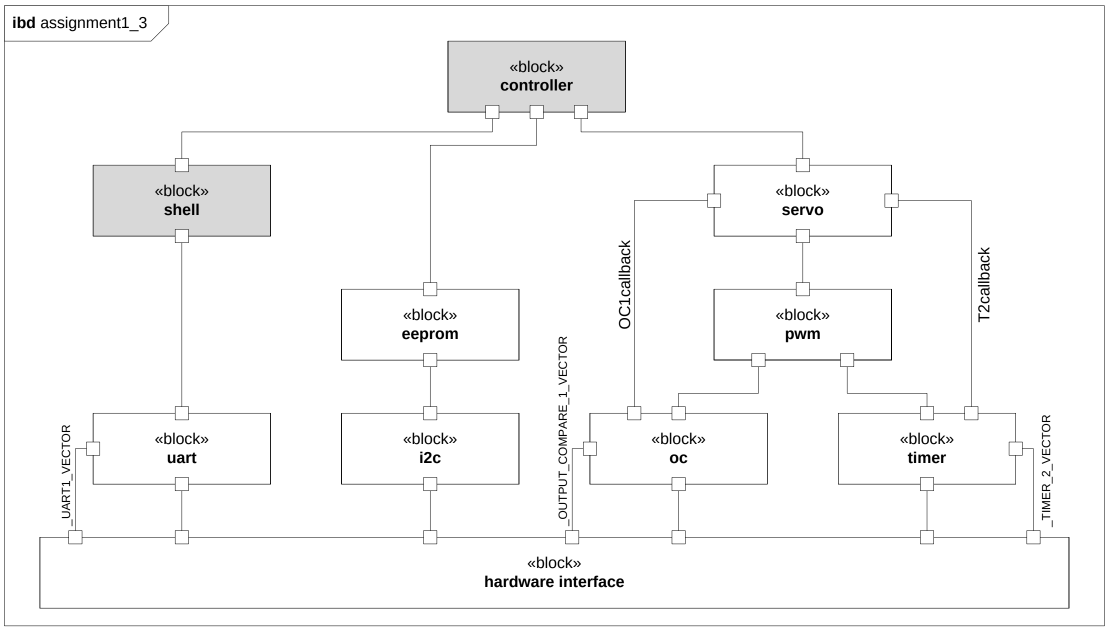


Figure 1: The internal block diagram of the software design of the assignment of week 1.3

Step 1 – creating the project and selecting the files

Start by downloading and unzipping the file *assignment1_3.zip* from our N@Tschool entry. In the folder *embc* on your drive, create a new folder named *assignment1_3*. Copy the required files from assignment 1.2 into the folder structure. Place the *uart.c* and *uart.h* files from the unzipped folder in the correct project folders.

Create a new project called *assignment1_3* by opening MPLAB by following the Wizard, which you can find in the menu Project → Project Wizard…

Follow the same steps as given in the description of assignment 1.1.

Step 2 – creating the terminal block

Create *shell.c* and *shell.h*, and add/copy the usual file information.

Implement 3 functions:

```
/* ======================================================================== */
/* function: shell_init( fpb );                                             */
/*                                                                          */
/* description: initialises the shell module.                               */
/*                                                                          */
/* pre:  fpb          - peripheral bus clock frequency in Hz                */
/*                                                                          */
/* post: return value - 0 if initialisation failed                          */
/*                      1 if initialisation succeeded                       */
/* ======================================================================== */
unsigned char shell_init
(
    unsigned int fpb
);

/* ======================================================================== */
/* function: shell_getcommand( command );                                   */
/*                                                                          */
/* description: checks whether the terminal has received a command. If a    */
/*              command is received, it will return a 1, and the command will */
/*              be placed in the command string.                            */
/*                                                                          */
/* pre:  command      - pointer to the first element of the string where the  */
/*                      command needs to be placed in.                      */
/*                                                                          */
/* post: return value - 0 if no command is available                        */
/*                      1 if a command was received, the command will be    */
/*                      placed in the command string                        */
/*       command      - will hold the command string if the return value is 1 */
/* ======================================================================== */
unsigned char shell_getcommand
(
    unsigned char *command
);

/* ======================================================================== */
/* function: shell_sendmessage( message );                                  */
/*                                                                          */
/* description: transmits a message, stored as a string in the message      */
/*              variable, to the terminal. The message will be terminated   */
/*              with a newline: "\n\r".                                     */
/*                                                                          */
/* pre:  message      - pointer to the first element of the string to be    */
/*                      transmitted to the terminal                         */
/*                                                                          */
/* post: return value - 0 if transmission failed                            */
/*                      1 if transmission was successful                    */
/* ======================================================================== */
void shell_sendmessage
(
    unsigned char *message
);
```

The initialisation function sets up the UART connection.

Implement the *shell_getcommand* function as a polled state machine. The function will return a 1 when the command has been received completely. In that case, the command variable will hold the command. The controller module needs to make sense of it and provide the right response.

Whenever the controller wants to send a message to the terminal (for example for returning the angle of the servo), it can make use of the *shell_sendmessage* function.

Implementation information

- The connection to the PC is realised via UART1.
- A command entered in the terminal will be submitted by pressing the carriage return (enter key), which can be detected as the character '\r'.
- To move the cursor to the beginning of the next line on the terminal, write a "\n\r" to the UART.
- Use *strcpy* (#include <string.h>) to copy the incoming command to the returning argument. Use Google for finding the prototyping of the function.

Step 3 – creating the controller block

Start by creating *controller.c* and *controller.h* in the right directories. Write the main function, using the requirements as a lead. It is allowed to create extra blocks and/or extra connections, for example if you need an extra timer in the controller block.

Implementation information

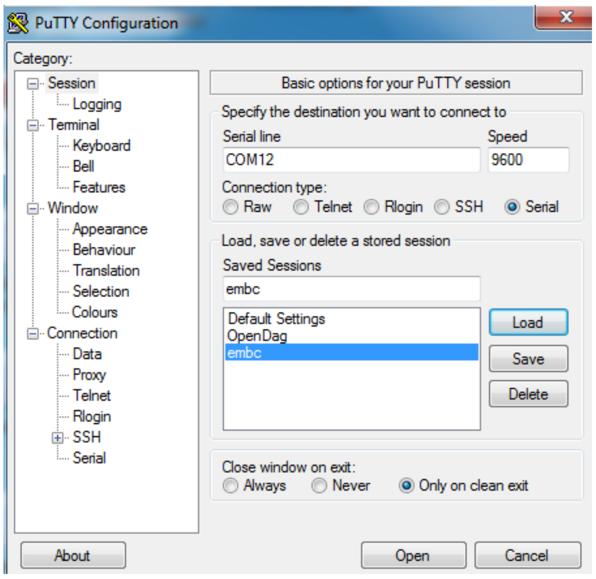
- You might need the functions *strncat* and *strcmp*, or *strncmp* for which you will need to #include <*string.h*>. Use Google for finding the prototyping of the functions.
- Converting a string value to an integer can be done by using the function *atoi*.
- For more information on manipulating strings: http://en.wikibooks.org/wiki/C_Programming/Strings

Step 4 – setting up and running putty

Run putty. Find the serial communication port to which the USB driver is connected (in Figure 2 it is COM12). Select *Serial* and enter 9600 as the speed of the connection. Next, select *Serial* in the left table, in the *Connection* list.

In the screen that follows (see Figure 3), adjust the settings for the number of data bits, number of stop bits, parity and flow control. Click *Open* to start the terminal session.

In the terminal settings (Figure 4), change *Local echo* to *Force on*.



Basic options for your PuTTY session

Specify the destination you want to connect to

Serial line: COM12
Speed: 9600

Connection type: ( ) Raw ( ) Telnet ( ) Rlogin ( ) SSH [✓ Serial]

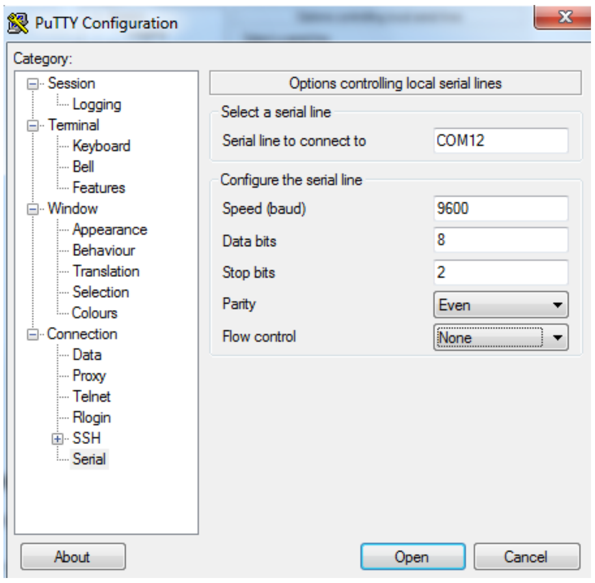
Load, save or delete a stored session

Saved Sessions: embc

Default Settings
OpenDag
embc

Close window on exit: ( ) Always ( ) Never [✓ Only on clean exit]

Figure 2: The first configuration screen of putty



Options controlling local serial lines

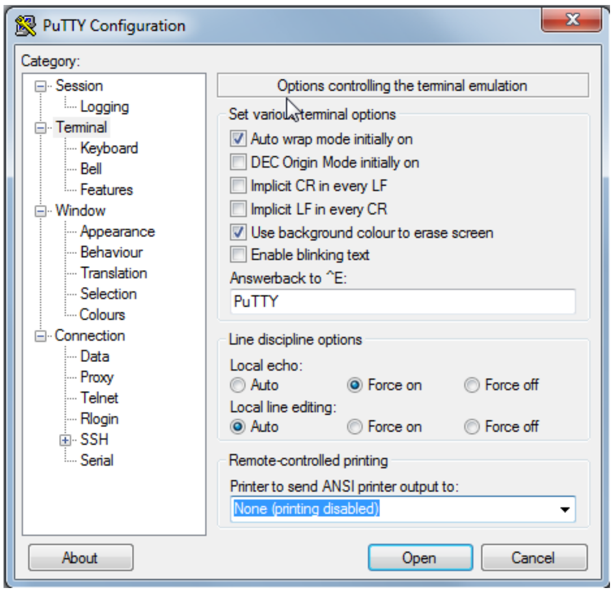
Select a serial line

Serial line to connect to	COM12

Configure the serial line

Speed (baud)	9600
Data bits	8
Stop bits	2
Parity	Even
Flow control	None

Figure 3: The configuration screen for the serial connection



Options controlling the terminal emulation

Set various terminal options
- [✓] Auto wrap mode initially on
- [ ] DEC Origin Mode initially on
- [ ] Implicit CR in every LF
- [ ] Implicit LF in every CR
- [✓] Use background colour to erase screen
- [ ] Enable blinking text

Answerback to ^E: PuTTY

Line discipline options

Local echo: Auto [✓ Force on] Force off

Local line editing: [✓ Auto] Force on Force off

Remote-controlled printing

Printer to send ANSI printer output to: None (printing disabled)

Figure 4: The configuration screen for the terminal behaviour. Change Local echo to Force on.

Step 5 – testing the system

Test the system by checking the requirements according to the following table:

requirement	description	pass/fail
1	The system works as specified in assignment 1.2	
2	Additionally, a connection with a terminal can be established through a UART connection	
3	The settings of the UART connection are 9600 baud, 8 data bits, 2 stop bits, and 1 parity bit (even)	

4	After startup, the microcontroller prints the following text to the UART connection:
	
	`embc shell`
	`==========`
	`embc>`
	
	where the last line is a command prompt.	

5	The user can enter two simple commands (where <value> is an integer value between -45 and 45):
	
	`setangle <value>`
	`getangle`	

6	The first command will position the servo at the desired angle. After the servo has set, the shell responds with the following text: `servo set to <value> degrees`	
7	The second command will show the current angle of the servo. It will be shown as follows: `current angle: <value> degrees`	

8	Any other command will result in the following response from the shell: `syntax error`	
9	After a command has been handled, the shell will transmit a new prompt: `embc>`	