ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE
BIOROBOTICS LABORATORY

Internship Report

Firmware development for Roombots

Student:
Xinkui FENG
Université Paris-Sud XI

Supervisors:
Rico Möckel
Stéphane Bonardi
EPFL BioRob

Professor:
Auke Jan Ijspeert
EPFL BioRob

Index

Acknowledgement

I want to give my special thanks to:

My supervisors Rico Möckel and Stéphane Bonardi who have spent much time on my works and helped me to revise my report.

Alessandro Crespi and Alexander Sprögwitz who have supported my works in the informatics field and mechanic field respectively.

Jesse van den Kieboom and Mostafa Ajallooeian who have given me suggestions and advices in CPG's part.

The secretary Sylvie Fiaux and the technician André Baderscher who have also given me a great help.

And Professor Auke Jan Ijspeert who has given me this precious opportunity to do this internship at BioRob Lab of EPFL.

I Introduction

1. EPFL and Lausanne

Lausanne is a city in Romandy, the French-speaking part of Switzerland, and is the capital of the canton of Vaud. Its history can be traced back to the Roman Empire when the city was first built up as a military camp. The city is situated on the shores of Lake Leman, and the Alps Mountain can be seen from there. The population of the city is over 122,000, and its urban area is almost 41.37 km^2. Also, the headquarter of the International Olympic Committee is located in Lausanne – the IOC officially recognizes the city as the Olympic Capital [1].

Figure 1 Lausanne, the Lake Leman and the Alps Mountain

The Ecole Polytechnique Fédérale de Lausanne (EPFL) is one of the two Swiss Federal Institutes of Technology and is located in Lausanne, Switzerland. EPFL is ranked as Europe's #1 and world's #15 university in the field of "Engineering/Technology and Computer Sciences" in the academic ranking of world universities (ARWU) by Shanghai Jiao Tong University. It is one of Europe's leading institutions of science and technology with the following main missions:

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Figure 2 EPFL's Logo

- Educate engineers and scientists
- Be a national center of excellence in the science and technology
- Provide a hub for interaction between the scientific community and industry

EPFL was founded in 1853 as a private school under the name of Ecole Spéciale de Lausanne. And, it became a world-wide renowned institution since 1969 when it was renamed under the current name and directly controlled by the Swiss Federal Government. In contrast, all other universities (beside EPFL and ETHZ) in Switzerland are controlled by their respective cantonal governments. It is located on the shores of Lake Geneva and brings together over 10,000 people on a single campus (under a slogan "one campus, one city"), including 7,200 students and 275 faculty members. More than 50% of the teaching staff comes from outside of Switzerland. It owns more than 250 laboratories and research groups on campus which insures its innovativity and productivity. The campus consists of 65 buildings with different types of architectures which have witnessed the growth of EPFL:

Figure 3 Buildings in EPFL, from left to right, top to bottom: IN, BC, AB

The building in the first picture is built for Electronic Engineering in the late 70s. The modern one in the second picture is built in 90s and belongs to the department of computer science and communication. The post-modern one in the third picture is build after 2000 for the school of life science.

EPFL is composed of 7 schools covering the domains from architecture & civil engineering to computer science, from pure mathematics to social sciences etc. My internship takes place in the institute of bioengineering, more precisely in the BioRobotics Laboratory.

2. BioRobotics Laboratory

The Biorobotics Laboratory (BioRob in short) leaded by professor Auke Jan Ijspeert is one part of the Institute of Bioengineering in the School of engineering at EPFL. His research teams are composed by several post-doc researchers, technicians, dozens of PhD students and master students (including trainees). They are interested in using robots and numerical simulations to study the neural mechanism of movement control, learning in animals and getting inspirations from animals to design new control methods for robots. Also they design novel robots capable of agile locomotion in complex environments. In general, the BioRob Lab has 6 research topics including: Dynamical Systems, Amphibious Robotics, Modular Robotics, Rehabilitation Robotics, Humanoid Robotics and Quadruped Robotics. The Lab has joint in several on-going European projects [2].

Figure 4 Robots developed in BioRob [2], from left to right: Salamander, Cheetah, Yamor

Their research is currently funded by the Swiss National Science Foundation, the European Commission (Information Society Technologies), the EPFL and the Swiss SystemsX initiative in Systems Biology.

3. My participation

I participate in the development of Roombots [3] which is in the research domain of modular robotics in the BioRob Lab. This project intends to design and control modular robots which can ultimately be used as adaptive furniture that moves, self-reassembles, self-reconfigures and self-repairs. The Roombots can change its shapes (reconfiguration) as well as moving around (locomotion) depending on the users' needs. The figure below illustrates this idea:

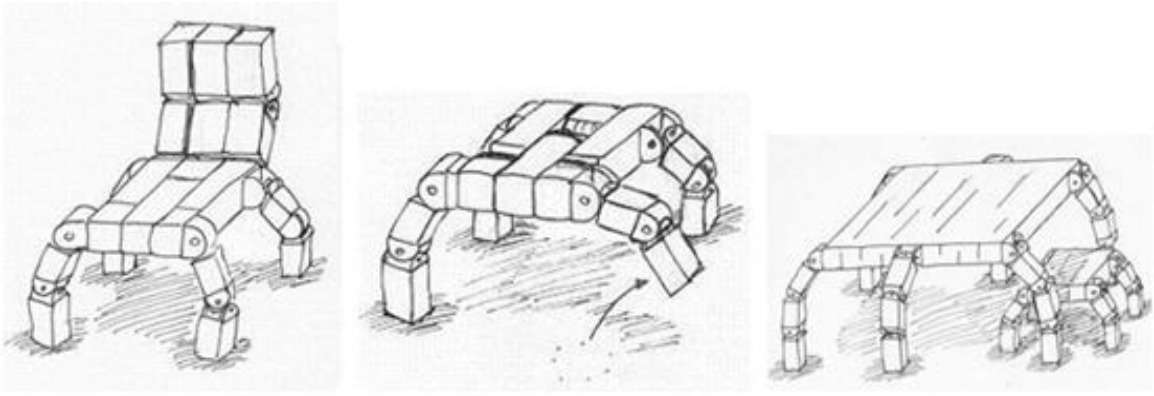

Figure 5 Initial design sketches of Roombots [3]

From the figures above and below, we can see the Roombots is made up of several identical modules as well as the Roombots have many different shapes, such as a chair-like shape, a table-like shape and a stool-like shape etc. The Roombots can move by using its legs. During my internship, I majorly concern on the embedded applications development for such a module.

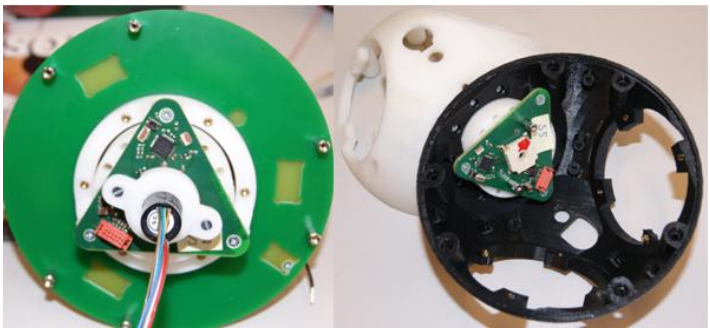

Figure 7 Inside view of an under-construct module

Figure 6 one module of Roombots [4]

More precisely, each module of Roombots is composed by several types of electronics boards. On every electronics board, one microcontroller is placed. My mission is to write programs for the microcontroller, say firmware[1] development. More details can be found in Chapter IV of this report. I just make a short introduction to the types of these electronics boards.

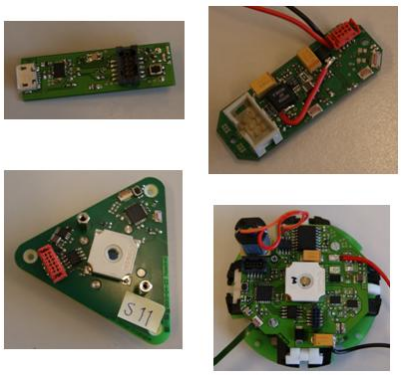

Figure 8 Electronics boards

From left to bottom, top to bottom. The first one is the communication board, it is like an electrical interface between PC and Roombots. The second one is the motor board, to control one actuator inside of Roombots. The third one is the sensor board, which provides the angular position of actuator. The forth one is the ACM board, active connection mechanism, which makes it possible for different modules to connecting together.

[1] Firmware: a kind of small program particular for the embedded devices.

My colleges are working on other aspects of the Roombots. Alexander Spröewitz is the main designer for the mechanical part; Rico Möckel is the main designer for the electronics part. Together, we want to build-up a full functioning module of Roombots. Soha Pouya works on the simulation part of locomotion studying new control methods for the Roombots. Stéphane Bonardi works on the simulation part of self-reconfiguration trying to design new algorithms for that.

To conclude this section, during my internship, I'm expected to:

- Design a generic library for the microcontroller inside Roombots
- Design firmware for different kinds of electronics boards inside Roombots
- Make some experiments on communication with the designed firmware
- Implement Center Pattern Generators on microcontroller
- Implement a real-time tool to measure the interested microcontroller output

All of my missions belong to the first objective of Roombots project – to design a novel robot [3]. And my works are co-supervised by Rico Möckel and Stéphane Bonardi.

4. State-of-art

The initiate idea of the Roombots project is emerging the hi-technology into our daily life by developing the so-called "adaptive furniture" [4]. It gives the possibility to user to create whatever shapes of furniture that he wants. To achieve this purpose, the concept of Self-Reconfiguration Modular Robots (SRMR) is used on the building block, say module. This kind of module can connect or disconnect with other modules or to external structured environment to form different kinds of shapes. The CEBOT developed in the year 1988 [5] is believed to be the first modular robot in this catalog. The main properties of modular robot system should include:

- A whole robot is made up of several modules
- Each module should be almost autonomous, i.e., it has its own actuators, power supply, processors etc.
- The shape of robot is task-dependent which means the most suitable shape could be chosen according to the tasks' needs.

These properties make the modular robot especially fit to exploit the unknown environment, compared to other "regular" robots. Such as the idea of developing the SpaceMolecubes which might be used to exploit the outer space proposed by the Cornell Computational Synthesis Laboratory [6].

How to control the modular robots to do reconfiguration is one of the major research topics in this domain, where decentralized strategies outweigh centralized strategies

because the latter on lacks of scalability [4]. The graph theory is adopted by centralized strategies. The transformation of robot is realized by adding/deleting nodes and edges in graph. The Roombots uses one decentralized strategy where the reconfigurations are described by "cluster flow" locomotion, for more details, you can check the reference [4].

Besides the reconfiguration part, the Roombots also has the ability of moving, either by forming a legged robot or by rolling forward/backward on the ground. To control the locomotion of Roombots, the Center Pattern Generators (inspired from the neuron science) are used. The CPGs-based control method is first used by Kamimura et al. in the modular robot MTRAN [7]. The CPGs models together with optimization algorithms for Roombots can be found in [8].

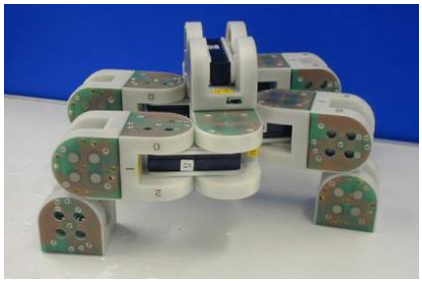

Figure 9 MTRAN [7]

Figure 10 Molecubes [6]

The Roombots is the successor of the Yamor Robot in the BioRob Lab. One of the important improvements of the Roombots from Yamor [错误！未定义书签。] is the adding of Active Connecting Mechanism (ACM). The ACM makes Roombots possible to do self-reconfiguration while the Yamor can only be configured manually.

Figure 11 Tripod-like Yamor Robot [错误！未定义书签。]

Figure 13 one module of Yamor [错误！未定义书签。]

Figure 12 Snake-like Yamor Robot [错误！未定义书签。]

The working principles of one module of Roombots is illustrated by the sketch below:

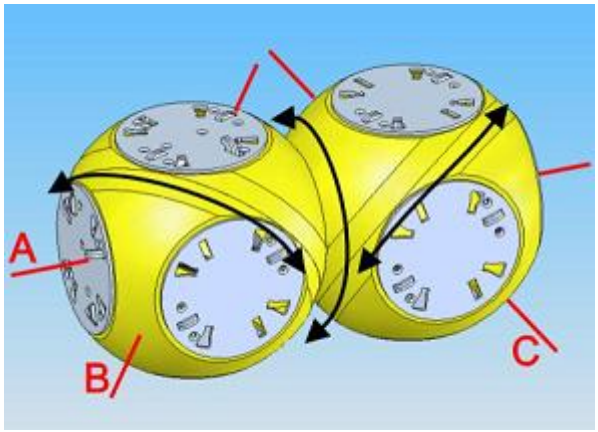

Figure 14 One module of Roombots [9]

The module of Roombots is made of two spheres. It has 3 degrees of freedom, marked by the black arrows and red axis. Each hemisphere can rotate against the axis. The ACM can be added under the gray areas. For example, by looking at the A side of

module, you can see small grapples of ACM. These grapples are used to connecting to other modules or structured environment. By writing the self-reconfigure algorithm in the microcontroller of every ACM, the Roombots get the ability of self-reconfiguring [9].

For more details of the published research paper of Roombots, you can go to the Roombots project main page [3]. In the next chapter, I begin to discuss the ASCII protocol design.

II Communication protocol design

1. ASCII Protocol

Communication protocols, like our human languages, are used by robots or electrical devices to communicate with each other.

The communication protocol that we are going to design is based on the bus RS485 which is ideal for our applications compared to other existing buses. The reasons are listed below:

- The differential transmission/reception mode is adopted by RS485 which greatly enhanced its ability to resist electrical noises.
- In general, RS485 can reach higher baud rate than the CAN bus or I^2C bus
- There's no predefined data protocol on RS485, users are allowed to design their own protocols according to the their needs.

The figure below shows the communication structure:

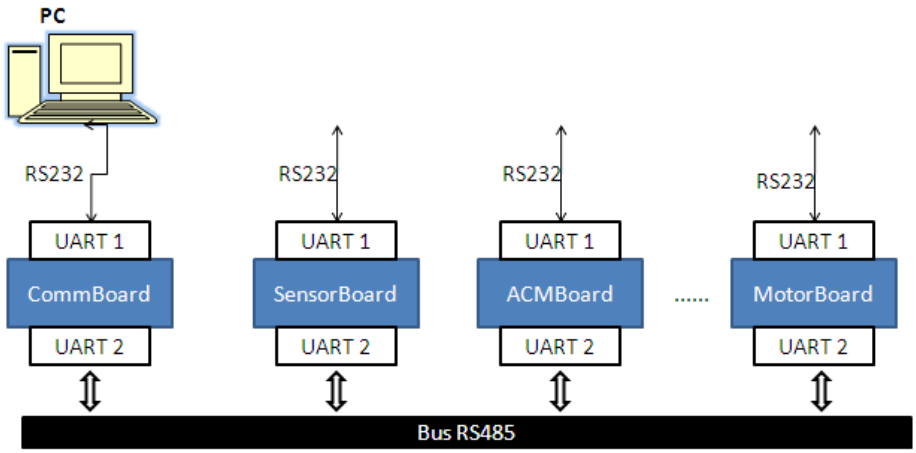

Figure 15 Communication structure

These devices are divided into two categories: the PC and communication board are

named “master”, because they can send data actively; the other boards are called “Slave”, because they can only send data if they have been demanded by “master”. The communication scenario is like this: the PC sends data to the communication board by RS232, communication board redirects data to the bus RS485 (remember? The communication board works as an electrical interface); then the target board will respond to the PC by sending back some data. The data reaches to the PC via the communication board.

We wish to design a simple protocol which is easy for debugging the Roombot. This protocol should be efficient, because in real application we have a large amount of data exchanges among different devices.

Designing an ASCII protocol is the first choice in terms of simplicity for debug use. The one we designed was inspired by the Pin-tilt Unit [10]. ASCII protocol means that one communication unit is composed by several ASCII characters. We reference the communication unit as “packet”. In the following part, I explain the different data fields composing a packet:

INST	BOARD	ID	PARAM	VAL	END

Note, all characters in data fields are case sensitive

INST: instruction, it is composed by one ASCII character.

- ‘s’ for setting, set a value to dsPIC
- ‘?’ for asking, ask a value from dsPIC
- ‘*’ for answer, the answer of dsPIC
- ‘!’ if error happens

Note: we distinguish two types of packets. One is the instruction packets sent by the master. These packets always begin with ‘s’ or ‘?’. Another is the status packets sent by a slave. These packets always begin with ‘’, or ‘!’*

BOARD: the type of boards, it is composed by one ASCII character.

- ‘S’ for sensor board
- ‘M’ for motor board
- ‘A’ for ACM board
- ‘C’ for communication board

ID: id of the board, it is composed by one or more numerical ASCII characters.

This information is necessary, because in one meta-structure, there are several boards with the same type. So we give each of them a different number.

Note: ‘0’ – ‘9’, ‘+’, ‘-’ are numerical ASCII character. BOARD and ID together are called the identifier of one electronics board.

PARAM: parameters to modify, it is composed by one ASCII character.

- ‘p’ stands for absolute position
- ‘r’ stands for relative position
- ‘v’ stands for velocity limitation
- ‘a’ stands for acceleration limitation
- ‘K’ stands for proportional constant in PID control
- ‘I’ stands for integral constant in PID control
- ‘D’ stands for differential constant in PID control
- ‘l’ stands for the Led of Board
- ‘t’ stands for Time-out constant

VAL: the value of a parameter, it is composed by one or more numerical ASCII characters.

END: the end of one packet, it is composed by one ASCII characters.

- ‘\n’ new line
- ‘\r’ carriage return
- EOF, end of file

Note: the END field is very important to keep the packet’s integrity. Lack of this item may cause communication failure.

Specification:

1. The baud rate of communication should be no smaller than 9600 bps
2. Slave must respond to the instruction packet (by sending status packet), to prevent some unexpected deadlocks.
3. The maximum time Master spends on waiting one status packet should be no larger than 500 ms.

Examples:

Instruction packet from Master (PC): ¨?A10p\n ¨
Status packet from Slave (A10): ¨*A10p4086\n ¨

The PC asks the angular position of the ACM Board which has ID 10, the board replies to the PC, its position is at 4086 unit resolution.

Instruction packet from Master(PC): ¨šS3l1\n ¨
Status packet from Slave (S3): ¨*\n ¨

The PC turns on the LED on sensor board id 3. ‘*’ is the status packet from sensor board which means operation has succeeded. Sensor board returns status packet to the PC’s ‘set’ instruction.

Instruction packet from Master(PC): ¨šM5r-200EPFL\n ¨

Status packet from Slave (M5): ¨!Invalid Packet\n ¨

PC sends out an instruction-like packet (which violates our definition) to Motor board 5. So the status packet from motor board 5 begins with '!' and indicate the verbose error information.

All the ASCII protocol related implementation code is put in the file *ascii.h* and *ascii.c*. For further information, you can take a look in the source code and its doxygen document.

Discussion:

The main advantages of using this ASCII protocol are the following:

- It is simple to develop
- It is convenient for debug use, because users can control Roombots by simply tape characters in terminal

The major drawbacks of this ASCII protocol are:

- Compare to binary protocol, it is less efficient. Especially, we need to write parse function to get the numerical values from packets.
- There's no check sum data field in this protocol. So when the internal electrical environment of Roombots is noisy, packets from drivers might get corrupted but receivers cannot notice that.
- The data fields of INST, TYPE and PARAM are represented by a single alphabet character. We could run out of letters, if the contents of them get expanded.

Once we have finished the design of the ASCII protocol, the microcontrollers and the PC have the language to communicate with each other. We just need to implement these rules in the firmware of the Roombots, then our goal is achieved. In the next chapter, we will discuss the Roombots firmware.

III Firmware design and optimization

In electronics and computing, firmware is a term often used to denote a fixed, usually rather small program [11]. There's no strict boundaries between firmware and software, or you can regard firmware as a kind of software used by embedded systems, such as microcontrollers. In a word, firmware has a closer relationship with hardware.

In my internship, one of my tasks was to write this kind of programs for the microcontrollers used in the Roombots so that PC can communicate with the Roombots via a USB cable.

1. Microcontroller: dsPIC33 family [12]

In the chapter I, I have introduced the 4 types of electronic boards used in Roombots, all of them have adopted the same type of microcontroller, that is the dsPIC33FJ128MC802 from the enterprise Microchip which is one of the leading providers in this domain (another famous enterprise in this domain is Atmel). This type of microcontroller is chosen because of its nice features:

- It has 16-bit 40 MIPS processor with most instruction at 1 cycle which means its computation ability is powerful.
- It has one build-in ADC module (Analog/Digital Converter) with high precision (12-bit precision).
- It has two UART modules which can support 10 M bps at maximum.
- It has a QEI (quadratic encoder) module which is desirable in PID control
- It has a hardware CRC (cyclic redundancy check) module to calculate the checksum

Because Microchip only provides low level access (at register level) to its microcontrollers, I have to spend some time on reading the references of the modules we are interested in and write wrapper C functions from scratch. Then we can use these wrapper functions to write the firmware for different boards. An alternative solution is to use the open source library named "molole" [13] (which is designed for the general use of dsPIC33F) developed in the *laboratoir de système robotique* [14] of EPFL. From the point of view of pure software engineering, the alternative solution could be more time saving. But if only considering the efficiency of code instead of portability or other features, the home developed library might be better.

In the next section, we begin to discuss the implementation of the C library for our microcontroller dsPIC33FJ128MC802.

2. Implementation of Libp33

Having known that one microcontroller has many different modules such as UART, oscillator, and timers etc, it is more efficient to put C functions of different modules into different source files. In general, we can compile these modules separately to get object files then zip these object files together to get the final version of libp33. This method is better than put all source code in one big file, because it saves both the develop time and compile time and it is easier to maintain. The standard C library has been developed in the very same way. In this section, I only give some remarks in the difficult points in developing C functions modules by modules. You can check the doxygen document and source code itself for more details.

General Purpose Input/Output module [15]:

The dsPIC33FJ128MC802 has two I/O ports (general purpose input/output) A and B, 21 reconfigurable pins in total (RA0~RA4, RB0~RB15). You have to consider initiating first this module before other things. By default, these pins are used as analog input or output, if you write digital level 1 or level 0 to an analog pin, strange things might happen. Only one function is written in for this module

```
1.  void port_init();
```

Every function in the library is named in this way `modulename_action(…)` module name followed by action.

Oscillator Module [16]:

It is one of the most important modules, because many other modules rely on the correctness of the oscillator. The dsPIC33FJ128MC802 microcontroller itself has a build-in fast RC (Resistor/Capacitor) internal circuit works at 7.3728 M Hz. Besides, you can add external crystal as the oscillator source. In principal, the external crystal is more precise than the internal RC circuit. But in the experiments of the Chapter V, we will find this internal RC circuit could also provide satisfactory performance.

Remark, the use of an external crystal as oscillator requires an external parallel resistor of 1MΩ to be connected.

The major challenges of design the `osc_init(…)` function are to decide which oscillator source to use (the internal RC circuit or the external crystal) and to calculate the correct PLL parameters for different frequencies input.

Interrupts module [17]

The microcontroller dsPIC33F has a rich set of interrupts, but we are not going to use

all of them. So the initiate function of this module `interrupts_init(…)` turns off all the interrupts and at later, the interested interrupts can be turned on by some other initiate functions (e.g., the UART module might turn on the interrupts for receiving characters; the timer module might use timer 1 as system clock etc). Also, in this module, we write functions to turn on/off the general interrupt switch. The trick lies in set the CPU's priority, when its priority is set to 7, all interrupts are deactivated; by contrast the priority at 0 means the general interrupt switch is on.

Remark: the nested interrupt mode is used, i.e., low priority level tasks can be interrupted by the higher priority level tasks.

Timer module [18]

In this module, we initiate timer 1 as the system clock which will provides the time reference for the other events (such as PID control happens every 10 ms, CPG control happens every 20 ms, LED on board blinks every 1000 ms for instances). You can take a look at the implementation of `sys_timer_init()` in this lib and write your own functions for other timers in the same manner.

Remark: the timer 1 generates an interrupt every 1 ms is good enough to deal with the most events in the RB robot. If you set a timer to generate an interrupt every 1 us (which is too fast), some unexpected behaviors may happens.

UART module [19]

The Universal Asynchronous Receive/Transmit module is the key stone of communication between the PC and the Roombots. The interrupt mechanism is used for receiving characters.

CRC module [20]

Cyclic Redundancy check is one robust algorithm to calculate the check sum for a given packet. The dsPIC33FJ128MC802 has an embedded CRC calculate circuit.

The implementation difficulties are that you have to declare 8-bit pointers to CRCDAT register, if you want to calculate the check sum for one string of 8-bit data and choosing appropriate generator polynomial.

3. The first version of Firmware

I developed the first version of firmware between 2010.05.10 – 2010.05.28 for the communication boards, ACM boards and sensor boards. The communication board and motor boards are assembled into one module of Roombots. This module is used during the Open Days demonstration in 2010.05.29 and 2010.05.30 at EPFL's Rolex

Learning Center, you could look the video of that demo at link given in the reference [3].

Figure 16 One finished module of Roombots used in the Open Days

As having been discussed in the Chapter II, the communication board is "Master" which can send instruction packets, other boards are "Slave"s which can feed back status packets to master. And the communication structure should look like the figure below:

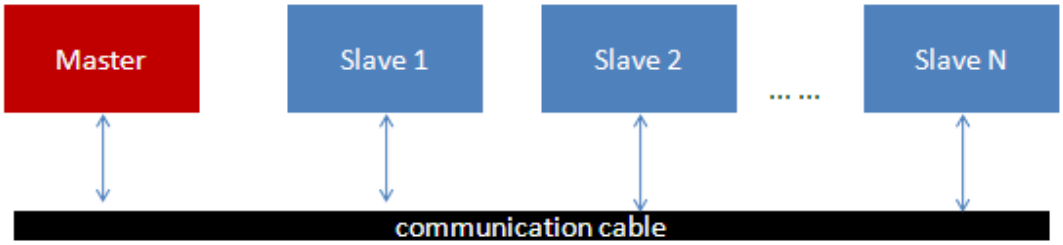

Figure 17 Master-slave structure or multi-drop structure

Remark: the bus RS485 is half-duplex. It cannot transmit and receive data at the same time.

This is the typical master-slave communication structure, on one bus there's only one master and several slaves. The finite state machines are used as the key component of the firmware. And the state machine used by master is quite familiar to the one used by slaves which is logical and reduces the difficulty of development.

Finite State machine of Communication board

The idea is: connect the uart1 module communication board to PC, and uart2 module to RS485 transceiver. In general case, the communication board can only receive characters from PC's side. The interrupt mechanism is used to receive the characters from PC's side, and put them in one big buffer. When communication board has finished receiving one instruction packet, it parses this packet and sent it out to RS485. Then change to reception mode, wait to receive status packet from RS485. Time out might happen, if the status packet is not received within the given time interval

(according to the ASCII protocol, this time interval must be smaller than 500 ms)

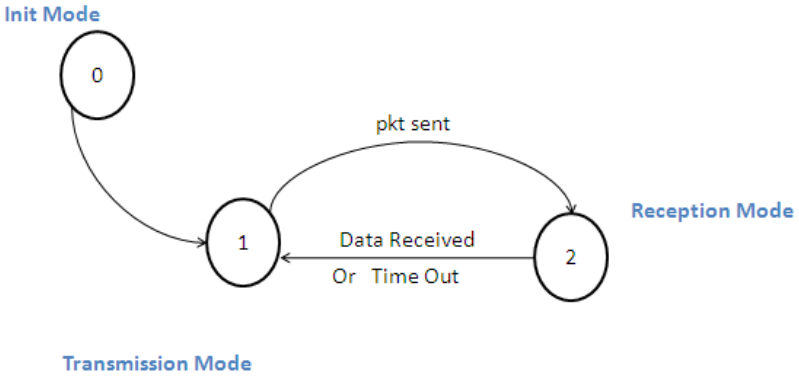

Figure 18 FSM of communication board

This state machine is composed by 3 states: state 0, initiate mode, program does some preparation work to begin the communication such as initiate the big buffers for instruction packet disable receiving characters from bus RS485. After state 0, program enters in the state 1, it receives characters from PC's side and store them in the big buffer for instruct packet. State 2, is the reception mode which means it is possible to receive characters from RS485's side (meanwhile it is impossible to send characters to RS485), program keeps on polling the U2RXREG and passes receive characters directly to PC (to U1TXREG).

The ASCII protocol part can be added in the state 1. E.g., PC sends one instruction packet like "sC2l1\n" which means it wants to turn on the LED on communication board 2. This instruction packet first goes to the communication board, then the communication board parses it. If this communication board has ID 2, then it turns on its LED on board, else it sends this packet to bus RS485. The figure below is the components of this firmware:

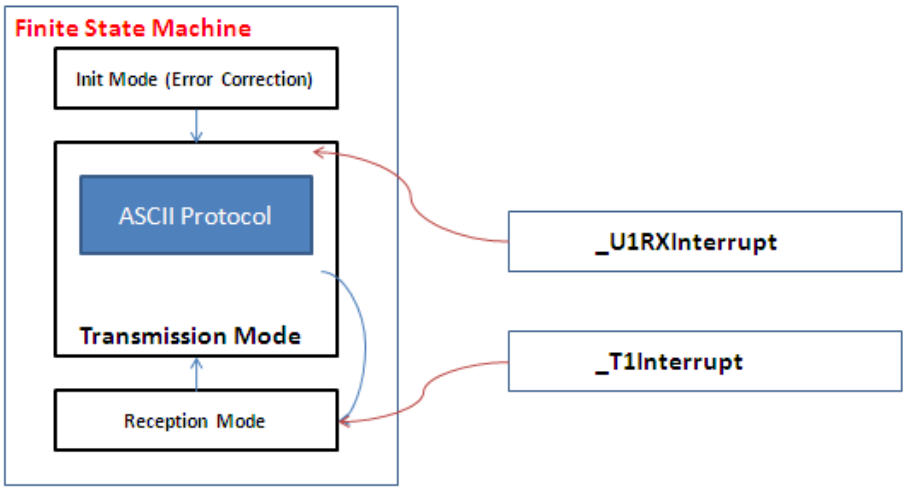

Figure 19 Firmware's components

In figure 11, the block at left side is what we have discussed above. At right side, is two interrupt service routines. `_U1RXInterrupt()` is used to receive characters from PC's side (PC is connected to uart1) which is coupled with Transmission mode, `_T1Interrupt()` is system clock which provides time out for the reception mode.

ASCII protocol part can be executed in these steps:

I. BOARD and ID matching
II. If matched
III. If INST == 's' Then switch (PARAM)
IV. Else form status packet
V. Else ignore this packet

The figure below depicts the interactions among the different components of firmware:

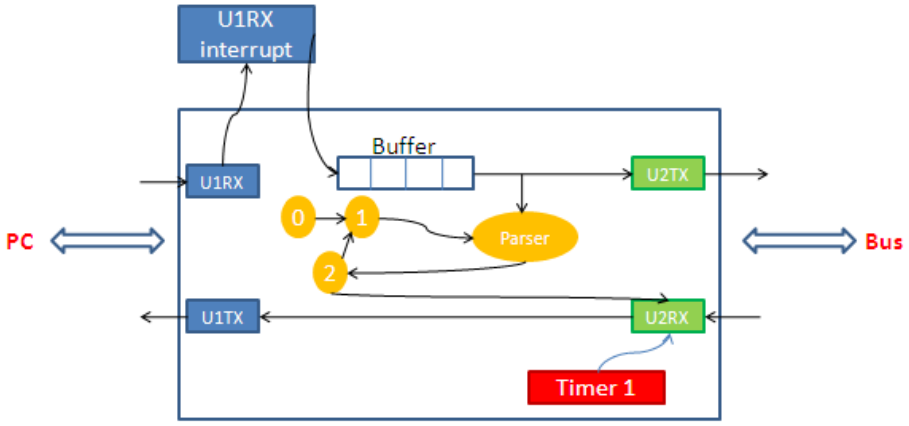

Figure 20 Interaction of components

The module UART1 is connected to PC, the interrupt mechanism is used for receiving characters from PC. The received characters then are put into the buffer. In state 1, when the board has finished receiving one inst packet, it sends the packet out to bus and parse the sent packet. The output of parser decides whether or not to change the state. Only in state 2, receiving chars from bus is possible. Timer 1 provides the timing for U2RX. The characters received from Bus are directly put in U1TX (sending to PC directly).

Finite State Machine for other boards

The state machine of other boards is similar to the communication board except that it majorly stays in the reception mode (communication board stays in transmission mode in most of time). The figure below is the state machine of other boards:

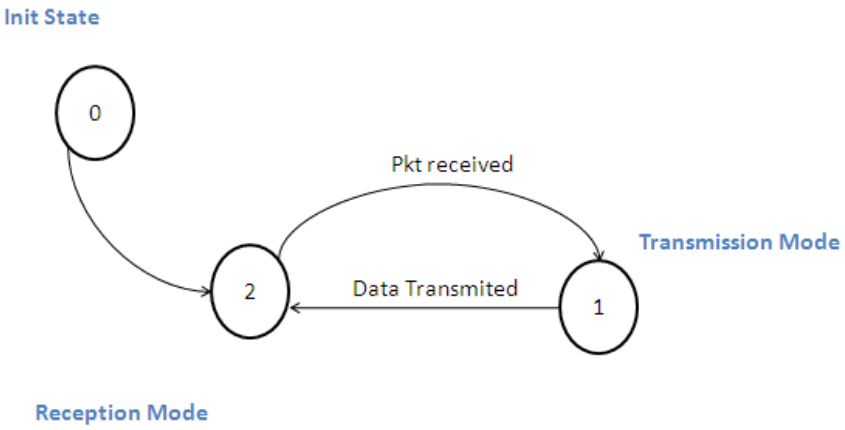

Figure 21 FSM of other boards

After the initiate work in state 0, program jumps into state 2 waiting to receive characters from RS485's side (the characters transmitted by communication board). The interrupt mechanism is adopted. The status packet is formed in state 1 to responds communication board's instruction packet.

The ASCII protocol part is added in state 2. The program's structure is similar to the one in communication board except the uart2 is used to receive characters and the system timer is used to deal with PID control event (in ACM board).

Some discussion

1. The data structure of the buffer which is used to store one instruct packet is not sufficient. In the communication board for instance, when program is parsing the received one packet, PC could continue to send characters. These characters are also appending at the end of the previous packet, but they are just cleaned when the communication board has finished parsing the first packet. Some more sophisticated data structure is needed so that communication board can store more than one instruct packet in its buffer.

2. Using the polling mechanism is a rather bad choice in a complicated real-time application. It wastes the CPU time, or even the whole program might get blocked forever in some unexpected cases. Writing blocking function is not a good practice in real-time applications.

3. Interrupt service routine is a part of firmware; different devices may use the same interrupt but do different things in the interrupt service routines. So the ISR should be put in the main file instead of in the generic library.

4. It is more practical and natural (from the point of view of debug) to let all devices stay in reception mode. If they have some packets to send, then send them and back to reception mode. *Remark: reception mode and transmission mode is referred to bus RS485, because it is half-duplex. RS232 is full-duplex, it can*

receive characters and sending characters at the same time.

5. Compared to binary code, the ASCII protocol is not efficient. To parse instruct packets and to form status packet, consume time. But the key advantage is that it is friendly for user to debug robot.

4. The optimized version of firmware (version 4)

To address the problems listed in the discussion of last section, I have made some optimization of the firmware. First, the more sophisticated data structure is used for the buffer of instruction packet, and I've added a buffer for the status packet. Then the polling mechanism is complete deleted from this version of firmware. And I have noticed the finite state machine is not necessary since it can be replaced by simpler functions. The library libp33 is full used (at the time when the first version of firmware was written, the design of this library was not finished).

Ring buffer [21]

Ring buffer (or circular buffer) is an abstract data structure which in fact is a special form of queue. Data is enqueue at the rear of queue and is dequeued from the front of queue. This data structure is capable to hold several packets. We declare and initiate two ring buffers, one for instruction packet, the other for status packet.

Firmware's general structure

We get rid of the state machine from this version of firmware and adopt round-robin structure in the background (or in main routine). Round-robin mode means all the background tasks has the equal chance to execute. And full interrupt mechanism is adopted. The general structure for all boards is looked like this:

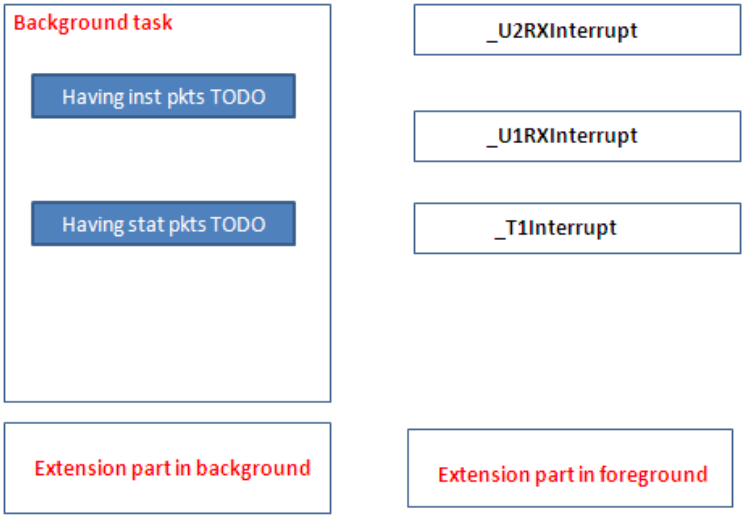

Figure 22 Firmware v4's structure

Both uart1 and uart2 are using interrupt mechanism to receive characters. Timer 1 is the system clock which can handle the event of time out, PID control, CPG control and so on.

In general case, receiving characters from RS485 is enabled (and sending chars to RS485 is disabled). If we have packets to send to RS485, first disable RS485 reception, then enable RS485 transmission. When the last character has been completely sent, then you can disable RS485 transmission and enable RS485 reception.

The ASCII protocol part can be put in "Having inst pkts TODO". The idea of ¨having inst packet TODO¨ is like this: if there're instruction packets in ring buffer and the RS485 bus is not hold, then send one instruction packet. Once have finished sending one instruction packet, parse it. If INST equal to '?', then hold the RS485 bus for a certain time (if not to do so, the data confliction on bus RS485 might happen). The RS485 bus is released either by finishing receiving one status packet or time out happens.

The idea of ¨having stat packet TODO¨ is much simpler: if there're status packets in ring buffer then send them to PC. It can be translated to two lines of code:

The figure below describes the internal interactions of firmware:

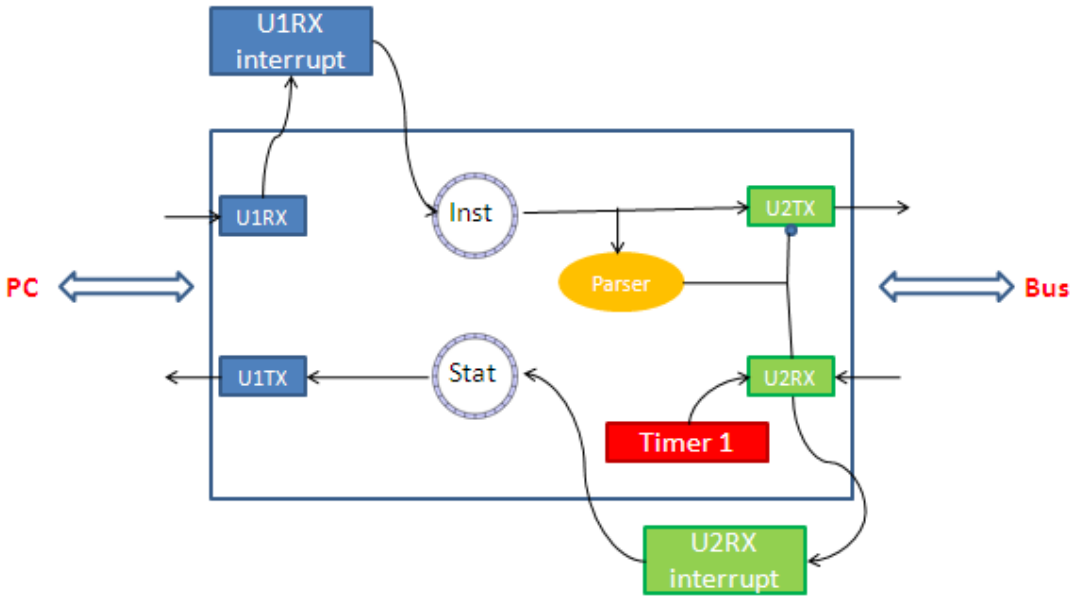

Figure 23 Interactions of firmware

Unlike the previous version of firmware, both UART1 and UART2 use interrupt mechanism to receive characters. If there are instruction packets in inst ring buffer and the bus is not currently hold, then send out one inst packet and parse the sent packet. The output of parser decides whether or not to hold the bus for a while (waiting to receive status packet). Timer 1 provides time reference to U2RX when the

latter one is waiting to receive characters.

IV Baud rate test experiments

One of the most important features of dsPIC33F is that its UART module can support the maxi baud rate up to 10 M bps according to the datasheet. Still I want to verify it by experiments. Besides, I want to figure out whether or not the internal RC circuit (which is not as precise as crystal) can support high speed communication.

1. Experiment I: combine RS232 and RS485

Objective: this test is design to verify that 10 M bps is reachable.

Material: one PC, two communication boards, and one communication bus RS485.

Configuration:

PC: 8 bit data, no parity, 1 stop bit, no hardware handshake. 200 millisecond, is the time out for reading data from serial port.
IC boards: 8 bit data, no parity, 1 stop bit, no hardware handshake. Internal RC circuit oscillator of 7.37 MHz is used.

Method:

1. Program the two boards with the same firmware (firmware v4 of communication board). Modify one line of code in the Interrupt Service Routine of sensor board, so that it puts the received characters in the instruction packet's ring buffer (instead of status packet's ring buffer).
2. Connect communication board to one USB port of PC, and connect sensor board to communication board via bus RS485. Make sure both boards are got powered.
3. Run the automated test environment which will automatically send N packets (both instruction packet-like packets and status packet-like packets) predefined in this program from PC to communication board in a random manner. And this program tries to receive response packets from communication board's side. After sending and receiving process have finished, this program will compare the sent packets with the received packets and get the mismatching rate as result. The figure at below illustrates our scenario of testing:

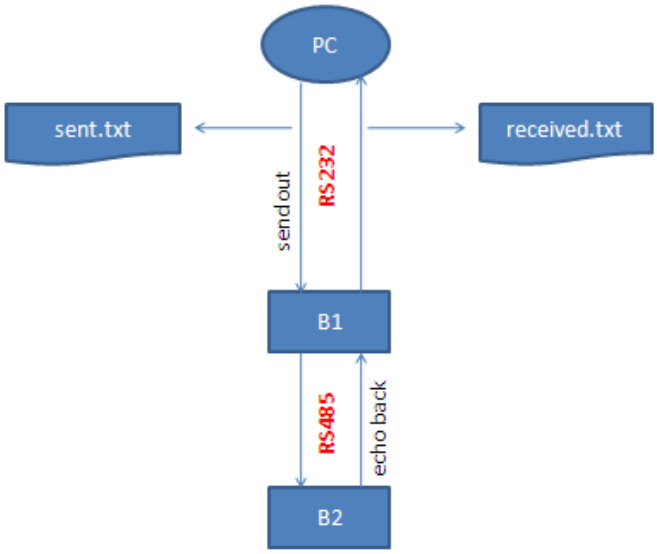

Figure 24 Test scenario of exp I: PC send packets to B1, B1 passes them to B2, B2 echoes them back to PC via B1.

4. Pick up the next baud rate listed in the ˝baud rate calculate sheet ¨and repeat step 1~3.

Result:

N °of trial	PC (baud)	Board (baud)	Packets (unit)	Error Rate (%)
1	38600	38600 (INT)	10000	0.00
2	1 M	1 M (INT)	10000	0.00
3	2 M	2 M (INT)	10000	0.00
4	2.5 M	2.5 M (INT)	100	100.00

The first column is the number of trial in this experiment. The second column is the baud rate of PC (actually it's the baud rate of USB-RS232 converter). The third column is the baud rate of boards, INT means internal oscillator is used. The forth column is the number of packets sent. The fifth column shows the results.

Conclusion:

Compare the trial 1 to 4, generally we can conclude that if the communication is possible, then the communication will go well, otherwise the communication will completely fail. Trial 1 to 3 are the successful communication with 0.00% error rate, but trial 4 is the failed one with 100.00% error rate.

The communication goes well for the baud rate between (38600 bps ~ 2 M bps), but it cannot pass the level of 2 M bps (at least in the present configuration). It seems 2 M bps is the upper limit of baud rate (which is far from enough). I doubt that this result might be caused by the impreciseness of internal RC circuit which is used as oscillator

of boards. To make a further investigation, the experiment II is needed.

2. Experiment II: only RS232

Objective: this test further investigates which factors constrain the upper limit of baud rate, and what is the upper limit of baud rate.

Introduction: In this experiment II, we try to figure out what are the constraint factors and whether or not we could reach higher baud rate.

Material: one PC, one communication board, one sensor board, one bus RS485, one evaluate board of Microchip (with dsPIC33FJ64GP802 on it).

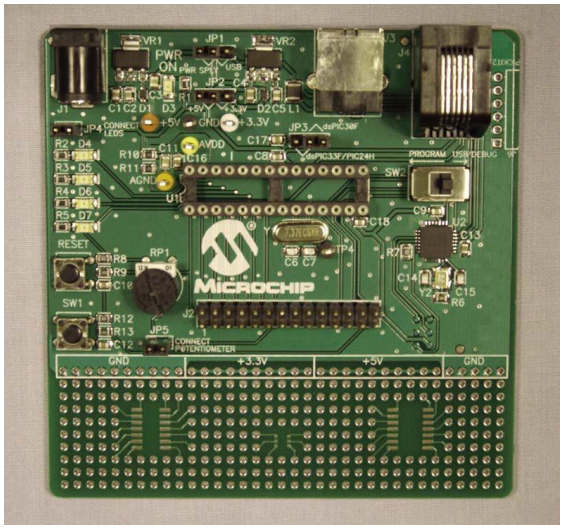

Figure 25 Evaluation board from Microchip

Configuration:

PC: 8 bit data, no parity, 1 stop bit, no hardware handshake. 200 millisecond, the time out for reading data from serial port.
Our IC boards: 8 bit data, no parity, 1 stop bit, no hardware handshake. Internal RC circuit oscillator of 7.37 MHz is used.
Demo board of Microchip: 8 bit data, no parity, 1 stop bit, no hardware handshake. Both Internal RC circuit oscillator of 7.37 MHz and external crystal of 7.37 MHz are used.

Analyst before experiments:

I had some further literature on the datasheet of the USB-RS232 converter [22] and the RS485 transceiver on our IC boards. I quoted an important phrase from the former one, *"The FT232R (USB-RS232 Convertor) supports all standard baud rates and non-standard baud rates from 183 Baud up to 3 M baud, Achievable non-standard baud rates are calculated as follows - Baud Rate = 3 M / (n + x)"*, which means this

converter has the same principle as the UART module of microcontroller and only some discrete baud rate is reachable.

From the data sheet of RS485 transceiver, I knew this transceiver can support any baud rate under 16 M bps.

To be clear, I list two import formulas to calculate reachable baud rate:

$$Baud\ rate = \frac{3\ M\ bps}{n + x}$$

$$Baud\ rate = \frac{40\ M\ bps}{4(UxBRG + 1)}$$

The first one is for the USB-RS232 converter, and the second one is for the UART module of dsPIC33 where n is the integer part in the range of [1-2^{14}], x is the fractional part represented by 3-bit.

Method:

1. Program the communication board's firmware on the demo board. So that the demo board will automatically echo the received packets to PC.
2. Connect demo board to PC via USB cable (there's a USB-RS232 converter on the demo board). Make sure the demo board is got powered.
3. Run the automated test environment on PC which sends out packets randomly and tries to receive the echoes from demo board. Then match the sent packets with the received packets. The figure below illustrate this idea:

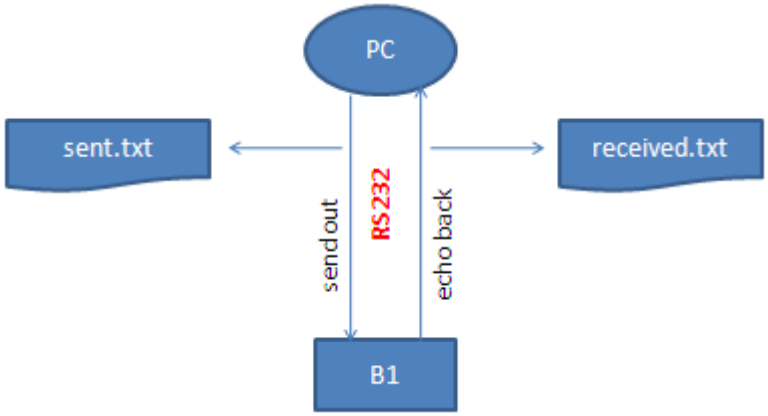

Figure 26 Test scenario of exp II

N°	PC (baud)	Board (baud)	Mismatch (%)	Packets (unit)	Error (%)
1	2.4 M	2.5 M (EXT)	4.08	10000	0.00
2	2.4 M	2.5 M (INT)	4.08	10000	0.00
3	2.66 M	2.5 M (EXT)	6.43	100	100

4	3 M	3.33 M (EXT)	9.91	100	100

The forth column is baud rate mismatching between PC and board.

Conclusion:

Compare trial 1 and 2, I conclude the 2.4 M bps is reachable, internal oscillator has the same performance as the external oscillator in this case. Compare trial 1 to 4, I notice that the baud rate mismatching between two devices is the key factor to constrain the communication. In the next experiment, I'm going to test the communication on RS485.

3. Experiment III: RS485

Method:

1. Programm the communication board so that it sends out a string ¨fo\n¨at a certain period.
2. Programm the sensor board so that it can receive characters from communication board via the bus RS485.
3. If the received char is one in the set of {'f', 'o', '\n'}, then the sensor board will toggle its green LED on board.

In theory, the communication between communication board and sensor board via RS485 transceiver will reach to 10 M bps without difficulties.

Result:

N °trial	Board1 (baud)	Board2 (baud)	LED blink normally ?
1	1 M	1 M	Yes
2	2.5 M	2.5 M	Yes
3	5 M	5 M	Yes
4	10 M	10 M	Yes

Internal oscillator is used in all trials.

Conclusion

The theory limit 10 M bps is reachable on RS485 communication. Internal oscillator is good enough to support the maximum baud rate.

4. Conclusion

Key points:

- The USB-RS232 converter can be a main bottleneck for the UART communication
- Oscillator's precision matters, and the Internal RC oscillator embedded in dsPIC33 is precise enough.

$$Baud\ rate = \frac{3\ M\ bps}{n+x} \quad (1)$$

$$Baud\ rate = \frac{40\ M\ bps}{4*(N+1)} \quad (2)$$

The first formula is used to calculate the non-standard baud rate supported by FT232R circuit where n is integer part [2^0, 2^{14}] and x is the fractional part with 3 bit precision (which implies the value of x belongs to the set {0, 0.125, 0.250, 0.375, … , 0.875}). The second formula is used to calculate the baud rate supported by UART module of dsPIC33 in 4x mode where N is the value in the register of baud rate generator UxBRG.

The reachable non standard baud rate for USB-RS232 converter:
1 M bps, 1.2 M bps, 2 M bps, 2.4 M bps, 2.66 M bps, 3 M bps
The reachable baud rate for dsPIC33F:
1 M bps, 1.25 M bps, 2 M bps, 2.5 M bps, 3.33 M bps, 5 M bps, 10 M bps
The reachable baud rate for RS485 transceiver:
All baud rates under 16 M bps.

The baud rate limit is at 2.4 M bps when having the intervention of PC. The baud rate limit is at 10 M bps on purely RS485.

V Conclusion

During my internship, I'm integrated in the develop team for the Roombots project. I've participated in the embedded applications development for the microcontrollers inside of Roombots.

First, I've design one library called libp33 majorly for the microcontroller dsPIC33FJ128MC802. Since Microchip only provides the access to its microcontroller at register's level, it lacks modularity and reusability from a programmer's perspective. So the motivation of this task is to solve this problem and make a good preparation for further developing firmware on microcontroller. The libp33 provides many basic C functions for programmers to access different modules of the microcontroller. This task is time consuming, because I have to read every interested module's reference manual and think carefully how to implement efficient

C functions for general usage. Thanks to this task, my knowledge about microcontroller programming is greatly deepened. In the future work, people could make this library even more generic to support more modules and more microcontroller devices. Also, people could get some inspirations of how to design library and how to talk to microcontroller at register's level. People who interest in this part are also suggest to take a look at the molele library [13] which is designed by LSRO library of EPFL.

Second, I've design an ASCII communication protocol based on RS485 standard. The major advantage of this protocol is that: it makes users extremely easy to debug Roombots simply by taping ASCII characters on the terminal. The main drawback of this protocol is that: it is not efficient from the point view of programming; more precisely for parsing packets, it is not convenient or even error-prone. One solution is to add another binary protocol on Roombot. Then using ASCII protocol for debug purpose, and using binary protocol for programming. This could be scheduled in the future work. People can also get some inspirations from the work has been done in the Dynamixel actuators [23].

Third, I've designed firmware for three electronics boards inside Roombots. They are communication board, ACM board and sensor board. And the communication board was used in the open day's demonstration at Rolex Learning Center. And I've given an optimized structure of firmware which is suitable for all boards. During my internship, I've spent most of my time on this part which makes me to realize: to develop embedded applications is really tricky -- you can encounter either software or hardware bugs/errors; your program must obey to both logical correctness and timing constraints. I think it is worthwhile to develop one lightweight real-time operating system for every microcontroller inside Roombots in the future. So the idea of concurrency and threading-programming can be introduced in. I give a suggest text book about system programming in the reference [24].
Fourth, I've implemented a small and useful plot tool in MATLAB which can be used to measure your interested parameter of microcontroller in real time, such as the output of ADC module, the output of PWM module and any module's output as you wish. You can use it to tune the parameters of PID control loop, and also observe the CPG commands. A more detailed document can be found at [25].

Fifth, I've acquired some basic knowledge about center pattern generators. I've implemented a simple version of CPG's library libcpg which can be used by both microcontroller and PC. One novel thing about this library is that: the Runge-Kutta integration method is used to calculate the output of CPGs, besides the Eular's method. A more detailed document can be found at [25]. This task enriches my knowledge, and I'm quite satisfactory after finishing this task.

Besides what I've done from the perspective of technique, I'm quite happy to working in the BioRob Lab to know people from different countries over the world and to

learn something about their cultures. I've seen the way of doing researching in top university such as EPFL in Switzerland. Also, I've heard from Prof. Auke Jan Isjpeert his own opinions about doing PhD, post-doc and working in industry. I think I'll profit a lot from these experiences in my future.

Appendix

```
/**
 * @file comm_firm.h header file of communication board's firmware
 *
 * date of creation: 2010.07.23
 * data of modification: 2010.08.13
 *
 * @author xinkui feng <xinkui.feng@epfl.ch>
 */

#ifndef COMM_FIRM_H
#define COMM_FIRM_H

// ---------------------- include ---------------------------------------------

// C function library of device dsPIC33F
#include "../libp33/libp33.h"
// include ASCII protocol part
#include "ascii.h"

// --------------------- macros -----------------------------------------------

// frequency of oscillator, M Hz
#define FIN 7.37

// pin of enable rs485 transmission, high level effective
#define PORT_RS485_TE _LATB6
// pin of enable rs485 reception low level effective
#define PORT_RS485_RE _LATB7
// macros to enable/disable rs485 transmision and reception
#define EN_RS485_TRAN(i) ( i == 0 ? (PORT_RS485_TE = 0) : (PORT_RS485_TE = 1) )
#define EN_RS485_RECV(i) ( i == 0 ? (PORT_RS485_RE = 1) : (PORT_RS485_RE = 0) )
// pin of rs485 transmission
#define PORT_RS485_TX _RP13R
// pin of rs485 reception
#define PORT_RS485_RX 14
// pin of rs232 transmission
#define PORT_RS232_TX _RP8R
// pin of rs232 reception
#define PORT_RS232_RX 9

// size of a small buffer, which can contents one inst packet
```

```
#define SIZE 16


// ---------------------- globle variables -----------------------------------

// board type
char board = 'C';
// board id
char id = '1';

// number of packets in ring buffer
int n_stat_packet = 0;
int n_inst_packet = 0;
// ring buffers
rbuf stat_packet_rbuf;
rbuf inst_packet_rbuf;


// uart communication baud rate, bps
unsigned long baud = 38400;
// uart1 mapps to rs232
REG* u1tx = (int*)&PORT_RS232_TX; // pass addr
REG* u1rx = PORT_RS232_RX;
// uart2 mapps to rs485
REG* u2tx = (int*)&PORT_RS485_TX; // pass addr
REG* u2rx = PORT_RS485_RX;


// parse inst packet
char flag_parse = 0;
// buffer which contents the inst packet to parse
char buf[SIZE] = {0};


// hold to send inst packet
char flag_hold = 0;


// time out flag
char flag_timeout = 0;
// time out for receiving stat packet, ms
unsigned int timeout = 10;
// ticks for time out
```

```
unsigned int timeout_tick = 0;


// system tick counters
unsigned long long sys_tick = 0;



// --------------------- function prototyping --------------------------------

void comm_startup();
char uart1_send_packet_rbuf(rbuf *rb);
char uart2_send_packet_rbuf(rbuf *rb);
void command(packet* p);

#endif // COMM_FIRM_H

/**
 * @file comm_firm.c the implementation of firmware
 *
 * version 4
 *
 * @author xinkui feng <xinkui.feng@epfl.ch>
 */

// header file of firmware, only include it once
#include "comm_firm.h"

// config of oscillator, fast RC circuit is choosed as primary oscillator
_FOSCSEL(FNOSC_FRC);
_FOSC(FCKSM_CSECMD & OSCIOFNC_OFF);

/*
 * Main routine of firmware
 */
int
main()
{
        // local variables
        packet inst_packet;

        // system startup function
        comm_startup();
```

```
	// body
	while(1) {
		// having status packets
		if(n_stat_packet) {
			// then send them to PC via uart1
			uart1_send_packet_rbuf(&stat_packet_rbuf);
		}

		// having inst packets
		if(n_inst_packet) {
			// if rs485 bus is not be hold
			if(!flag_hold) {
				// then could sent inst packet to devices via uart2
				uart2_send_packet_rbuf(&inst_packet_rbuf);
			}

			// if one packet has been sent, parse this sent packet
			if(flag_parse) {
				int i;

				// parse the content in buffer to inst_packet
				int status = parse(buf, &inst_packet);

				// valid packet
				if (status == 0) {
					// device matching
					if(board == inst_packet.board && id ==
inst_packet.id) {
						// make action
						command(&inst_packet);
					} else {
						if(inst_packet.inst == '?') {
							// hold rs485 bus for a while
							flag_hold = 1;
						}
					}
				}

				// clr buf
				for(i = 0; i < SIZE; i++) {
					buf[i] = 0;
				}

				// clr flag
```

```
                        flag_parse = 0;
                }

        }

        // if time out happens
        if(flag_timeout) {
                // add error information to ring buffer
                rbuf_write_str("time out\n", stat_packet_rbuf);
                // increase number of stat packet in ring buffer
                n_stat_packet ++;
                // clr flag
                flag_timeout = 0;
        }

        // TO ADD
    }

    return 0;
}

// ------------- definition of functions ----------------------------------

/*
 * System startup function for communication board
 */
void
comm_startup()
{
    // first init oscillator, internal source is used
    osc_init(0, FIN);
    // init I/O ports
    port_init();
    // interrupts config
    interrupts_init();
    // init uart modules with 4x sampling mode
    uart1_init(1, baud, u1tx, u1rx);
    uart2_init(1, baud, u2tx, u2rx);
    // init cyclic redundence check module
    crc_init();
    // init and start system timer
    sys_timer_init();

    // watch dog is disabled
```

```
	_SWDTEN=0;

	// initiate ring buffers
	rbuf_init(&stat_packet_rbuf);
	rbuf_init(&inst_packet_rbuf);

	_TRISB3 = 0; // set Green LED as output
	_TRISB7 = 0; // set RS485 receiver enable to output
	_TRISB6 = 0; // set RS485 transmitter enable to output

	// rs485 transmission is disabled and reception is enabled
	EN_RS485_TRAN(0);
	EN_RS485_RECV(1);
}

/*
 * note: uart1 is connected to rs232
 * send one packet (in ring buffer) out via uart1,
 * it is a non-blocking function
 */
char
uart1_send_packet_rbuf(rbuf *rb)
{
	char c;

	// while transmitter's register is not full
	while(!U1STAbits.UTXBF) {

		// read one character out of ring buffer
		rbuf_read(&c, rb);

		// chk if it is the end of one packet
		if( is_stop_char(c) ) {
			// finish sending one packet
			return 0x00;
		}

	}

	// one packet is partially sent
	return 0x01;
}

/*
```

```
 * note: uart2 is connected to rs485
 * send one packet (in ring buffer) out via uart2,
 * it is a non-blocking function
 */
char
uart2_send_packet_rbuf(rbuf *rb)
{
	static int i = 0;
	char c;

	// disable rs485 reception
	EN_RS485_RECV(0);
	// enable rs485 transmission
	EN_RS485_TRAN(1);

	// while transmitter's register is not full
	while(!U2STAbits.UTXBF) {

		// read one character out of ring buffer
		rbuf_read(&c, rb);

		// save char in a small buffer, wait to be parsed
		buf[i++] = c;

		// chk if it is the end of one packet
		if( is_stop_char(c) ) {
			i = 0;
			// set flag to parse
			flag_parse = 1;
			// finish sending one packet
			return 0x00;
		}
	}

	// wait the last char has been completely sent out,
	while(!U2STAbits.TRMT);

	// then disable rs485 transmission
	EN_RS485_TRAN(0);
	EN_RS485_RECV(1);

	// one packet is partially sent
	return 0x01;
}
```

```
/*
 * board react to inst packet
 */
void
command(packet* p)
{
	switch (p->param) {
		// LED
		case 'l': {
			if(p->inst == 's') {
				(p->value == 0) ? (PORT_LED = 0) : (PORT_LED = 1);
			} else if (p->inst == 'r') {
				// TODO
			}
			break;
		}
		// Time Out
		case 't': {
			if(p->inst == 's') {
				if(p->value < 0) {
					// its value cannot be negative
					p->value  = 0;
				}
				// set timeout value
				timeout = p->value;
			} else if (p->inst == 'r') {
				// TODO
			}
		}
	}
}


// ------------- interrupt service routine --------------------------------

/*
 * System timer, interrupt every 1 ms
 */
void _ISR
_T1Interrupt()
{
	// clr INT flag
	IFS0bits.T1IF = 0;
```

```
	// if rs485 bus is hold
	if(flag_hold) {
		// begin to count time out ticks
		timeout_tick ++;
	}

	// time out happens
	if(timeout_tick >= timeout) {
		// release rs485
		flag_hold = 0;
		// set time out flag
		flag_timeout = 1;
		// clr counter
		timeout_tick = 0;
	}

	// increase system ticks
	sys_tick++;
}

/*
 * interrupt generated by uart1 when receiving one char
 */
void _ISR
_U1RXInterrupt()
{
	char c;

	// clr INT flag
	_U1RXIF = 0;

	// read all chars out of buffer U1RXREG
	while(U1STAbits.URXDA) {
		// get one char
		c = U1RXREG;

		// put it in ring buffer of inst packet
		rbuf_write(c, &inst_packet_rbuf);

		// chk if one packet is finished
		if( is_stop_char(c) ) {
			// increase number of packets in ring buffer
			n_inst_packet++;
```

```
                }
        }
}

/*
 * interrupt generated by uart2 when receiving one char
 */
void _ISR
_U2RXInterrupt()
{
        char c;

        // clr INT flag
        _U2RXIF = 0;

        // read all chars out of buffer U2RXREG
        while(U2STAbits.URXDA) {
                // get one char
                c = U2RXREG;

                // put it in ring buffer of stat packet
                rbuf_write(c, &istat_packet_rbuf);

                // chk if one packet is finished
                if( is_stop_char(c) ) {
                        // release rs485 bus, if any
                        flag_hold = 0;
                        // increase number of packets in ring buffer
                        n_stat_packet++;
                }
        }
}

// that's all, folks
```

Reference

[1] Lausanne, *Wikipedia*, http://en.wikipedia.org/wiki/Lausanne
[2] The BioRob Lab's main page, http://biorob.epfl.ch/
[3] Roombots project's main page, http://biorob.epfl.ch/page38279.html
[4] A. Spröwitz, P. Laprade, S. Bonardi, M. Mayer and R. Möckel et al. Roombots-Towards Decentralized Reconfiguration with Self-Reconfiguring Modular Robotic Metamodules. *Proceedings of IEEE IROS 2010*, Taipeh, Taiwan, October 18-22, 2010.
[5] FUKUDA T., NAKAGAWA S., "Self organizing robots based on cell structures - cebot", in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems,* 1988, pp. 145-150.
[6] CCSL's webpage, http://ccsl.mae.cornell.edu/
[7] KAMIMURA A., KUROKAWA H., "Distributed adaptive locomotion pattern generation for modular robots", in *IEEE international Conference on Robotics and Automation (ICRA2003)*, 2003.
[8] MARBACH D. and IJSPEERT A. J., "Online optimization of modular robot locomotion", in *Proceeding of the IEEE int. Conference on Mechatronics and Automation (ICMA 2005)*, 2005, pp. 248-253.
[9] MEYER M., Roombot modules – Kinematics Considerations for Moving Optimization, 2009, http://birg.epfl.ch/page69834.html.
[10] Pan-tilt unit, developed by DirectedPerception Company.
[11] Firmware, *Wikipedia*, http://en.wikipedia.org/wiki/Firmware
[12] All the reference manual for device dsPIC33F, Microchip Company, http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2573
[13] molole library's webpage, http://mobots.epfl.ch/molole.html
[14] The LSRO Lab's main page, http://lsro.epfl.ch/
[15] Microchip Company, section 10 ¨I/O ports¨,
[16] Microchip Company, section 7 ¨Oscillator¨
[17] Microchip Company, section 6 ¨Interrupts¨
[18] Microchip Company, section 11 ¨Timers¨
[19] Microchip Company, section 17 ¨UART¨
[20] Microchip Company, section 36 ¨CRC¨
[21] Circular buffer, *Wikipedia*, http://en.wikipedia.org/wiki/Circular_buffer
[22] FTDI Chip company, The datasheet of device ¨FT232R USB UART IC¨
[23] Company Robotis,the reference manual of ¨Dynamixel-AX12¨.
[24] RANDAL B. and O'HALLARON D., "Computer Systems: A Programmer's Perspective", *Prentice Hall*, 2011.
[25] The webpage of Xinkui FENG's internship at EPFL, http://biorob.epfl.ch/page41751.html