BOTZILLA PROJECT

Dominique Corseaux, Gregoire Maillet, Jean-Baptiste Maillet

January 18, 2003

Contents

Part I

Introduction

Last updated:

`$Date: 2003/01/12 17:17:25 $`

Botzilla is a team project aiming to compete for the coupe de France de robotique/eurobot 2002, an French/European robotic competition. This involve embedded software development, electronic and mechanical engineering. Most of all, the goal is to learn, experiment and have fun using the non-trivial specifications of the eurobot. This is a competition between engineer's school and universities. These competitions are organized by Planete Sciences, formely known as ANSTJ (Association Nationale Sciences et Techniques Jeunesse = Youth Science and Technical non-profit Organization)

note:

We are three former students (read: professionals in our fields since a few years now), but we hope to be able to compete as outsiders!

For information about the Coupe de France de robotique/Eurobot, see:

- the Coupe de France de Robotique page (in French, http://www.anstj.org/robot/index.html)
- the Eurobot page (in English, http://www.anstj.org/robot/concours/eurobot/garde_en.html)

Technical fields involved include: cross-development (GNU tools, gcc), real-time executive (rtems), i2c, microcontrolers (Microchip PIC, Motorola 68332), analog and numerical electronics, PWM... applied for BSP, host and target device drivers, unit and regression test, remote debugging, simulation. Methodologies, mechanical, electrical and software engineering come into play too.

This may vary : competencies at hand include embedded and device driver development, IA, imaging analysis, radio-frequency transmissions, power-switching... and the eurobot rules changes every year.

Part II

News

Last updated:

`$Date: 2003/01/12 17:17:25 $`

Significant content changes:

2002/11/30:
Started the “PID regulation” section, see part X.
Add the “Serial / I2C Interface” chapter, see 19.
2002/11/27:
Started the “Architecture” section, see part VI.

2002/11/25:
Started this “News” section.
Started the “Engineering advice” section, see part VII.

Part III

How to use this site

This site is quite unstable, at least to this date:

`$Date: 2003/01/12 17:17:25 $`

It's a whole work on progress, and its parts may move in a parallel way. So you should not bookmark anything (you've been warned) except the main location http://botzilla.free.fr/

Nevertheless, this site contain technical material. As users ourselves of such documentation, we chose to use a publishing system that will hopefully make it easy to maintain booth a brows-able and a printable document. For the gory details, we use LaTeX and its converters with a Makefile, all under CVS control (in fact two CVS servers on two remote sites).

This means that:

- You'll always find a consistent table of contents, allowing you to find what may be of interest to you.
- You'll always find download-able, printable and up-to-date versions of this site,
- as postscript here (http://botzilla.free.fr/dldoc/botzilla_ps.tgz)
- as pdf here (http://botzilla.free.fr/dldoc/botzilla_pdf.tgz)
- and a brows-able off-line version here (http://botzilla.free.fr/dldoc/botzilla_html.tgz)

These postscript, pdf and html versions are compressed with tar/gunzip. UNIX/Linux users will know what to do with the tgz compressed files.

For MS Windows users: Zip can manage tgz files. It'll just ask you first if you want to uncompress it in a temporary folder first.

For Apple users: with Mac OS X, you fall under the category "UNIX users". Use the terminal or Stuff-It Expander.

Part IV

Images

For the moment there are few images in this documentation, but you could found some photos here (http://botzilla.free.fr/botzimage/)

Note that these images are not part of the printable or downloadable documentation.

Part V

Strategy

Last updated:

`$Date: 2002/11/11 19:23:37 $`

This part deals with the general strategy for the conception of a robot for the Coupe de France de Robotique (in French, http://www.anstj.org/robot/index.html) / the Eurobot (in English, http://www.anstj.org/robot/concours/eurobot/garde_en.html).

Chapter 1

How to move on the field

In this competition, we use a flat rectangular field (2m x 3m).

What wee need:

- Go fast (we have only 1min30s to win a match)
- Abitily to turn round and round (the field is small)
- Robustness: ability to go through the game field strewed with pucks and balls.
- Easy to control.

If the field is flat, the simplest way to achieve these requirements it to use 2 motrice wheels, and 2 free rollers. Of course, at any moment only 1 of the 2 rollers touch the ground. The robot switch form one roller to the other when it accelerate, brake, turn or bump into something. For the ability to turn round in the corners, we propose a circular robot with the 2 motrice wheels on a diameter, and the 2 rollers on the perpendicular diameter.

For a simple and efficient control, we propose to use only 2 types of deplacement: straight line and turn round. Curves are to complex and seem not necessary in this competition.

Chapter 2

How to manage time

We have 1min30 to win, it's short and maybe it's important that the robot adapt his behavior with the remaining time. For example, in a first time it will try to ensure a minimal point, then it could try to make complex things (like stack up 3 pucks), and a the end, it has to score the last point (for example put a single puck rather than to keep it for a stack).

It's also very important for the robot, to detect dead lock situations.

Match are very short, so it's better to have multi-tasks robot that could quickly adapt it's behavior with output events.

How to do that, well you need some timers, an interrupt handler, and ... it will be easier with a real-time multi-tasking system (we use RTEMS, see part VIII).

Chapter 3

Relative versus Absolute Positioning

Of course, an absolute positioning (this means that you precisely know were you are in the field) is great, but it has a cost and it's not always necessary. It mainly depend on the subject of the competition: for a soccer or basket ball game it's necessary, but this year the robot could works fine without on absolute positioning. Of course, it's a good thing to cover the field when you look for pucks, but you could ensure that without knowing your position.

Part VI

Architecture

Last updated:

`$Date: 2002/12/04 19:42:10 $`

Some "glue" that will help understand the rest of this document:

Chapter 4

Global hardware architecture

In figure 4.1 is presented the system as a whole (well, except the mechanical parts, OK).

We use two CPUs board at the time, possibly more in the future.

Rationale for this:

- Pros:
  - allow parallel development (we don't see each other every day)
  - allow subdividing the problem in smaller parts
  - increased flexibility
  - allow experimenting with more targets and technologies, eurobot is all about this
- Cons:
  - imply a bus in order for CPU to exchange information (but a bus is needed anyway for "smarts" peripherals)
  - increased integration cost
  - increased hardware cost (more parts)

First and main CPU is a Motorola 68000 based board, the NF300 and its application board. This CPU is responsible for high level strategy and decision making based on information fed up by other CPUs and/or peripherals.

Rationale for this:

- Pros:
  - we have one
  - classic, well know 32 bit micro-controller CPU
  - supported by the almighty GCC compiler

  - ideally suited for this kind of embedded project (though when it was bought, we did not had the eurobot in mind)
  - allow cross-development which is the whole fun of embedding. Not Intel.
  - powerful enough to support high-level operating system
  - lots of peripherals, booth built in the 68332 CPU and on the application board
  - flexible (but complex)
  - very powerful (though touchy) debugging capabilities, from inside the processor itself
  - small, lightweight, low electrical power consumption for a 32 bit CPU
- Cons:
  - NF300 CPU board as a very good quality/price ratio, but the application board is quite expensive
  - complex (because flexible)
  - touchy debugging system

The second CPU we use is a Microchip PIC 16F877. Right now it's dedicated to PID control of robot's moving.

Rationale for this:

- Pros:
  - cheap
  - fast
  - widespread
  - simple
  - popular, and as a consequence tons of applications and documentations available
- Cons:
  - no debug to this date
  - poor, if any, free/open-source compiler support

Booth CPUs are connected via an I2C bus.

Rationale for this:

- Pros:
  - cheap
  - simple
  - reasonably low I/O consumption to implement on a CPU compared to features

  - more than enough addressing space
  - native to Microchip PIC
  - lots of application circuits
  - off the shelves peripherals available and usable for robotics applications
  - flexible (master/slave mode)

- Cons:
  - serial, and so limited bandwidth

Main points, this modular design design allow us:

- *concurrent development, which is mandatory for us*
- *expand ability and thus flexibility through a bus and "modular" is always good, this design can be recycle on other projects*

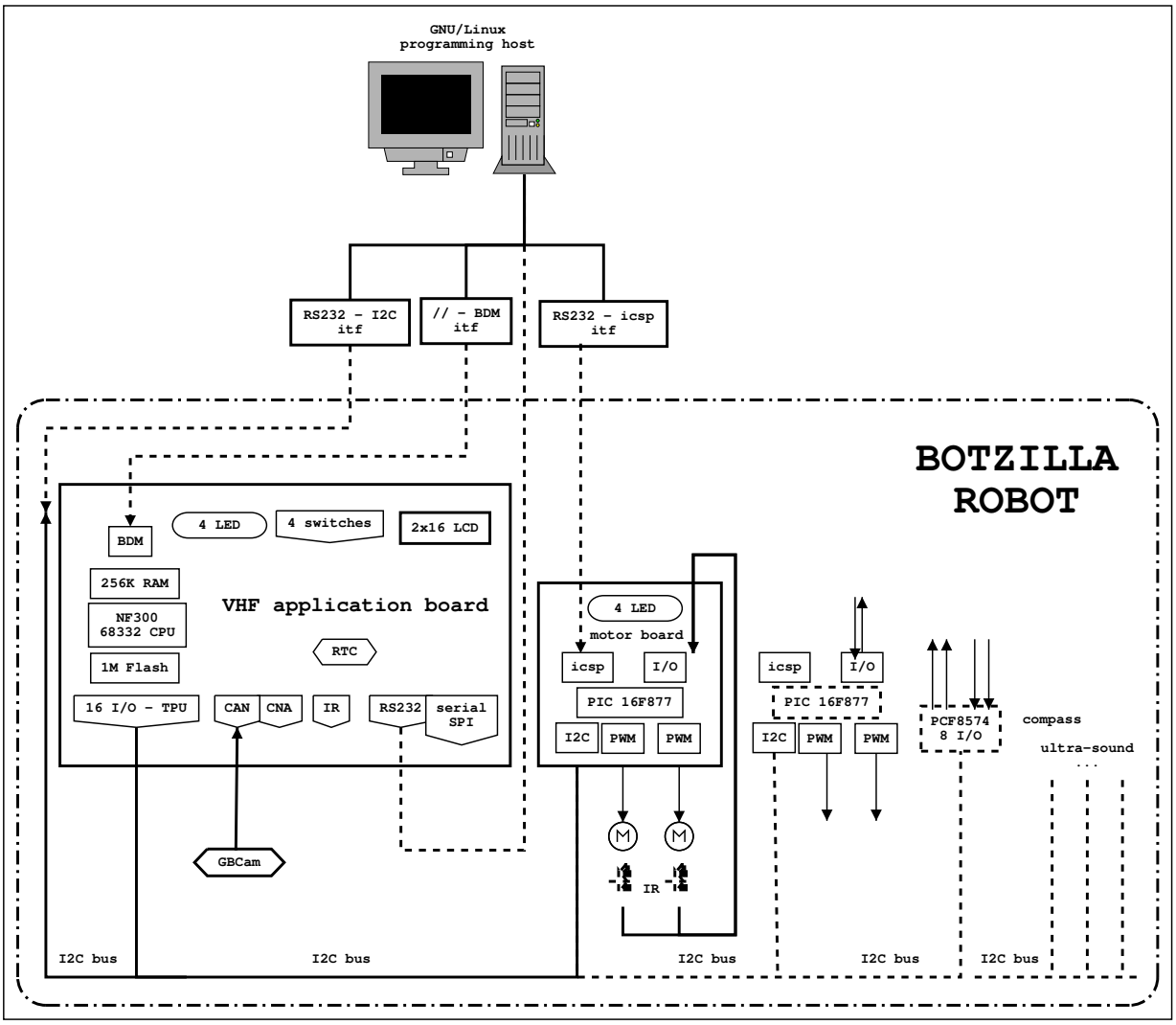

Figure 4.1: Global hardware

Chapter 5

Global software development

Figure 5.1 outline the development system. Software tools and re-used code are parts of the robots as well as gear and electric wires.

The NF300 board run the RTEMS real-time executive.

Rationale for this:

- Pros:
  - free and open source
  - mature
  - hard real-time
  - small footprint
  - GNU tool-set based
  - cleverly designed and small enough so that you can actually understand it
- Cons:
  - free and open source (DIY, read the source)

  For a more in-depth presentation of RTEMS, see part VIII.

On the host side, RTEMS relies on GCC and newlib as well as other GNU tools. Newlib is an open source implementation of the C library specially targeted towards embedded applications (small footprint and fast, for a reasonable functionality loss cost). Programming and debugging is made trough BDM (Background Debugger Mode) built in CPU32 Motorola architecture, by the way of an host driver and a parallel interface - these two last elements being quite delicate - resulting on 80% of the power of an ICE (In Circuit Emulator) for 20% of the price. The serial port built-in the 68332 provides logging capabilities.

The PIC board, right now performing mainly date acquisitions and actuators,

runs an interrupt driven applications. The development chain is an assembly of various tools, ranging from a free but close source compiler to a user space driver: the GPASM assembler is fed through the C2C compiler. Picprog is then used to program the device. The I2C bus, by the way of a serial/I2C interface (19), then give us logging capabilities as a substitute for debugging. Up to now, the PIC application is both:

- very real-time dependent (mainly data acquisition and output update)
- simple enough

... so that a purely interrupt driven code, that's to say without an OS is well suited.

For a more in depth presentation of the PIC tool chain, see part IX.

Synthesis till now:
The key point is the assets: this combination of hardware and software chain is flexible enough to be re-used on many kinds of embedded projects, being robotic or not, and allow experimentation on many techniques and tools.

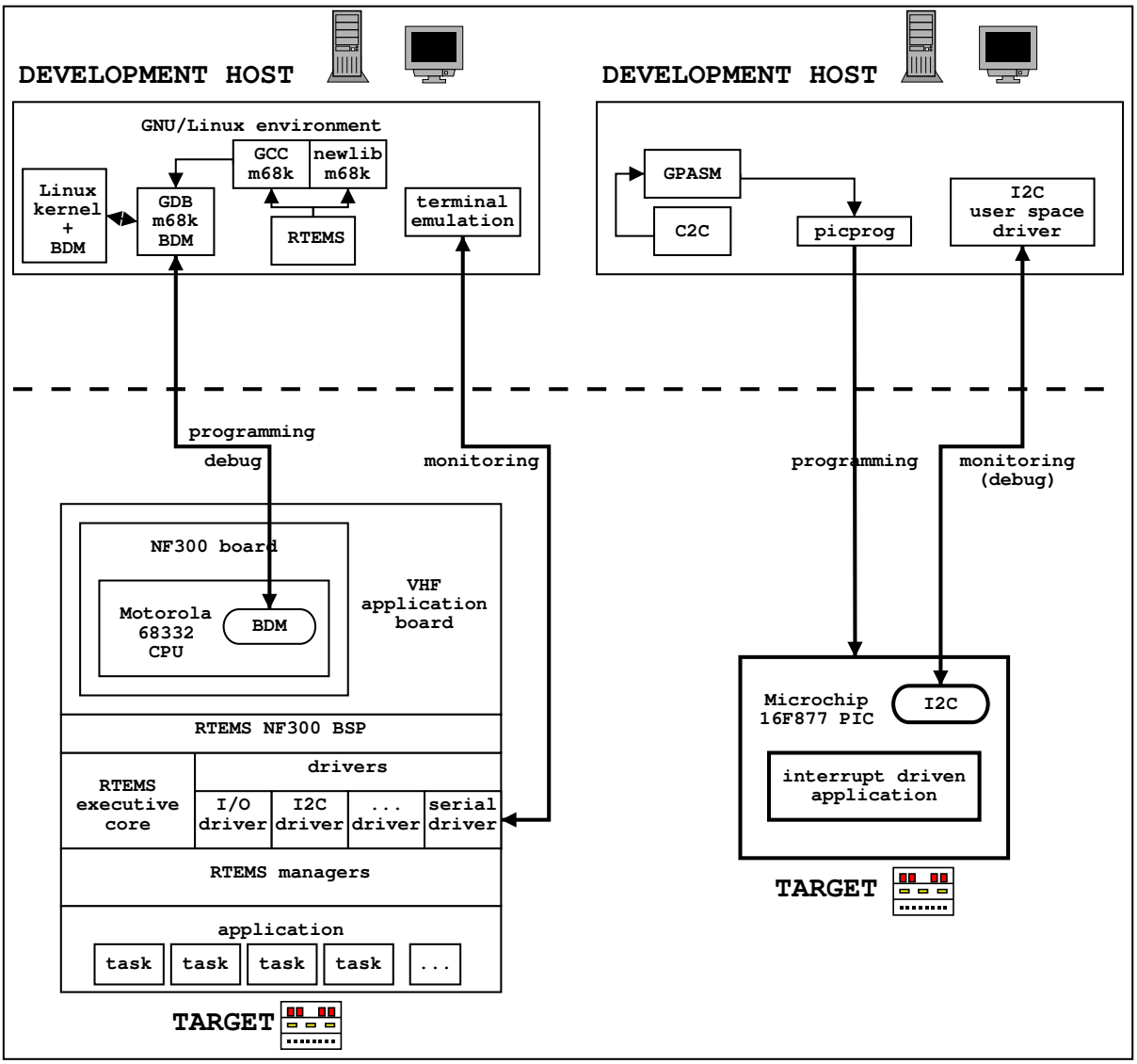

Figure 5.1: Global software development architecture

Chapter 6

Global software system

Finally, figure 6.1 show, err, the application software as we know it to this date. The NF300 will hold the knowledge about the match rules, and thus will be responsible for strategy and decisions. It will provide event logging for post-match (or post-mortem) analysis. On a smaller scale it will be responsible for:

- self positioning, based on field knowledge and input provided by the PIC motion board
- anti-collision
- target identification, grip and storage
- probably more... (to be continued)

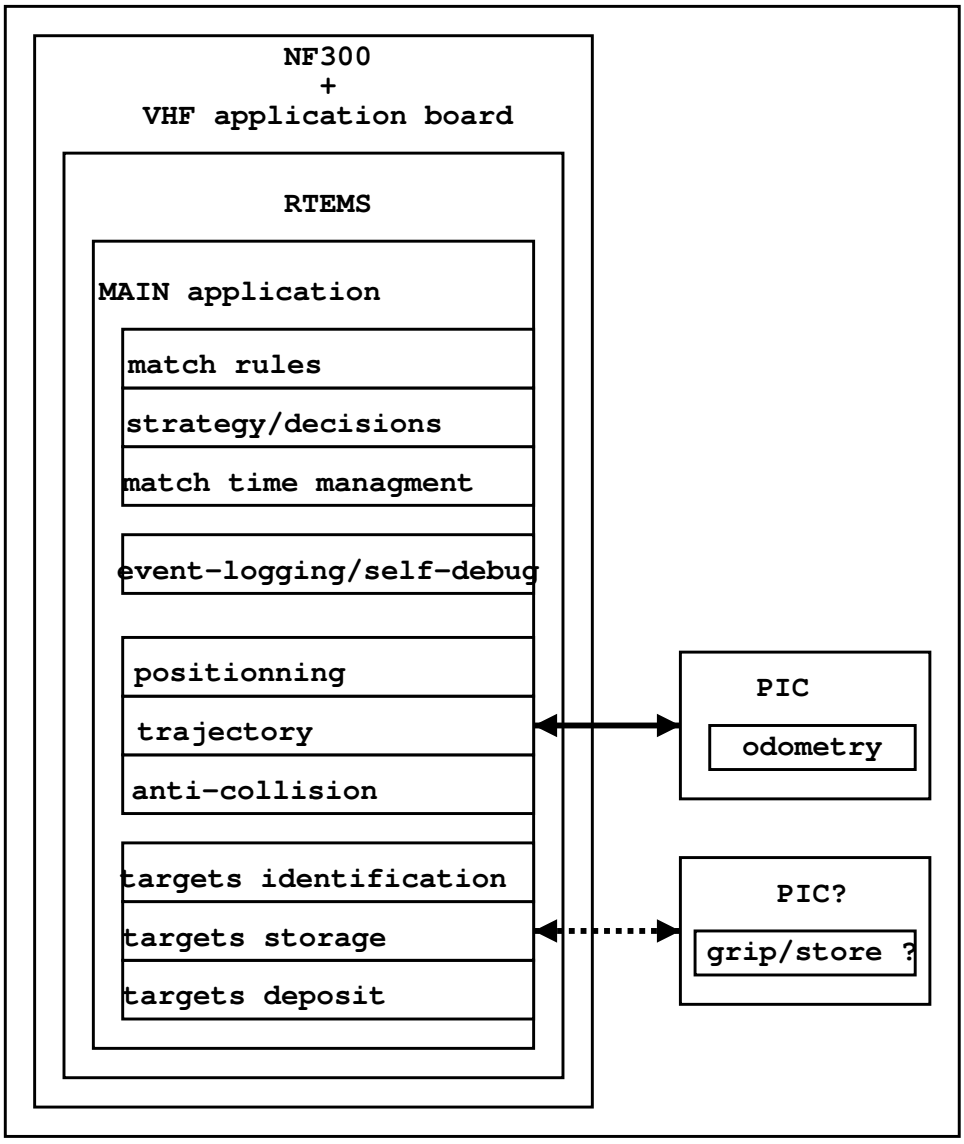

Figure 6.1: Global software system architecture

Part VII

Engineering advice

Last updated:

`$Date: 2002/11/27 20:05:54 $`

This section is especially targeted towards students aiming to enter the eurobot competition.

So far was the project was presented in a very high-level manner, so before we dig in some nifty details we would like to say this:
The three of us added reach an age of more than a century. We're supposed to have some experience, here we will try to transmit some of this experience. You will find here some general advice, questions that you should ask yourself, and other philosophical stuff. Though we hope not to, some of this may sound pedantic, provocative or "do as I say, not as I do". That's OK, just think about it.

This is not about engineering matters as in OO versus imperative programming, UML design, SART, and such. Not that these subject are not interesting, but:

- they go far beyond the scope of this document,
- they are generally introduced by your teachers, being a mandatory part of a cursus (if they're not, study them by yourselves!)
- these is about engineering as opposed to technical, low level details, and not about hardware or software engineering *methods*.

A robot is mainly a software project, not a hardware one. I'm terribly sorry if this comes as a shock. Note: the hardware guy on our own project is the first to agree with that. Love it or not, providing you have the budget everything is there for the hardware. On the other hand the robotic field as a whole is still struggling with software problems. So how does it come that most teams are from hardware cursus one may ask? Simply because a hardware person, these days, can't do without knowing a minimum of programming (note: I said "programming", not "software engineering"). There's not much they can do without this minimal software knowledge. Whereas software persons are not taught hardware - they don't need it. A hardware-based team will be able to come up with a viable robot. Notice again I said "viable", not "good". A software-based team won't be able to bring a robot at all. So if you want a "viable" robot you need hardware people. But if you want a "good" robot, you need some software people.

So a robot is mainly a software challenge, but many if not most eurobot team come with a strong hardware bias. *You do need a multidisciplinary team.* If everyone want to do the hardware, it ain't gonna work. If everyone want to do the code, it may be worse! This may be a problem - did you considered presenting your school as a whole, with various competencies from different sections, rather than a team coming from a specialized class? Believe me it's a lot of fun and much more effective to work with "another-side person": focusing on your side, your motivated because you get the feeling that your special talent is used as it may be, you learn from each other, can delegate the part that you don't feel at ease with, and ultimately this is the just the way it's done in the real

world (the industry).

Who do you need? A bare minimum would be *an application software programmer, an hardware designer, and an embedded software programmer* (that's to say someone with a good understanding of booth hardware and software) to make the joint. A mechanical designer is handy to: this is real craft, don't underestimate it. A system-administrator would be a great plus for all of them, and this need will grow as your team. If you find someone who's really good and willing at analog electronic catch him by all way, because this is a rare talent and one can can't be build overnight. Try to build such a team. There's no need at throwing more members to a project which is already unbalanced in the first place. Especially if the project is late on schedule: it takes time to integrate someone. If there's no other way, have the hardware people do so software, but never ever the other way around. Note: I (Jean-Baptiste Maillet) am a software guy, I did hardware too, believe me, I know.

Don't necessarily use what you have, use what you need. This one's for sponsored team. Example: because you have a sponsor which can provide you PCB, do not spent your time designing PCB just for the thrill of it. And don't over-design your PCB, making multi-layered design when single face would be enough. Because: your loosing time having fun with tools and means, loosing the project goal, you over-design, spending too much resources on side-features (especially bad in the industry), you become overly dependent on the sponsor. Because you would have a wheel sponsor would you put additional wheels on your robot?

Do you know what you need as technical tools and supply? Don't hesitate to challenge yourself. Don't necessarily use the tools at hand just because they are there. Don't try to adapt tools that you already know/use just because you already know/use them. This is the bad side of re-use (see below for the good side). This leads to immobility, narrow view and "bloating". If you're used to this micro-controller family, is it well suited? What about this other one? Don't forget the fun factor. You're still young and at school, there's no real risk, enjoy it and discover new things. In the industry, you'll be supposed to come up with new solutions to new problems: be creative and curious.

By all means avoid overkill. Don't use a 100,000 gates FPGA when a 10,000 would be enough. Don't use a Pentium III when you can use a 386. This is the worst kind of engineering that you can do: spending much more money than needed on a product. This is not the way it works in the industry (read this phrase again: this is not the way it works in the industry). In 1976, two spacecraft arrived on the planet Mars and after a 34 millions miles travel, collected samples, analyzed them and send back the results to earth. Do you think that more than 25 years from now they used Pentium? Or even 386? More recently the pathfinder mission launched another mission to mars: the rover vehicle is equipped with an 8 bit processor and 512 KB of memory. Right now in 2002, the launching system used for the NASA space shuttle is still made up 74xxx circuit and processor with a 64 KB address space (well OK, here it's a re-design budget versus maintenance budget, but still...) Let's take some example closer to us: during the 80's, the old Sun station a.k.a. the pizza box ran Unix sys-

tem on Motorola 68k processors at a speed of a few tens of MHz, and Apple Macintosh on the same kind of processors ran Microsoft World, Excell, or some electronic CAD system. Today, look at the processing power of your mobile phone, of your PDA or of you hand-held video-game. Do you need a Pentium III to run the eurobot?

So this is supposed to be a software project, but we're advised to use some small CPUs, then you mean we shouldn't use an operating system? Yes *you must use an OS* for anything but the simplest tasks.. What other alternatives do you have? Wired logic is OK for specialized and/or sequential subsystems. An interrupt driven state-machine is still OK for simple automate-like systems, let's say a coffee machine or a data acquisition machine. It's complexity will grow endlessly and it will become a real burden with more and more regressions as your robot will grow up. Ultimately it'll explode in complexity for the task at hand in the eurobot. How will you had devices drivers? How will you perform such simple tasks as "wait for 2 seconds on this job to see if anything happens, but in the meantime continue performing the jobs at hand", or "if the eurobot match is to be finished in 15 second, then go for the secret emergency plan right now". Will your robot be able to do several things in the same time, like moving, analyzing its environment, and store the collected targets. You do need an OS. The world hasn't waited for today's CPU to come up with operating systems, and there's an hidden world between finite state machines and full-bloated OS like Linux and Windows. OS comes in all shapes and size, from a few hundred KB to several MB. Did you considered real-time executives, like uCOS, eCOS or RTEMS? these can fit in a few hundreds of KB, provide hard real-time capabilities and a more feature rich API that you'll need. Even on 8-bit CPU like Microchip PIC, primitive operating systems are available TODO: link here.

But what if it don't get the drivers I need? *Do you mean I may have to develop my drivers myself? Yes,* I do mean that. As an hardware designer it will be you job to provide the software team with a good enough specification and possibly firmware so that they're able to do so. As an embedded software programmer it will be you daily job to deal with the bare-silicon and make it alive. Be prepared, this will be your job for years. But don't forget to re-use. If you choose an OS by its driver set, then you're probably choosing a CPU and a development chain too as a consequence of the OS - you're taking it the wrong way.

Re-use. The rule here is *"find the guy who already did it".* Admit one for all that you're probably not the first one to stumble on a given problem. Come on, you're not so special - I'm not, I know. Forget the "here, we have a really special problem that nobody can help us with" attitude. The Internet is a wonderful tool: source code, and to a lesser extent electronic designs are just there waiting for you. You want to do I2C? Do you know that not only tons of hobbyist code is available for it, but also that rock-solid production code for I2C is available in the Linux kernel? Need to do some USB? Same thing. Need a driver, a BSP? Are you sure someone didn't make it already, did you looked for? Even if you can't use it as is, there may be there the precise bit of information you just needed. And dealing with other's production will be your job too in the real world. The same goes with hardware: do you really think you have to develop a CPU board? Did you look at the existing one? Wouldn't your time

be better spend on some peripheral design (and still, there are lot of peripheral boards available). Don't worry, you'll have enough to do with integration. This can't be emphasized enough: do not re-invent the wheel.

Choose software tools first, hardware second. The hardware almost always dictate the software, the reverse is rarely true. You don't want to be dictated things. Hardware is cheap: think of the capabilities you can have with a cheap USB web-cam. Software is expensive: think of the cost of the development tools MS (Visual Studio/MS SDK?) and possibly underlying OS (Microsoft Windows?), consequently underlying hardware (Intel PC?) host CPU associated, associated with this web-cam. Does this as a whole fit your needs? Do you want to design your system as a whole because of this USB web-cam? This would be taking the problem reverse-side.

The fun factor. TODO.
The reality factor. TODO.
Invest on yourself. TODO.
Document, document, document. TODO.

Forget the ego factor. In France at last thanks to media coverage, the eurobot as became really important for some schools. For such school's teams, the goal is not to compete, the goal is to actually win. This as some unfortunate sides effects on the competition spirit itself, which is a problem, and on some team/individual behavior, which is another problem. On the former, this is ANSTJ's job. On the latter, we would like to convince you that winning is not the goal. You may be convinced that a good score at the Coupe de France de robotique/eurobot is a good point on your resume for when you'll leave school. You' re wrong. First, this score is supposed to be a team result. You can't take personal credit for it. Second, and most importantly, a good score on such competition may be a valuable point on your resume when fresh out of school, though not necessarily, but in a couple of hear as your experience will grow up this will look funnily childish and anecdotal, if not geek-ish on its worst sense. Your score on the competition will be meaningless. Whereas if you can say/put on your resume "I used/developed an operating system on 8/16 bit CPU", "I developed device drivers for this and this peripherals", "I was responsible for the system administration of a 15 persons development team", "I integrated an image analysis library in an embedded development environment", this will be hard-facts that you'll be able to take credit for. These realizations - yours - won't wear out before long and will proof your professional attitude from school time. In order to do this you should of course document your work and be ready to display it proudly (no ugly code and straps all over your boards please). I don't talk about the nifty implementation details: I talk about the how's and why's of your project. *We're not in marketing, we're in engineering: enhance your craft, not your market score.*

Part VIII

RTEMS

Last updated:

$Date: 2002/11/25 20:24:19 $

Chapter 7

What is RTEMS

RTEMS is an open-source real-time multi-tasking embedded executive. Open-source means that you have access to the whole source code of RTEMS. Thus you can tailor it to your needs, use code from others, modify the code, distribute it. Your free in the sense of "free beer" as well as in "freedom". Multi-tasking means that RTEMS can do several things at a time, and if your familiar with a modern operating system like UNIX/Linux, Microsoft Windows or Mac OS X should be familiar (think multiple-process, or more accurately in the case of RTEMS multi-threaded). Real-time is a difficult to define concept. In the case of RTEMS it means hard real-time: a predictable worst-case latency. Embedded means that RTEMS is designed to be used in small systems, or systems with less or a different interaction from the user. Embedded systems are everything that old a computer inside, but don't look like an usual computer with a keyboard, a big display on a monitor, a hard drive. Example of embedded systems may be your cell-phone, a cash-register, your VCR. If you've done some embedded or real-time work before, these concepts should be familiar.

The "executive" part of my first sentence may be trickier. Read "executive" as you may read "operating system". So what's the difference? Contrary to an operating system, an executive is linked to its application in a single binary executable. As a consequence, the executive (operating system) and its application do not have to sustain the burden of dynamically launch applications, which may imply manage a file system, a dynamic linker, multiple memory spaces (virtual memory). On the negative side, a crash of the application can also crash the whole executive. But on the positive side, as a consequence an executive can be much smaller and faster than a usual operating system. Conclusion: cheaper hardware for the same level of performances. RTEMS can run in a few tens of kilobytes. Classic example of executive are iRMX from Intel, or VRTX from Ready System. More contemporary examples may include eCOS from Cygnus/Red Hat or Micro-C/OS-II from Labrosse.

The good points about RTEMS are its open-source license, its small footprint, its level of maturity (RTEMS has passed version 4.5 after years of development) and above all its GNU/gcc base. RTEMS is based on the gcc (as GNU compiler collection) development tool-chain. By using it, you instantly benefit from gcc-s assets:

- gcc is cross-target. gcc can output code for i386, m68k, Power-PC, SPARC, Hitachi SH, ARM, MIPS, HP Power Architecture processors. gcc really shines in this field. Just say "32 bits" (or more) and gcc is there. It can do this in several object format, and can of course remote-debug all this. This is a tremendous asset. Time spend on gcc is time well spent.

- gcc is a serious, respected and mature piece of software. It is used not only for GNU/Linux based systems, but also by Sun Solaris, Apple's Mac OS X, WindRiver VxWorks. Let's synthesize: everybody uses gcc except Microsoft. Time spend on gcc is time well spent.

- gcc support development in several languages: C, C++, Java, Ada, Objective-C. RTEMS uses some: you can code in plain C, C++ and Ada. A sensible choice for embedded work. Time spend on gcc is time well spent.

- RTEMS benefits from the dynamic development of the gcc and the GNU tool-chain. RTEMS is developed and maintained by Joel Sherrill, member of gcc's steering board. Time spend on gcc and RTEMS is time well spent.

- gcc is standard compliant. Though providing some special extensions, gcc pays much attention to following standards. It may be be the only mainstream development chain to do so. Time spend on gcc is time well spent.

- gcc is free as in "free beer" as well as in "freedom". You'll always bee able to use gcc. Time spend on gcc is time well spent.

- gcc is very well supported. You can buy commercial support, or simply type the error message you get from gcc in a Google search. Chances are few that you're the first one to encounter a problem. You'll be surprised by the amount of pertinent answers you'll get from a simple Google search. Time spend on gcc is time well spent.

What part of "time spend on gcc is time well spent" didn't you understood?

Cross-development, mature and widespread, free, standard compliant, good support, what would you want more?

If you know other serious alternatives with such features please contact me (jmaillet@club-internet.fr), I'm very much interested. Serious.
Furthermore,

- RTEMS is cleanly designed. Its source code layout is divided between APIs and libs, CPUs, boards and devices. You can switch APIs, CPUs or boards without having to go through the learning curve again.

- a UNIX BSP is available. This means that you'll be able to learn RTEMS without having an target board.

- a bunch of BSPs are available. See a list here (http://www.oarcorp.com/RTEMS/Features/Ports_BSPs/po
  This list is probably outdated: it's that of RTEMS v4.5.0, dating 2001, whereas OAR releases an RTEMS snapshot every two or three months.

For more information about rtems, see OAR site (http://www.oarcorp.com/).
OAR is the company responsible for RTEMS's development. The FAQ is available here (http://www.oarcorp.com/rtemsdoc-4.5.0/rtemsdoc/html/FAQ/FAQ00001.html)

Chapter 8

Install RTEMS

8.1 Introduction

8.1.1 Goals

This is a unofficial micro HOW-TO install and build rtems.

This document will help you to install rtems on your host system for the UNIX BSP plus a specific target BSP. The process is quite straightforward and should not take more than an hour (plus download time).

It was inspired by "RTEMS on the MRM332", (c) 2002 by Joel Sherrill:

http://www.oarcorp.com/ joel/rtems/mrm332.html

This was the real starter when I first installed RTEMS. I hope to be more generic, still using a target BSP as an example, but also adding the UNIX port building procedures and a few hints.

The official documentation for this ("Getting Started with RTEMS for C/C++ Users" or "Getting Started with GNAT/RTEMS") may be a bit confusing when just starting out and/or outdated. Nevertheless, you should read this official documentation afterwards, once you'll have a basic understanding of the system: it provide much more in depth information.

8.1.2 Prerequisites

I'll assume you have a UNIX based host, ideally a Linux one on Intel x86 with an RPM-based distribution. You should also have a basic knowledge of the configure/make/install steps and their meaning. This document also assume that you want an rtems install not only for your favorite BSP/target processor family but also for the UNIX target (simulation), in C/C++. Though lacking some features, the UNIX port is handy for learning RTEMS without the burden of a cross-development environment. Regarding the target BSP, the example given is for the efi332 BSP, an m68k (Motorola 68000) based board.

8.2 Check that you have enough spare space on your system

A cross-development environment takes a lot of disk space.

Rest assure nonetheless that you won't have to download the amount announced here: these are disk space after installation on a living system.

8.2.1 RTEMS source directory

You can't do much without this. That's around 95 MB.

8.2.2 RTEMS build directory

Size vary. For a full blown BSP with C++, network, and all bells and whistles: approximately 340 MB, among these 45 MB for the documentation (that you can remove from this build directory once you have installed them). Still, you will probably want two of these build directory (one for UNIX, one for your target). Think ahead that you'll need to keep this in your home directory, or a directory shared by a group of users.

The base tools, cross-development tools, library and documentation will be installed in `/opt`, and takes around 175 MB for one each target processor family.

8.2.3 Sum it up

95 + 295 + 295 + 175 = a round up total of 860 MB

Consider your partition setup versus this and again take notes that the most part, that is to say the sources and build directory will live in user space (your home directory for instance). Also think in advance that you may want to keep in sync with OAR RTEMS snapshot releases, this may mean CVS.

8.3 Select what to download

As mentioned before I'll take the example of Linux/PC and will install Intel x86 RPM packages, but you can use `alien` for a `.deb` based distro, and rest assure that all sources are at disposal in case of more exotic situations.

The precise names I mention here, as well as the version number, where the ones I found in March 2002. Of course, this is subject to changes. You should take the latest snapshot at your disposal, and the `LATEST-xxx` tools or `TRUSTED-xxx`.

You will find the file on OAR FTP site:

`ftp://ftp.oarcorp.com/pub/`

For example for the C/C++ tools (binutils, gcc/newlib):

`ftp://ftp.oarcorp.com/pub/rtems/snapshot/c_tools/LATEST_BINUTILS/`

note: during october 2002, OAR made some changes:
http access to their ftp files is now possible:
http://www.oarcorp.com/ftp/
The documentation from the latest snapshot is now available online. Do use it at:
http://www.oarcorp.com/rtemsdoc-current/
Finally, you can now find a CD image here:
ftp://ftp.oarcorp.com/pub/rtems/cd-working

8.3.1 The basic tools

- autoconf-rtems-2.52-0.noarch.rpm 400 KB
- automake-rtems-1.5-0.noarch.rpm 320 KB

note:

There can be a misconceptions that `autoconf` and `automake` are needed in order to do the configure and make steps on source packages. This is not the case: these tools are needed by the maintainer and distributor of the sources, not by the user of these sources. If you intend to just run the `configure` script, not generate it for instance then you don't need `autoconf`. Furthermore, your Linux system probably have `autoconf` and `automake` already. On the other hand these two applications are quite versions dependent when working together. This, plus OAR providing these target independent packages in spite of the fact that they are provided by every Linux distribution lead me to the conclusion that they are mandatory. I may be wrong, information welcome on this point.

TODO: check if there's an RPM dependency between these and rtems-gcc

- rtems-base-binutils-2.11.2-1.i386.rpm 1.5 MB
- rtems-base-gcc-gcc2.95.3newlib1.9.0-3.i386.rpm 770 KB
- rtems-base-gdb-5.0-7.i386.rpm 200 KB

The `binutils` include the `gas` assembler and `ld` linker, silently called by `gcc`. `gcc` is, well, the GNU C Compiler and if you've read so far shall need no introduction. The `newlib` is a replacement C library for the `glibc`, more suitable for embedded work. `gdb` the GNU debugger may or may not be usable for your target board "as is". These and other RTEMS `gdb` packages are optional. Foe example I use so far a custom remote `gdb` for m68k.

8.3.2 Tools for you target processor

In my example:

- m68k-rtems-binutils-2.11.2-1.i386.rpm 5.8 MB
- m68k-rtems-gcc-gcc2.95.3newlib1.9.0-3.i386.rpm 9.5 MB
- m68k-rtems-gdb-5.0-7.i386.rpm 1.6 MB

8.3.3 RTEMS itself

RTEMS, as the real time executive to be compiled on your host with your application and then loaded and run on your target, is not distributed as an rpm but as sources in a bzip2 archive, always available from:
`ftp://ftp.oarcorp.com/pub/rtems/snapshots/rtems/current`
The naming schemes is:
`rtems-ss-YEARMONTHDAY.tar.bz2`
For example the latest at the date of this writing is:
`rtems-ss-20020301.tar.bz2`
It's approximately 9 MB in size.

The base tools, plus tools for a target, plus RTEMS sources: a total to download, round up, of 20 MB.

Reminder: if you want to check what's an `rpm` for and where do its file go, issue:

```
rpm -qilp name_of_the_rpm.rpm
```

All OAR RPMs will install in `/opt`, except the documentation that goes cleanly in `/usr/doc` in an `rtems` sub-directory.

8.4 Install on your host

Reminder: to install an rpm, issue:

```
rpm -ivh name_of_the_rpm.rpm
```

These will check for dependencies first.

8.4.1 The basic tools

You should install first the generic, target independent tools if needed (see above):

- autoconf-rtems-2.52-0.noarch.rpm
- automake-rtems-1.5-0.noarch.rpm

Then proceed with the base:

- rtems-base-binutils-2.11.2-1.i386.rpm
- rtems-base-gcc-gcc2.95.3newlib1.9.0-3.i386.rpm
- rtems-base-gdb-5.0-7.i386.rpm

8.4.2 Tools for you target processor

Same `rpm -ivh` command, with my m68k example target for:

- m68k-rtems-binutils-2.11.2-1.i386.rpm

- m68k-rtems-gcc-gcc2.95.3newlib1.9.0-3.i386.rpm
- m68k-rtems-gdb-5.0-7.i386.rpm

Don't forget to edit your shell configuration files to have `/opt/rtems/bin` in your path, since this is where the tools have been installed. Still with the m68k example, you should see with your new path setup:

- m68k-rtems-addr2line
- m68k-rtems-ar
- m68k-rtems-as
- m68k-rtems-c++
- m68k-rtems-c++filt
- m68k-rtems-cpp
- m68k-rtems-g++
- m68k-rtems-gasp
- m68k-rtems-gcc
- m68k-rtems-gdb
- m68k-rtems-ld
- m68k-rtems-nm
- m68k-rtems-objcopy
- m68k-rtems-objdump
- m68k-rtems-protoize
- m68k-rtems-ranlib
- m68k-rtems-readelf
- m68k-rtems-size
- m68k-rtems-strings
- m68k-rtems-strip
- m68k-rtems-unprotoize

8.5 Building

8.5.1 Building for your target

In your home create an rtems directory, uncompress and untar the archive in it, this will make an `rtems-ss-VERSION` subdirectory holding the sources. I want to build for an efi332 board, so besides this rtems sources create a `b-efi332i`. This will be the build directory. Move in this build directory, and configure:

```
../rtems-ss-VERSION/configure [options]
```

More specifically, example here for the efi332 which is an m68k target:

```
../rtems-ss-20020301/configure --target=m68k-rtems --enable-rtemsbsp=efi332
```

This should takes less than 1 minute. Or even better, just a bit longer:

```
../rtems-ss-20020301/configure --target=m68k-rtems --enable-rtemsbsp=efi332 \
--disable-itron --enable-cxx --enable-tests --enable-docs --enable-networking
```

Explore rtems sources, `README` and configure script for available options. Try `configure --help`.

The latter example will enable the POSIX API (default), but not the ITRON API, enable C++, enable the test application suite, the docs (a must), and networking. Launch:

```
make
```

Wait a few minutes. Your build directory will grown up to 140 MB.

Reminder: this build directory now contain a `config.status` file, generated automatically by the configure step. It can be run (= it's a shell script) to recreate the current configuration again.

But don't do this! Once you're able to build with a bloated configure like described above:

Issue a `make clean`, then `make` again but this time logging the build. Keep this log for future reference (for instance: how many and what kind of warnings with a clean source tree for this full blow configuration).

Then do a `make install`, or at least a `make install-doc`. The documentation, as html, ps, pdf and xdvi will go in `/opt/rtems/share/rtems/`. Now, the build process you have gone through is long, and you probably don't want all the features you enabled in the configure options. Even just scanning through the doc sources to check the dependencies without rebuilding anything will consume some time for the `make` program.

So let's clean the build directory and make the build shorter with a more reasonable configuration:

```
../rtems-ss-20020301/configure --target=m68k-rtems --enable-rtemsbsp=efi332 \
--disable-itron --enable-tests
```

And then `make` again.

Where are the application files?

You will find some `.exe` files and your build rtems: these may be, depending on your target CPU, Intel HEX or Motorola SRECORD files. Such type of files are ASCII translation of addresses and binary code or data. You'll use this if you have a device programmer of some sort, and in production stage only. For daily code/debug cycle, you will want to use the `.nxe` files: these are elf object files with the debugging symbols. `gdb` handle this. For example, if your build directory is `b-efi332`:
`b-efi332/m68k-rtems/c/efi332/tests/samples/hello/o-optimize/hello.nxe`
This `hello.nxe` is the classic "hello world" app, supposed to print on a serial port of your target.

8.5.2 Building for your UNIX host

Side by side with you `rtems-ss-DATE` and `b-TARGET` directories, create a `b-unix` directory and move to it. Then issue:

```
../rtems-ss-DATE/configure --target=i586-pc-linux --disable-itron \
--enable-tests --enable-docs --enable-cxx --enable-networking
make
```

Then `make install` if you which, but there's no real point on installing this: these are demonstration application and object code that you'll use from the build directory, not "real" programs.

Note that the exact target option may vary. I had to scan through the build log to see what was missing and made symlink between what RTEMS was looking for and my gcc tools. I built RTEMS-UNIX on other similar hosts without such problems (sorry, did not kept notes for this).

There's no `*.nxe` files for the UNIX port. The `*.exe` files are regular elf files. So that you can now try:

```
./i586-pc-linux/posix/tests/hello.exe
```

You should see:

```
*** HELLO WORLD TEST ***
Hello World
*** END OF HELLO WORLD TEST ***
```

That's it, it runs.

8.6 Where to go from here

Essential readings among the various documentation that you installed in the steps above (look in your `/opt/rtems/share/rtems/html/`) are:

- RTEMS FAQ
- RTEMS development environment guide
- Getting Started with RTEMS for C/C++ Users

You should also perform and record the output of the various test suite to get an idea of where you start from. The sp (Single Processor) tests also uses most of rtems library and thus makes an excellent learning material.

For any remaining questions, first search on rtems user mailing list archive:
http://www.oarcorp.com/rtems/maillistArchives/rtems-users/

TODO - a chapter on "Remote debugging"
TODO - a chapter on "Doing an RTEMS board support package for your target"
TODO - Maybe a chapter with "A few hints"

8.7 RTEMS BSP for NF300 board

For information on RTEMS, see:
OAR site (http://www.oarcorp.com/)
For information on NF300, see the `README` file from this BSP:
README (http://botzilla.free.fr/rtems/README)
Here you will find a patch adding an NF300 BSP to rtems 2002/03/01 snapshot (about 920 KB):
rtems-ss-20020301_nf300.patch.bz2 (http://botzilla.free.fr/rtems/rtems-ss-20020301_nf300.patch.bz2)
This patch was generated with:

```
LC_ALL=C TZ=UTC0 diff -Naur rtems-ss-20020301 \
rtems-ss-20020301_nf300 > nf300rtemsbsp.patch
```

My `rtems-ss-20020301` directory being a clean RTEMS snapshot, and my `rtems-ss-20020301_nf300` directory being the same snapshot with added support for the NF300 board.

To apply this patch:

Uncompress the patch, you'll obtain an 8MB patch file. With this patch side to side to a clean RTEMS source directory (snapshot 20020301), issue:

```
patch -Np1 -verbose < rtems-ss-20020301_nf300.patch
```

Now you can build for NF300, for example:

```
mkdir b-nf300
cd b-nf300
../rtems-ss-20020301/configure -target=m68k-rtems -enable-rtemsbsp=nf300 \
-disable-itron -enable-cxx -enable-tests -enable-networking
make
```

A few remarks:

- This BSP is beta, it has not been fully tested. It builds and run. It prints, but unreliably. It has only been run from RAM, not tested on FLASH.
- It has been ported from the `efi332` BSP. The differences between these two BSPs are:
- different linker scripts,
- references to `NF300_XXX` in the code instead of `EFI_XXX` (this should almost be `sed` work...) to avoid confusing references,
- some differences in the chip selects due to different memory layout,
- no `GET_CTS` macro in the console code,
- default baud rate set to 9600.

So it should run no more, no less than the efi332 BSP.

Additional info:

CVS tag and changelog from OAR's files have not been edited/deleted, autoconf/automake files have been edited by hand to provide the new make targets. Yes I know, the `configure` scripts and various `Makefile.in` should really be regenerated.

note: considering that OAR releases a new RTEMS development snapshot every two or three month, this BSP based on the 2002/03/01 snapshot is clearly outdated. My RTEMS tree is placed under CVS control, and it shouldn't be a big deal to update it. Thanks to the clear layout of RTEMS source tree, it isn't a big deal anyway. I did this kind of work three times already, it takes no more than a couple of hour. On the other hand, considering that this project is done on my spare time, I do not have "a couple of hour" to devote to such work for now, and would rather spend this time on drivers and application development or documentation. This would be update frenzy. If you are an NF300 owner, or for any valuable other reason you would like to see this BSP updated, please contact me. We'll sort this out.

Chapter 9

RTEMS build system

So far you've build, and hopefully loaded and debug on your target some sample applications delivered with RTEMS source package.

In order to have a fully functional development environment, you still miss a convenient way to build your applications without touching rtems source tree (you don't want to do some dirty hacking in your source tree, don't you?). Such a build can be quite complicated in the link phase to say the less, unless you're an expert at GNU make, autoconf and automake. You don't need to have such an expert at hand: OAR provides a build system.

Furthermore, this build system is very convenient and well documented. The only thing to know is that it's not documented in the main user doc in `/opt/rtems/share/rtems/` pointed above(TODO: send an rtems FAQ source patch about this?).

In `opt/rtems/make/`, you'll find a `README`. Read it. I won't go into much details here because all there is to know is in this `README`, but rather demonstrate it.

Everything rely on a single environment variable that you have to define, `RTEMS_MAKEFILE_PATH`. If you study a bit the templates makefiles, you'll see that it's relative to this path that the build system expect to find other makefiles, config files and libraries ready to be linked with your source code. For such files to be available when the build will need them, you must perform the `install` step for the BSP that you'll want to use. Let's say, to continue our example, for the efi332 and UNIX BSPs. You don't want to do such `install` every other day, so you can do this with the more bloated configuration that you can think about so that libraries ready to be linked will be ready to use once for all: `--enable-cxx` and `--enable-networking` are good candidate even if you don't think you'll be using these features for a while.

Once you've done such `install` for the UNIX BSP and your target BSP (example efi332), you'll find:

- in `/opt/rtems/m68k-rtems/efi332` a `lib` subdirectory with object code ready to be linked, and a `make` directory containing configuration info,
- and similarly, in something like `/opt/rtems/i586-pc-linux/posix` a sim-

ilar tree for the UNIX applications.

These are the directories where your environment variable `RTEMS_MAKEFILE_PATH` should point to, according to the target you want to build for.

Put this to use. In your home working directory, create a tree like this copying the sources from the 'hello' sample of the RTEMS sources.

```
|
‘-- my_hello
    |-- init.c
    ‘-- system.h
```

Now copy the templates makefiles from `/opt/rtems/make/Templates`:

```
|-- Makefile.dir
‘-- my_hello
    |-- Makefile.leaf
    |-- init.c
    ‘-- system.h
```

Rename these makefiles and edit them according to the build system described in the `README`. That's to say, in the directory makefile the line:

```
SUBDIRS=my_hello
```

And in the leaf makefile the lines:

```
H_FILES=system.h
C_PIECES=init
```

Don't forget to clean up the lines where OAR states things like ‘‘`add your local stuff here`’’. You end up with this:

```
|-- Makefile
‘-- my_hello
    |-- Makefile
    |-- init.c
    ‘-- system.h
```

Now is the time to define our environment variable:

```
export RTEMS_MAKEFILE_PATH=/opt/rtems/m68k-rtems/efi332
```

You can now issue `make` and voila, you have a nice hello application build in an new `o-optimize` subdirectory, ready to load and run on your target. Want to change for an UNIX build?

```
make clean
export RTEMS_MAKEFILE_PATH="/opt/rtems/i586-pc-linux/posix
make
```

It's done.

If you've ever tried setting such a cross-development build system from scratch, you'll be glad OAR did this for us (truth to be said, if they didn't RTEMS would be very difficult to use at all). Enjoy.

Chapter 10

Other

Here are some pieces of RTEMS information that don't fit in other sections but still may be useful.

10.1 An HTML tables of content for RTEMS single processor test suite

Save yourself a few keystrokes and wanderings in RTEMS sources using these tables of content.

What is it?

In the RTEMS documentation, RTEMS C User's Guide is RTEMS API reference manual. As a reference manual, it's not a tutorial showing RTEMS primitives in action on real applications. For this learning purpose, the sp (Single Processor) test suite in the source code of RTEMS itself is best suited, and said to "test 98% of RTEMS library". You'll need it also to understand the code layout that is supposed to be used (i.e. `system.h, init.c,` then probably some `task_xxx.c` files).

Of course you can explore randomly among these sp test source directory: each of them include a `.doc` file describing the features tested. Unfortunately, there is no central `.doc` file to use as a table of content when you're looking for something in peculiar.

Say you're looking for something about messages: this is demonstrated in the sp13 test.

The files below are simply concatenation of the sp test's `.doc` files, hence the table of content, in HTML. For each test a link allow a visit of the source directory.

Testing with Mozilla, Galeon, Konqueror, lynx, Netscape Navigator and MS Internet Explorer, some on Linux, Mac OS X or MS Windows raises the need for two files:

- The single window version, sptest_toc.html (http://botzilla.free.fr/rtems/sptest_toc.html) Once gone into a source directory, when coming back you'll find yourself at the top of the toc, not from where you came from. HTML's anchor seems useless. lynx does not have this annoying behavior.
- The open-other-window version, sptest_toc_ow.html (http://botzilla.free.fr/rtems/sptest_toc_ow.html) Will open a new window when going to source directories. lynx will be fine with that too in it's spartan terminal.

My advice: use lynx.

Furthermore, lynx display C syntax in color. Of the other navigator tested, only Konqueror do that.

Just save this files, put them above your RTEMS source directory, and browse (you may have to edit the path to your RTEMS sources).

Part IX

PIC

Last updated:

`$Date: 2002/12/04 19:42:10 $`

PIC are microcontroller made by Microchip, not very expensive, easy to use and quite popular.

In this document, the only PIC we present is the `PIC16F877`:

- RISC CPU (only 35 instructions) 20MHz
- 8K of FLASH Program Memory
- In-Circuit serial programming via 2 pins
- 368 bytes of Data Memory
- 256 bytes of EEPROM Data Memory
- 14 Interrupts
- 3 timers
- 2 PWM modules
- I2C module

Chapter 11

How to use a PIC

11.1 A PIC tool box

In a minimal tool chain you need:

- a text editor,
- a pic assembler,
- a pic programmer: way to program the pic (soft and hard to actually transfer your program on the microcontroller).

But it's also a good idea to use a more high level programming language like C. All these tools are freely available under Linux (you could also find it under DOS, except for the C compiler that you have to pay).

Unfortunately, there is no simple and cheap way to debug on a PIC.

Chapter 12

The assembler: GPASM

GPASM is a part of the gputils package. To install it under Linux download the tgz file (I use the version 0.10.2) and follow the instructions of the INSTALL file (./configure, make, and make install).

Then, to build a `.hex` file from an `.asm` file:

```
gpasm file.asm
```

the result is a file named `file.hex` ready to be transfer on the PIC.

There are 2 important options: –dos (if you add to produce a hex file with dos newlines), and -w1 to suppress useless warning like

```
(Message [302] Register in operand not in bank 0.  Ensure bank bits
are correct.).
```

Chapter 13

C Compiler for PIC

13.1 Install C2C

Unfortunately, to this date, there is no efficient free and open-source C compiler for PIC. But `C2C` is an efficient postcardware. In fact, the new version of C2C for Linux is no more a postcardware: it's a shareware without evaluation period (44.76 Euro) available here: C2C (http://www.picant.com/c2c/c.html). I have never test this new shareware version.

To install the free postcardware version under Linux, you download a binary archive C2C Linux Version 3.27 (http://botzilla.free.fr/pic/c2cl327e_version_linux.zip).

```
This application is distributed as postcardware for non-profit use and
as shareware for commercial use.
For non-profit use you may use the compiler as long as you wish. If you
like it I ask you to send me any nice postcard.
For commercial use you have to register the compiler. To register run
the 'register.exe' application under MS Windows or go to the compiler
home page and follow the instructions.
```

For this Linux version there is no code size, date, or target restriction. It's a fully operational C compiler. C2C is not very big (735Ko) and could be put in `/usr/local/bin` to be usable for all users. It's distributed with a small help file, and some examples.

13.2 C2C limitations

C2C is not GCC, there are some major limitations that you have to known:

- No real local variables: A call to function which uses local variables may change the values of these variables in the caller.
- Only one constant type is supported: const char.
- Maximum of 2 operands in a 16 bit expression.

And as you may already known, there are also some limitations due to PIC microcontroller:

- Only unsigned data types.
- No floating point data types.

There is a last point that you have to be aware of (even if it's not a limitation, more a feature): C2C will automaticly generate the necessary bank changed in your code.

13.3 using C2C

To compile a program for a `PIC16F877`, try for instance:

```
c2c -SRC -PPIC16F877 file.c
```

The `-SRC` option is used to insert C-code as comments into assembler file (file.ASM) (it's far more easy to read).

After what, to use the assembler file generate by C2C you have to add some header lines (with configuration options). For example:

```
;=========================================
        list p=16f877
        #include <p16f877.inc>
        __config _CP_OFF & _WDT_OFF & _BODEN_ON & _PWRTE_ON & _HS_OSC & _WRT_ENABLE_ON & _LVP
;=========================================
```

In an assembler file, use `;` to comment a line. And every line have to start with a tabulation. Then, go to the GPASM chapter to build an hex file ready to be programmed to your PIC.

Chapter 14

Picprog: A Pic programmer

The choice of a programmer closely depends on the hardware you want to use. We use `Picprog1.1` to program the PIC via a PC regular serial port (see chapter 18 for the hardware design). Picprog is simple, efficient and open source. This programmer has been design for `PIC16F84` but it works fine with the whole `PIC16xx` microcontroller family. For more informations go to Picprog 1.1 (http://gyre.weather.fi/ jaakko/pic/picprog.html).

Chapter 15

Makefile

Since, you use UNIX, all these commands can be put in a Makefile like this:

```
# ====================================================================
C_FILE=file.c
# ====================================================================
ASM_FILE=$(addsuffix .asm, $(basename $(notdir $(C_FILE))))
ASM_COMPLET_FILE=$(addsuffix _complet.asm, $(basename $(notdir $(C_FILE))))
HEX_FILE=$(addsuffix _complet.hex, $(basename $(notdir $(C_FILE))))

all:
        c2c -SRC -PPIC16F877 $(C_FILE)
        cat header.asm $(ASM_FILE) > $(ASM_COMPLET_FILE)
        gpasm --dos -w1 $(ASM_COMPLET_FILE)

prog:
        picprog --erase --burn --device=pic16f877 --input $(HEX_FILE) --pic /dev/ttyS0

clean:
        rm -f $(ASM_FILE)
        rm -f $(ASM_COMPLET_FILE)
        rm -f $(HEX_FILE)
# ====================================================================
```

Chapter 16

PIC 16F877 tutorial

Of course, you HAVE TO read the data sheet from Microchip. But, it's not easy to understand without some examples, and unfortunately the examples provided by Microchip are poor and not very clear (and written in assembler). In this chapter, I'll just give you some tips to start.

16.1 IO

16.1.1 PORTA

Port A is a 6 bits bi-directional port. It's not the easiest IO port because pins RA0-RA3 and RA5 are multiplexed with analog inputs, and pin RA4 is multiplexed with Timer 0 module clock input.

To use IO rather than AN module use the `ADCON1` register:

```
ADCON1=ADCON1|0x06
```

To switch pins as Output or Input use the `TRSIA` register (0 for output, 1 for input).

```
// Example1 from http://botzilla.free.fr/
// Code design for PIC16F877
// To compile with c2c -SRC -PPIC16F877 example1.c

char ADCON1@0x9;  //(bank 1)
// C2C automaticaly define some classical register (PORTA, TRISA,...) but not ADCON1
// so you have to define it (go to C2C doc for more informations about this syntax)

void init_portA()
{
   PORTA=0;//No need to switch to bank 1, C2C do it.
   TRISA=0;//using PORTA as output
   ADCON1=ADCON1|0x06;//using IOPort instead of AN module
}

void wait()
```

```
{
   char i;
   for(i=0;i<255;++i)
     nop();// generate an empty instruction (a busy wait)
}

main()
{
   init_portA();
   PORTA=0;
   while(1)
     {
        PORTA=1;//0001
        wait();
        PORTA=3;//0011
        wait();
        PORTA=7;//0111
        wait();
        PORTA=15;//1111
        wait();
        PORTA=0;
        wait();
     }
}
/*
    EOF
*****************/
```

To test this program:

- Compile:

  ```
  c2c -SRC -PPIC16F877 example1.c
  ```

- Add your configuration header lines:

  ```
  cat header.asm example1.ASM > example1_with_header.ASM
  ```

- Build the HEX file:

  ```
  gpasm --dos -w1 example1_with_header.ASM
  ```

- And transfer it on the PIC:

  ```
  picprog --erase --burn --device=pic16f877 --input example1_with_header.ASM --pic /dev/tty
  ```

16.1.2 Others IO ports: PORTB, PORTC, PORTD, PORTE

For the other IO ports, it's more or less the same thing: use the TRISB, TRISC, TRISD or TRISE to choose input or output for each bits. But, PORTB pins RB7:RB4 could be use for Interrupt on change feature. This kind of interrupt

inputs are very precious so keep it in stock for later. For the PORTC, pins RC4:RC3 could be use for I2C bus communication, and pins RC2:RC1 could be use for PWM commands. etc,... As you see, you always have to check the data sheet to choose the right initialize for each IO port.

16.2 PWM

An important feature of this PIC16F877 is the PWM modules. Pulse Width Modulation (PWM) is an efficient light dimmer or DC motor speed controller. In our robot, we use these 2 modules to control the two motors.

The important steps to configure the PWM modules are:

- configure the Timer2 for the PWM period
- use CCP1CON and CCP2CON to choose the PWM mode and the 2 less significants bits of the duty cycle
- use CCPR1L and CCPR2L to choose the duty cycle (the speed of motor1/motor2)
- make the CCP2, CCP1 pin an output by clearing the TRISC<1:2> bits

0ne problem with the PWM for speed motor control is the choice of a correct PWM period. We have obtained good results with the longest available period: 0.81ms (1220Hz).

```
void init_pwm_and_timer2()
{
   CCP1CON=0;
   CCP2CON=0;
   T2CON=0;// Timer2 control register (PostScale|Off|PreScale)
   set_bit(T2CON,1);//PreScale 1:16
   TMR2=0; // Reset timer2
   // set the pwm period by writing to the PR2 register
   PR2=0xFF;// period is ((PR2)+1)x(prescale)x(50x4)ns=0.81ms
   // set the PWM1 duty cycle by writing to the CCPR1L register an CCP1CON<4:5>
   CCP1CON=0x0C;
   // 0 -> 2 LSbs of the PWM duty cycle
   // C -> PWM Mode
   //CCPR1L=0xDF;// duty cycle -> 1101 1111 00 = 892x50xprescale=0.71ms (87%)
   //CCPR1L=0x80;// duty cycle -> 1000 0000 00 = 512x50xprescale=0.4ms (50%)
   CCPR1L=0xFF;// duty cycle -> 1111 1111 00 = 1020x50xprescale=0.8ms (100%)
   // set the PWM2 duty cycle by writing to the CCPR2L register an CCP2PON<4:5>
   CCP2CON=0x0C;
   // 0 -> 2 LSbs of the PWM duty cycle
   // C -> PWM Mode
   //CCPR2L=0xDF;// duty cycle -> 1101 1111 00 = 892x50xprescale=0.71ms (87%)
   //CCPR2L=0x80;// duty cycle -> 1000 0000 00 = 512x50xprescale=0.4ms (50%)
   CCPR2L=0xFF;// duty cycle -> 1111 1111 00 = 1020x50xprescale=0.8ms (100%)
   //make the CCP2, CCP1 pin an output by clearing the TRISC<1:2> bits
   TRISC=TRISC&0xF9;
```

```
   set_bit(PIE1,1);//Timer2 IE
   clear_bit(PIR1,1);//Timer2 clear IF
   //set_bit(T2CON,1);// Prediviseur a 16
   set_bit(T2CON,2);// Turn ON Timer2

}
```

16.3 Interrupt

With C2C, the interrupt dispatch is done in a special function with this reserved prototype:

```
void interrupt(void);
```

The context is automaticly saved, and restored. Keep in mind that an efficient interrupt routine has to be short to avoid overflow interrupt problems. And don't forget that you often have to reset an interrupt flag (for timer, I2C, or an input interrupt).

```
void interrupt(void)
{
   // context is automaticaly saved, and restored at the end of interrupt
   if (INTCON&00000100b) timer0_interrupt();
   if (PIR1  &00000001b) timer1_interrupt();
   if (PIR1  &00000010b) timer2_interrupt();
   if (INTCON&00000001b) sensor_interrupt();
   if (PIR1  &00001000b) i2c_slave_interrupt();
}
void timer0_interrupt()
{
   clear_bit(INTCON,2);//reset the interrupt flag
}
void timer1_interrupt()
{
   clear_bit(PIR1,0);//reset the interrupt flag
}
void timer2_interrupt()
{
   clear_bit(PIR1,1);//reset the interrupt flag
}
void sensor_interrupt()
{
   clear_bit(INTCON,0);//reset the interrupt flag
}
void i2c_slave_interrupt()
{
   if (SSPSTAT&0x04)
     {
        //I2C Slave Transmission
        SSPBUFF=0;
```

```
        set_bit(SSPCON,4);// CKP, SSPBUFF must be write before
        clear_bit(PIR1,3);//reset the interrupt flag
     }
   else
     {
        //I2C Slave Reception
        tempI2C=SSPBUFF;
        clear_bit(PIR1,3);//reset the interrupt flag
     }
}
```

16.4 I2C

As you could see in the `init_i2c_slave` function, to initialize I2C on the PIC, the major steps are:

- custom your `SSPSTAT` register
- use `TRISC` to put `SCL` and `SDA` as inputs
- custom your `SSPCON` register
- choose an address for the PIC with `SSPADD` register
- start the I2C interrupt handler with `PIE1` and `PIR1`

There are two kinds of I2C behavior: master or slave. For the moment, I still haven't used the master mode.

Slave mode, doesn't mean that the PIC could only receive data, but that all the exchanges are initiated by an external master. In the previous example we use the slave mode. When, a master send a message to the PIC, this generate an interrupt with a flag on `PIR1` register.

```
void init_i2c_slave()
// I2C IE : PIE1<3>
// I2C IF : PIR1<3>
{
   set_bit(SSPSTAT,7);// standard speed mode
   clear_bit(SSPSTAT,6);// input level conform to I2C spec
   clear_bit(SSPSTAT,0);// BF buffer full bit
   //SCL and SDA as input
   set_bit(TRISC,3);
   set_bit(TRISC,4);
   //SSPCON<5> enable bit (1)
   //SSPCON<3:0> 0110 (slave, 7 bits address)
   clear_bit(SSPCON,7);// WCOL Write Collision Detect bit
   clear_bit(SSPCON,6);// SSPOV Receive Overflow Indicator bit
   set_bit(SSPCON,5);// Synchronous Serial Port Enable
   set_bit(SSPCON,4);// CKP
   clear_bit(SSPCON,3);//0110: I2C slave mode: 7 bits address
   set_bit(SSPCON,2);
```

```
   set_bit(SSPCON,1);
   clear_bit(SSPCON,0);
   SSPADD=0x0E;// 0000 1110 -> 7 bits address= 0000 111
   //SSP IE
   set_bit(PIE1,3);
   clear_bit(PIR1,3);
}
```

Then, we test if it's an incoming message, or a request for a transmission with `SSPSTAT` register. If it's a reception you have to read the `SSPBUFF` register (if you don't it will stop the I2C interrupt handler). If it's a transmission you have to write in the `SSPBUFF` register. In all the case, at the end, don't forget to reset the interrupt flag (here `PIR1:3`).

```
void i2c_slave_interrupt()
{
   if (SSPSTAT&0x04)
     {
       //I2C Slave Transmission
       SSPBUFF=0;
       set_bit(SSPCON,4);// CKP, SSPBUFF must be write before
       clear_bit(PIR1,3);//reset the interrupt flag
     }
   else
     {
        //I2C Slave Reception
        tempI2C=SSPBUFF;
        clear_bit(PIR1,3);//reset the interrupt flag
     }
}
```

Chapter 17

Useful links

Microchip (http://www.microchip.com):
For all the device data-sheets.

GPutils (http://gputils.sourceforge.net):

Picprog 1.1 (http://gyre.weather.fi/ jaakko/pic/picprog.html):
A simple way to program PIC microcontroller (Hardware and Open source soft for Linux).

A PIC development Tutorial (http://www.mastincrosbie.com/mark/electronics/pic/pic.html):
A good way to start with some advises and examples for using libraries and Makefile.

Fribotte: PIC et Linux (http://fribotte.free.fr):
A good way to start (in French).

Part X

PID

When you design a rolling robot with 2 motors (the classic differential steering configuration), one of the very first problem is to go on a straight-line path. Even a small difference between the 2 motors introduce an important deviation, so it's necessary to use a regulation process. PID regulation is a well known algorithm that takes in account Proportional, Integral and Derivative control.

There is lot of information about PID regulation and robots, for example, there are very precise details about the ways to tune the 3 ponderations (KI, KP, KD) in thousands of web pages, but, very often there is no information about a basic implementation choice: the clear definition of the system. In fact, all you need to start is in this article "Using a PID-based Technique For Competitive Odometry and Dead-Reckoning"
http://www.seattlerobotics.org/encoder/200108/using_a_pid.html
but, you may miss the the key point (as I did first), which is, you have to clearly choose the global regulation strategy. There are several ways to view this regulation system:

- 2 independent regulations (one for each motor)
- 2 independent regulations, `PLUS` a global regulation (to control the real position of the robot)
- 2 shared but independent regulations: one for the speed, and one for the orientation
- only one regulation on the orientation (because the actual speed is not very important).

Lots of people don't clearly say the type of regulation strategy they used, and it seems that beginners often use the first strategy because it's the simplest (tough probably the least effective). As it was said in "Using a PID-based Technique For Competitive Odometry and Dead-Reckoning", the last option (a single regulation on orientation), is a reasonable solution (simple and efficient).

Let's explore this solution: we consider that there is only one global system (the `Robot`), and not 2 (on for each motor). We start with a common command on the 2 motors (`Cini=50%`), and we make a regulation on the shift between these the right command (`Cr`) and left command (`Cl`) (Figure 17.1).

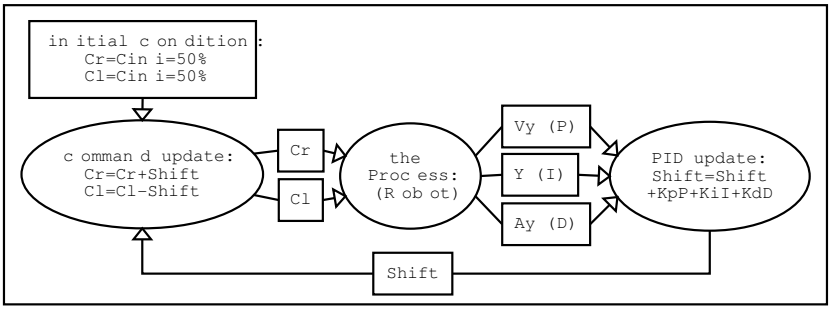

Figure 17.1: PID strategy, one regulation on the shift between Right and Left command (Cr and Cl) by using the 3 terms: Proportional (lateral velocity Vy), Integral (lateral error Y) and Derivative (lateral acceleration Ay)

So, now that we have defined the regulation strategy, we go back to a more classical discussion about the importance of the 3 regulation terms (Proportional, Integration, and Derivative) with some simulations. On all these simulations, differences between the motors are the same, and ponderations between terms (when they exist) are the same.

First, with only P regulation: on figure 17.2, you see that there is no real convergence of the system. There is a kind of over-correction due to a lack of anticipation, so the system stay in oscillation.

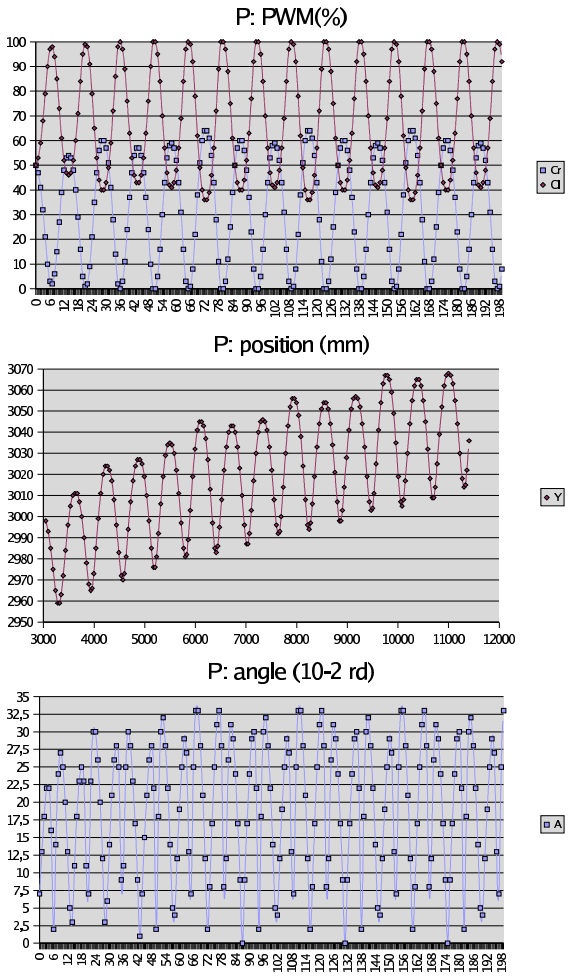

Figure 17.2: P regulation

Then, with P and D regulation: on figure 17.3, there is a convergence in the end, but the system keep a bias, because it has no idea about the effect of error accumulation before the convergence: it's a lack of memory.

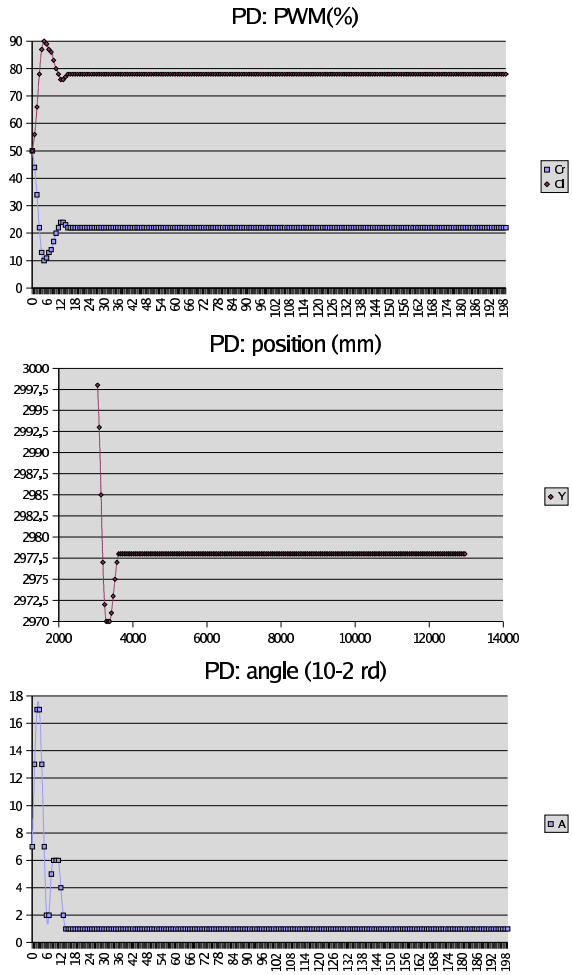

Figure 17.3: PD regulation

And, finally, with a complete PID regulation (see figure 17.4), there is no oscillations, and no bias.

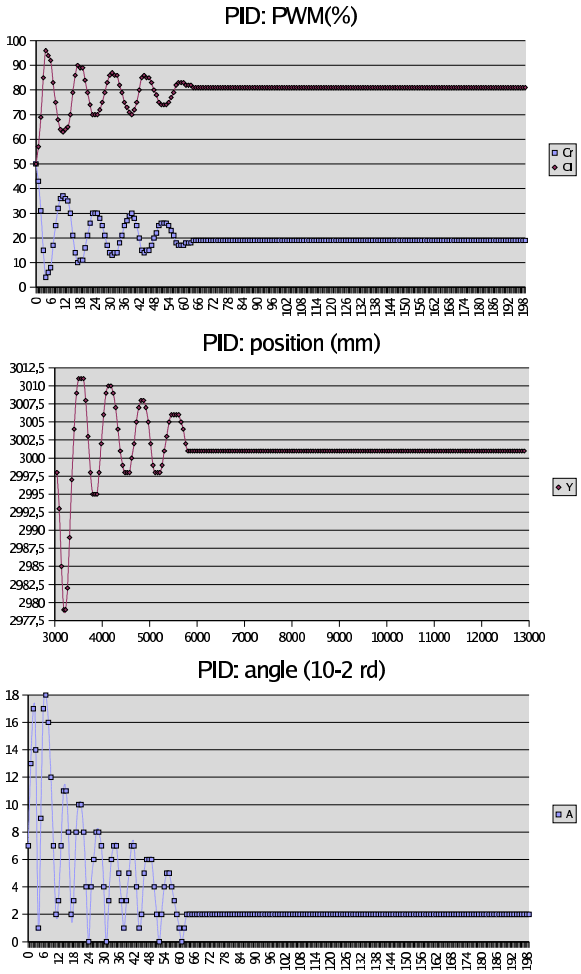

Figure 17.4: PID regulation

These simulations have been done with this program: http://botzilla.free.fr/pid/pid_simul.tgz.

Part XI

Hardware

Chapter 18

Faster and cheaper PIC programming: ICSP

Last updated:

`$Date: 2003/01/18 18:33:38 $`

Some Microchip PIC micro-controllers are programmable once soldered on the application board. This technique, known as `ICSP` (In-Circuit Serial Programming), has been implemented by Microchip to allow industrial users to program the controller at the very last moment before shipping, thus limiting the number of parts to store (unprogrammed, programmed, on board) and increase flexibility on the assembly line (different software versions per country, special customer tailored versions, evolutions of stocked boards).

For the hobbyist, the advantage on the hardware side is that the programming device is dramatically simplified, and that costly ZIF (Zero Insertion Force) supports, up to 40 pins for some PIC, aren't needed anymore. For the software developer this results in fewer physical manipulations, fastening the code/load/test cycle.

The constraint is that during programming, the PIC must be isolated from the application board regarding power pins (Vcc and gnd), and that one must have access to 2 pins having an alternate use: 2 I/O pins then become programming clock and serial data. Furthermore, the reset pin receive the Vpp programming voltage.

In order to isolate the PIC at programming time on the application board, one can use relays, or jumpers, or may use a dual line of pins connector (18.1). The drawback of a relay is not only its cost but also its size. Jumpers manipulation is tedious and error prone. Whereas the pro of the connector system is a risk-less use and low real-estate consumption on the board.

On this site (http://www.iki.fi/hyvatti/pic/picprog.html) one can see an unusual use of this technique, in the sense that it's not used to program PIC on application board, but rather to do a dedicated programmer board still usable

with all PIC supporting ICSP (pins used for alternate functions are not the same from a Microchip family to another). Nevertheless, the electronic used to link the serial port of a PC to the PIC is mature, cheap, and perform very well.

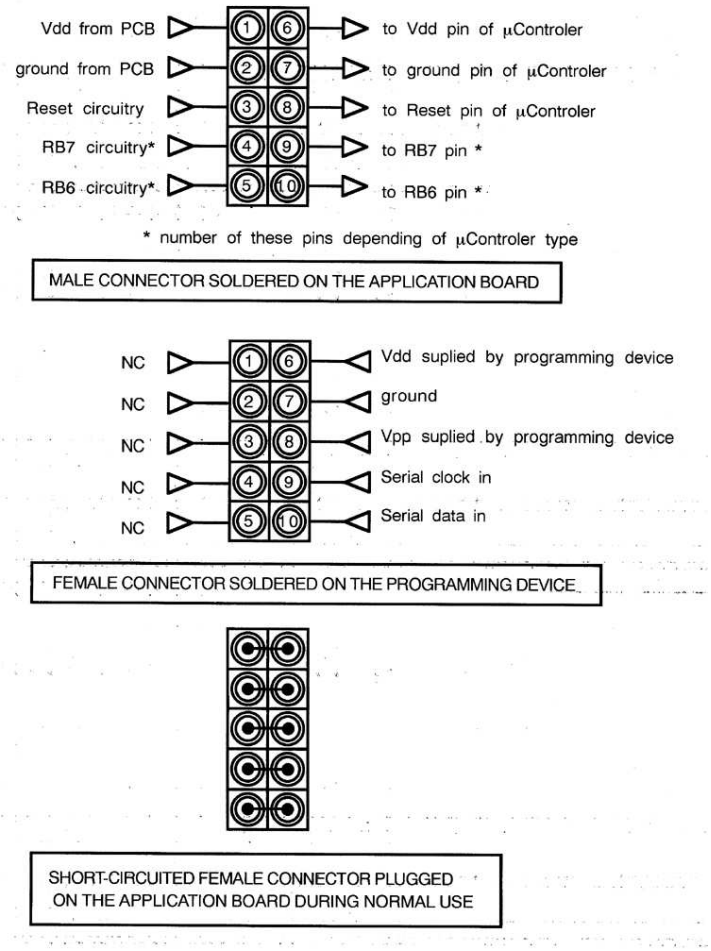

Figure 18.1: ICSP programming connector

This is the circuit that we use, redrawn to be be adapted to a 10 points in 2 rows connector (figure 18.1), ending in a very compact and simple ICSP programming system:

- During normal use, the connector is short-circuited by the use of a plug.
- In programming mode, the programmer take the place of the plug.

You can see some pictures in part IV.

There can't be no forgetting of a strap or commutating mistake. For the software used with this programming device, see chapter 14.

Chapter 19

Serial/I2C Interface

19.1 Hardware

19.1.1 Presentation

This module is intended to allow the use of the I2C bus from the serial port of a standard PC or from an other computer or controller fitted with a serial port using RS232 specifications. It can also be used to visualize the operation of an I2C bus by the way of a monitoring PC (connected to the serial port) or by the embedded control circuitry.

The module contain two parts:

- A bidirectional level translator between the RS232 standard to the I2C bus.
- An I2C component to visualize and/or act on the I2C bus.

19.1.2 How does it work

The circuit is build around the PCF8574 (Philips semiconductors) which is a specialized component normally used to provide a 8 bits wide bidirectional extension for microcontrollers. This circuit is available under two ordering numbers, PCF8574 and PCF8574A. The difference between them is the base address which increase flexibility in the use of the bus.

- The bits 0 to 3 of the port of PCF8574 (see Tab:19.1) have a switch connected to ground to emulate inputs.
- The bits 4 to 7 (see Tab:19.1) have a led connected to +5 volts to visualize outputs.
- The pins A0 to A3 (see Tab:19.1) are address select.
- The pin 13, not used on this card is an interrupt signal, down when a change occurs on an input.

Since the PCF8574 is +5volts powered, and the serial port of a PC has +/-12volts levels in accordance with the RS232 specifications, a level translator is needed between them. It is provided by a MAX232 (Maxim semiconductors) which is used here in his most typical application circuit from the Maxim application handbook. The six inverters circuit 74LS06 provide bidirectional communications between the bufferized serial port (4 wires) and the I2C bus (2wires).

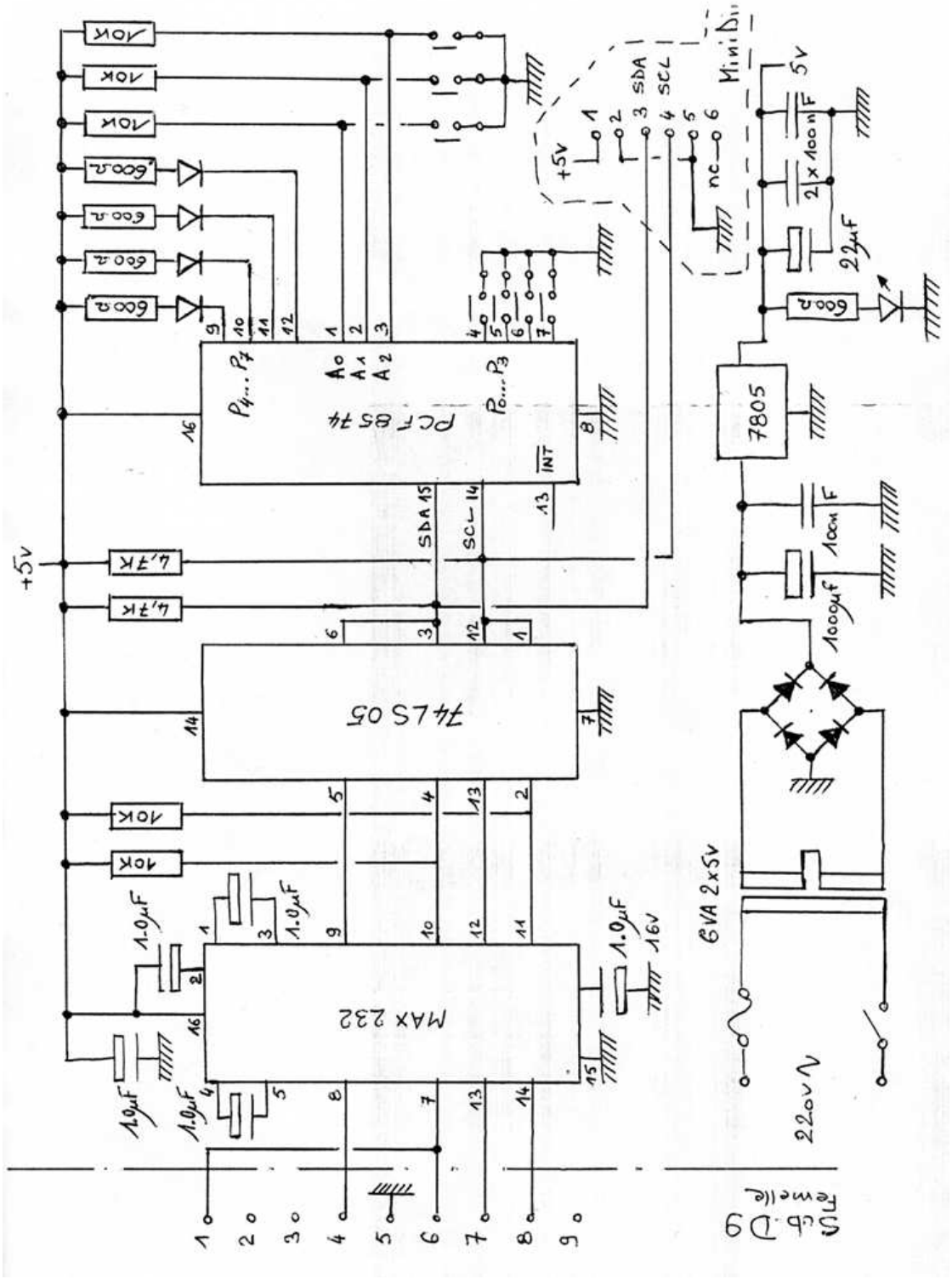

Figure 19.1: I2C Interface

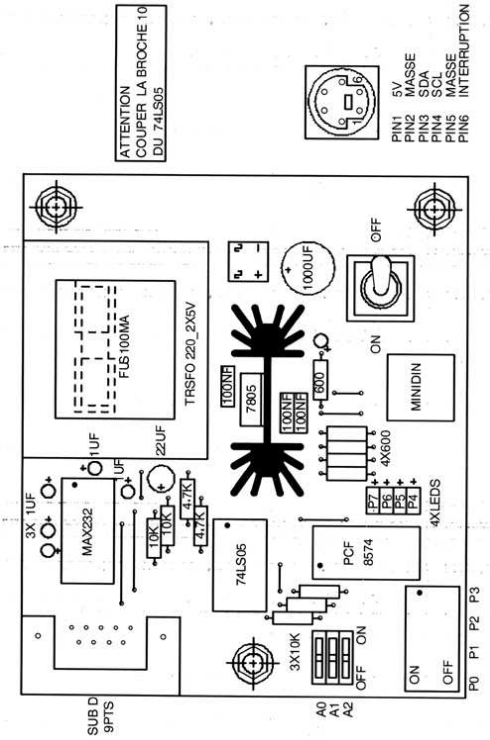

Figure 19.2: I2C Interface

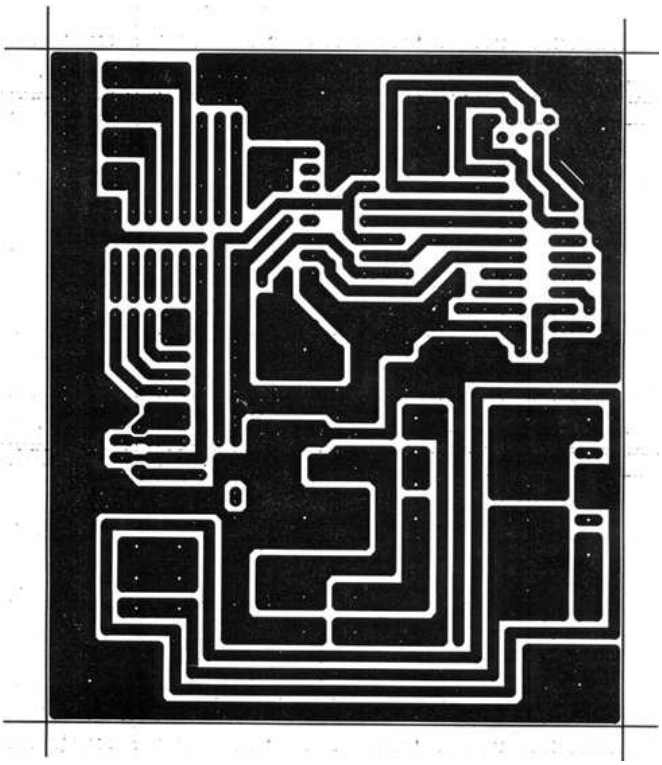

Figure 19.3: I2C Interface

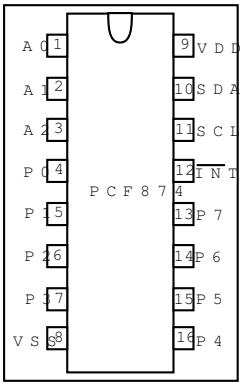

Figure 19.4: Pin configuration PCF8574

SYMBOL	PIN	DESCRIPTION
A0	1	address input 0
A1	2	address input 1
A2	3	address input 2
P0	4	quasi-bidirectional I/O 0
P1	5	quasi-bidirectional I/O 1
P2	6	quasi-bidirectional I/O 2
P3	7	quasi-bidirectional I/O 3
Vss	8	supply ground
P4	9	quasi-bidirectional I/O 4
P5	10	quasi-bidirectional I/O 5
P6	11	quasi-bidirectional I/O 6
P7	12	quasi-bidirectional I/O 7
$\overline{INT}$	13	interrupt output (active LOW)
SCL	14	serial clock line
SDA	15	serial data line
Vdd	16	supply voltage

Table 19.1: Pin configuration

19.2 software

Here (http://botzilla.free.fr/hard/appli.tgz), there is a simple application to use this Serial / I2C Interface on a PC (Linux or Windows9x).

- `lowlevel.c` and `lowlevel.h`: the low level part of the code (using ioperm, outb and inb).
- `highlevel.c` and `highlevel.h`: the system independent part of the code (using the I2C specifications).
- `appli.c`: an example of application.

There is a Makefile with these files, to compile this program (Linux) type:

```
make
```

Because, it's not a real driver (use of ioperm), you have to be root (for Linux user) to use it. This program will test `COM1` and `COM2` port, and look for a valid PCF target (scan all possible address). When a valid target is found, the program send some bytes to switch ON or OFF the 4 leds connected on the PCF.