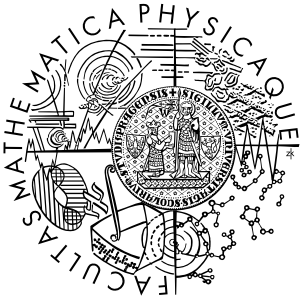
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. František Kačmarik

RTEMS Support for Lego NXT

Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Tomáš Bureš, PhD.
Study program: Informatics
Specialization: Software Systems (ISS)

Prague 2013

I would like to thank my supervisor, doc. RNDr. Tomáš Bureš, PhD. for his guidance, support and feedback during the entire time of working on this thesis.

I thereby declare that I carried out this master's thesis independently and used exclusively the cited resources.
I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague July 24, 2013 Bc. František Kačmarik

Title: RTEMS Support for Lego NXT
Author: Bc. František Kačmarik
Department: Department of Distributed and Dependable Systems
Supervisor of the master's thesis: doc. RNDr. Tomáš Bureš, PhD.
Supervisor's e-mail address: bures@d3s.mff.cuni.cz

Abstract: RTEMS is an open source real-time operating system used in embedded devices across a wide range of processor architectures. Lego Mindstorms NXT is a programmable robotics kit from Lego, which may be used as an educational tool to build a model of an embedded system with computer-controlled electromechanical parts.

The goal of the thesis is to allow development of RTEMS-based applications for Lego Mindstorms NXT platform with the aim of employing the results of the thesis as a runtime platform used in the Embedded and Real-Time Systems Course. The thesis thus identifies a suitable way of porting RTEMS to Lego NXT, provides the corresponding runtime environment and discusses basic means for deployment and debugging of RTEMS application on Lego Mindstorms NXT.

Keywords: RTEMS, Lego Mindstorms NXT, board support

Názov práce: RTEMS Support for Lego NXT
Autor: František Kačmarik
Katedra: Katedra distribuovaných a spolehlivých systémů
Vedúci diplomovej práce: doc. RNDr. Tomáš Bureš, PhD.
E-mail vedúceho: bures@d3s.mff.cuni.cz

Abstrakt: RTEMS je voľne šíriteľný operačný systém reálneho času, používaný v mnohých vstavaných zariadeniach zahrňujúci rôzne procesorové architektúry. Lego Mindstorms NXT je programovateľná robotická sada spoločnosti Lego, ktorá môže byť použitá ako výukový nástroj pre výstavbu modelov vstavaných systémov s počítačom riadenými mechanickými časťami.

Cieľom diplomovej práce je umožniť vývoj RTEMS aplikácií pre Lego Mindstorms NXT, výsledkom práce je výuková platforma použiteľná v predmete Vestavěné systémy a systémy reálneho času. Práca preto identifikuje vhodný spôsob portovania RTEMS na Lego NXT, poskytuje behové prostredie a uvádza spôsob nahrávania a ladenia RTEMS aplikácií na platforme Lego Mindstorms NXT.

Kľúčové slová: RTEMS, Lego Mindstorms NXT, podpora hardware

Contents

List of Figures

Chapter 1

Introduction

Real-time systems span over many sectors of industry. They can be found in automotive industry, space and defense systems, medical equipment, network devices, etc. Probably the most known characteristic of real-time systems is that they have to response to the given event in a specific time. They are usually used in systems, where the results of computation depends on the time when these results are produced. Many real-time systems are embedded as integral parts of electronic devices, therefore such a component is called an embedded system. The most known examples of real-time embedded systems are ABS and airbag in cars, manufacturing robots, multimedia systems for video processing, weapon systems.

Real-time systems have many characteristics and are thoroughly studied. Embedded and real-time systems course's goal is to introduce students into the world of real-time systems. During this course, students should acquaint themselves with the basics of real-time system design, scheduling, synchronization, communication, become familiar with definitions, characteristics and classification of real-time systems.

To gain hands-on experiences with real-time systems, students come in contact with multiple real-time systems during labs, where they create small real-time applications. One of the application that students have been creating is a punch press robot. It is programmed using RTEMS operating system. Created application runs in a simulated environment using QEMU, which is an open source machine emulator and virtualizer. RTEMS operating system is chosen specifically because it is able to run in x86 QEMU simulator. Another application developed during labs is the line following robot using nxtOSEK operating system and Lego Mindstorms NXT robotic kit. Using real hardware, students will become more aware of real hardware constrains, such as computation speed or memory size. They will also experience the impact of physical constrains, for example: motor has some momentum and can not be stopped at once, if robot wants to turn to follow the line, turn will take time and will move robot into different position relative to the followed line, than it will return to the followed line more steeply, etc.

1.1 Goals and structure of the thesis

Since necessary NXT hardware is already obtained and used during labs, students could develop their RTEMS real-time applications for NXT hardware. At this time, punch press is the only application developed using RTEMS during Embedded and real-time systems course. RTEMS running on NXT hardware would widen possibilities of creating many different applications using variety of sensors and motors. Since NXT is primarily destined for young children, it has rugged design, sensors and motors are easily connectable, building frames for variety of modeled systems is only a question of imagination.

The thesis is aimed to meet following requirements:

- **discuss suitable way of porting RTEMS to NXT:** RTEMS already has a support for boards with ARM processor. For RTEMS applications to run on NXT, suitable way of porting RTEMS should be found.
- **provide basic means for application compilation, deployment and debugging:** Means for development RTEMS application is already prepared, since Eclipse IDE is already used during the labs. Building environment must be put together, because NXT is using ARM, therefore cross-compiler toolchain should be chosen. For created applications to be deployed, easy way of uploading application into the NXT should be provided. Also simple way of debugging RTEMS applications should be available.
- **provide support for basic peripherals:** Once RTEMS application is successfully running on NXT brick, at least support for basic sensors, e.g. touch sensor, color sensor (HiTechnic), ultrasonic sensor and motors must be provided.
- **create API for accessing peripherals:** nxtOSEK operating system already have API called ECRobot, that is well documented and students already use it during labs, it would be appropriate to use the same API. It would eliminate the necessity of students learning different API and they would be able to access devices in the same way and create applications faster.

The thesis is divided into ten chapters:

- **The Chapter 2** introduces RTEMS as real-time operating systems, defines used terms, presents RTEMS architecture.
- **The Chapter 3** describes NXT and its internal hardware and architecture, address space and memory mapping. Also presents constrains and requirements that are caused by choice of application management.
- **The Chapter 4** discusses suitable solutions for each part of the RTEMS to NXT porting process.
- **The Chapter 5** provides information on porting RTEMS on NXT platform, in detail discusses most crucial parts of implementation, describes basis of used drivers, system initialization and added startup screen.

- **The Chapter 6** shows instruction to prepare building environment and discuss options that can be used in RTEMS application.
- **The Chapter 7** zooms in on debugging RTEMS applications.
- **The Chapter 8** evaluates memory requirements and memory usage of RTEMS application running on NXT.
- **The Chapter 9** gives overview of related systems.
- **The Chapter 10** concludes the thesis.

Sample RTEMS applications should be provided. Therefore multiple small applications, like line following robot, example mutex and semaphore usage, are enclosed on DVD.

ECRobot API is enclosed on DVD in form of a web page, which sums up all available methods.

Chapter 2

RTEMS

The Real-Time Executive for Multiprocessor Systems or RTEMS is a open source fully featured Real-Time Operating System or RTOS that supports a variety of open standard application programming interfaces (API) and interface standards such as POSIX and BSD sockets. It is used in space flight, medical, networking and many more embedded devices across a wide range of processor architectures including ARM, PowerPC, Intel, Blackfin, MIPS, Microblaze and more.

Following chapters will explain basic terminology, provide overview of RTEMS internal structure and describe architecture of RTEMS application.

2.1 Real-time and embedded system

Real-time systems have a complex set of characteristics. Generally, they must adhere to more rigorous requirements. The real-time system is usually defined as a system where correct behavior depends not only on the value of the computation but also on the time at which the results are produced [1].

Real-time application system must receive and respond to a set of external stimuli within rigid and critical time constraints referred to as deadlines. Deadlines can be further characterized as either hard or soft. If hard deadline is missed, it will result in a catastrophic event, e.g. airbag will not be inflated in time. Missing a soft deadline may not have such a devastating consequences. Another requirement of real-time application systems is the ability to coordinate or manage a large number of concurrent activities. Sometimes the activities - tasks are spread over a set of processors instead of a single processor.

Real-time operating systems or real-time executives serve as a cornerstone on which to build the application system. A real-time multitasking executive allows an application to be cast into a set of logical, autonomous processes or tasks which become quite manageable. Each task is internally synchronous, but different tasks execute independently, resulting in an asynchronous processing stream. Tasks can be dynamically paused for many reasons resulting in a different task being allowed to execute for a period of time. The executive also provides an interface to other system components such as interrupt handlers and device drivers. System components may request the executive to allocate and coordinate resources, and to wait for and trigger synchronizing conditions.

RTEMS, Real-Time Executive for Multiprocessor Systems, is a real-time executive (kernel) which provides a high performance environment for embedded military applications including the following features:

- multitasking capabilities
- homogeneous and heterogeneous multiprocessor systems
- event-driven, priority-based, preemptive scheduling
- optional rate monotonic scheduling
- intertask communication and synchronization
- priority inheritance
- responsive interrupt management
- dynamic memory allocation
- high level of user configurability

By using RTEMS, the real-time applications developer is freed from the problem of controlling and synchronizing multiple tasks, does not need to develop, test, debug, and document routines to manage memory, pass messages or provide mutual exclusion. The developer is then able to concentrate solely on the application. The time and cost required to develop sophisticated real-time applications is then significantly reduced.

Embedded systems bear their name, because they are usually embedded as a part of a bigger product, dedicated to specific task. They can be optimized for specific functionality, which can increase their performance and rentability.

2.2 RTEMS application architecture

One important design goal of RTEMS was to provide a bridge between two critical layers of typical real-time systems. As shown in the following figure, RTEMS serves as a buffer between the project dependent application code and the target hardware. Most hardware dependencies for real-time applications can be localized to the low level device drivers.

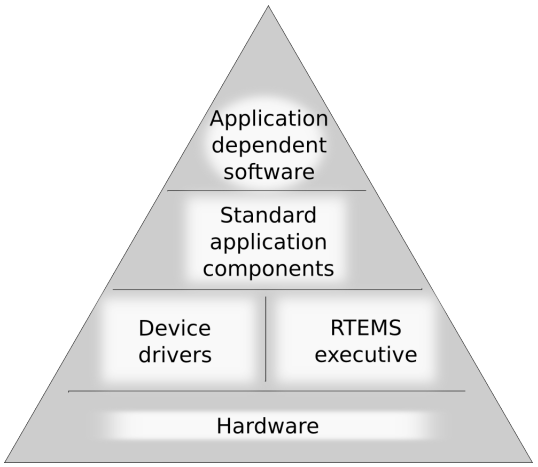


Figure 2.1: RTEMS application architecture

2.3 RTEMS internal architecture

RTEMS can be viewed as a set of layered components that work in harmony to provide a set of services to a real-time application system. The executive interface presented to the application is formed by grouping directives into logical sets called resource managers. Functions utilized by multiple managers such as scheduling, dispatching, and object management are provided in the executive core. The executive core depends on a small set of CPU dependent routines. Together these components provide a powerful run time environment that promotes the development of efficient real-time application systems. The following figure illustrates this organization:

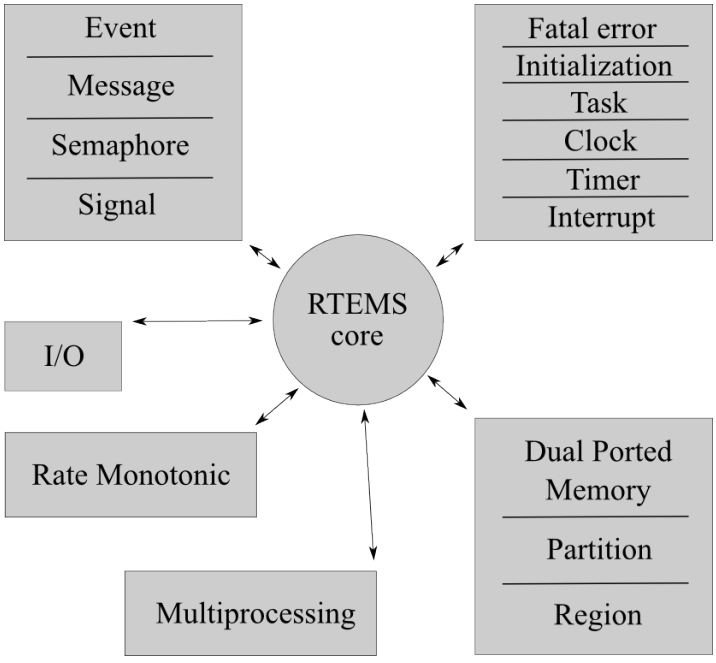


Figure 2.2: RTEMS internal architecture

Chapter 3

NXT

Lego Mindstorms NXT 2.0 is the second set from LEGO's Lego Mindstorms series. Main component is a programmable brick computer that controls the system. A set of modular sensors, e.g. color, light, ultrasonic, touch sensor, gyro and acceleration sensor, motors can be connected to the brick. Main building blocks to create various structures for mechanical systems are parts from the Technics line. LEGO Mindstorms NXT 2.0 may be used to build a model of an embedded system with computer-controlled electromechanical parts. Many kinds of embedded systems may be modeled, for example: line following robot, robot balancing on the ball, segway, image scanner...

Figure 3.1: Lego Mindstorms NXT, from [2]

3.1 Hardware

Lego Mindstorms NXT 2.0 consists of the following hardware:

- 32-bit Atmel AT91SAM7S256 main microcontroller (256 KB flash memory, 64 KB RAM), built on ARM7TDMI core (ARMv4T architecture). It is the heart of the NXT brick.
- 8-bit Atmel AVR ATmega48 microcontroller at 4 MHz (4 KB flash memory, 512 Bytes RAM) is also present. It manages the pulse width modulation (PWM) of the motors, can shut down the brick.
- 100x64 pixel LCD Screen
- Loudspeaker - 8 kHz sound quality, 8-bit resolution, 2-16 kHz sample rate
- Bluetooth Class II V2.0
- USB Port
- four 6-pin input ports (ports 1-4)
- three 6-pin output ports (ports A-C)
- four Push Buttons
- NXT 2.0 is powered by six AA batteries or the NXT Rechargeable DC Battery

3.2 Communication architecture

Following Figure 3.2 provides a graphical overview of Lego Mindstorms NXT 2.0 system architecture according to LEGO MINDSTORMS NXT Hardware Developer Kit.

Communication between the main ARM7TDMI processor and Sensors/Servo Motors is done via the ATMEL AVR co-processor. Exception is the Ultrasonic Sensor and acquisition of Servo Motor revolutions.

To access Sensors/Servo Motors, processor communicates with the co-processor via I2C serial bus. This system architecture influences the software run-time environment. The main ARM7TDMI processor accesses Sensors (to read sensor A/D - analog-to-digital converter value) and Servo Motors (to set PWM duty ratio and break mode) through a 1 ms periodical interrupt service routine of periodical timer interrupt. Servo Motors revolutions are directly captured by pulse triggered interrupt service routine of motor driver. Ultrasonic Sensor has its brain to directly communicate with the main ARM7TDMI processor via another I2C communication channel.

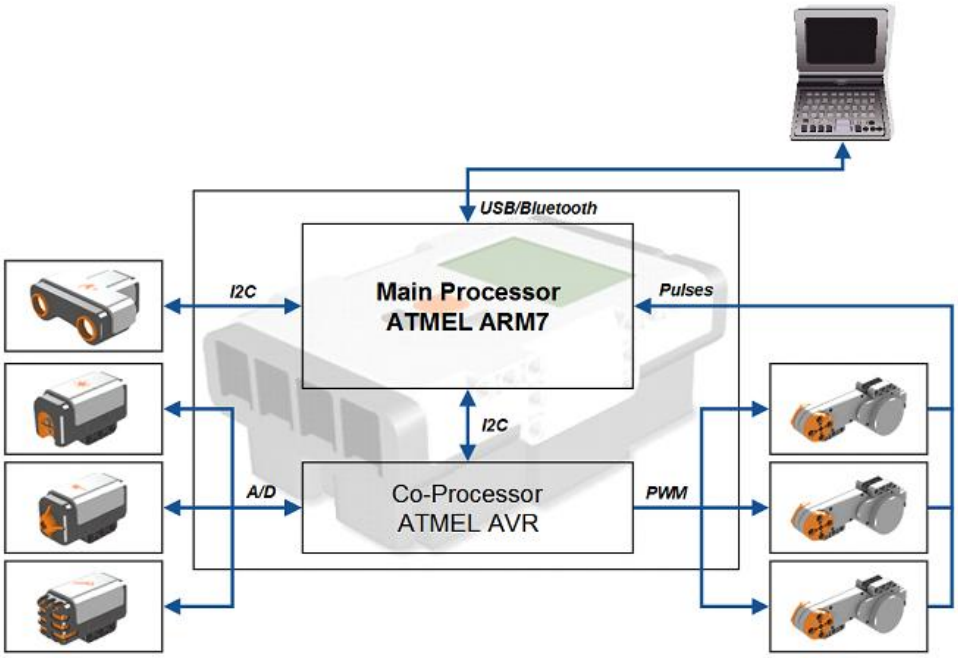


Figure 3.2: NXT communication architecture, from [3]

3.3 Address space

Lego Mindstorms NXT 2.0 has a 32-bit processor, which allows 32-bit memory addresses. Memory address space starts at `0x0000 0000`, ends at address `0xFFFF FFFF`. Processor can address 2^{32} = 4,294,967,296 bytes of memory. Flash and SRAM memories are mapped at the beginning of the address space. NXT has 256 Kbytes of Flash Memory, single plane, 1024 pages of 256 bytes, Protection Mode, lock bits to secure contents of the Flash. Memory has 10,000 write cycles, 10-year data retention capability. Fast SRAM memory has capacity of 64 Kbytes and single-cycle access at full speed.

Internal peripherals, such as power management controller, interrupt controller, display, real time clock are mapped at the end of address space. Following Figure 3.3 describes NXT memory layout, presents memory addresses of internal devices and shows their internal memory mapping. Flash and SRAM memory address mapping can be change using Remap command. Memory address mapping of Flash and SRAM memories will be characterized in Section 3.3.1.

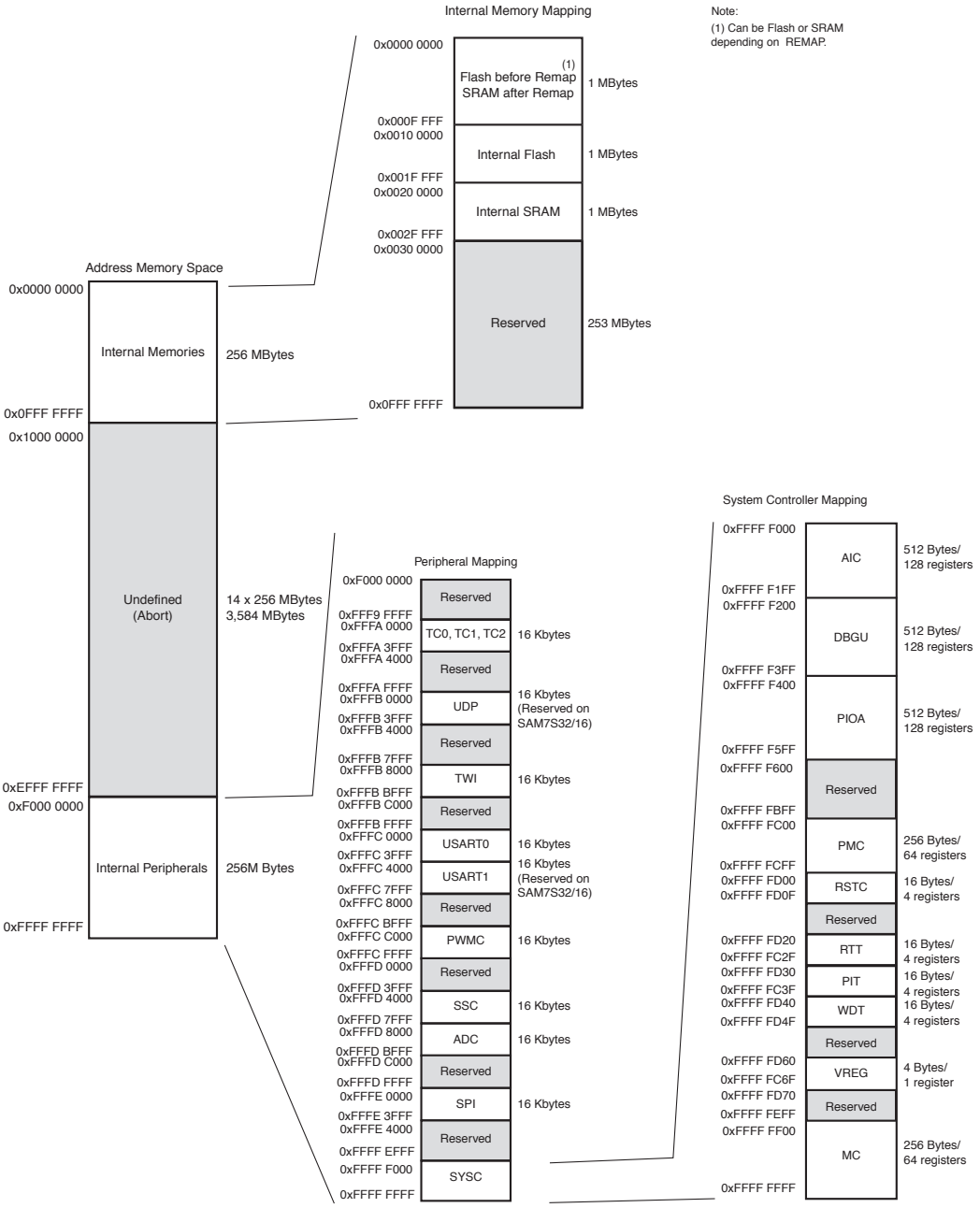


Figure 3.3: Address space mapping, from [4]

3.3.1 Flash-SRAM memory mapping

The AT91SAM7S256 has a capability to remap its internal Flash and SRAM memories to the different memory addresses. At any time, the Flash memory is mapped to address `0x0010 0000`. It is also accessible at address `0x0000 0000` after the reset and before the Remap Command is used.

SRAM is always accessible at address `0x0020 0000`. If Remap Command was used, SRAM is also accessible at address `0x0000 0000`.

Remap Command switches Flash – SRAM memory at address `0x0000 0000`. Remap Command is used during RTEMS startup to map SRAM memory to address `0x0000 0000`, where interrupt vectors table will be stored. SRAM memory

has to be remapped, because processor always jumps at address `0x0000 0000` to `0x0000 001C`, according to the occurred exception. In Flash memory, these addresses and also the beginning of the Flash memory are used by NXT BIOS. To be able to use NXT BIOS for uploading RTEMS applications, interrupt vectors table must be located in SRAM memory, therefore it must be remapped to address `0x0000 0000`. More information on NXT BIOS usage can be found in Section 3.4.

Following Figure 3.4 illustrates memory mapping, when NXT is turned on:

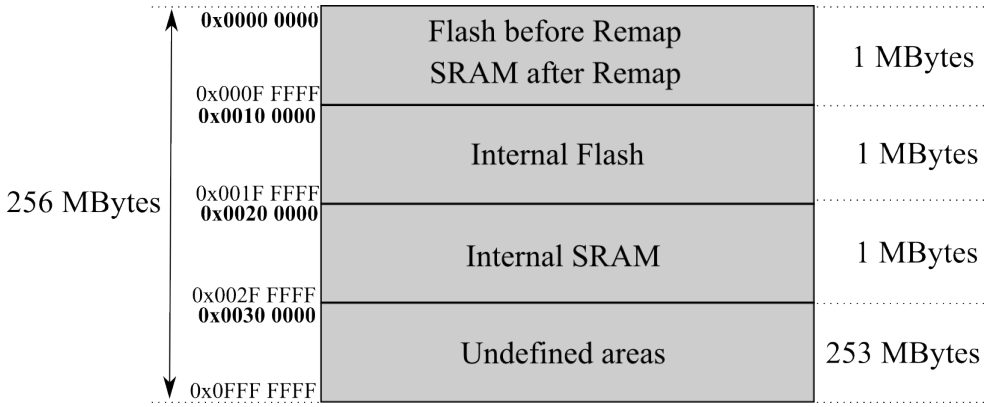


Figure 3.4: NXT memory remapping

3.4 nxtOSEK NXT BIOS

nxtOSEK [6] is an open source operating system for LEGO MINDSTORMS NXT platform. nxtOSEK consists of device driver of leJOS NXJ C/Assembly source code, TOPPERS/ATK (Automotive Kernel, formerly known as TOPPERS/OSEK) and TOPPERS/JSP Real-Time Operating System source code that includes ARM7 (ATMEL AT91SAM7S256) specific porting part, and glue code to make them work together. Part of the nxtOSEK is the nxtOSEK NXT BIOS. Is an application which allows to upload nxtOSEK application to the NXT. Using NXT BIOS, max. 224Kbytes single nxtOSEK program can be uploaded and stored to Flash memory. NXT BIOS is located at the beginning of the Flash memory. It is the place where processor jumps to read instructions after NXT is started. Therefore RTEMS application will be always started from NXT BIOS. Usage of NXT BIOS also created a constrains for the memory addresses for building RTEMS application and necessity of memory remapping when RTEMS application is started, see Section 3.3.1

To install NXT BIOS, follow [14] and [15].

Chapter 4

RTEMS to NXT porting analysis

To be able to run RTEMS on NXT platform, it is necessary to analyze main steps that will be needed to successful execution of RTEMS applications on the NXT, such as:

- building environment, building process
- application deployment
- application management in RTEMS
- support for CPU and internal/external peripherals in RTEMS, RTEMS porting
- API for accessing peripherals
- debugging methods

All of these steps will be described in following chapters.

4.1 Building environment

When porting to NXT platform, first thing to do is to create an environment for building application. NXT contains ARM processor, therefore it will be necessary to use some cross-compiling toolchains to compile on non-ARM workstations. Multiple toolchains are available, for example GNU ARM toolchain [20], YAGARTO [21], CrossWorks for ARM [22]. RTEMS provides pre-built environments for some hosts and also provides guidelines, how to build RTEMS using GNU ARM, therefore GNU ARM toolchain is chosen. Toolchain built from sources using Linux host will be discussed, due to usage of Linux hosts environment during labs in Embedded and real-time systems course. Process of preparing environment and building RTEMS is described in detail in Chapter 6.

For application development, Eclipse CDT with RTEMS plugin can be used. Eclipse is already used during Embedded and real-time systems course. Tutorial for simple project creation is enclosed in Appendix C.

4.2 Application deployment

Built RTEMS on NXT application has to have a specific structure. After start, NXT starts to execute code from the beginning of the Flash memory. Therefore it is usually the place where small assembly code, which starts with board basic initialization is located. This should be the place where RTEMS on NXT is located. Using this approach, RTEMS on NXT itself should have some BIOS-like utility to handle uploading of new applications. Uploading must consist of two parts: tool for sending compiled program running on the computer and receiver running on the NXT, which will save received data into Flash memory.

For NXT part, existing systems for NXT platform usually have a BIOS-like firmware, which represents operating system, through which an applications can be uploaded, managed and executed. To get rid of implementing application management in RTEMS on NXT, firmware from nxtOSEK is used to handle application management. nxtOSEK uses NXT BIOS, which can upload and run nxtOSEK applications. With NXT BIOS, students can upload RTEMS on NXT and nxtOSEK applications the same way, which simplifies environment preparation for Embedded and real-time systems course and students are already familiar with uploading process. Usage of NXT BIOS impose specific requirements, which are discussed in Section 4.2.1, for location of the application in the memory and therefore to the application structure. NXT BIOS jumps to the beginning of the RTEMS on NXT, which than takes complete control over the entire NXT.

nxtOSEK already provides tools for application uploading. nxtOSEK expects to use `appflash.sh` script, which uses `NEXTTool`, to upload applications in conjunction with NXT BIOS. Though it is working fine in Windows operating system, they had troubles working in Linux environment, therefore JAVA-based uploader for 64-bit Linux platform is enclosed. It is compatible with NXT BIOS and applications can be uploaded via USB cable. Uploader implementation is described in the Section 5.11.

4.2.1 Imposed memory requirements

RTEMS application is expected to be uploaded and run via NXT BIOS, which means that memory layout of RTEMS application must be compatible with nxtOSEK application. RTEMS application is uploaded to address `0x0010 8000`, which is ensured by using enclosed uploading application. When NXT BIOS starts, it checks whether it should run uploaded application or go into mode, where new application can be uploaded. It also checks NXT BIOS version record, whether it has changed. NXT BIOS version is stored at address 0x0010 7F00, which is the last NXT BIOS containing page in memory.

To check whether new application should be uploaded, address `0x0010 7E00` is checked. That is the last but one BIOS containing page in memory. Flash new application mode is entered, if value at specified location is set to

```
"Jumpin' Jack Flash"
```

Address `0x0010 8000` is the place where BIOS stores uploaded application and also it is the starting point of nxtOSEK applications. If application should be

run, processor jumps to address `0x0010 8000` and starts executing instructions. In case of RTEMS on NXT application, system initialization routines should be stored at this address and from this point, uploaded application will take complete control over NXT.

Following Figure 4.1 displays location of NXT BIOS and nxtOSEK or RTEMS application in the memory.

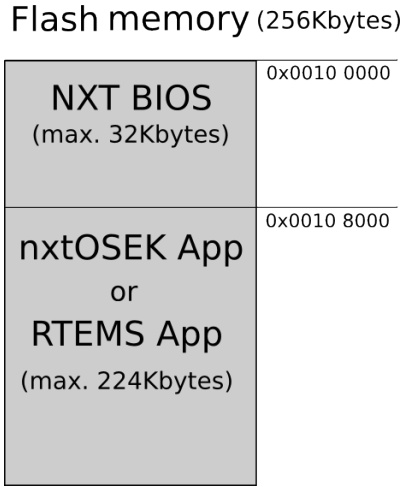


Figure 4.1: Application stored in memory using NXT BIOS

nxtOSEK SRAM application starts with copying itself into SRAM memory. This mechanism restricts the size of a nxtOSEK SRAM application up to 64Kbytes. It is not necessary to copy entire application into SRAM, only some specific parts will be copied. RTEMS on NXT application code stays in the Flash memory, only the `.data` and `.bss` segments, IRQ routines and a routine which sets flags into memory for NXT BIOS to flash new application are copied into the SRAM. This way, application size can be much bigger.

4.3 RTEMS on NXT application management

To be able to flash new RTEMS on NXT application, shutdown and also to comfortably connect to the remote computer screen for debugging purposes, it will be necessary to provide simple user interface. In different systems running on NXT, this functionality is included in firmware. RTEMS on NXT implements startup screen, which enables user to connect to the remote screen application, shutdown NXT, run application or set new application flash mode via pressing buttons. Ability to shutdown and set new application flash mode should also be present when RTEMS on NXT application is executed. This can be achieved via separate task or as part of some periodically occurring handler, such as clock tick handler. Solution using clock tick handler is used, because it does not need application programmer to include special tasks, it is transparent and also this way it can not be forgotten to include this special task.

4.4 Porting RTEMS

RTEMS is designed to be portable to a variety of computer architectures and boards. NXT is using ARM processor, which is already supported by RTEMS. With processor supported, only NXT board with its peripherals will need to become operational. In RTEMS, support for specific board is added in form of BSP - Board Support Package. BSP holds specific implementations of board initialization and device drivers for particular board. A minimal Board Support Package includes device drivers for a clock tick, console I/O, startup and miscellaneous support code.

In BSP, RTEMS on NXT execution starts with small assembly script which initializes internal devices, such as clock generator, sets processor mode, initializes stack for each operating mode, sets interrupt handlers and interrupt controller, copies necessary parts of code into SRAM memory. Then all RTEMS internal structures are initialized together with drivers for internal and external devices.

BSP package, together with implementation details for RTEMS on NXT crucial parts and functionality is described in Chapter 5. Interrupt handling possibilities are discussed in Section 4.4.1.

4.4.1 Handling interrupts

Interrupt handling on NXT is done in a following manner: as soon as exception occurs (e.g. data abort, undefined instruction, IRQ), NXT processor jumps to the corresponding instructions situated at addresses from `0x00` to `0x1C`. From there, RTEMS on NXT jumps to the standard interrupt handler. RTEMS on NXT uses single point of entry for interrupts, where it reads AIC_IVR (Interrupt Vector Register) to get an interrupt number. This number is expected to be the number, which determines the position in the table, where all registered interrupt handlers are stored. Interrupt vector numbers are set during advanced interrupt controller initialization.

Another approach to interrupt handling supported by the NXT is called interrupt vectoring. The interrupt handler addresses corresponding to each interrupt source can be stored in the registers. This way, the processor will be able to jump to the interrupt handler in a single instruction. Since AIC_IVR is at address `0xFFFF F100`, the interrupt vector for interrupt handler stored at address `0x0000 0018` will contain following instruction:

```
LDR  PC,[PC,# -&F20]
```

This instruction will read AIC_IVR register, which at that time will store address for particular interrupt handler and jump to the appropriate handler. Though this approach is a bit faster, it is more memory consuming, because each interrupt handler must have a logic for saving registers, interrupt nesting, context switching... For a specific tasks, which should be serviced very fast, such as voice/video buffers or some peripheral readings, this approach should be suitable and effective.

RTEMS does not have support for interrupt vectoring. Interrupt vectoring is a hardware specific feature and RTEMS is expected to be multi-platform, therefore it handles interrupt in a way that can be used on various platforms. nxtOSEK,

which is destined to run solely on NXT utilizes interrupt vectoring for handling interrupts.

4.5 API for accessing peripherals

nxtOSEK has built-in C/C++ API, which is called ECRobot. ECRobot C API was originally designed to develop a MATLAB&Simulink based Model-Based Design environment (called Embedded Coder Robot) for LEGO MINDSTORMS NXT. To make NXT internal and external devices accessible to the programmer in RTEMS on NXT, ECRobot-like interface is used. ECRobot-like API is chosen, because during the Embedded and real-time systems course, students meet with nxtOSEK, create nxtOSEK applications, therefore they will be familiar with this API and won't need to explore new one. In comparison with nxtOSEK ECRobot API, nxtOSEK specific functions are omitted and newly created RTEMS specific functions are added.

4.6 Debugging methods

Traditionally it is quite hard to debug embedded systems. Usually, after making any change in code, application has to be compiled, linked, uploaded and executed to determine whether change was successfully applied. It is a quite time consuming activity. There is also a problem with examining state changes in program. Usually if developers don't want to open up the device, solder connections or simply don't have hardware debugger, they have to suffice with simple techniques, such as some debug output on LCD screen, playing different sounds, etc. For low-level control of hardware, hardware debugger is necessary. Some of the development systems, for example RobotC, provide another approach. They generate instructions for an intermediate virtual machine rather than direct instructions for the NXT CPU. This way developer can connect to the virtual machine using USB or Bluetooth and remotely debug running application using debugger build in into development environment IDE.

With RTEMS, it would be difficult to use an intermediate virtual machine for debugging. Goal of this thesis is to run RTEMS on NXT as is, not with such a huge RTEMS functionality change. Usage of hardware debugger is possible, since NXT is capable to utilize JTAG interface, but hardware modifications of NXT brick are necessary, see Section 7.2.1. Debugging via JTAG is further described in Chapter 7.

leJOS operating system has ability to use computer as a remote console to display tracing statements generated in NXJ program. It has capability to write to the remote computer screen using Bluetooth or USB connection in conjunction with `nxjconsoleviewer` application. On the computer side, `nxjconsoleviewer` Java application is used to display output on the screen. Instead of implementing new communication protocol and application to display outputs from NXT, existing leJOS solution was reused. RTEMS on NXT is able to utilize the same application and provide this simple debugging feature.

Chapter 5

RTEMS on NXT platform

In following text, this chapter will describe necessary steps that were undertaken during porting RTEMS to NXT platform. The most significant parts of the RTEMS on NXT initialization, NXT board initialization, interrupt handling and drivers implementation will be discussed. In the end, startup screen and application uploader implementation will be presented.

This chapter can also be viewed as a tutorial for porting to NXT platform.

5.1 NXT Initialization

RTEMS on NXT execution starts with assembly code, that should be located in BSP's `start/start.S` file. It is the starting point for every BSP. It sets stacks for different processor modes (*IRQ*, *FIQ*, *Abort mode*, *Supervisor*) and set processor back to *Supervisor* mode. Than interrupt vectors, handlers and clock have to be put into operation.

Processor starts its execution in *Supervisor* mode with disabled interrupts. Processor named ARM7TDMI has more operating modes:

- **User:** The normal ARM program execution state
- **FIQ:** Designed to support high-speed data transfer or channel process
- **IRQ:** Used for general-purpose interrupt handling
- **Supervisor:** Protected mode for the operating system
- **Abort mode:** Implements virtual memory and/or memory protection
- **System:** A privileged user mode for the operating system
- **Undefined:** Supports software emulation of hardware coprocessors

Current processor mode is set in CPSR register. It holds information about most recently performed ALU operation, controls the enabling and disabling of interrupts and sets processor operating mode. RTEMS on NXT applications do not utilize all operation modes. Applications do not utilize protection that can be achieved via using multiple processor modes, therefore initialization and also

RTEMS on NXT applications run in *Supervisor* mode. Processor changes mode to *IRQ* when interrupt occurs.

CPSR register in detail:

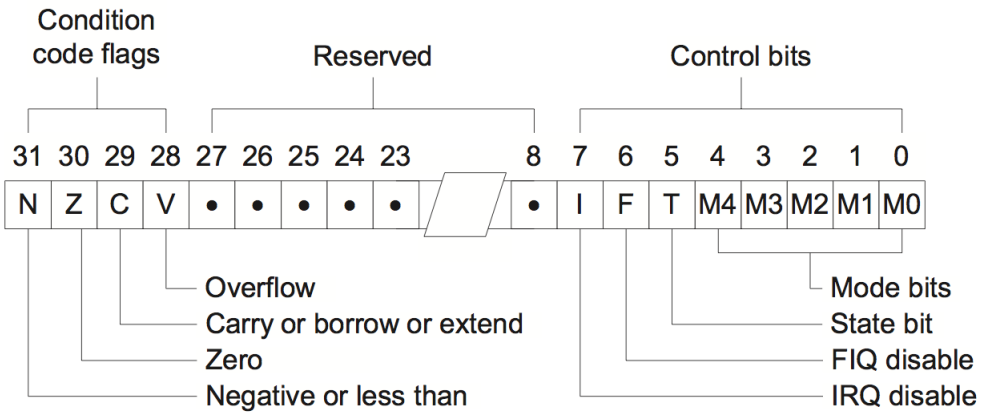


Figure 5.1: Overview of CPSR register, from[19]

Some modes also have own Saved Program Status Register (SPSR), that holds the CPSR of the task immediately preceding the raised exception. Number of available registers differs between ARM and Thumb state, ARM state uses 32-bit instruction set and Thumb uses 16-bit instruction set. Most of the RTEMS on NXT code is compiled into Thumb instruction set because of size optimization. *IRQ* mode, which is set when exception occur, starts expecting ARM instructions. For each mode, stack pointers have to be initialized. Segments .bss and .data should also be initialized.

Interrupt vectors are expected to be at address `0x0000 0000` – `0x0000 001C`. These interrupt vectors should be stored in the program at the beginning of the memory. Since this place in memory is occupied by NXT BIOS, interrupt vectors are copied to SRAM during startup, then the SRAM memory is remapped. Details on memory remapping can be found in Section 3.3.1. Interrupt vectors can contain pointers to exception handlers or instruction to jump to the AIC (Advanced Interrupt Controller). When pointers are used, each handler has to find out what kind of interrupt occurred, what is the interrupt source and determine correct handling. When instruction to jump to the AIC is used, NXT's AIC is able to determine interrupt source and set the interrupt source into specific register. When AIC jumps to the correct handler by itself, this feature is called interrupt vectoring. RTEMS on NXT does not have support for interrupt vectoring, therefore AIC contains exception handler number instead of exception handler address. Interrupt vectoring as well as interrupt handling is reviewed in Section 5.3 and Section 4.4.1. Each interrupt that wants to be raised, must be registered in AIC. Settings for AIC can be found in Section 5.2.1.

To be able to raise timer interrupt for scheduler periodically, Master clock has to be set. Detail description of enabling and setting master clock is in Section 5.4.

At this point, low level initialization should be complete and initialization of devices and RTEMS on NXT structures can continue in C language methods. RTEMS on NXT internal initialization is discussed in Section 5.7.

Other devices that require interrupt registration should be registered before interrupts are enabled. In RTEMS on NXT, it is done in `bsp_predriver_hook`, see Section 5.7. It is also the place where initialization of drivers occurs. Drivers for motors, sensors, internal devices are stored in BSP package. Some of the NXT devices require to run handling routines periodically. For example, link with AVR co-processor has to be sustained every 1ms, quadrature for motors has to be decoded, USB connection must be invoked periodically, buttons state should be checked, etc. In RTEMS on NXT, these specific routines are executed every 1ms, from timer interrupt. It would not be right to oppress programmer with creation of specific tasks for execution of this hardware specific routines. He could also omit some of them, which will lead to installation of new NXT BIOS. Drivers will be further discussed in Section 5.6. RTEMS already contains support for ARM processors, therefore routines such as context-switch, nested interrupt handler, register definitions do not need to be implemented.

If initialization is successfully completed, NXT is ready to start executing user code.

When application is being executed on NXT, it is very useful to be able to shut it down in a way other than pulling out batteries. NXT does not have on/off switch, it has to be implemented in application. NXT shut down is initiated from AVR co-processor. Details can be found in Section 5.5.4. In RTEMS, task execution can be terminated from code via `rtems_shutdown_executive`. It would be convenient to be able to shut down NXT from outside the application, just by pressing buttons. RTEMS on NXT implements this functionality together with setting NXT BIOS into flash new application mode. From periodical timer interrupt, buttons state is checked. If gray square button was pressed, NXT will shut down. If orange and left-arrow buttons were pressed, NXT BIOS application flash mode will be set, see Section 5.10.

5.2 Interrupts

All interrupts of the system are managed by the AIC - Advanced Interrupt Controller. Every interrupt source must register itself in AIC to be able to raise interrupts and also has to register its interrupt handler. In following chapters, process of initializing interrupt controller, registering interrupts and interrupt handling will be described.

5.2.1 Advanced interrupt controller

In the following text, initialization of the advanced interrupt controller (AIC) will be presented.

For each interrupt source, a particular interrupt handler can be set, i.e. a function which is called when the corresponding interrupt occurs. Interrupts can also be individually enabled or masked, and have eight different priority levels.

To initialize the AIC, first thing to do is to disable and clear all interrupts.

```
// Disable all interrupts
AT91C_BASE_AIC->AIC_IDCR = 0xFFFFFFFF;
```

```
// Clear all interrupts
AT91C_BASE_AIC->AIC_ICCR = 0xFFFFFFFF;
```

NXT support fast interrupt. Fast interrupt (FIQ) can be used as higher priority interrupt then a normal interrupt, for example for some very high priority task. RTEMS disables this feature. The fast forcing feature redirects any internal or external interrupt source to provide a fast interrupt rather than a normal interrupt. To disable fast interrupts, set Fast Forcing Disable Register (AIC_FFDR):

```
AT91C_BASE_AIC->AIC_FFDR = 0xFFFFFFFF;
```

Next step in AIC initialization is to acknowledge any previous interrupt, so the AIC will become idle, not handling any interrupt:

```
AT91C_BASE_AIC->AIC_EOICR = 1;
```

When AIC is being initialized, individual interrupt sources can be adjusted. Interrupt controller has 32 sources. Interrupt source 0 is reserved for fast interrupt. Interrupt source 1 is always located at System Interrupt, such as System Timer, Real Time Clock, Power Management Controller and the Memory Controller. Another interesting interrupt sources are UDP (11), TWI (12), SPI (13).

For each interrupt source, priority and source type must be set. It is done via separate register AIC_SMR (Source Mode Register) for particular interrupt source. During initialization, all AIC_SMR registers are set to 0 and each AIC_SVR is set to its matching number, which will be used when resolving incoming interrupts:

```
AT91C_BASE_AIC->AIC_SMR[interrupt_peripheral_id] = 0;
AT91C_BASE_AIC->AIC_SVR[interrupt_peripheral_id] =
                              interrupt_peripheral_id;
```

To install interrupt handler into AIC, interrupts must be disabled:

```
AT91C_BASE_AIC->AIC_EOICR = 1 << interrupt_peripheral_id;
```

For each interrupt source, set Source Mode Register with corresponding priority and source type:

```
AT91C_BASE_AIC->AIC_SMR[interrupt_peripheral_id] = desired_value;
```

Detail of Source mode register structure is shown in Figure 5.2:

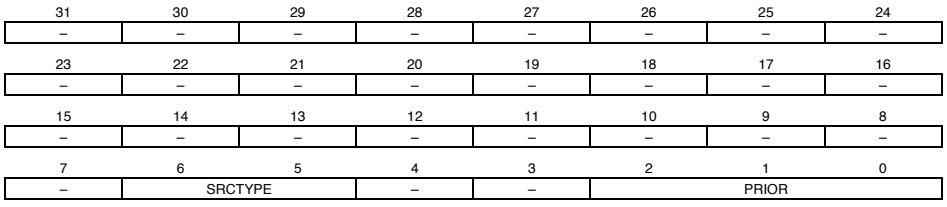


Figure 5.2: Overview of SMR register, from [4]

Priority in `PRIOR` field of SMR register can be set between 0 (lowest priority) to 7 (highest priority).

`SRCTYPE` field of SMR register can be set to the values displayed in the following table:

SRCTYPE		Internal Interrupt Sources	External Interrupt Sources
0	0	High level Sensitive	Low level Sensitive
0	1	Positive edge triggered	Negative edge triggered
1	0	High level Sensitive	High level Sensitive
1	1	Positive edge triggered	Positive edge triggered

The active level or edge is not programmable for the internal interrupt sources. When AIC_SMR register is set, to enable interrupts, set

```
AT91C_BASE_AIC->AIC_IECR = 1 << interrupt_peripheral_id;
```

Advanced interrupt controller initialization is located in `irq/irq.c` file.

In RTEMS on NXT, to register a particular interrupt, interrupt source and also its handler method, device drivers use added function `rtems_handler_install_nxt`.

5.2.2 Advanced interrupt controller protect mode

To be able to read AIC registers via debugger, it is necessary to set the Protect Mode, which permits reading the Interrupt Vector Register (IVR) without performing the associated automatic operations. This is necessary, because debugger might read AIC_IVR register and thus undesired consequences might occur, such as:

- If an enabled interrupt with a higher priority than the current one is pending, it is stacked.
- If there is no enabled pending interrupt, the spurious vector is returned.

When AIC_IVR is read while no enabled interrupt source is pending, AIC detects a spurious interrupt. When this happens, the AIC returns the value stored by the programmer in AIC_SPU (Spurious Vector Register). In RTEMS, default exception handler is stored in AIC_SPU.

Accessing AIC_IVR changes system behavior and is not desired. To enable protected mode, set

```
AT91C_BASE_AIC->AIC_DCR = 1;
```

Spurious vector initialization and protected mode enabling is also located in `irq/irq.c` file.

5.3 Interrupt handling

In following text, RTEMS on NXT interrupt handling will be presented.

As soon as exception occurs (e.g. data abort, undefined instruction, IRQ), processor jumps to the corresponding instructions situated at addresses from `0x00` to `0x1C`. Those are addresses of exception handlers, which were copied to the SRAM memory after remap during RTEMS initialization:

```
LDR PC, __Reset_Addr /* Reset handler address*/
LDR PC, __Undefined_Addr /* Undefined instruction handler*/
LDR PC, __SWI_Addr /* Software interrupt */
LDR PC, __Prefetch_Addr /* Prefetch abort*/
LDR PC, __Abort_Addr /* Data abort*/
NOP /* reserved */
LDR PC, __IRQ_Addr /* Interrupt handler*/
LDR PC, __FIQ_Addr/* Fast interrupt handler*/
```

Processor starts to execute IRQ interrupt handler in ARM state and is switched to IRQ operating mode. Interrupt handler method `_ARMV4_Exception_interrupt` for ARM processor is already present in RTEMS. It saves registers, saves context, switches from ARM to Thumb mode and calls `bsp_interrupt_dispatch` method. This method determines the interrupt vector number of interrupt, that caused the exception by reading AIC_IVR (Interrupt Vector Register) from AIC. Reading AIC_IVR acknowledges interrupt. Current interrupt is determined according to its priority at the time when AIC_IVR is read. If multiple interrupt sources of equal priority are pending and enabled when the AIC_IVR is read, the interrupt with the lowest interrupt source number is serviced first. Interrupts with higher priority can be handled during the service of interrupt with lower priority.

To be able to know, which interrupt source triggered interrupt, value of corresponding AIC_SVR register (Source Vector Register) is returned when reading AIC_IVR (Interrupt Vector Register). These values were written to AIC_SVR registers during AIC initialization. RTEMS uses single point of entry for interrupts, where it reads AIC_IVR to get interrupt number. This number is expected to be the number, which determines the position in the table, where all registered interrupt handlers are stored. This table is stored in SRAM memory right behind the exception vectors.

To end the interrupt treatment, write to the End of Interrupt Command Register (AIC_EOICR) must be performed. It does not matter, what value will be written, only the write itself is necessary.

5.4 Master clock

Master clock is the clock provided to all the peripherals and the memory controller. It must be set to the specific value for devices to work properly. In following text, instructions for setting Master clock to desired frequencies will be discussed.

In NXT, clock is generated by the Clock Generator. It provides the clock for Master clock.

Clock generator is made up of a PLL (Phase Lock Loop), a Main Oscillator and an RC Oscillator. It provides the following clocks:

- SLCK, the Slow Clock, which is the only permanent clock within the system
- MAINCK is the output of the Main Oscillator
- PLLCK is the output of the Divider and PLL block

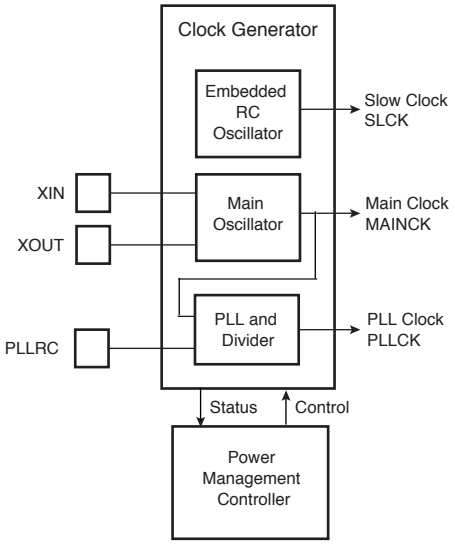


Figure 5.3: Overview of a Clock Generator, from [4]

5.4.1 Master clock initialization

The Clock Generator User Interface is embedded within the Power Management Controller, which starts at address `0xFFFF FC00`. Main Oscillator is enabled in Main Oscillator Register (CKGR_MOR) at address `0xFFFF FC20` by writing a value `0x0000 0601`. This value is computed as enabled MOSCEN bit (bit number 0) in Main Oscillator Register and value of the oscillator startup time, which is computed by looking at the DC characteristics given in the datasheet of the board. Oscillator frequency is set to 18.432 MHz. When oscillator is started, time is needed for oscillator to stabilize. Oscillator is stabilized when bit number 0 at address `0xFFFF FC68` (register PMC_SR) is set to 1.

Once the oscillator is started and stabilized, the PLL can be configured. The PLL is made up of two chained blocks: the first one divides the input clock, while the second one multiplies it. The MUL and DIV factors are set in the PLL Register at address `0xFFFF FC2C`. These two values must be chosen according to the main oscillator (input) frequency and the desired main clock (output) frequency. Since Master clock is expected to run at 48 MHz (48054857Hz), which will be frequency output divided by 2, frequency output should be close to 96109714. DIV and MUL values will be 14 and 72.

Like the main oscillator, a PLL startup time must also be provided. It is also computed from datasheet. PLL is locked when bit number 2 at address `0xFFFF FC68` (register PMC_SR) is set to 1.

Now the remaining prescaling value of the main clock must be set and the PLL output must also be selected. Value 0x7 is written into Master Clock Register (PMC_MCKR) at address `0xFFFF FC30`, which means that PLL Clock is selected as Master clock and Processor Clock Prescaler is set as selected clock (PLL Clock) divided by 2. Master clock is ready when bit number 3 at address `0xFFFF FC68` (register PMC_SR) is set to 1.

Clock initialization is done in assembly code, at the beginning of the RTEMS on NXT initialization. It is located in `start/start.S` file.

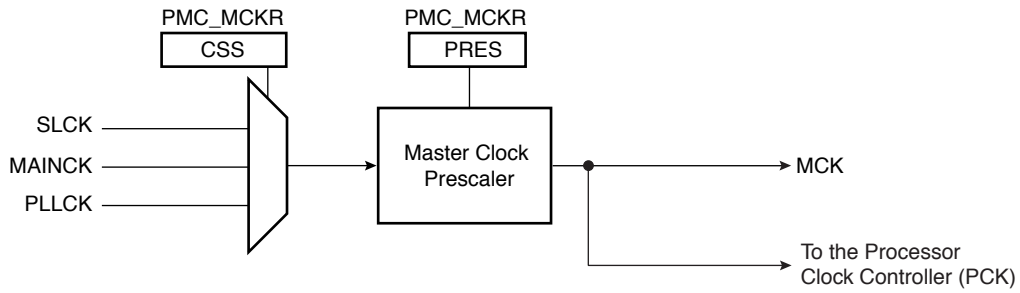


Figure 5.4: Overview of Master Clock controler, from [4]

5.4.2 Clock interrupt

File `clock/clock.c` stores routines for measuring time, initializes Periodic Interval Timer (PIT) which generates the operating system's periodic scheduler interrupt. The PIT provides a programmable overflow counter and a reset-on-read feature. It consists of two counters: a 20-bit CPIV counter and a 12-bit PICNT counter. Both counters work at Master Clock /16. The first 20-bit CPIV counter increments from 0 up to a programmable overflow value set in the field PIV of the Mode Register (PIT_MR) at address 0xFFFF FD30. When the counter CPIV reaches this value, it resets to 0 and increments the Periodic Interval Counter, PICNT. The status bit PITS in the Status Register (PIT_SR, address `0xFFFF FD04`) rises and triggers an interrupt, provided the interrupt is enabled (bit PITIEN in PIT_MR).

PIT_MR register should trigger interrupt every 1ms, which corresponds to a PIV value of 0xBBA, which is Master Clock / 16 / PIT_FREQUENCY(1000) -1. Bit PITEN, which enables interrupt triggering and bit PITIEN, which enables timer are added, so the final value to be set to the PIT_MR register is `0x3000BBA`.

Structure of PIT_MR register is shown in Figure 5.5.

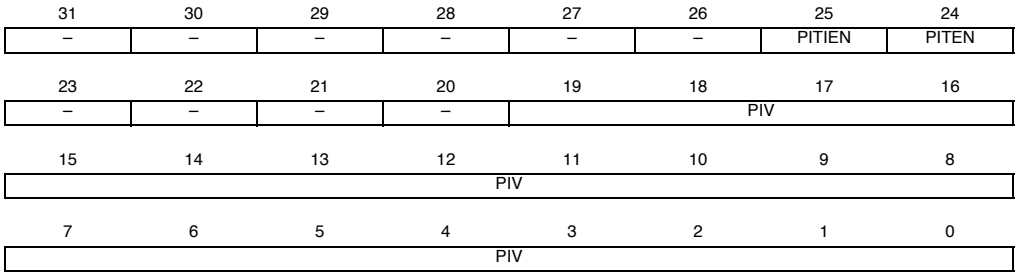


Figure 5.5: Overview of PIT_MR register, from [4]

When CPIV and PICNT values are obtained by reading the Periodic Interval Value Register (PIT_PIVR), the overflow counter (PICNT) is reset and the PITS is cleared, thus acknowledging the interrupt. The value of PICNT gives the number of periodic intervals elapsed since the last read of PIT_PIVR. When CPIV and PICNT values are obtained by reading the Periodic Interval Image Register (PIT_PIIR), there is no effect on the counters CPIV and PICNT, nor on the bit PITS. For example, a profiler can read PIT_PIIR without clearing any pending

interrupt, whereas a timer interrupt clears the interrupt by reading PIT_PIVR. The PIT is stopped when the core enters debug state.

When timer interrupt occurs, `Clock_driver_support_at_tick` is called. It reads PIT_PIVR register and executes routines that are expected to run every 1ms, such as link to AVR co-processor must be sustained, motor interrupt is raised so the quadrature can be computed, buttons are checked whether some of them were pressed, if debug output is enabled then debug messages are sent. Because of these restrictions, following directive must be present in RTEMS on NXT application:

```
#define CONFIGURE_MICROSECONDS_PER_TICK 1000
```

Directive for clock driver can not be omitted:

```
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
```

More information on application directives can be found in Section 6.3.1.

5.4.3 Time in RTEMS

RTEMS automatically stores elapsed ticks and increments this value every 1ms. Number of elapsed ticks, which corresponds to number of elapsed ms is accessible via `systick_get_ms` method.

For counting elapsed time in nanoseconds, RTEMS on NXT provides counter `nxt_start_ns` and function `nxt_get_ns_count` for retrieving elapsed time. For example, they can be used for rough estimate of task duration. Since there is no nanoseconds counter, for nanoseconds precision they read PIT_PIIR register and compute elapsed nanoseconds from its current value, therefore the value is just an rough estimate.

RTEMS has structures to store real time clock, but such information is useless in NXT and this time will always be lost when NXT is shut down. For applications, time from NXT start or time elapsed between some events is more valuable. If real time is set in application (using `rtems_time_of_day` structure), it can be used to wake task at a precise time.

5.5 Board support package

RTEMS has a multi-layered approach to portability. It is oriented to maximize the amount of software that can be reused, although some parts of the executive are specific to hardware in some sense. RTEMS classifies target dependent code based upon its dependencies into one of the following categories: CPU dependent, Peripheral dependent and Board dependent.

Reusable peripheral dependent files, device drivers for peripherals and for standard controllers can be found at `c/src/lib/libchip`. NXT specific device drivers are stored in board support package.

CPU dependent files are stored in `cpukit/score/cpu/arm`. As mentioned earlier, CPU support for ARM processor is already present in RTEMS. CPU dependent support files for NXT are stored in `c/src/lib/libcpu/ARM/AT91SAM7`. Its

subdirectories contain additional support routines, such as clock driver, timer support, debug unit driver, header files for device address definitions, device drivers for peripheral controllers.

The `automake` source file `c/src/lib/libcpu/arm/Makefile.am` together with the `autoconf` source file `c/src/lib/libcpu/arm/configure.ac` hold necessary information for compiling parts/subdirectories of processor specific routines for NXT.

For support of variety of devices, RTEMS stores device dependent functionality in bundles called BSP - Board Support Packages. To add support for a new device into RTEMS, new board support package should be created.

Board dependent files are stored under `c/src/lib/libbsp`.The BSP source directory is further subdivided by the CPU family and BSP. NXT BSP is stored in `arm/nxt`. Under BSP directory, all BSPs follow subdirectories naming convention. Directories console, include, make, start and startup are commonly used between BSPs. Under NXT, following directories joining common functionality are found:

- **console:** is technically the serial driver for the BSP rather than just a console driver, it deals with the board UARTs (Universal Asynchronous Receiver/Transmiter). NXT does not have a console, it is implemented only to prevent errors when programmer uses print() function. To decrease application size, console support including should be disabled via macro.
- **drivers:** contains drivers for NXT devices, such as display, Bluetooth, motors, sensors, AVR co-processor...
- **ECRobot:** API for accessing internal and external peripherals
- **include:** include files for NXT platform
- **irq:** interrupt handling routines
- **make:** flags to be used during compilation
- **start:** assembly code for board initialization
- **startup:** overridden RTEMS startup routines

In following chapters, each of the BSP's directories and its functionality will be presented in detail.

5.5.1 ECRobot

ECRobot API consists of low level device API and ECRobot wrapper API (prefix of the API is `ecrobot_`). In comparison with nxtOSEK ECRobot API, some functions were omitted, such as nxtOSEK system hook routines. Overall usage and access to the sensors, motors, Bluetooth, etc. remained the same. ECRobot API was extended with `debugPrint` function for displaying debug output via `nxjconsoleviewer`. Usage of `nxjconsoleviewer` is further described in Section 7.1.1. Complete API is enclosed on DVD in form of a web page.

5.5.2 make

Custom configuration file `nxt.cfg` is used for building RTEMS application. Once the BSP is installed to be used to build new applications, the same configuration must be used when building the application. So it must include a build configuration file. `nxt.cfg` holds information for compiler, such as flags for optimization, processor version, instruction set, floating point usage. By default, optimization flags is set to *-0s -s*. *-Os* enables all *-O2* optimizations that do not typically increase code size. -s option removes symbol table and relocation information from the executable. Debugging symbols for GDB can be added via *-g* option. HW Floating point unit is not used, because NXT does not have a FPU. Floating point operations are emulated by compiler, which is enforced by *-mfloat-abi=soft* option. *-mthumb* and *-mthumb-interwork* are used to set default generation of Thumb instructions and to generate code that supports switching between ARM and Thumb instruction sets.

5.5.3 start

`start` directory contains NXT assembly startup code, `start.S`, which is responsible for correct NXT devices initialization. Detail overview of initialization is described in Section 5.1. After assembly initialization is completed, RTEMS continue with system initialization as is described in Section 5.7.

5.5.4 Startup and shutdown

In `startup` directory, overridden startup initialization files for NXT are stored. File `bootcard.c` provides framework for the BSP initialization sequence. It was updated with startup screen step. Startup screen is discussed in Section 5.9. Changed implementation of startup steps are located in `bspstart.c`.

`bspreset.c` implements `rtems_shutdown_executive`, it shuts down NXT. NXT is shut down via AVR processor, which controls NXT power management. AVR communicate with ARM7TDMI processor though I2C bus via sending data of the following structure:

```
typedef   struct
{
   UBYTE Power;
   UBYTE PwmFreq;
   SBYTE PwmValue[NOS_OF_AVR_OUTPUTS];
   UBYTE OutputMode;
   UBYTE InputPower;
} IOTOAVR;
```

When powering down the NXT, the Power byte should be set to `0x5A` and the `PwmFreq` byte should be set to `0x0`. It will turn of the NXT and NXT will be awaken when the orange “Select” button will be pressed.

`linkcmds.nxt` describes locations of parts of the code in memory for linker. It is used as a part of the linker scripts. Parts of program that are intended to be located in SRAM will be copied to their appropriate location during system initialization.

5.6 Drivers

Drivers for peripherals were divided into subdirectories named by the devices they handle. They implement access to the hardware devices. Most of their functions are not meant to be called directly from application code. Drivers are heavily inspired by nxtOSEK and leJOS. These systems already have implementations for all necessary devices. Modifications had to be made to be able to use drivers in RTEMS environment. nxtOSEK has different approach to interrupt handlers, as mentioned in Section 4.4.1 and also enables/disables interrupts differently. LeJOS uses JAVA, therefore some of the used structures were changed. Also missing support for HiTechnics color sensor v2 was added.

Following chapters briefly discuss each of the drivers according to their location in subdirectories.

5.6.1 AVR

Driver handles communications between the NXT main ARM processor and the AVR co-processor. The AVR provides support for the motors, analogue sensors, keyboard and power. The two processors are linked via a TWI connection. It is used to exchange a message every 1ms. The message alternates between sending commands to the AVR and receiving status from it. Reading status data from AVR takes place via DMA/Interrupt handling.

If application is running and NXT brick still emits "ticking" sound, it is a sign that AVR was not set up correctly. This sound is also emitted if NXT brick is in a firmware flashing mode.

5.6.2 Bluetooth

Directory contains driver for Bluetooth device, implements sending messages up to 256 characters long, handles buffers for receiving messages. Functions for initiating connections, setting Bluetooth address, wrappers for sending and receiving messages are part of ECRobot API.

5.6.3 Display

Driver contains functions to convert strings, numbers to actual characters on the screen. Its 100x64 pixels are divided into 8 lines of 16 characters. In memory, display is represented as a char array, which has one extra line:

```
static uint8_t display_buffer[DISPLAY_DEPTH+1][DISPLAY_WIDTH];
```

This helps to use only one DMA transfer to update display. Default behavior is to use DMA refresh. Display can also be refreshed without DMA usage, using `nxt_lcd_force_update`. It is necessary if display needs to be refreshed with disabled interrupts, otherwise display would never be refreshed. It takes about 17ms for entire display to be rendered. To communicate with display hardware and start display refresh, an SPI – Serial Peripheral Interface is used.

5.6.4 Flashprog

Routines to write into Flash memory to set flags for NXT BIOS, as mentioned in Section 5.10. It prepares memory for writing and handles writing of selected page. Flash memory must be written by entire pages.

5.6.5 I2C

I2C is an Inter-Integrated Circuit, which is a multimaster serial single-ended bus for attaching low-speed peripherals. Directory contains implementation of the software-based I2C bus, which handles communication with compatible devices connected via the sensor ports that are located at the bottom of NXT.

5.6.6 Motor

Motor power management is handled by AVR processor. Driver serves as interface for setting and reading motor parameters. To be able to read motor quadrature, interrupt is raised from periodical timer interrupt, which ticks every 1 ms, to decode quadrature and save number of counts for each connected motor into global structure.

5.6.7 Sensors

Some sensors are accessed via PIO – Parallel Input/Output Controller. Driver provides interface for sensors, which do not use I2C bus for retrieving sensor values.

5.6.8 Sound

Support for audio output is provided that enables the tone generation. Is uses pulse density modulation to actually produce the output. To produce a tone, a single pulse density modulation encoded cycle is created (having the requested amplitude), this single cycle is then played repeatedly to generate the tone. The actual output of the bits is performed using the built in Synchronous Serial Controller (SSC). This is capable of outputting a series of bits to port at fixed intervals.

5.6.9 UDP

Driver implements USB support, uses mixture of interrupt driven and directly driven I/O. Interrupts are used for handling the configuration/enumeration phases, which allows to respond quickly to events. For actual data transfer, data are handled directly thus removing need to have data buffers available during interrupt. Implementation uses the standard Lego identifiers (and so can be used from the PC side applications that work with the standard Lego firmware, in addition to leJOS applications)

5.7 RTEMS on NXT Initialization

When RTEMS on NXT initialization is switched from assembly code into C code, boot_card method in `startup/bootcard.c` is executed. Then RTEMS on NXT initialization continues initializing its data and internal structures, etc. Figure 5.6 illustrates program flow during initialization.

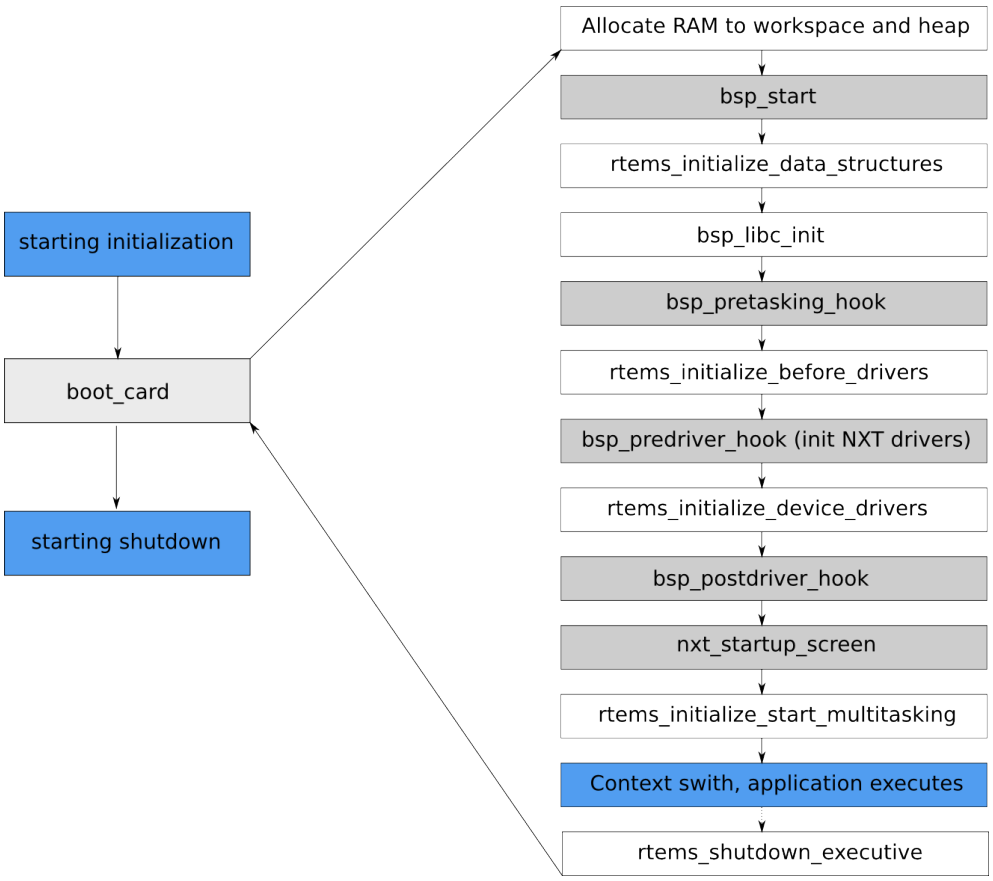


Figure 5.6: RTEMS initialization overview

RTEMS enable programmer to override default startup methods to add initialization of necessary drivers for devices. Default methods like `bsp_start`, `bsp_predriver_hook` are overridden in RTEMS on NXT, in file `bspstart.c`. The `bsp_predriver_hook` executes initialization methods for display driver, avr driver, i2c bus, Bluetooth, etc. An additional step has been added – startup screen, which is implemented by `nxt_startup_screen` method. It was added to achieve more user-friendly usage of RTEMS applications. More on startup screen can be found in Section 5.9.

To shutdown NXT from application, `rtems_shutdown_executive` should be called. It returns program execution back to the `bootcard.c` and will shut NXT down properly.

5.8 Configuration table

For adding directives, for example for debug print, file `confdefs.h` was modified. It is a C language header file that based on the setting of a variety of macros can automatically produce nearly all of the configuration tables required by an RTEMS on NXT application.

The primary configuration table for RTEMS application, was extended with `NXT_Configuration` structure, which holds flags according to the directives inputted by application programmer in RTEMS on NXT application header file.

Newly added directives are `SHOW_STARTUP_SCREEN` and all directives mentioned in Section 6.3.1, which begin with NXT_.

5.9 Startup screen

If `#define SHOW_STARTUP_SCREEN` is set in application, execution of RTEMS on NXT application can be postponed. When RTEMS on NXT is fully initialized, just before first task is executed, startup screen is displayed. Startup screen shows battery status, Bluetooth address or status of USB/Bluetooth connection, description for buttons functionality. At this point, user can run application, shut down NXT, configure NXT BIOS to flash new application at the next start or connect to the remote console output via USB or Bluetooth.

This feature can be helpful. For example, imagine you have line following application on your NXT and you want to retrieve readings of the color sensor. Immediately after you turn on the NXT, NXT starts to move. You have to catch it or hold it until you will connect with the remote output and after the connection is established, you will release it so it can follow the line. If you get distracted for a second when you are establishing the connection and NXT is on your table, it can fall down and brake. With the startup screen, you can also retrieve debug messages from the time when the first task is executed.

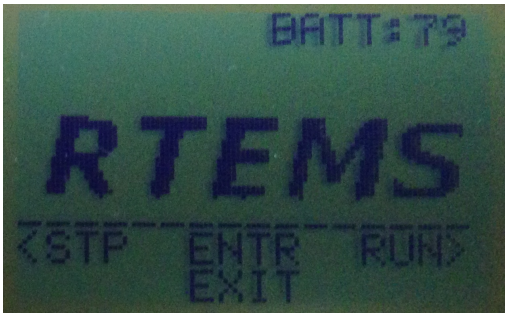


Figure 5.7: RTEMS startup screen

5.10 Flash application mode

To enter flash application mode in NXT BIOS after NXT is turned on, `"Jumpin' Jack Flash"` must be set to Flash memory at specified location, as described in Section 4.2.1. NXT Flash memory has a restriction. It can not be read and

written at the same time. Code, that is writing to the flash memory must be copied to the SRAM memory, executed from there. Code is copied to the SRAM memory during RTEMS on NXT startup.

In file `start.S`, routine labeled `fast_test_copy` loads the compiled code from section `.bsp_fast_text` into SRAM memory. In fact, only one method belongs to this section and will be copied during startup: `flash_write_req_to_page`.

Flash application mode is set, when user presses start and left buttons simultaneously for at least 100ms. It can be done when application is running or when RTEMS on NXT is waiting in startup screen. Check, whether buttons were pressed is executed during each timer interrupt. Cleaner solution would be to have special task which is executed periodically at specified time, but this approach would affect user applications. Programmer would always have to add specific task for checking buttons. If he forgot, he would not be able to flash new application, new NXT BIOS would have to be installed.

5.11 Application uploader

Uploader is based on leJOS NXJ [7], which is a firmware replacement for Lego Mindstorms NXT, includes a JAVA Virtual machine and multiple useful utilities. Uploader uses leJOS NXJ package `lejos.pc.comm` for communication between computer and NXT. It uses LibNXT [11], which is a low level utility library for talking to the NXT. LibNXT handles USB communication and locates NXT in USB devices tree. LibNXT is used only for Linux based platforms, where `libusb 0.1` is supported. For different platforms, such as Windows, Lego's Fantom driver is used instead of LibNXT.

Uploader is a wrapper using methods from `lejos.pc.comm` package, especially class `lejos.pc.comm.NXTCommUSB` has all methods for searching, opening and writing to the USB device. It is implemented by `NXTCommFantom` or `NXTCommLibnxt`, depending on used platform. Uploader starts by finding NXT device in computer USB devices tree. To access NXT over USB, root privileges are necessary thus uploader should be run using `sudo` utility or user needs to have sufficient privileges to access USB device. Once NXT is found and accessible, it establishes connection with NXT BIOS. Newly uploaded application is transmitted divided into small 64 bytes chunks. NXT BIOS stores chunks in buffer until it reaches 256 bytes. Then received bytes can be saved as one entire page into FLASH memory.

Chapter 6

Environment preparation and application building

Following chapter will provide step-by-step description, how compilation toolchain for RTEMS on NXT was built from sources. Instructions for building RTEMS are provided. Also steps for building RTEMS on NXT applications will be discussed.

RTEMS build tools are available for multiple platforms (development hosts). Following instructions should be used to prepare building environment on Linux based host. For different operating system, follow [30]. RTEMS on NXT does not impose any constrains on used development host.

6.1 Cross-compiler toolchain

To build cross-compiler toolchain from sources, following utilities will be necessary:

- Binutils
- GCC (gcc-core)
- GCC (gcc-g++)
- GDB
- Newlib
- GMP (Multi-precision arithmetic library)
- MPC (Multi-precision complex floating-point library)
- MPFR (Multi-precision floating-point computation library)

In chosen directory location, create two directories: `tools` and `archive`. All mentioned utilities can be found at [18] as tarballs. Download mentioned archives into `archive` directory. After completion, directory structure should look like and contain following files:

```
~$ find .
./archive
./archive/binutils-x.yy.tar.bz2
./archive/binutils-x.yy-rtems4.11-YYYYMMDD.diff
./archive/gcc-x.y.z.tar.xz
./archive/gcc-x.y.z-rtems4.11-YYYYMMDD.diff
./archive/gdb-x.y.tar.bz2
./archive/gdb-x.y-rtems4.11-YYYYMMDD.diff
./archive/newlib-x.yy.z.tar.gz
./archive/newlib-x.yy.z-rtems4.11-YYYYMMDD.diff
./tools
```

There are also some RTEMS specific patches, that need to be applied to downloaded utilities before building. The most recent patches should be used.

Go to `tools` directory, unpack archives and apply most recent patches:

```
~/tools$ tar xf ../archive/binutils*.tar.*
~/tools$ tar xf ../archive/gcc*.tar.*
~/tools$ tar xf ../archive/newlib*.tar.*
~/tools$ tar xf ../archive/gdb*.tar.*

~/tools$ cd binutils-x.yy/
~/tools/binutils-x.yy$ cat ../../archive/binutils*.diff | \
                           patch -p1
~/tools/binutils-x.yy$ cd ../gcc-x.y.z/
~/tools/gcc-x.y.z$ cat ../../archive/gcc*.diff | \
                           patch -p1
~/tools/gcc-x.y.z$ cd ../newlib-x.yy.z/
~/tools/newlib-x.y.z$ cat ../../archive/newlib*.diff | \
                           patch -p1
~/tools/newlib-x.y.z$ cd  ../gdb-x.y
~/tools/gdb-x.y$ cat ../../archive/gdb*.diff | \
                           patch -p1
~/tools/gdb-x.y$ cd ..
```

6.1.1 Building binutils, gcc, gdb

For `binutils` compilation, target platform and location for built executables has to be specified. It can be set by exporting shell variables, which will be used during configuration.

```
$ export TARGET=arm-rtems4.11
$ export PREFIX=/opt/rtems4.11
```

When installing for Debian system, install missing requirements first:

```
sudo apt-get install m4 patch build-essential texinfo cvs
     libncurses5-dev libgmp3-dev libmpfr-dev libmpc-dev libtool
```

RTEMS also needs `autoconf` and `automake` tools to be available on the host.

`binutils` should be build into newly created directory using following commands:

```
~/tools$ mkdir b-binutils
~/tools/binutils-x.yy$ cd b-binutils
~/tools/binutils-x.yy$ ../binutils-x.yy/configure
                            --target=${TARGET}
                            --prefix=${PREFIX}
~/tools/binutils-x.yy$ make all
~/tools/binutils-x.yy$ make info
~/tools/binutils-x.yy$ sudo make install
~/tools/binutils-x.yy$ cd ..
```

You need to include built executables into your `PATH` variable, so that they can be found in further compilation:

```
$ export PATH=/opt/rtems-4.11/bin:${PATH}
```

`gcc` must be build with a standard C library. RTEMS uses `newlib`:

```
~/tools/b-binutils$ cd gcc-x.y.z/
~/tools/gcc-x.y.z$ ln -s ../newlibx.yy.z/newlib .
~/tools/gcc-x.y.z$ cd ..
~/tools$ mkdir b-gcc
~/tools$ cd b-gcc/
~/tools/b-gcc$ ../gcc-x.y.z/configure \
                            --target=${TARGET} \
                            --with-gnu-as \
                            --with-newlib \
                            --verbose \
                            --enable-threads \
                            --enable-languages="c,c++" \
                            --prefix=${PREFIX}
~/tools/b-gcc$ make all
~/tools/b-gcc$ make info
~/tools/b-gcc$ sudo make install
~/tools/b-gcc$ cd ..
```

`gdb` needs to be build only if you plan to debug NXT using JTAG adapter:

```
~/tools$ mkdir b-gdb
~/tools$ cd b-gdb/
~/tools/b-gdb$ ../gdb-x.y/configure \
                        --target=${TARGET} \
                        --prefix=${PREFIX}
~/tools/b-gdb$ make all
~/tools/b-gdb$ make info
~/tools/b-gdb$ sudo make install
~/tools/b-gdb$ cd ..
```

6.2 RTEMS building

When building RTEMS, it is not necessary to build all BSPs. Target BSP can be chosen by specifying *–enable-rtemsbsp=nxt*. Options for disabling networking, posix, C++ can be used. For example, during development, *–enable-tests=samples* option can be used to build sample applications together with building RTEMS. Samples can be used as small testing applications without necessity to install RTEMS into installation directory and then building testing application from a different environment again.

RTEMS is designed to be compiled outside its source directories. Assume that RTEMS sources are stored under `rtems` directory and RTEMS will be build in newly created `rtems-build` directory:

```
~$ cd rtems
~/rtems$ export PATH=/opt/rtems-4.11/bin:$PATH
~/rtems$ ./bootstrap
~/rtems$ cd ..
~$ mkdir rtems-build
~$ cd rtems-build
~/rtems-build$ ../rtems/configure --target=arm-rtems4.11
           --prefix=/path/to/directory/rtems-build
           --enable-rtemsbsp=nxt
           --enable-posix
           --disable-networking
           --disable-cxx
           --enable-tests=samples
~/rtems-build$ make
~/rtems-build$ sudo PATH=/opt/rtems-4.11/bin:${PATH} make install
```

6.3 Application building

Once RTEMS was build and installed on the host, RTEMS application can be build. To build application, `RTEMS_MAKEFILE_PATH` must be specified. Variable `RTEMS_MAKEFILE_PATH` is the same as `$PREFIX` used when building RTEMS, with addition of `arm-rtems4.11/nxt`.

```
~$ cd exampleApplication
~$ export RTEMS_MAKEFILE_PATH=${PREFIX}/arm-rtems4.11/nxt
~$ make
```

Examples of entire makefile can be found in enclosed sample applications.

6.3.1 Application optimization

RTEMS contains directives to include or exclude parts of application to be built. To minimize memory requirements, some unnecessary components can be disabled. Also some configuration directives are necessary for proper functioning of

NXT. In following text, it will be discussed which directives must be present and which directives will help to minimize RTEMS application memory footprint.

Clock driver has to be present, otherwise NXT won't work as expected

```
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
```

NXT clock is expect to create interrupt for scheduler every 1ms. For proper time computation, RTEMS needs to know how many microseconds elapses in one timer tick.

```
#define CONFIGURE_MICROSECONDS_PER_TICK 1000
```

Usually, applications have console driver. In RTEMS on NXT, console driver is not necessary. Therefore following directive should not be present.

```
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
```

Since RTEMS does not need the console driver, there is no reason to configure termios.

```
#define CONFIGURE_TERMIOS_DISABLED
```

RTEMS uses no drives, so no filesystem and libio support is necessary, also any stdio.

```
#define CONFIGURE_APPLICATION_DISABLE_FILESYSTEM
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 0
```

Disable reentrancy support in the C library. It is usually not something an application wants to do unless the developer is committed to use library routines that are known to be reentrant.

```
#define CONFIGURE_DISABLE_NEWLIB_REENTRANCY
```

To enable startup screen, do not forget to define

```
#define SHOW_STARTUP_SCREEN
```

Depending on using debug print output, define either Bluetooth or USB debug output. Though both, Bluetooth and USB debug output can be enabled at the same time, it is not recommended. Also when using Bluetooth debug output, Bluetooth should not be used in RTEMS application, for example for communication with another NXT.

```
#define NXT_ENABLE_BT_DEBUG_OUTPUT
#define NXT_ENABLE_USB_DEBUG_OUTPUT
```

When using Bluetooth to debug, programmer can specify name for NXT and also a Bluetooth pairing key. Device name can be used in `nxjconsoleviewer` application to identify the NXT. Default NXT name is `"MYNXT"` and default pairing key is `"1234"`. Maximum length for NXT name and pairing key is 16 characters. To change default values, following macros should be set:

```
#define NXT_ENABLE_BT_DEBUG_OUTPUT_NXT_NAME "MYNXT"
#define NXT_ENABLE_BT_DEBUG_OUTPUT_NXT_PIN "1234"
```

For RTEMS Workspace memory pool and C Program Heap memory pool to be occupying the same memory area, use following macro. Having both pools together in one area can improve memory usage.

```
#define CONFIGURE_UNIFIED_WORK_AREAS
```

To be thorough, header file should contain at least following includes to be able to use macro definitions and call NXT specific functions:

```
#include <rtems/confdefs.h>
#include <bsp.h>
#include <ecrobot_interface.h>
```

6.4 Application uploading

To use uploader for application uploading, NXT BIOS must enter into flash application mode. To enter flash application mode, see Section 5.10. NXT must be connected to computer using USB cable. Uploader takes path to the built application image - `.rom` as first argument. During upload, uploader as well as NXT BIOS shows upload status. Once application is uploaded, FINISHED is shown at LCD display.

Following Figure 6.1 shows LCD display before, during and after application is flashed.

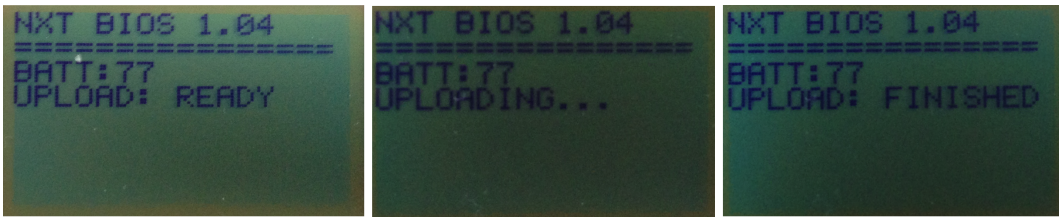


Figure 6.1: NXT LCD screen during application uploading

During development and debugging, application uploading is much easier done via JTAG adapter. Code can be uploaded and debugged via the same interface, you won't need to connect USB and run NXT BIOS to upload a newly built code. With application, which will not have upload functionality, or which does not have this functionality yet, using JTAG is the only way. JTAG debugging will be discussed in Section 7.2.

Uploader can be used on both Linux and Windows systems. For uploader to work in Linux environment, JAVA and `libusb0.1` must be present. In Windows environment, Lego's Fantom driver must be installed and 32-bit version of JAVA must be present.

Chapter 7

Debugging

In following chapters, two ways of debugging RTEMS will be described. RTEMS can be debugged using debug output to the remote console, or using more hardware oriented approach, JTAG adapter.

7.1 Debug print

RTEMS application can send output to the remote computer. It can communicate with remote computer either via USB cable or wireless via Bluetooth. At the remote computer, `nxjconsoleviewer` application must be running.

7.1.1 Communication with nxjconsoleviewer

For RTEMS on NXT to communicate with `nxjconsoleviewer`, functionality of leJOS `lejos.nxt.comm.RConsole` class had to be implemented. Establishing connection is done only when NXT startup screen is shown. After application is started, connection will not be established and all debug messages will be discarded. Connection is established using standard USB or Bluetooth methods in ECRobot API, that is available to programmer. Connection establishing is implemented in `ecrobot_rtems_startup_screen` function, which executes method `ecrobot_init_bt_slave` for initiating Bluetooth connection as a slave device and `ecrobot_init_usb` for USB connection. For USB, connection holding function must be invoked every 1 ms. It is done from regular timer tick interrupt handler. If user wants to use USB in RTEMS on NXT application, he has to initialize the connection and invoke `ecrobot_process1ms_usb` function every 1 ms by himself. When Bluetooth connection is establish, there is no need to do anything to sustain the connection.

When the connection is successfully established, to send debug output to the `nxjconsoleviewer`, function `debugPrint(const char * format, ...)` must be used. It has the same input syntax as standard printf function. Limitations for message size are 64 chars for USB and 254 chars for Bluetooth. Internally, `debugPrint` uses standard `vsnprintf` function for input conversion. This operation is quite time consuming, therefore using `debugPrint` might have negative impact on the task execution speed. Remember this when using `debugPrint`.

`ECRobot_bt_data_logger` and `ECRobot_bt_data_logger_string` are method that can also be used for debugging. `ECRobot_bt_data_logger` sends pre-fixed internal NXT information, such as sensors values, motors revolutions, time. It is designed to be used with NXT GamePad application. It sends raw numeric values in specific packet format. Method `ECRobot_bt_data_logger_string` is its equivalent, it send the same information in string format, that can be displayed in `nxjconsoleviewer`. This method uses `debugPrint` for sending data, therefore it uses the same debug output channel and can have the same impact on the task execution speed.

Although `nxjconsoleviewer` has a built-in functionality to display NXT LCD screen, this feature is not used in RTEMS.

7.1.2 Connecting to nxjconsoleviewer

When NXT is started, RTEMS should display startup screen and wait for user input. This is the time when the `nxjconsoleviewer` should be connected to the NXT. If the Bluetooth debug output is enabled, the Bluetooth interface address of the device and NXT name will be present at the NXT LCD display.

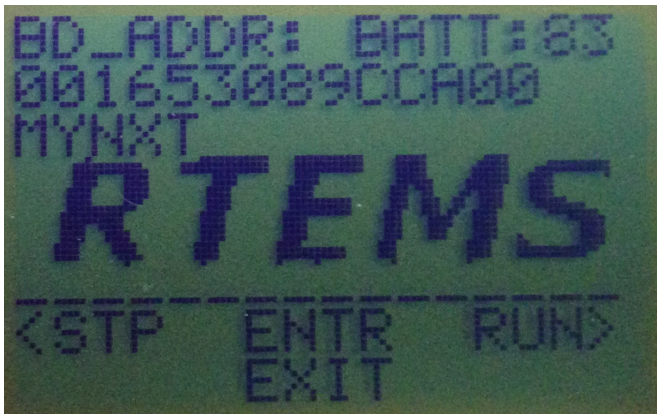


Figure 7.1: RTEMS startup screen

This address or name can be used in `nxjconsoleviewer` to identify the NXT. To connect to the NXT from `nxjconsoleviewer`, select the USB or Bluetooth connection type. If connecting via Bluetooth, then fill in the NXT Bluetooth interface address or name, that is shown at NXT LCD display and click `Connect`. Address on LCD display is displayed with two trailing zeros. Bluetooth address in `nxjconsoleviewer` must be entered without them. For example, if NXT displays address `0016530CA62F00`, input value for `nxjconsoleviewer` will be `0016530CA62F`. When only one NXT is in range of `nxjconsoleviewer` host, connection will be established without need to input name or address. Otherwise name or address must be provided. Connection is successfully established when you can see message `"BT Connected"` or `"USB Connected"` at the NXT display.

When prompted for Bluetooth pairing key, default key is set to `"1234"`.

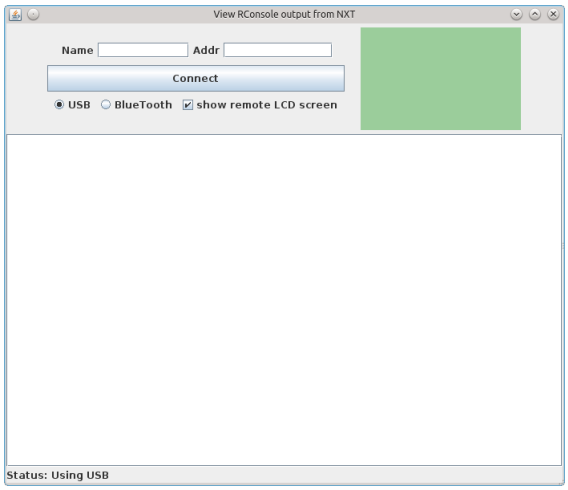


Figure 7.2: nxjconsoleviewer application

7.2 Debugging via JTAG

JTAG (Joint Test Action Group) was an industry group formed in 1985 to develop a method to test circuit boards after manufacture. In1990 Intel released the first processor with JTAG interface - the 80486. Today JTAG is used as the primary mechanism for debugging embedded systems which may not have any other debug-capable communications channel. On most systems, JTAG-based debugging is available from the very first instruction after CPU reset, letting it assist with development of early boot software which runs before anything is set up. An in-circuit emulator (ICE), also called a "JTAG adapter", uses JTAG as the transport mechanism to access on-chip debug modules inside the target CPU. Those modules let software developer to debug the software of an embedded system directly at the machine instruction level when needed, or in high level language source code. Using JTAG, processor can be halted, single stepped, or let run freely, code or memory access breakpoints can be set. NXT has only two hardware break points available for debugging.

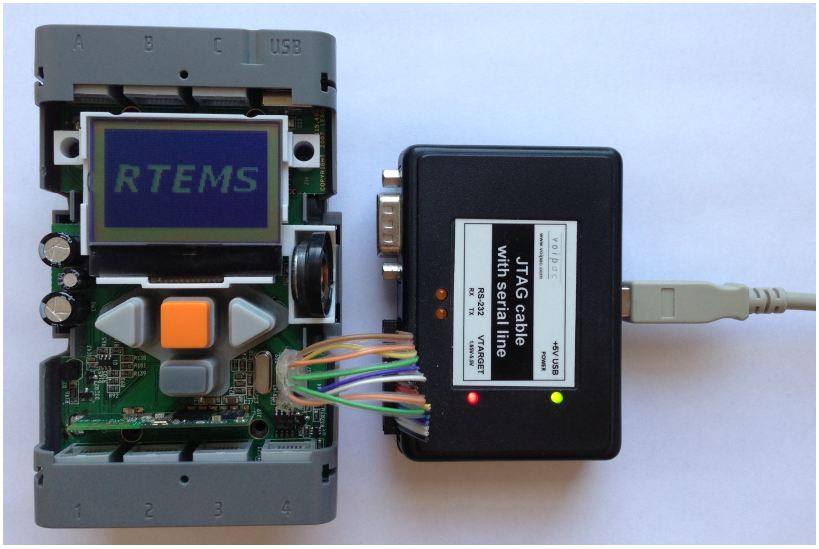


Figure 7.3: NXT connected via JTAG adapter

7.2.1 Connector soldering

NXT has JTAG capability, although it does not have JTAG connector soldered. Connector is needed to connect to the JTAG debug probe. To solder connector, NXT has to be opened and connector has to be soldered on the circuit board. Such actions invalid NXT warranty and you may damage your NXT in process. If you are fully aware of the risks, follow [16] and Lego NXT Hardware Developer kit [17] to create cable which will connect NXT and JTAG adapter.

To connect to the processor JTAG interface, hardware debugger – JTAG adapter is needed. Hardware debugger will be connected to the NXT JTAG connector and to the computer using USB. During porting RTEMS to NXT, VOIPAC JTAG 27M JTG 000 adapter was used.

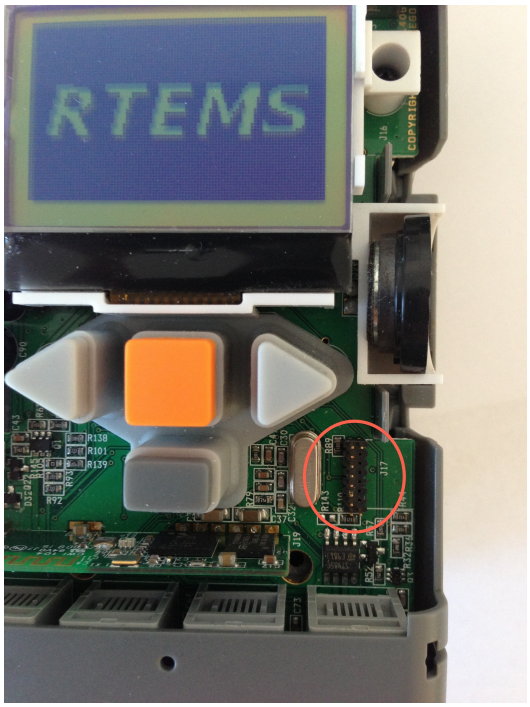


Figure 7.4: JTAG connector inside NXT

7.2.2 OpenOCD

Communication with the JTAG adapter is done via OpenOCD. OpenOCD (Open On-Chip Debugger) has been created by Dominic Rath as part of a diploma thesis at the University of Applied Sciences, FH-Augsburg. First, configuration file has to present. It contains information of JTAG Clock speed. Speed is set via parameter `adapter_khz 6000`. Configuration file `speed.cfg` will consists of just one line containing `adapter_khz 6000`.

To start OpenOCD server, execute command

```
~$ openocd -s /usr/share/openocd/
           -f interface/vpaclink.cfg
           -f board/atmel_at91sam7s-ek.cfg
           -f speed.cfg
```

It specifies JTAG adapter type and target board. at91sam7s-ek is an evaluation board, almost completely similar to NXT board, therefore it is the best available option.

Connection to the OpenOCD from the terminal is done via telnet. By default, OpenOCD expects connections at port 3333 and 4444. Port 3333 is destined for debugger, port 4444 is to be used from terminal. Following example shows how to upload NXT BIOS at the beginning of the FLASH memory using JTAG:

```
~$ telnet localhost 4444
>halt
>arm7_9 dcc_downloads enable
>arm7_9 fast_memory_access enable
>flash write_image erase unlock \
           /nxtOSEK/ECRobot/bios/nxt_bios_rom.rfw 0x00100000
>arm7_9 fast_memory_access disable
>reset init
>reset run
```

NXT BIOS will be flashed only once. During development, application will be flashed multiple times. Application must be stored at address `0x0010 8000`. When application is flashed, processor's program counter register (PC) is set to execute instruction from address `0x0010 8000` using command `step`.

Application can be flashed using following set of commands:

```
~$ telnet localhost 4444
>halt
>arm7_9 dcc_downloads enable
>arm7_9 fast_memory_access enable
>flash write_image erase unlock <PATH_TO_APPLICATION> 0x00108000
>arm7_9 fast_memory_access disable
>reset init
>step 0x00108000
>resume
```

Review of the most used commands for debugging NXT:

- **halt:** halts processor, stops command execution. Almost all commands can be used only if target (NXT) is halted.
- **reset <init|run>:** command reset processor to the state, as it was just turned on. Depending on the second parameter, it starts executing code or waits for next command
  - **init:** Sets processor to the state as it was just turned on and waits
  - **run:** Sets processor to the state as it was just turned on and starts executing code from the beginning of the memory
- **arm7_9 dcc_downloads <enable|disable>:** Enable the use of the debug communications channel (DCC) to write larger (>128 byte) amounts of memory.

- **arm7_9 fast_memory_access <enable|disable>:** Allow OpenOCD to read and write memory without checking completion of the operation.
- **flash write_image erase unlock *<path_to_application>* *<memory_address>*:** Flash application image into memory.
- **step *<address>*:** Set processor program counter register to address, or if address is omitted, execute one following instruction
- **resume:** Processor resumes normal execution
- **mdw *<address>* *<count>*:** Read count bytes from memory, starting at address.
- **bp *<address>* *<count>* hw:** Set hardware breakpoint into at memory address for count bytes. NXT has only 2 hardware breakpoints available at the same time.
- **rbp *<address>* :** Remove the breakpoint at memory address.
- **regs:** Print all processor registers.

7.2.3 Eclipse CDT

Eclipse CDT can be used as a programming IDE as well as a debugging environment. Eclipse can use GDB debugger in conjunction with OpenOCD server to debug NXT via JTAG interface.

To start using Eclipse, new project containing RTEMS on NXT sources should be created.

RTEMS on NXT is meant to be build in different directory than the one containing source code. To build RTEMS on NXT, custom build script is used. Set variable `BUILD_DIR` to the directory, where you want RTEMS on NXT to be built.

```
#!/bin/bash

BUILD_DIR=/rtems-build-directory

cd ${BUILD_DIR}
export PATH=/opt/rtems-4.11/bin:${PATH}

/usr/bin/make clean

/usr/bin/make --jobs=8 all
```

To be able to build RTEMS from Eclipse using `Run Build` button, new `External Tools Configuration` will have to be created.

Figure 7.5 shows Externals tools configuration tab:

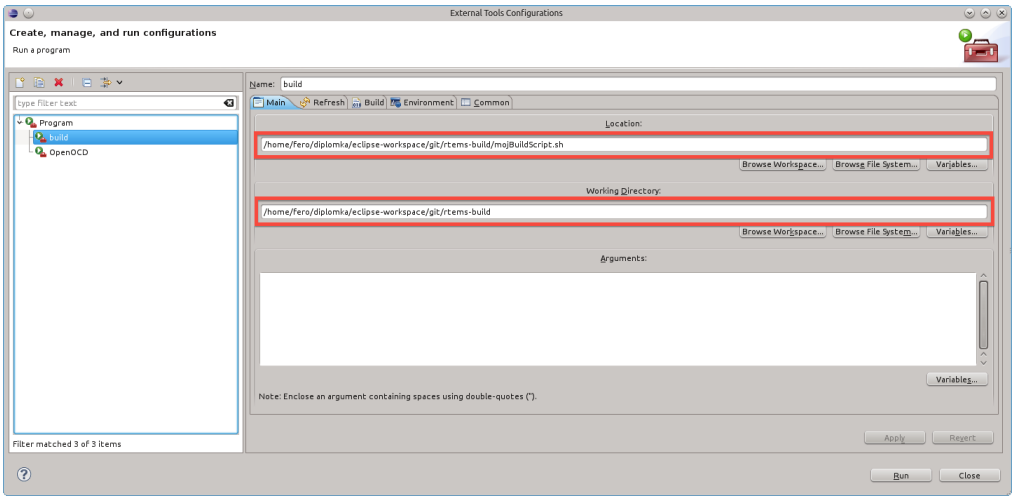


Figure 7.5: Eclipse debug configuration - External tools configuration

In `Location`, path to the build script must be entered. `Working directory` should also be specified.

When RTEMS is built with option *–enable-tests=samples*, all samples under directory `testsuites/samples` will be built. These applications can be modified, specific functionality can be added and later debugged.

To upload application into NXT, new debug configuration should be created. It has to be new configuration under `GDB Hardware Debugging`.

In the Main tab, to the field `C/C++ Application` fill in the full path to the created executable, that should be uploaded.

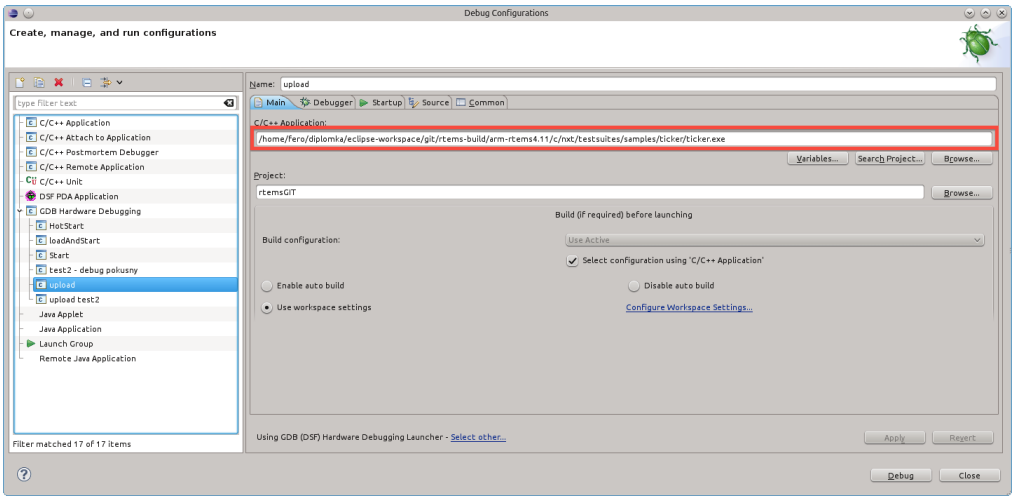


Figure 7.6: Eclipse debug configuration - Main tab

In the Debugger tab, to the field `GDB Command` fill path to the GDB compiled for RTEMS (`arm-rtems4.11-gdb`) and built application. It should look like following:

```
/opt/rtems-4.11/bin/arm-rtems4.11-gdb /rtems-build/ \
      arm-rtems4.11/c/nxt/testsuites/samples/ticker/ticker.exe
```

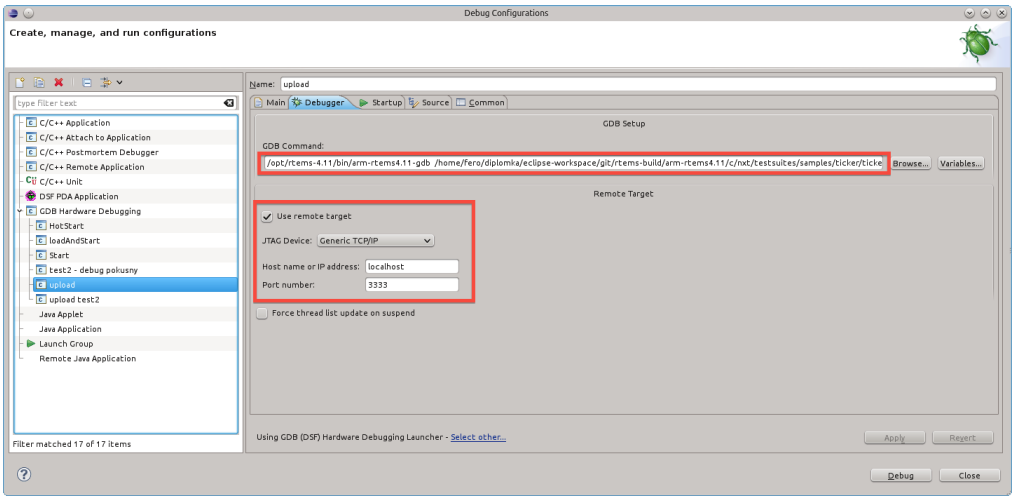


Figure 7.7: Eclipse debug configuration - Debugger tab

Make sure that `Use remote target` is checked, in the field `JTAG Device` Generic TCP/IP is chosen, `Host name or IP address` is localhost and `Port number` is 3333.

The last tab that needs to be changed is Startup.

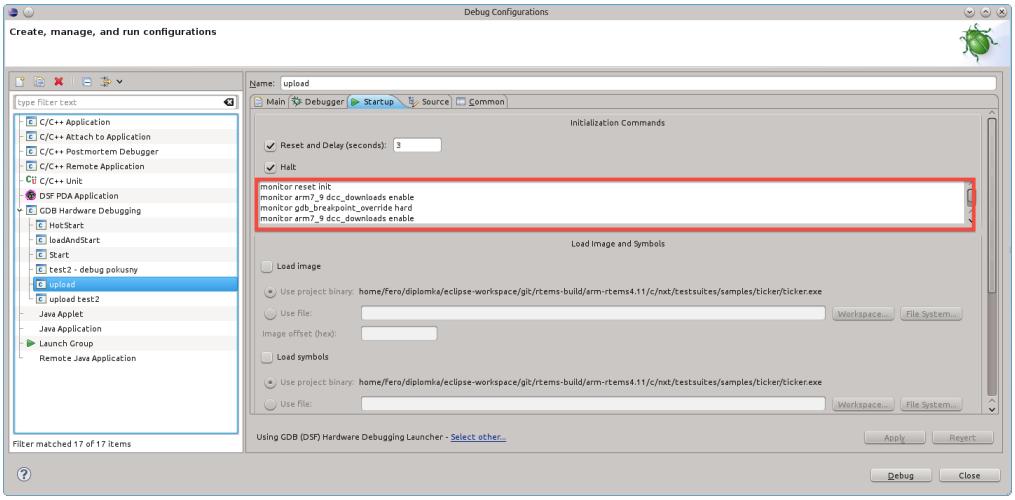


Figure 7.8: Eclipse debug configuration - Startup tab

This is the place, where actual commands for uploading will be set. Make sure that only `Reset and Delay` and `Halt` fields are checked and fill in following commands into marked area (set correct path to the .rom file, should be located right besides .exe file):

```
monitor reset init
monitor arm7_9 dcc_downloads enable
monitor gdb_breakpoint_override hard
monitor arm7_9 dcc_downloads enable
monitor arm7_9 fast_memory_access enable
monitor flash write_image erase unlock git/rtems-build/
    arm-rtems4.11/c/nxt/testsuites/samples/
```

```
    ticker/ticker.rom 0x00108000
monitor arm7_9 fast_memory_access disable
monitor reset init
monitor step 0x00108000
```

To start upload, connect JTAG adapter with turned on NXT and computer, start OpenOCD server with command mentioned in Section 7.2.2. Once connected, execute configuration.

To debug application, similar configuration has to be created. Tabs `Main` and `Debugger` will be exactly the same as in upload configuration. Difference is in `Startup` tab. It will contain following commands, which will set processor command register to the address `0x0010 8000`, where application is located in the Flash memory and set GDB to use hardware breakpoints:

```
monitor reset init
monitor arm7_9 dcc_downloads enable
monitor gdb_breakpoint_override hard
monitor step 0x00108000
```

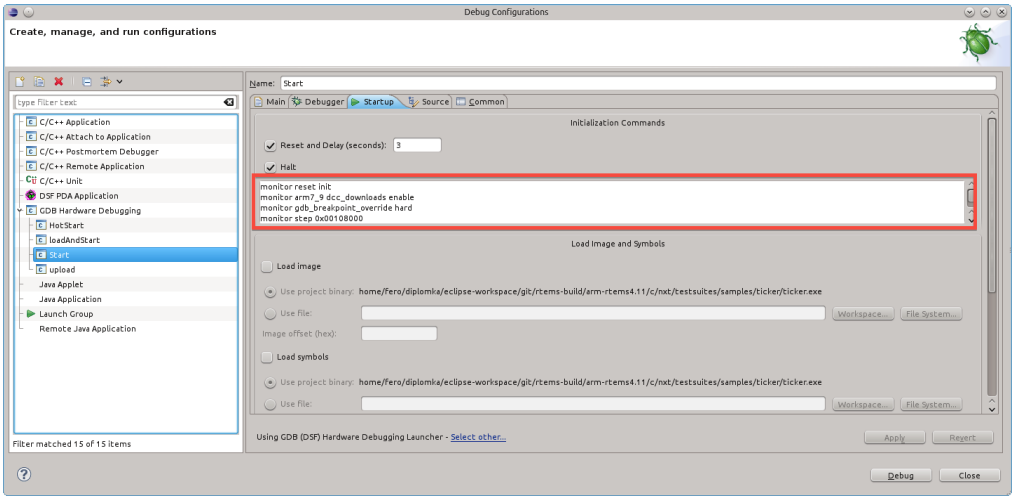


Figure 7.9: Eclipse debug configuration - Debugger startup commands

Once configuration is executed, debugger will be stopped in assembler file `start.S`. From this point, debugging can continue using standard Eclipse debugger commands. Because NXT has only two hardware breakpoints, make sure that only two breakpoints are set at the same time.

Chapter 8

Evaluation and case study

To be able to design and build RTEMS application, it is necessary to know the limitations of RTEMS on NXT. One of the apparent factors is a memory size. Application memory usage will be evaluated in Section 8.1. Memory usage with enabled POSIX API will be discussed in Section 8.1.1.

8.1 Memory usage

First of all, RTEMS image, that will be uploaded into Flash memory has to have less than 224Kbytes. It can be easily checked:

```
~$ ls -al o-optimize/semaphores.rom
-rwxrwxr-x 1 fero fero  108040 Jul 13 16:16 semaphores.rom
```

Since parts of the RTEMS must be located in SRAM memory, their size have to be less than 64Kbytes. SRAM memory has to accommodate following segments:

- `.ramvectors`: interrupt vectors place holder
- `.vector` interrupt vector table
- `.fast_text`: functions that resides in memory (e.g. function that writes into the Flash memory)
- `.data`: initialized global, static variables
- `.bss`: uninitialized data, variables
- `.work`: RTEMS Workspace and C Program Heap

Segments `.ramvectors`, `.vector` and `.fast_text` have fixed size. Size of `.data` and `.bss` segments is given according to used variables. Size of `.work` segment is determined as joint size of RTEMS Workspace and C Program Heap. RTEMS Workspace size is determined during application build, according to number of used tasks, structures, semaphores, etc. C Program Heap occupies the rest of the memory. They both serve as memory pools:

- **RTEMS Workspace:** memory pool used by RTEMS to allocate control structures like tasks, semaphores, task structures, system data structures.

- **C Program Heap:** memory pool from where `malloc` function allocates free memory.

To illustrate real size of application, example application `semaphore` will be used. It is simple application containing 2 tasks and one initialization task, they use semaphore for synchronization and print their priority on LCD when they run. Application also periodically allocate small size of memory using `malloc`. Size of segments can be determined after application is successfully compiled using

```
~$ arm-rtems4.11-size -A o-optimize/semaphores.exe
o-optimize/semaphores.exe  :
section              size      addr
.ramvectors            32   2097152
.start                692   1081344
.vector              1478   2097554
.fast_text            168   2099032
.text               93240   1082204
.init                  20   1175444
.fini                  16   1175464
.rodata              9943   1175480
.eh_frame               4   1185424
.ctors                  8   1185428
.dtors                  8   1185436
.jcr                    4   1185444
.data                3936   2099200
.bss                11856   2103136
.work               47696   2114992
...
```

To determine usage of RTEMS Workspace and C Program Heap, JTAG debugger is used to read actual values from running application. Actual statistics for RTEMS Workspace can be read from global variable `_Workspace_Area`:

```
(gdb) p _Workspace_Area
$1 = {free_list = {prev_size = 0,
           size_and_flag = 0,
           next = 0x206948,
           prev = 0x209dc8
           },
      page_size = 8,
      min_block_size = 16,
      area_begin = 2114992,
      area_end = 2137794,
      first_block = 0x2045b0,
      last_block = 0x209eb8,
      stats = {instance = 0,
           size = 22792,
           free_size = 4720,
           min_free_size = 240,
           free_blocks = 2,
```

```
            max_free_blocks = 2,
            used_blocks = 21,
            max_search = 1,
            allocs = 25,
            searches = 25,
            frees = 4,
            resizes = 0
            }
      }
(gdb)
```

Statistics for C Program Heap are stored in `RTEMS_Malloc_Area`:

```
(gdb) p RTEMS_Malloc_Area
$2 = {free_list = {prev_size = 0,
            size_and_flag = 0,
            next = 0x209f38,
            prev = 0x209f38},
      page_size = 8,
      min_block_size = 16,
      area_begin = 2137794,
      area_end = 2162688,
      first_block = 0x209ec8,
      last_block = 0x20fff8,
      stats = {instance = 1,
            size = 24880,
            free_size = 24768,
            min_free_size = 24768,
            free_blocks = 1,
            max_free_blocks = 1,
            used_blocks = 7,
            max_search = 1,
            allocs = 7,
            searches = 7,
            frees = 0,
            resizes = 0
            }
      }
(gdb)
```

If using JTAG debugger is not an option, statistics can also be displayed from application on LCD display or using `nxjconsoleviewer`. By reading values from global variables `_Workspace_Area` and `RTEMS_Malloc_Area`, programmer can obtain memory usage information, but he has to start with a small application and gradually add new functionality.

Total memory size is shown for each pool in variable `size`, free memory space is stored in variable `free_size`. Example application `semaphore` has about half of the memory available:

```
.work (47696) - _Workspace_Area(22792) - "mallocated space"(112)
```

Pool size and actual application needs are two different things. For example, size for task's stack is determined from number of tasks multiplied by RTEMS default stack size. If application creates task with two times bigger stack, application must also specify

```
#define CONFIGURE_EXTRA_TASK_STACKS (N * RTEMS_MINIMUM_STACK_SIZE)
```

for RTEMS to take it into memory requirements calculation. Adding new semaphore into example application raises memory usage only slightly:

```
size = 22920, free_size = 4720
```

Objects increase memory usage only slightly, stacks for tasks are the biggest memory consumers. Each task takes at least `RTEMS_MINIMUM_STACK_SIZE`, which is 4Kbytes. Therefore if you add 6 tasks into example `semaphore` application(8 tasks in total), it will not be able to allocate enough memory. If memory limitations are exceeded, RTEMS will be unable to start. As a fail safe, it will set flags for NXT BIOS to flash new application after restart.

Another example application is `posixMutex`. It is a simple application using pthreads, which lock and unlock on a shared mutex. Memory usage can again be obtained from `_Workspace_Area` and `RTEMS_Malloc_Area` variables.

Following table presents memory usage with number of created threads. If application uses more than 4 threads at a time, it will run out of memory.

Number	`_Workspace_Area`		`RTEMS_Malloc_Area`	
of threads	total size	free size	total size	free size
2	26248	208	21424	21424
3	35128	248	12544	12544
4	44016	304	3656	3656

Macro `#define CONFIGURE_UNIFIED_WORK_AREAS` will join RTEMS Workspace pool and C Program Heap pool into one memory area. It can help better memory utilization. If macro is defined, memory usage statistics can be found in `_Workspace_Area`. Using dynamic memory allocation - `malloc` in real time systems is not usual, therefore huge memory allocation is not expected.

8.1.1 POSIX memory requirements

RTEMS has an build option to include or exclude the POSIX API. When POSIX API is excluded, memory space can be saved. Example application `semaphore` is compiled with RTEMS, that was compiled with POSIX API (`--enable-posix`) and without POSIX API (`--disable-posix`).

Application size comparison:

With POSIX API	Without POSIX API
108064	103680

Application segments size comparison:

Segment	With POSIX API	Without POSIX API
.text	93264	89408
.data	3936	3860
.bss	11856	10096
.work	47696	49520

Application memory usage size comparison:

	With POSIX API		Without POSIX API	
Memory pool	Total size	Free size	Total size	Free size
_Workspace_Area	22920	4720	21880	4432
RTEMS_Malloc_Area	24752	24704	27616	27536

Size of application and its memory usage is only slightly bigger when POSIX API is included, it is up to programmer to decide whether POSIX API should be used.

Chapter 9

Related work

Lego provides different programming environments for NXT: NXT-G, ROBOLAB. There are also open-source community based environments, such as RobotC and Bricxcc. Open-source operating systems, such as leJOS NXJ and nxtOSEK, which support NXT platform also exist. They differ in development environment(graphical, textual), used language (C, C-based, Java), used firmware and in debugging capabilities.

In following chapters, summary of used firmware and debugging capabilities for selected systems will be provided. Drivers for peripherals are usually very similar, because they are hardware dependent. They differ only in some system specific parts, such as interrupt handler registration, but overall functionality remains unchanged. API for accessing peripherals differs according to used system.

9.1 NXT-G and ROBOLAB

They are both using standard Lego firmware, which has internal virtual machine for running applications. Applications are stored in special `.RXE` files. The NXT firmware consists of several modules, including those for sensors, motors, and a virtual machine. When the virtual machine runs a program, it reads the encoded `.RXE` file format from Flash memory and initializes a 32KB pool of RAM reserved for use by user programs. The `.RXE` file specifies the layout and default content of this pool. After the RAM pool is initialized, the program is considered active, or ready to run. Most program functionality involves modifying this pool of RAM and performing I/O actions based on the values stored in RAM.

`.RXE` bytecode application, that was converted from graphical representation to the text descriptions - bytecode, will is uploaded to the NXT. Bytecode is used by internal virtual machine. It provides access to the internal devices and sensors using built-in drivers. Firmware is also used as a tool to upload new applications. Lego provides open-source firmware, that can be useful source of information when porting for NXT.

9.1.1 NXT-G

NXT-G provides graphical environment for NXT programming, is bundled with NXT. Program is created by drag-and-dropping code blocks, represented by icons,

into flowchart diagrams. It was developed by National Instruments for Lego. It is primarily targeted to children or people with no programming experience and it requires Windows host.

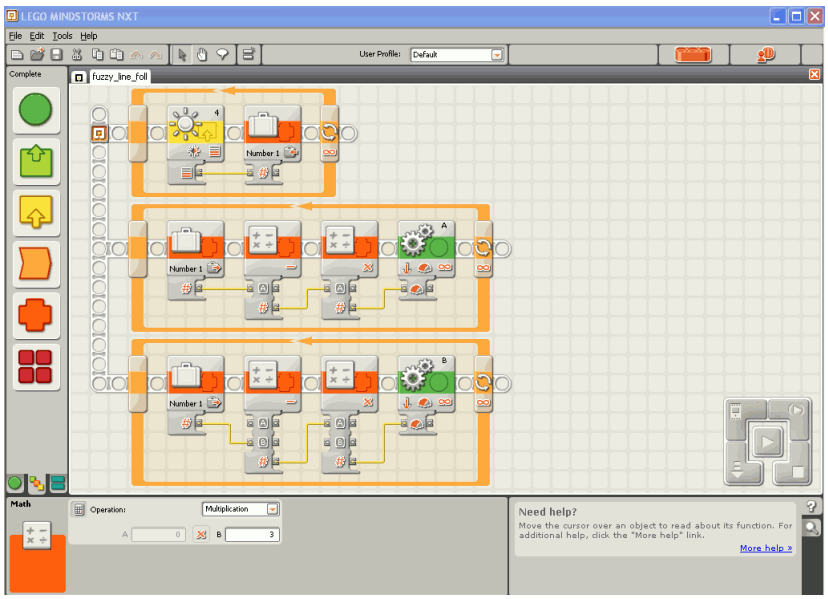


Figure 9.1: NXT-G application example, from [23]

9.1.2 ROBOLAB

It is another graphical environment for programming NXT. It was developped by Tufts University for Lego to by used in programming older version of Lego Mindstorms - RCX. It is good choice if you want to have graphical programming environment for both, older and newer Lego Mindstorms. ROBOLAB is using LabVIEW system as its core technology, was developed by National Instruments. Graphical environment, such as NXT-G or ROBOLAB are easy to use, very intuitive, but become tedious very quickly for experienced programmers.

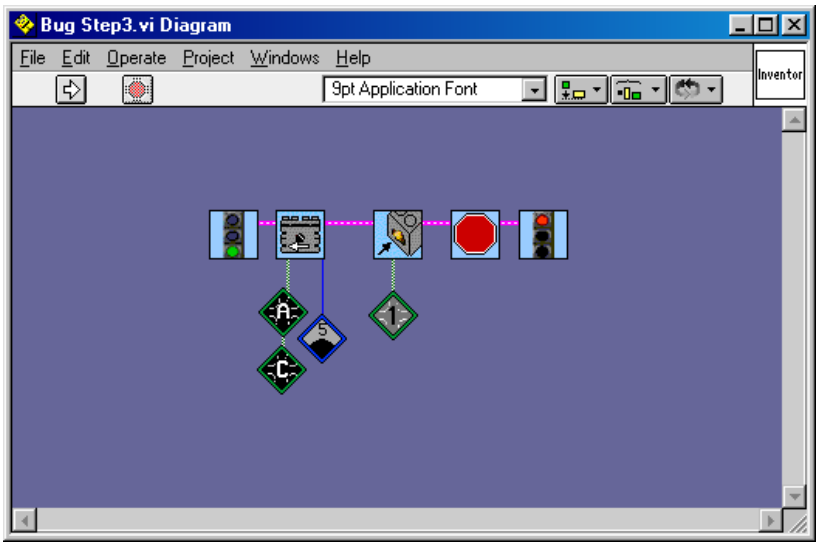


Figure 9.2: ROBOLAB application example, from [24]

9.2 Bricx Command Center

Bricx Command Center (BricxCC) is an integrated development IDE for Windows host. It can be used to program all Lego Mindstorms family bricks. NXT can be programmed using Not eXactly C (NXC) or Next Byte Codes (NBC) programming language. Next Byte Codes (NBC) is a simple open source language with an assembly language syntax, Not eXactly C (NXC) is an open-source language similar to C. BricxCC can use standard firmware, same as is used by NXT-G and ROBOLAB, or NBC/NXC firmware, which is standard NXT firmware with BricxCC specific enhancements.

BricxCC contains utilities to upload programs and firmware, view NXT screen at remote screen, convert sound files...

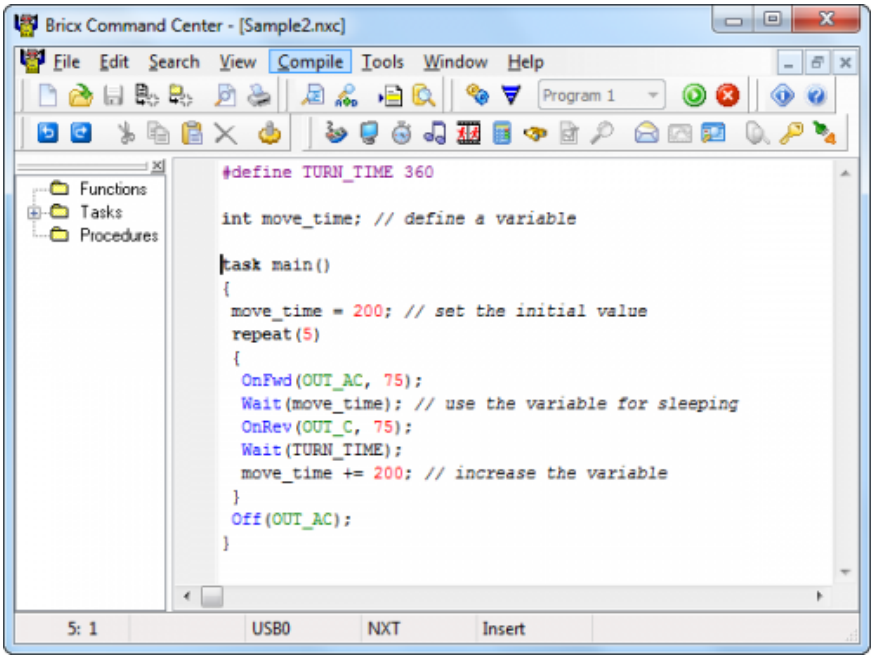


Figure 9.3: BricxCC application example, preview [25]

9.3 RobotC

RobotC is an integrated development environment with debugging capability and a programming language for educational robotics and competitions, it is targeted to students. It can be used to program Lego NXT, RCX, Arduino... It was developed by the Robotics Academy at Carnegie Mellon University. It has its own programming language that is based on C language and uses its own firmware. ROBOTC have an interactive, PC-based, real-time debugger.

Debugger is using internal virtual machine. When a program is run on the NXT, a virtual machine interpreter within the NXT firmware begins interpreting the intermediate instructions of the program. With some extra added features, such as suspend, resume, read memory, functionality of external hardware debugger can be simulated. Firmware also contains drivers for accessing peripher-

als. API to accessing peripherals is implemented via setting or reading specific variables, for example:

```
motor[motorB] = 75; // motor B is run at a power level of 75
bFloatDuringInactiveMotorPWM = false;  // brake mode for motore
```

Disadvantage is, that RobotC is a commercial product, therefore software license must be purchased. And it also requires Windows host.

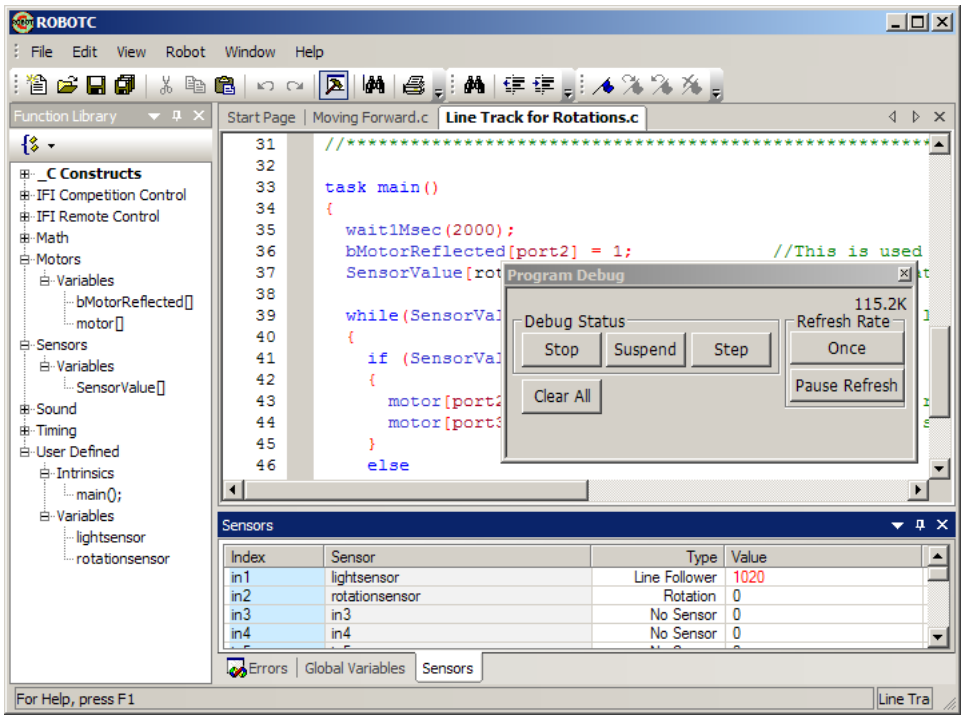


Figure 9.4: RobotC application example, from [26]

9.4 MATLAB and Simulink

MATLAB is a high-level programming language for numerical computing, data acquisition and analysis. It can be used to control LEGO NXT over a Bluetooth or USB connection, using the RWTH - Mindstorms NXT Toolbox, which is free and open-source.

Simulink is a MATLAB-based environment for modeling and simulating dynamic systems. Using Simulink, students can create algorithms for control systems and robotics applications. They can apply industry-proven techniques for Model-Based Design to verify that their algorithms work during simulation. They can automatically generate C code for those algorithms and upload the compiled code onto the NXT.

MATLAB and Simulink uses custom firmware, that must be flashed into NXT. Development environment provides interactive development and debugging. API to the peripherals is hidden from programmer, modeling of components representing devices is used instead of writing code to call methods.

MATLAB and Simulink Support for LEGO MINDSTORMS NXT programming is freely available, but MATLAB and Simulink themselves are licensed.

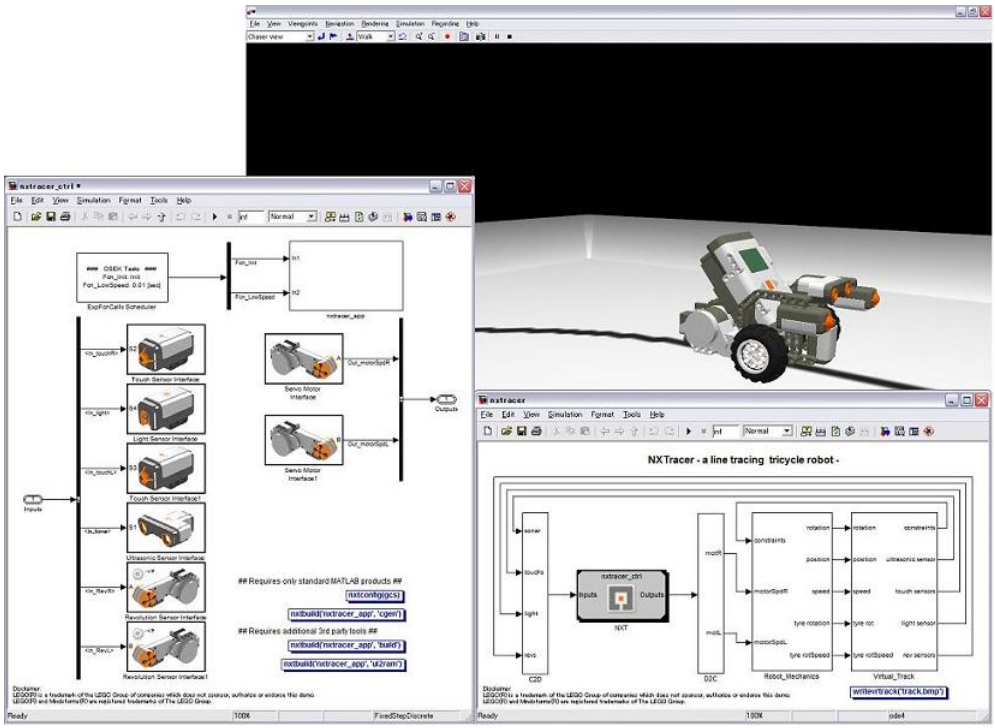


Figure 9.5: MATLAB application example, from [27]

9.5 leJOS NXJ

leJOS NXJ is Java based programming environment for NXT. It uses standard Java language and compiler but with a much smaller Class library. The standard Class library is too large for the NXT memory. leJOS includes small Java virtual machine.

As leJOS uses a firmware replacement, the new leJOS NXJ firmware must be flashed onto the NXT, and will replace the standard LEGO MINDSTORMS firmware. As programming environment, Eclipse or Netbeans can be used and leJOS NXJ has a custom plugins for both of them.

leJOS NXJ has also many utilities, such as console viewer, data viewer for viewing values from all sensors/motors and some of NXT internal devices at once, program uploader... Its debugging capabilities are limited to debug outputs to remote console.

leJOS NXJ provides API to the peripherals in form of classes. They contain methods for accessing and setting values to devices.

9.6 nxtOSEK

nxtOSEK is a real-time operating system. nxtOSEK consists of the I/O driver part of leJOS NXJ, TOPPERS OSEK real-time operating system used in automotive industry and a glue code for them. nxtOSEK enables Lego Mindstorms NXT robots to be programmed in ANSI-C/C++ and application can be build using GCC development environment. It also contains ECRobot API for accessing

I/O devices, which was chosen as device API in RTEMS. nxtOSEK uses custom firmware from leJOS. It has no debugging capabilities.

Since nxtOSEK kernel contains only Japanese comments and documentation, its quite difficult to understand. Regardless, it is a small system and if you are not interested in kernel itself, it can be used as a source of information for NXT hardware internal functionality. nxtOSEK was used as an inspiration for peripherals drivers implementation used in RTEMS.

Chapter 10

Conclusion and future work

The thesis has proved that it is possible to run RTEMS on Lego Mindstorms NXT platform. It provided implementation of Board Support Package for NXT, together with other necessary support routines, drivers for internal and external devices. Discussed necessary steps for porting to NXT platform, provided detailed description of most crucial hardware devices, such as clock generator and interrupt controller, memory mapping, evaluated possible approaches to interrupt handling.

For building and developing RTEMS applications, guidelines for installation of build tools were presented. They can be applied in environment where labs for Embedded and real time system course takes place.

Another important feature, that was presented is the usage of NXT BIOS for application management, which enables students to upload nxtOSEK application and RTEMS application the same way. That simplifies demands on working environment setting. Presented implementation was able to overcome constrains that were imposed by usage of NXT BIOS.

The work identified means for debugging RTEMS application. For debugging RTEMS application without installed JTAG connector, implementation provided methods to display debug prints on the remote computer screen. For low-level debugging, detailed directions for usage of JTAG hardware debugger in conjunction with Eclipse IDE settings were provided.

Possibility to follow up with extension of this thesis presented itself with implementation of a debug output. As some of the different development platforms for NXT, such as RobotC, use a virtual machine as an intermediate between application and hardware, it would be possible to use such intermediary that could have added functionality to provide more extensive debugging ability.

Bibliography

[1] Buttazzo, G. C.: *Hard Real-Time Computing Systems: Predictable Scheduling, Algorithms and Applications*. Springer, 2005.

[2] Lego NXT Robot Challenge page: http://bradfordschools.net/blog/lac/2012/06/28/lego-nxt-robot-challenge/

[3] nxtOSEK C API Reference page: http://lejos-osek.sourceforge.net/ECRobot_c_api_frame.htm

[4] ATMEL AT91SAM features: http://www.atmel.com/images/doc6175.pdf

[5] LEGO MINDSTORMS NXT wiki page: http://en.wikipedia.org/wiki/Lego_Mindstorms_NXT

[6] NxtOSEK home page: http://lejos-osek.sourceforge.net

[7] leJOS NXJ home page: http://lejos.sourceforge.net

[8] RTEMS_Eclipse_Plug-in wiki page: http://www.rtems.org/wiki/index.php/RTEMS_Eclipse_Plug-in

[9] LEGO MINDSTORMS home page: http://mindstorms.lego.com

[10] RTEMS Real Time Operating System page: http://www.rtems.org

[11] LibNXT page: http://code.google.com/p/libnxt/

[12] LEGO MINDSTORMS NXT Support from MATLAB page: http://www.mathworks.com/hardware-support/lego-mindstorms-matlab.html

[13] LEGO MINDSTORMS NXT Support from Simulink page: http://www.mathworks.com/hardware-support/lego-mindstorms-simulink.html

[14] nxtOSEK Installation in Linux page: http://lejos-osek.sourceforge.net/installation_linux.htm

[15] Upload a nxtOSEK program by using the NXT BIOS page: http://lejos-osek.sourceforge.net/howtoupload.htm#UploadtoFlash

[16] Installing the JTAG connector page: http://www.iar.com/Global/Campaigns/LEGO%20Mindstorms/QSSOLDER.pdf

[17] Mindstorms support page: http://mindstorms.lego.com/en-us/support/files/default.aspx

[18] RTEMS sources page: http://www.rtems.org/ftp/pub/rtems/SOURCES/4.11/

[19] ARM7TDMI Technical reference manual: http://www.atmel.com/Images/DDI0029G_7TDMI_R3_trm.pdf

[20] GNU ARM toolchain for Cygwin, Linux and MacOS page: http://www.gnuarm.com

[21] YAGARTO home page: http://www.yagarto.org

[22] CrossWorks for ARM home page: http://www.rowley.co.uk/arm/

[23] DTU Robocup page: http://nxtgcc.sourceforge.net/wiki/DTU_Robocup

[24] ROBOLAB application preview: http://www-lehre.inf.uos.de/~fsjaetzo/rit00/images/robolab.png

[25] Bricx Command Center application preview: http://dasl.mem.drexel.edu/~ducNguyen/wp-content/uploads/2011/06/BricxCC_Panel-500x375.png

[26] ROBOTC application preview: http://www.robotc.net/files/robotcforvex.png

[27] What is Embedded Coder Robot NXT? page: http://lejos-osek.sourceforge.net/ECRobot_nxt.htm

[28] NXT Programming Software page: http://www.teamhassenplug.org/NXT/NXTSoftware.html

[29] LEGO MINDSTORMS TUTORIAL http://nxtgcc.sourceforge.net/EMSOFT-2009-Mindstorms-Tutorial.pdf

[30] RTEMS Development Hosts page: http://www.rtems.org/wiki/index.php/RTEMS_Development_Hosts

[31] Swan, Dick: *Programming Solutions for the LEGO Mindstorms NXT - Which approach is best for you?*. http://www.kirp.chtf.stuba.sk/moodle/pluginfile.php/46108/mod_resource/content/2/Programming%20Solutions%20for%20the%20LEGO%20Mindstorms%20NXT.pdf

Appendix A

Contents of the enclosed DVD-ROM

Enclosed DVD containing this thesis in digital form together with source code for RTEMS including BSP package, routines and tools needed for RTEMS support of NXT.

Structure of enclosed DVD:

- *ecrobot_api*: Directory containing ECRobot API web page.
  - *ecrobotAPI_files*: ECRobot API documentation files.
  - *ecrobotAPI.html*: ECRobot API documentation index file.
- *master_thesis.pdf*: Master's thesis in PDF format.
- *sources*: Directory containing RTEMS sources.
  - *rtems-sources.tar.gz*: Compressed RTEMS sources.
- *readme.txt*: A description of the contents of the enclosed DVD.
- *sampleApplications*: Directory containing sample applications.
- *uploader*: Directory containing application uploader.
  - *src*: Source files for uploader.
  - *uploader*: Scripts for executing uploader.
  - *readme.txt*: Simple instructions for running uploader.

Appendix B

Sample application: Read color sensor value crossing the line

Simple application that demonstrates displaying values on remote computer using `nxjconsoleviewer`, work with sensors, motors and will present multiple approaches to task scheduling. Its purpose is to gain values that are return from color sensor when robot is crossing the line. This values can be further used in line following robot example.

It is expected that RTEMS is already installed, see Chapter 6. Entire application sources are enclosed on DVD in directory `samples/readLineValues`. It consists of multiple files:

- `Makefile`: contains information necessary to build application
- `init.c`: Represents declaration and initialization of tasks and sensors. It will create tree separate tasks for reading values, displaying values and the last one for operating motors.
- `system.h`: contains configuration information for RTEMS, it is responsible for specifying which modules will be compiled with application and also includes RTEMS, NXT and ECRobot header files.
- `tasks.c`: contains implementation of individual tasks. Tasks use different scheduling approaches, such as relative delay, rate monotonic scheduling and absolute timing.

In the following text, the entire process of building, uploading and reading values will be described step by step.

Before compilation, it is necessary to export variable containing path to the location where RTEMS was built. It can be done via exporting variable in console, using following command:

```
~$ export RTEMS_MAKEFILE_PATH=/path/to/RTEMS
```

Once path to RTEMS is set, application will be built executing

```
~$ make
```

Application will be built in `o-optimize` directory. Three files will be created: `readLineValues.exe`, `readLineValues.num` and `readLineValues.rom`. After successful built, application can be uploaded into NXT. NXT must have NXT BIOS installed and NXT must be in application flash mode. To get there, press middle orange button with left arrow button simultaneously from running application.

NXT must be connected to the computer via USB cable. To upload built file, from directory where source files are located, execute

```
~$ /path/to/uploader.sh o-optimize/readLineValues.rom
```

`uploader.sh` might be necessary to run with `sudo` command, because root privileges to access NXT over USB may be required.

Preview of the NXT LCD display in update mode, during uploading and when uploading is finished is shown in Figure 6.1, Section 5.11.

To exit upload application mode, press small gray rectangle button. Application `readLineValues` uses HiTechnic color sensor, touch sensor and two motors. They must be plugged in the following manner: color sensor must be plugged in NXT Port 1, touch sensor will be in NXT Port 2. Motors should occupy NXT PORT A and B. Assembled robot should look like the one in the following Figure B.1:

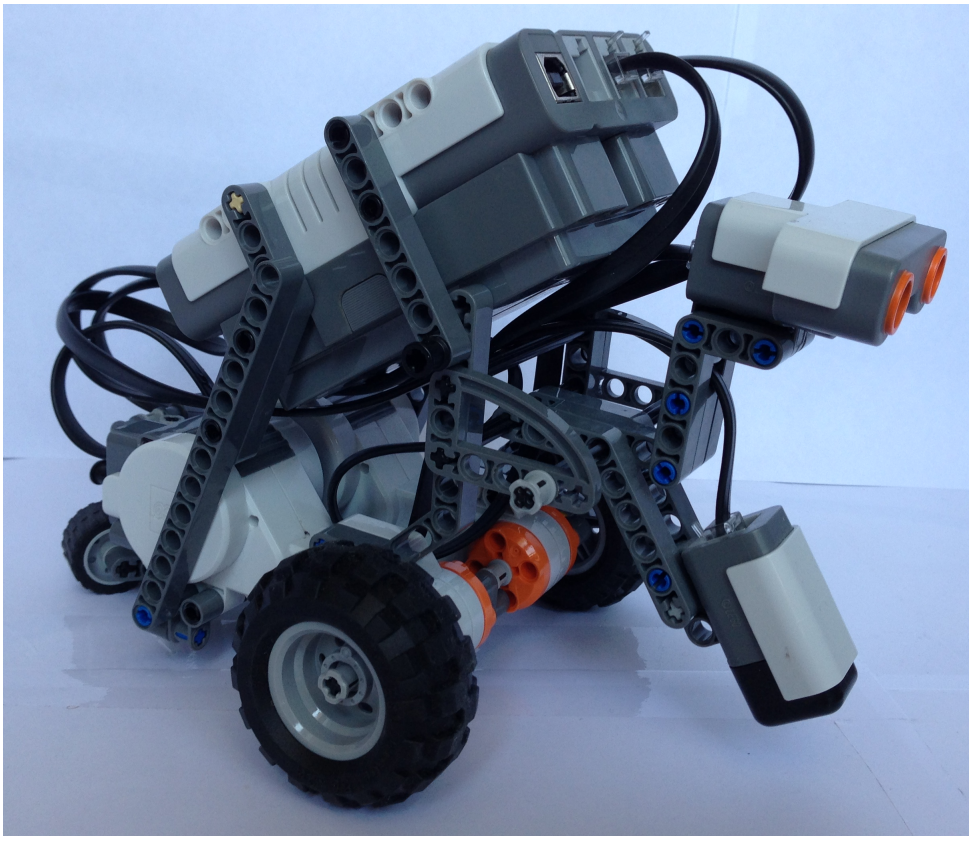


Figure B.1: NXT configuration for reading line values

With assembled robot, press square orange button. NXT startup screen will be presented. It is the time when robot will be connected to the remote console implemented by `nxjconsoleviewer` application. On remote computer, run `nxjconsoleviewer` application:

```
~$ /path/to/nxjconsoleviewer
```

It might be necessary to use `sudo` command to gain necessary privileges. To connect to the NXT, insert NXT address displayed on the LCD screen without two trailing zeros into Addr field. Connection is established when "`BT Connected`" string is shown at LCD screen and `nxjconsoleviewer` application shows **Status: Connected to NXT** in bottom left corner.

At this point, NXT should be placed in front of the line. When right arrow button is pressed, NXT starts to move forward and it will send color sensor values into remote application. NXT sends comma separated values in format:

```
number of ms from start, red value, green value, blue value
```

Once NXT crosses the line, it can be turned off using small gray rectangle. Received values should look similar to values in the following picture:

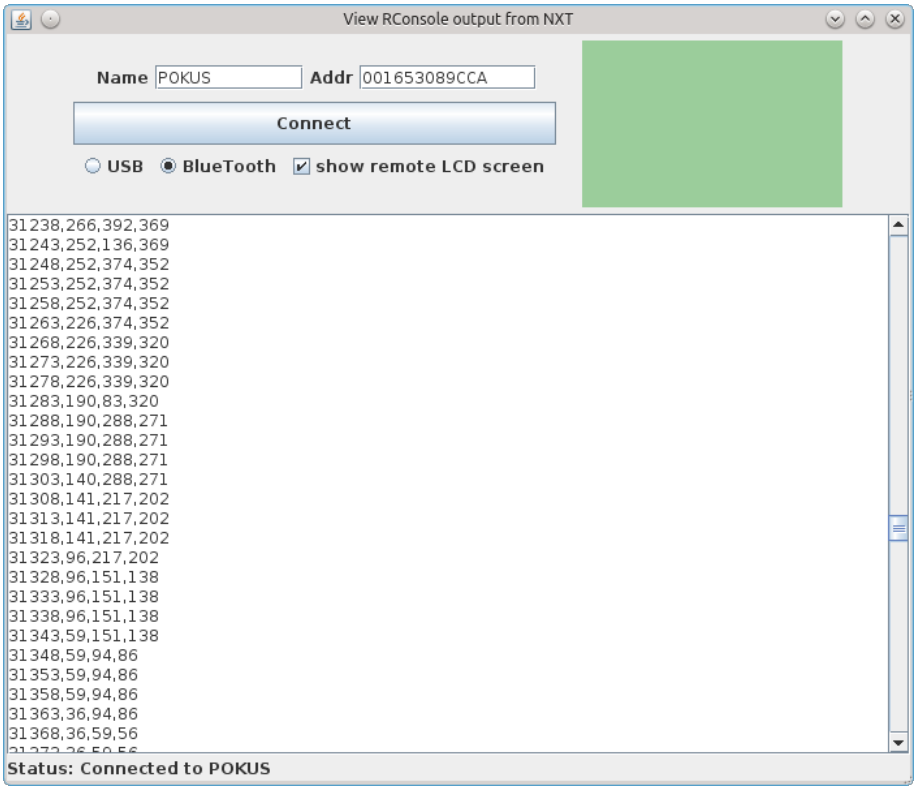


Figure B.2: Read color sensor value output in nxjconsoleviewer application

To shutdown robot and upload new application, hold left arrow button and orange square button simultaneously for a second.

Appendix C

Sample application: Line following robot

Sample line following application will be produced using RTEMS plugin for Eclipse CDT. It will show how to set up, create, build and upload RTEMS application into NXT.

It is assumed that Eclipse CDT is already installed together with RTEMS CDT Support plugin. Installation instruction can be found here [8].

In Eclipse, create new C project, using `File > New > C Project`. Choose `RTEMS Executable` and set project name.

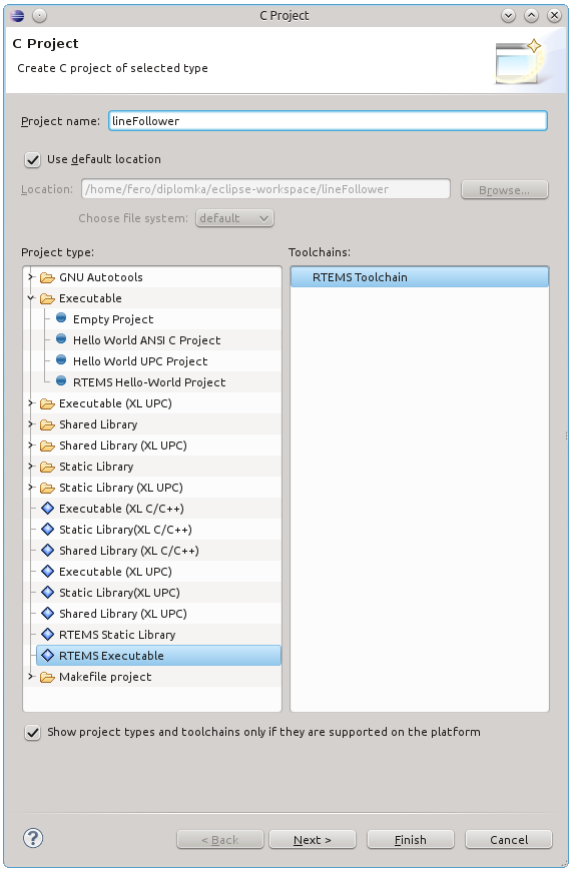


Figure C.1: Creating RTEMS project

Press `Next >` button two times, until RTEMS Setup step is displayed. Here, Base path to RTEMS should be set, according to path where RTEMS tools were installed. It is the path stored in variable `PREFIX`, mentioned in Section 6.1.1. BSP path is path to the directory where the NXT Board Support Package was installed. It is the `rtems-build` directory as mentioned in Section 6.2 with appended `arm-rtems4.11/nxt`.

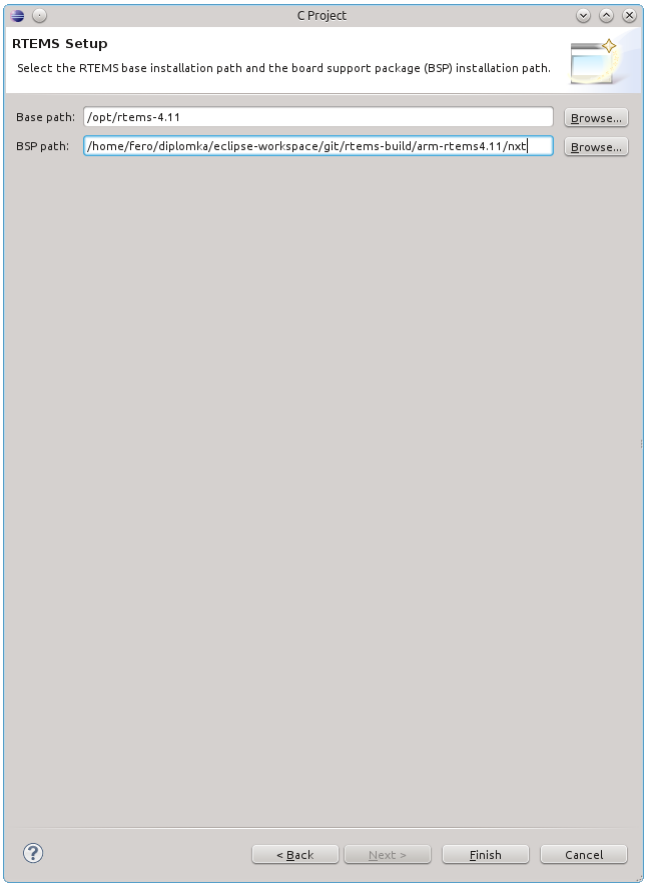


Figure C.2: RTEMS project setup

Hit `Finish` and new project is created. Under this project, copy contents of the `samples/lineFollower/` folder, that is enclosed on the DVD, into this newly created project. Copy of the `Makefile` file is not necessary.

RTEMS Eclipse plugin does not make application images, that could be uploaded into NXT. It is necessary to add post-build step to create this image. It can be set in `Project > Properties > Settings`. Assuming that RTEMS base path, which was set when project was created is `/opt/rtems-4.11`, command should be following:

```
/opt/rtems-4.11/arm-rtems4.11/bin/objcopy
     -O binary ${ProjName}.exe ${ProjName}.rom
```

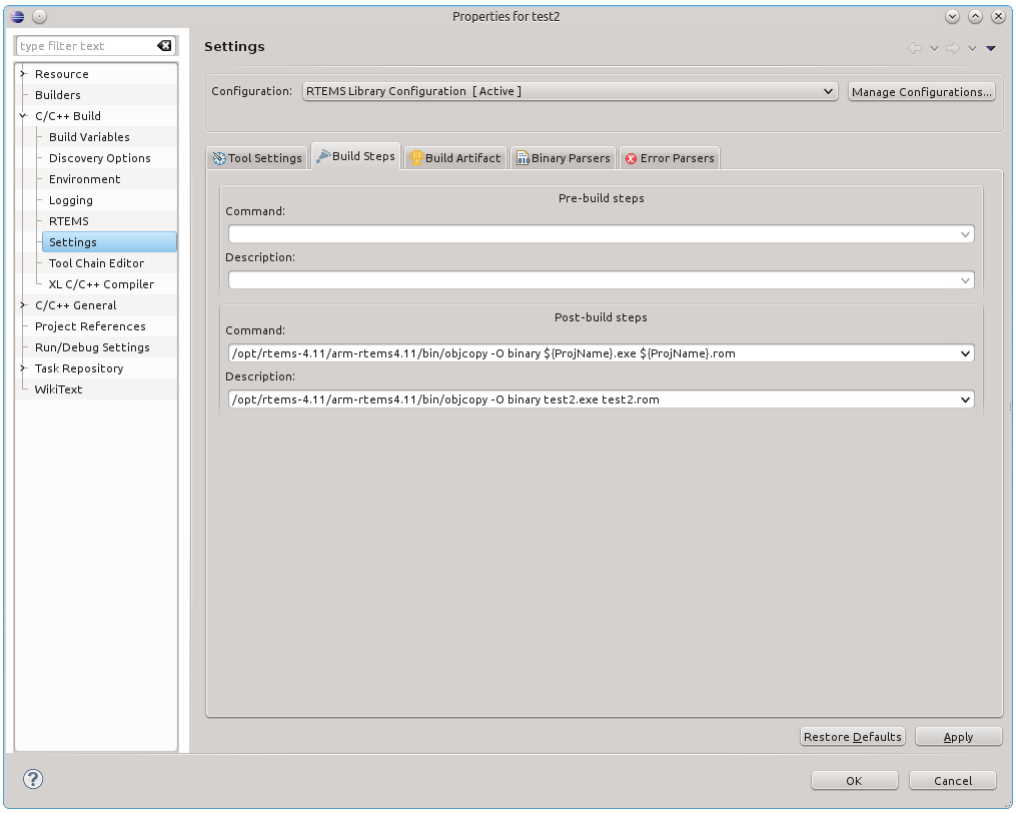


Figure C.3: RTEMS build settings - Build steps

In Build Artifact tab, make sure that field Artifact type is set to `Executable`

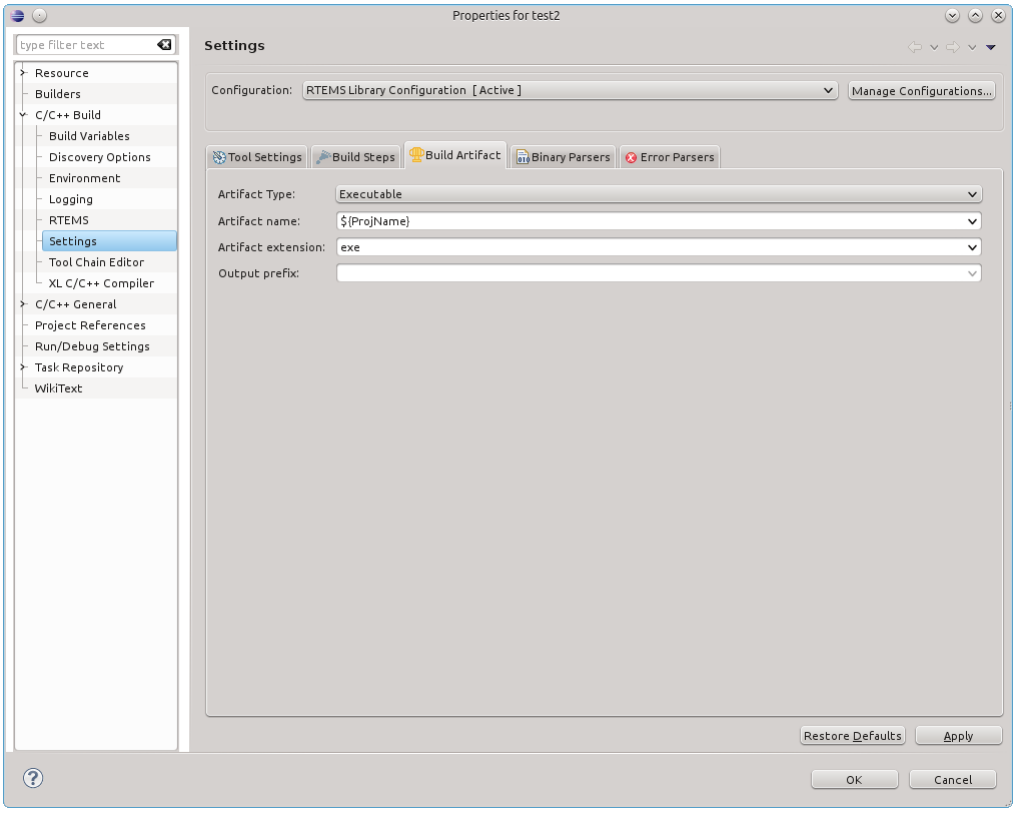


Figure C.4: RTEMS build settings - Build Artifact

Application can be built pressing `Ctrl+B`.

After application is successfully built, to upload application to NXT, newly build application can be uploaded using enclosed uploader:

```
~$ /path/to/uploader.sh lineFollower.rom
```

or new `Run build` configuration for calling uploader directly from Eclipse can be created.