



# LT200



## User Manual C/C++ Software Development Kit

P DOC LT200 002 E - V02



Leroy automation  
Boulevard du Libre échange  
31650 Saint Orens / Toulouse France  
Tel : +33 (0) 5 62 24 05 50 Fax : +33 (0) 5 62 24 05 55  
e-mail : [info@leroy-autom.com](mailto:info@leroy-autom.com)  
web site : [www.leroy-automation.com](http://www.leroy-automation.com)





## **Introduction**

Congratulations on your purchase of a LT200 automated device.

LT200 is a family of automated products with two available versions:

- LT200 programmable in C on Linux, using the Eclipse IDE.
- LT200 programmable in the IEC 61131-3 languages using the ICS Triplex ISaGRAF software workshop

The hardware setup is explained in the wiring manual that is available on our website.

## **Prerequisites**

Developing applications for the LT200 platform requires advanced knowledge of C/C++ programming and the Linux OS.

The hardware setup for the LT200 requires electrical and industrial automation skills.

Required hardware:

- A development PC running Windows XP or Linux, equipped with a serial port and an Ethernet card
- An RS232 crossover cable between the development PC and the LT200
- An Ethernet connection between the development PC and the LT200

## **Version**

This documentation describes the features available in the BSP LT200 V0.8

## **Proprietary Statements**

LT200 is a trademark of Leroy Automatique Industrielle.

The LT200 embedded Linux software is under a GNU/GPL license.

ISaGRAF is a trademark of ICS Triplex.

Windows XP is a trademark of Microsoft Corporation.

Leroy Automatique Industrielle develops and improves its products on a regular basis. The information contained in this documentation may change without notice and does not represent any commitment on behalf of the company. This manual may not be duplicated in any form whatsoever without the approval of Leroy Automatique Industrielle.

## **Contact**

 Leroy Automatique Industrielle  
Boulevard du Libre Echange  
31650 SAINT-ORENS

 33 (0) 5.62.24.05.50

 33 (0) 5.62.24.05.55

 [\*\*http://www.leroy-automation.com\*\*](http://www.leroy-automation.com)  
technical support:

 33 (0) 5.62.24.05.46

 [\*\*mailto:support@leroy-autom.com\*\*](mailto:support@leroy-autom.com)



# Table of Contents

Chapter 1. <b>General Overview</b> .....	<b>1</b>	<i>Installing and Configuring the Development Workstation</i> .....	17
<i>Introduction</i> .....	1	<i>LT200 Operating Modes</i> .....	18
<i>Physical Resources for the LT200</i> .....	1	<i>Development Environment</i> .....	21
<i>Overview of the LT200 BSP</i> .....	2	<i>Debugging in Application Mode</i> .....	22
<i>LT200 System Startup</i> .....	2	<i>Reloading the u-boot File</i> .....	23
<i>Structure of the Embedded Linux File     System</i> .....	3	<i>Reloading the uImage, initrd, and jffs2     Files</i> .....	23
<i>The LT200's file system has a specific     structure.</i> .....	3	<i>Library Modifications</i> .....	24
<i>Description of the Structure in Flash     Memory</i> .....	3	<i>Adding a Driver</i> .....	24
Chapter 2. <b>Getting Started in     Windows XP</b> .....	<b>5</b>	Chapter 5. <b>LT200 Configuration</b> .....	<b>25</b>
<i>Introduction</i> .....	5	<i>Introduction</i> .....	25
<i>Installing the CodeSourcery Open     Compilation Chain</i> .....	5	<i>Ethernet Configuration and Daemon     Activation</i> .....	25
<i>Installing Eclipse</i> .....	6	<i>Daemon Configuration</i> .....	25
<i>Importing an Existing Project into Eclipse</i>	6	<i>DHCP Daemon</i> .....	26
<i>Creating a New Project</i> .....	7	<i>SNMP Daemon</i> .....	27
<i>Compiling</i> .....	7	<i>FTP Daemon</i> .....	27
<i>Loading</i> .....	8	<i>HTTP Daemon</i> .....	28
<i>Debugging an Executable</i> .....	8	Chapter 6. <b>Programming Your     Application</b> .....	<b>29</b>
Chapter 3. <b>Getting Started in Linux</b> .	<b>10</b>	<i>Introduction</i> .....	29
<i>Introduction</i> .....	10	<i>User Applications: Starting and Stopping</i>	30
<i>Installing and Configuring the     Development Workstation</i> .....	10	<i>Managing Input/Output Cards</i> .....	33
<i>Installing and Configuring the Eclipse IDE</i>	11	<i>Watchdog Management</i> .....	37
<i>Development Environment</i> .....	12	<i>Using the Millisecond Counter</i> .....	38
<i>Importing an Existing Project into Eclipse</i>	13	<i>Managing the LEDs on the CPU Card</i> .....	39
<i>Creating a New Project</i> .....	13	<i>Modbus Protocol</i> .....	40
<i>Compiling</i> .....	13	<i>Using the Modbus Slave Protocol</i> .....	42
<i>Loading your Executable</i> .....	14	<i>Using the Modbus Master Protocol</i> .....	43
<i>Debugging</i> .....	15	<i>Modifying the LT200's IP Settings from a     Modbus Serial Master</i> .....	45
Chapter 4. <b>Advanced Linux     Programming</b> .....	<b>17</b>	Chapter 7. <b>Main Terminal Commands</b>	<b>46</b>
<i>Introduction</i> .....	17	Chapter 8. <b>GLOSSARY</b> .....	<b>47</b>
		Chapter 9. <b>INDEX</b> .....	<b>48</b>



# General Overview

## Introduction

The LT200 is based on a Linux 2.6.12 kernel.

It can therefore be programmed using any distribution of Linux. The LT200 kernel can be regenerated, if needed.

If the kernel does not need to be regenerated, LT200 applications can be developed on Windows XP using the Eclipse IDE and Window libraries for ARM processors.

Installing and configuring the Eclipse workshop for use with Windows and Linux is described in Chapters 2 and 3, respectively.

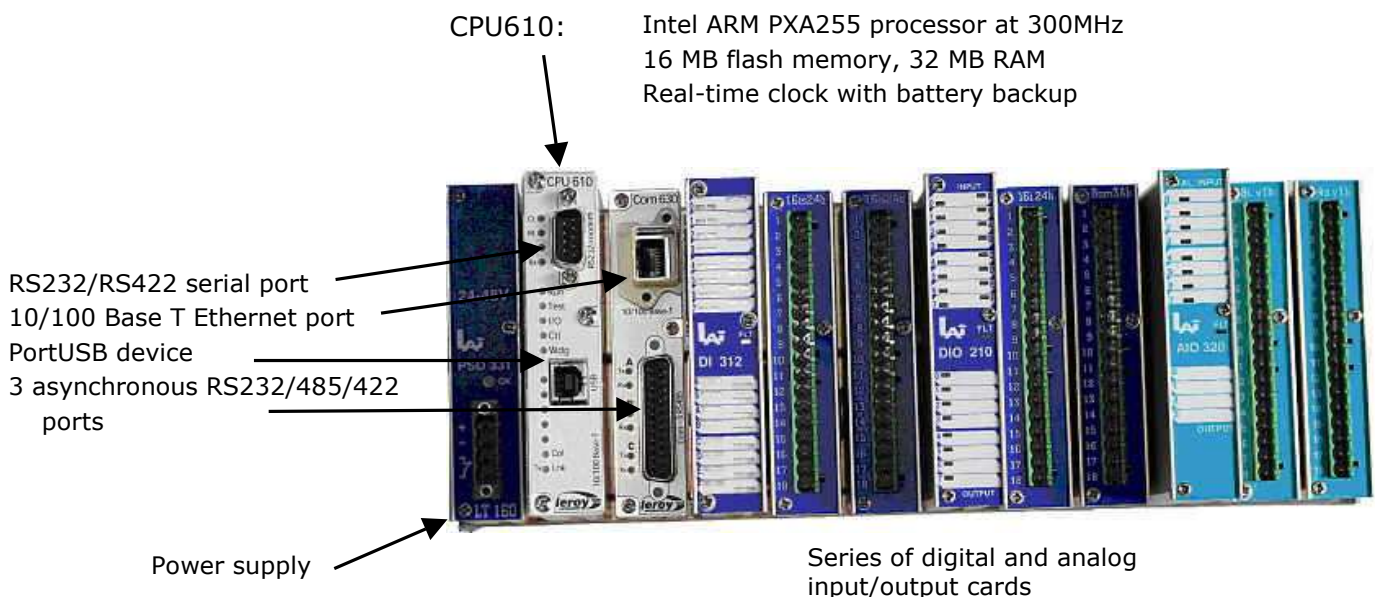
Creating a project, programming using Leroy libraries (memory, input/output drivers, communication functions, etc.), and compiling are all detailed in Chapter 6.

This chapter covers the following:

- Physical resources for the LT200
- Overview of the LT200 BSP
- Starting up the LT200 system
- Structure of the embedded Linux file system
- Description of the structure in Flash memory

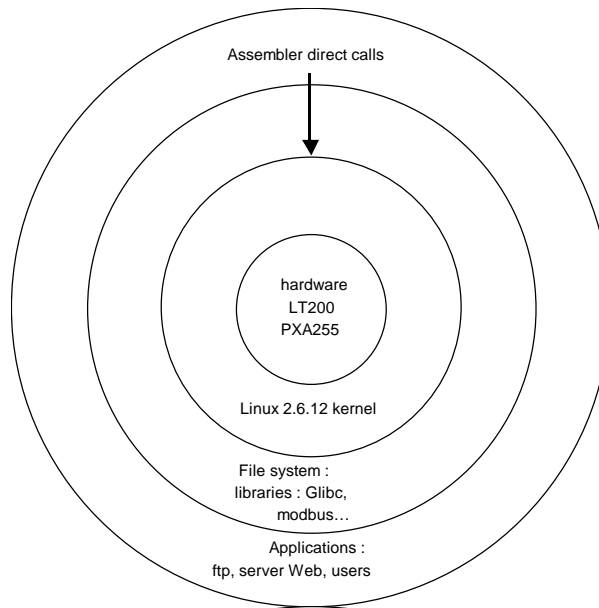
## Physical Resources for the LT200

LT200 is a physical database. As a result, the application must be executed and use available physical resources:



## Overview of the LT200 BSP

The LT200 BSP (Board Support Package) is an in-house Linux distribution. Its structure can be represented as follows:



The hardware consists of the processor, RAM, flash memory, and the following devices:

- External communication: Ethernet, asynchronous serial, USB
- Input and output cards – digital and analog
- Real-time clock

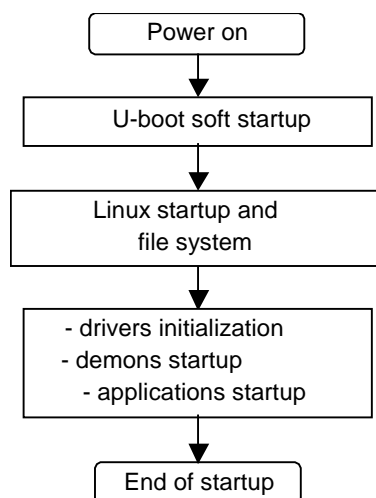
The Linux kernel and file system are the main component. This is the only interface between the system and hardware. It is primarily responsible for task management, memory management, and device monitoring.

The libraries interface with the applications that launch automatically on startup.

## LT200 System Startup

After the system powers on, the first software executed is U-Boot, which initializes the components on the CPU card (microprocessor, clocks, RAM, flash memory, Ethernet component, etc.) and launches Linux in RAM.

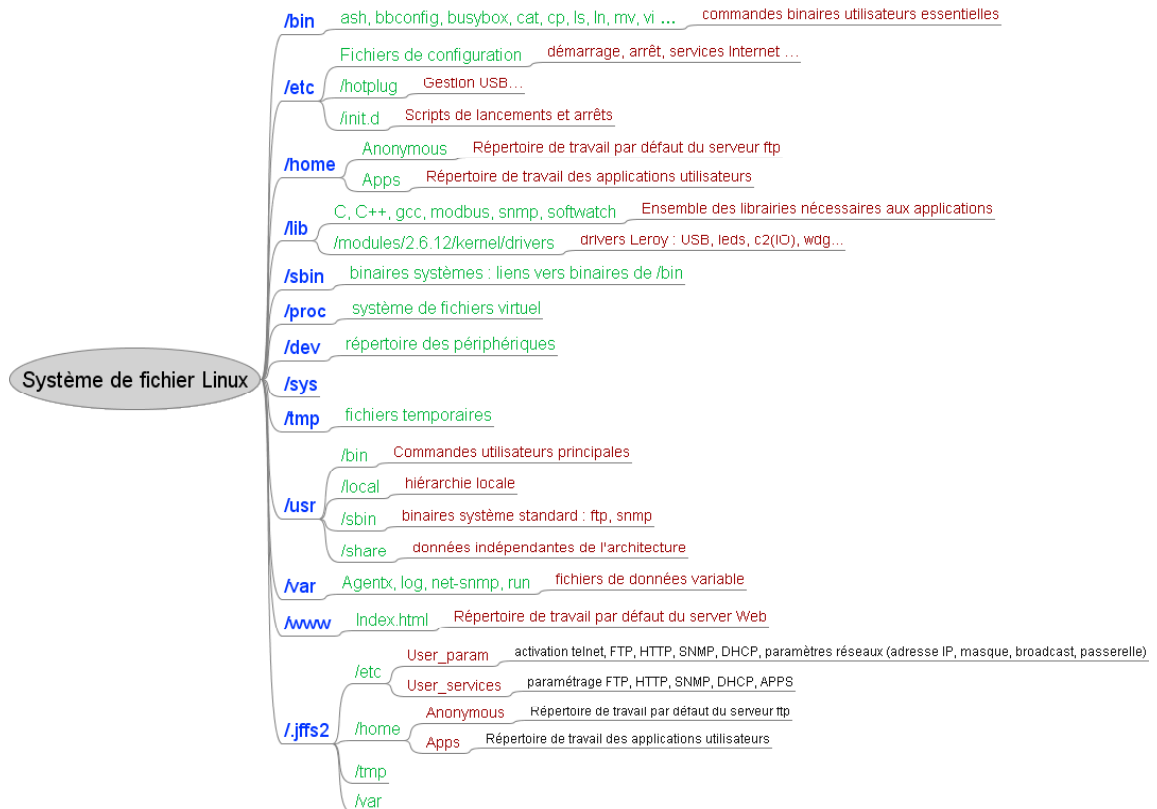
The drivers are then initialized, and the daemons and applications are launched from the file `/etc/sysinit`.



## Structure of the Embedded Linux File System

The LT200's file system has a specific structure.

The main commands used by the Linux system are incorporated into symbolic links referring to the Busybox application. This application provides a set of simplified system commands. The appendix lists the main commands.



We provide the user with all of the resources needed to modify the Linux kernel or the file system.

Under current use, without modifying Linux or the file system, the following directories primarily affect the user:

- /home/anonymous: default directory for the ftp protocol
- /home/apps: default directory for automatically launching applications
- /.jffs2/etc: directory for storing LT200 configuration files

By default, there is a web page available for modifying the LT200's Ethernet settings.

## Description of the Structure in Flash Memory

The LT200 has four key files in flash memory:

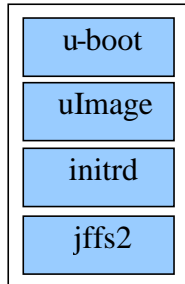
- u-boot: This file initializes the hardware when the machine powers on and launches the system startup process.
- uImage: This file contains the Linux 2.6.12 kernel. It is copied to RAM during startup so that it can be executed.
- Initrd (*INITial RamDisk*): An image of the minimal system initialized at system startup. The initial RAM disk is in read/write mode, but it not persistent.
- jffs2 (Journaling Flash File System version 2): An image of the logged file system. The jffs2 file system is in read/write mode, and it is persistent. The jffs2 system is mounted into RAM on the initial RAM disk:
  - The root of the jffs2 file system (/) is mounted to the .jffs2 directory.

- The jffs2 file system's /home directory is mounted to the /home directory on the initial RAM disk, the jffs2 file system's /tmp directory is mounted to the /tmp directory on the initial RAM disk, and the jffs2 file system's /var directory is mounted to the /var directory on the initial RAM disk.

In practice, this means that:

- If a file is created in the /home, /tmp, or /var directories, it is saved even when the power is cut.
- If a file is created in another directory (excluding /.jffs2), it is lost when the power is cut.

Mémoire Flash LT200



Flash memory addressing:

The following table shows the flash memory addressing:

Start Address	End Address	Size	File
0x00000000	0x00040000	256 KB	Bootloader
0x00040000	0x00080000	256 KB	Bootloader_env
0x00080000	0x000c0000	256 KB	Fast_save
0x000c0000	0x00100000	256 KB	Config
0x00100000	0x00280000	1.5 MB	Kernel
0x00280000	0x00a80000	8 MB	Initrd_FileSystem
0x00a80000	0x01000000	5.5 MB	Jffs2_FileSystem
	Total:	16 MB	

Bootloader\_env: Contains the LT200 environment variables, such as:

- ethaddr: MAC address
- ipaddr: IP address at startup
- netmask: subnet mask
- mode: startup mode

Fast-save: Contains the data saved during a power failure.

Config: Contains the network configuration data.

The client application will be stored in the jffs2 persistent file system. Its base size is 1.5 MB, leaving 4 MB available for the user.

# Getting Started in Windows XP

## Introduction

In this chapter, we describe the following steps:

- Installing the CodeSourcery open compilation chain
- Installing the Eclipse IDE
- Importing an existing project
- Creating a new project
- Compiling
- Loading an executable
- Debugging an executable

## Installing the CodeSourcery Open Compilation Chain

The source file is distributed on the CD-ROM for the C Pack.

You can also download it from the CodeSourcery website, <http://www.codesourcery.com>, choosing the ARM processor version "Sourcery G++ Lite 2008q1-126 for ARM GNU/Linux" or later. The download file is "IA32 Windows Installer", which is around 100 MB in size.

The tested version is: Sourcery G++ Lite 2008 q1-126.

The file must be executed. The installation starts up:



- A "C:\Program Files\CodeSourcery\Sourcery G++ Lite" directory is created.
- At the "Add product to the Path" prompt, select "Modify Path for current user" (or "for all users", if necessary).
- Finish the installation.
- Check that the path has been modified successfully. In a shell command window (cmd.exe), type:
  - "path". You should see the following in the path list:  
"C:\Program Files\CodeSourcery\Sourcery G++ Lite"  
"C:\Program Files\CodeSourcery\Sourcery G++ Lite\bin"
    - "arm-none-linux-gnueabi-g++ -v". The last line should contain "Sourcery G++ Lite 2008 q1-126".

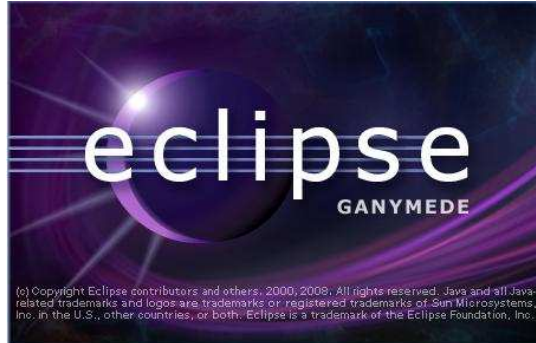
When installing on Windows Vista, the path is not modified automatically. In a shell command window, type the command:

```
"setx "%PATH%; C:\Program Files\CodeSourcery\Sourcery G++ Lite\bin"
```

Check that the path has been modified successfully, as shown above.

## Installing Eclipse

Download "Eclipse IDE for C/C++ Developers" from the Eclipse site, <http://www.eclipse.org/downloads>. The download file is around 60 MB. The tested version is version 3.4.1.



Unzip the downloaded file, which creates an "eclipse" directory. Drag it to your "C:\Program Files\CodeSourcery\Sourcey G++ Lite" directory.

Make a shortcut on your desktop to the Eclipse.exe file.

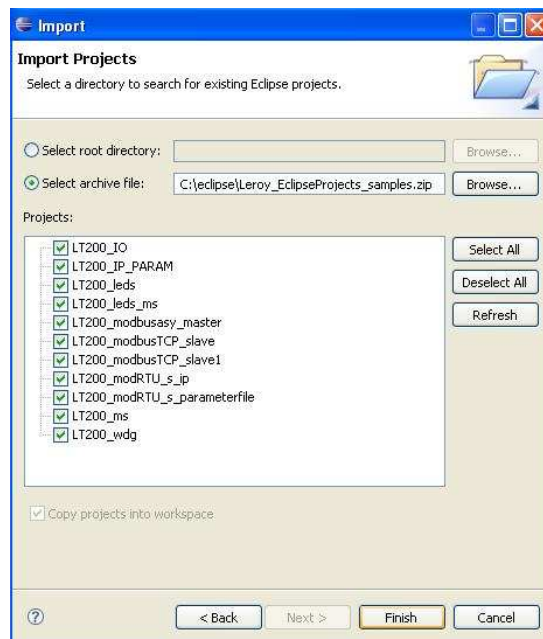
When starting Eclipse, select a directory for your projects: "Select a workspace".

## Importing an Existing Project into Eclipse

Use the Eclipse project explorer to import a sample project.

In Eclipse:

- Select the "File/Import" menu.
- Select the "General/Existing projects into Workspace" option.
- Select the archive file.
- Select the project(s) to be restored from the archive file, and then click on "Finish".



- The selected projects appear in the project list.

## Creating a New Project

You can create a new project.

In Eclipse:

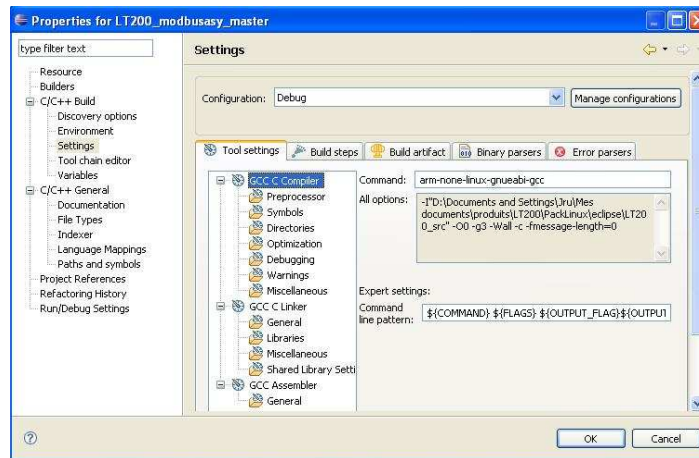
- Select the "File/New/CProject" menu.
- In the "C Project" window, select:
  - Project types: "Executable/Empty Project".
  - Tool chain: "Linux GCC". (Uncheck the "Show project types and toolchains only if they are supported on the platform" box.)
  - Provide a name for your project.

Your project then appears in the project list.

## Compiling

You can compile your project.

To check the compilation options for your project, open the Properties window for your project.



Next, verify the following options:

- In the "C/C++ Build" window:
  - Builder Type: "External Builder"
  - Build command: "cs-make"
    - In the "Discovery options" window:
  - Compiler: "GCC C Compiler"
  - Compiler Invocations command: "arm-none-linux-gnueabi-gcc"
    - In the "Settings / GCC C Compiler" window:
      - Command: "arm-none-linux-gnueabi-gcc"
      - Directories: your project path
        - In the "Settings / GCC C Linker" window:
          - Command: "arm-none-linux-gnueabi-gcc"
      - Libraries: Leroy libraries to include:

Ex. modbus library: library name: "libmodbusd.so", name to assign in Eclipse: "modbusd", specifying the directory path to where the library is located

- In the "Settings / GCC C Assembler" window:
  - Command: "arm-none-linux-gnueabi-as"

After the compilation, you will have generated an executable file called "S01application", and you can then move on to the next step: testing your application.

## Loading

You can load your executable file onto the LT200.

On your PC, you should have the following open at once:

- An FTP client for transferring your application to the LT200. The user name is "anonymous", and you can use an email address for the password.
- A Teraterm terminal (equivalent to the Windows hyperterminal: which emulates a serial (115200 bauds, 8bits, no parity) or TCP/IP Telnet (port 23) connection, to perform operations on modifiable files on the LT200.

Your executable should be in the "home/apps" directory so that it automatically runs at startup.

Via the terminal: Delete the current executable file.

- From the root, go to the "apps" directory:  
# cd home/apps/
- List the current files in the directory:  
# ls
- Delete the current application file:  
# rm S01application
- List the active processes:  
# ps
- Stop the "S01application" process:  
# kill + processnum

Via your FTP client: Transfer the new "S01application" executable file.

Via the terminal:

- Go to the "home/apps" directory:  
# mv ../anonymous/S01application .
- Change the rights on the new file:  
# chmod 777 S01application

## Debugging an Executable

There are two methods for debugging an executable:

- Booting without control, display messages in the console that are generated by the application program
- Booting with controls available as the program runs (stop points, step through, etc.), with a list of variables and gdb

Method #1: Booting without control, display of application messages

Via the terminal:

- Run the new file:  
# ./S01application -d
- List the active processes:  
# ps
- Verify that the new file is executed.

Method #2: Booting with gdb

Via the terminal:

- Run gdbserver on the LT200: The IP address for the development workstation is 192.168.1.152, and the open TCP port number for that PC is 5000.  
# gdbserver 192.168.1.152:5000 home/apps/ S01application

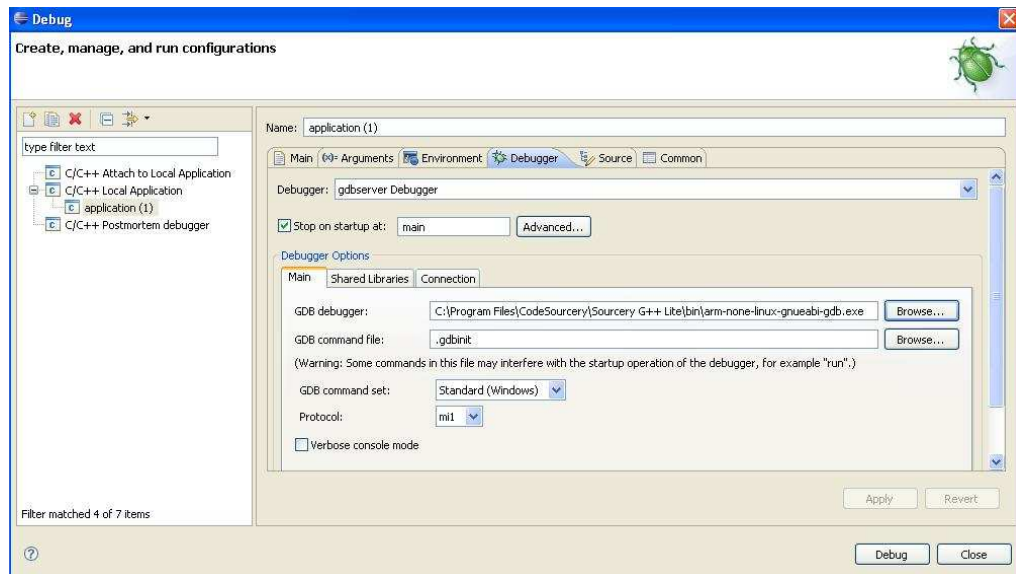
The following message should appear:

```

Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control Window Help
# gdbserver 192.168.1.151:5000 application
Process application created: pid = 529
Listening on port 5000

```

- On your development machine, change the debug options in Eclipse. Open the Debug window from the "Open debug dialog" window, and go to the "Debugger" tab. In the "GDB debugger" field, input the path and "arm-none-linux-gnueabi-gdb.exe" executable.



- In the "Connection" tab, input the LT200's IP address (192.168.1.180 here) and the TCP port number to use:



- Click the "Debug" button. The debug process starts in Eclipse, and the following messages appears in the console:

```

Tera Term Web 3.1 - COM1 VT
File Edit Setup Web Control Window Help
# gdbserver 192.168.1.151:5000 application
Process application created: pid = 529
Listening on port 5000
Remote debugging from host 192.168.1.151

```

All of the Eclipse debug functions are available when running your program.

# Getting Started in Linux

## Introduction

This chapter describes the C installation for LT200 on the Suse 10.3 distribution of the Linux operating system.

Other Linux distributions may be used, but they may require adjustments to run the various services used for the LT200.

For a PC running Windows XP, we have successfully tested virtualization with the VMware server in Windows XP and the installation of a Suse 10.3 distribution image.

In this chapter, we describe the following steps:

- Installing and configuring the development workstation
- Installing and configuring the Eclipse Europa IDE
- Importing an existing project
- Creating a New Project
- Compiling a project
- Loading an executable
- Debugging an executable

## Installing and Configuring the Development Workstation

### Linux system firewall:

The firewall must be deactivated. To do this, use the YaST utility:

```
# Yast
  > Sécurité et Utilisateurs [Security and Users]
    > Firewall
      > Arrêter le pare-feu maintenant [Stop the firewall now]
```

### Installing and configuring minicom

The *minicom* software is used as a serial terminal. It lets you watch the LT200 startup and access the backup console.

In order to install and configure it, you must have administrator rights (the system's *root* user). You can also install it from YaST.

- While logged in as administrator, install the *rzsz* and *minicom* packets:

```
# rpm -i rzsz-0.12.20-838.i586.rpm
# rpm -i minicom-2.1-144.i586.rpm
```

- While logged in as administrator, configure minicom:

```
# minicom -s
  > Serial port setup
    > Serial device = /dev/ttyS0
    > LockFile location = /var/lock
    > Bps/Par/Bits = 115200 8N1
    > Hardware flow control = no
    > Software flow control = no
```

```

> Save setup as dfl
> Exit from minicom
• While logged in as administrator, set up the following access rights:
# chmod 777 /var/lock
# chmod 777 /dev/ttyS0

```

## Installing and Configuring the Eclipse IDE

### Prerequisites

Eclipse is the recommended IDE. It is programmed in Java and therefore requires a JRE in order to run. You can find a list of JREs at [adresse](#). Once you have downloaded a JRE, you need to install it according to the procedure from its provider.

### Installing the software

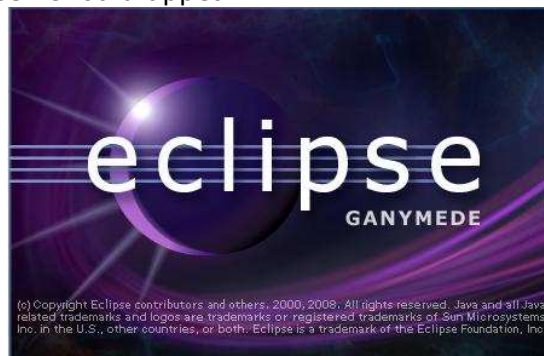
The Eclipse website is [www.eclipse.org](http://www.eclipse.org). The "Eclipse IDE for C/C++ Developers" software can be downloaded from [adresse](#). Once the software has downloaded, do the following:

```

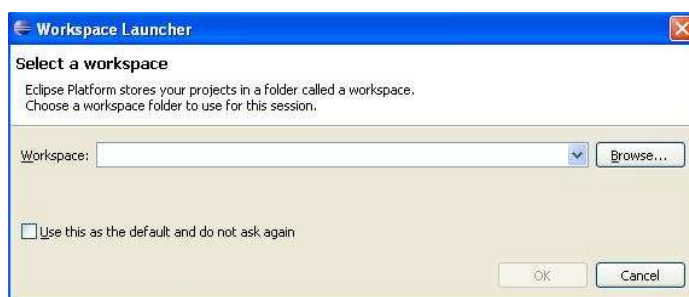
• Decompress the downloaded file:
# tar xvzf eclipse-SDK-3.4.1-linux-gtk.tar.gz
• Check that the installation worked properly:
# ls
eclipse      eclipse-SDK-3.4.1-linux-gtk.tar.gz
# ./eclipse/eclipse &

```

The Eclipse welcome screen should appear:



This dialog box should then appear:



Complete the dialog box with the path to your application directory, and then click OK.

## Development Environment

### Installation

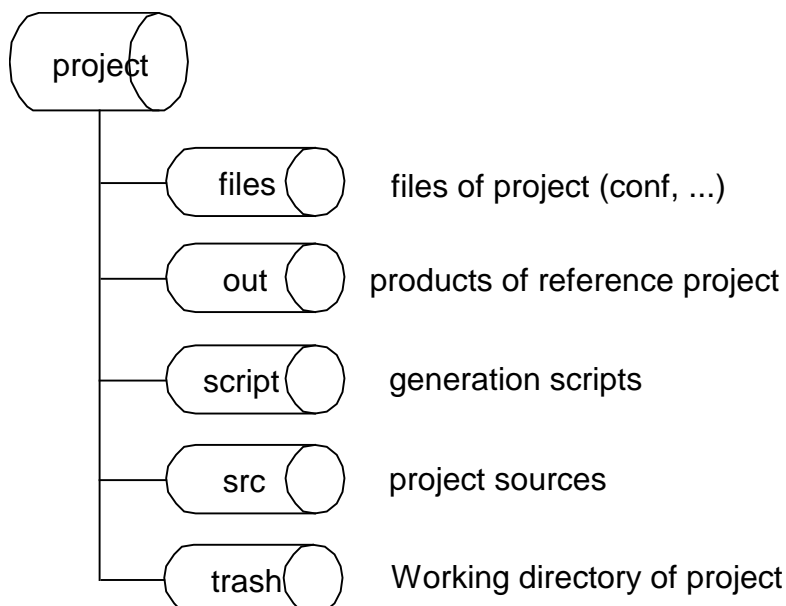
- Copy the "lt200-vX.Y" directory (X.Y corresponds to the software version) from the CD to the development machine.
- Modify the rights to the new directory created above on the development machine:

```
# cd lt200-vX.Y
# chmod -R a+w .
```

- Go to the "out" directory and decompress the nfsroot-lt200.tar.gz and include.tar.gz files:

```
# cd lt200-vX.Y/out/
# tar xvzf nfsroot-lt200.tar.gz
# tar xvzf include.tar.gz
```

Project structure:



### BSP project description

The "lt200/src" folder has the packaged projects that make up the product. They are all decompressed in the "build" folder using the "extract-lt200.sh" script:

- Go to the product directory and execute the script to extract all of the projects:

```
# cd lt200-vX.Y
# ./extract-lt200.sh
```

### Cross tools generation

This is required to be able to compile for the LT200. Go to the "project/scripts" directory and type the following command:

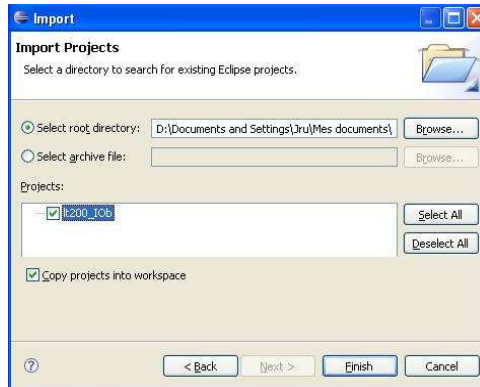
```
# ARM_COMPILER=/home/lt200-vX.Y/build/cross-tools-src-0.1/trash/arm-compiler ./build.sh
```

## Importing an Existing Project into Eclipse

Use the Eclipse project explorer to import an archived project.

In Eclipse:

- Select the "File/Import" menu.
- Select the "General/Existing projects into Workspace" option.
- Select the directory where the project file is located.
- Select the "Copy projects into workspace" option, and then click on "Finish".



- Your project then appears in the project list.

## Creating a New Project

You can create a new project.

In Eclipse:

- Select the "File/New/CProject" menu.
- In the "C Project" window, select:
  - Project types: "Executable/Empty Project".
  - Tool chain: "Linux GCC". (Uncheck the "Show project types and toolchains only if they are supported on the platform" box.)
  - Provide a name for your project.
    - Your project then appears in the project list.

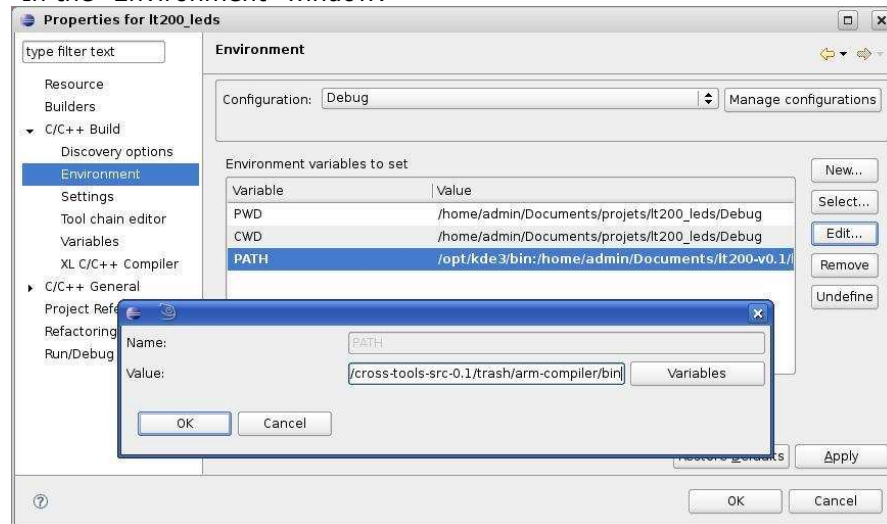
## Compiling

You can compile your project.

To check the compilation options for your project, open the Properties window for your project and verify the following options:

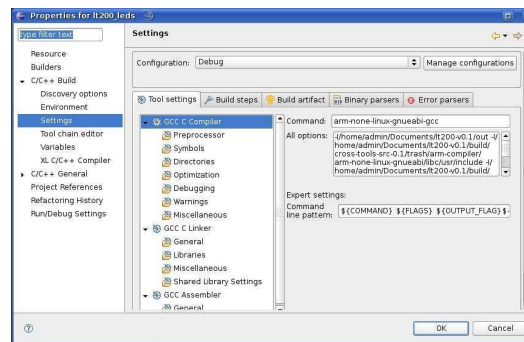
- In the "C/C++ Build" window:
  - Builder Type: "External Builder"
  - Build command: "Use default build command"
    - In the "Discovery options" window:
  - Compiler: "GCC C Compiler"

- In the “Environment” window:



Fill in the path to the cross-compiler directory:

- Name: “PATH”.
- Value: “:/home/admin/documents/lt200-vx.x/build/cross-tools-src-0.1/trash/arm-compiler/bin”. This value should be modified to correspond to the paths used in your installation.



- In the “Settings / GCC C Compiler” window:

- Command: “arm-none-linux-gnueabi-gcc”
- Directories: your project path
  - In the “Settings / GCC C Linker” window:
- Command: “arm-none-linux-gnueabi-gcc”
- Libraries: Leroy libraries to include:

Ex. modbus library: library name: “libmodbus.so”, name to assign in Eclipse: “modbusd”, specifying the directory path to where the library is located

- In the “Settings / GCC C Assembler” window:

- Command: “arm-none-linux-gnueabi-as”

After the compilation, you will have generated an executable file called “S01application”, and you can then move on to the next step: testing your application.

## Loading your Executable

You can load your executable file onto the LT200. The LT200 should be in operational mode.

On your PC, you should have the following open at once:

- An FTP client for transferring your application to the LT200. The user name is “anonymous”, and you can use an email address for the password.
- A minicom or telnet connection

Your executable should be in the “home/apps” directory so that it automatically runs.

Via the FTP client:

- Connect to the LT200 FTP server.
- Copy your application to the default directory on the FTP server: "/home/anonymous".

Via the terminal or Telnet client:

- From the root, go to the "apps" directory:  
# cd home/apps/
- List the current files in the directory:  
# ls
- Delete the current application file:  
# rm S01application
- List the active processes:  
# ps
- Stop the "S01application" process:  
# kill + processnum

Via your FTP client: Transfer the new "S01application" executable file, and go to the "home/apps" directory.

Via the terminal:

- Change the rights on the new file:  
# chmod 777 S01application

## Debugging

There are two methods for debugging an executable:

- Booting without control, display messages in the console that are generated by the application program
- Booting with controls available as the program runs (stop points, step through, etc.), with a list of variables and gdb

Method #1: Booting without control, display of application messages

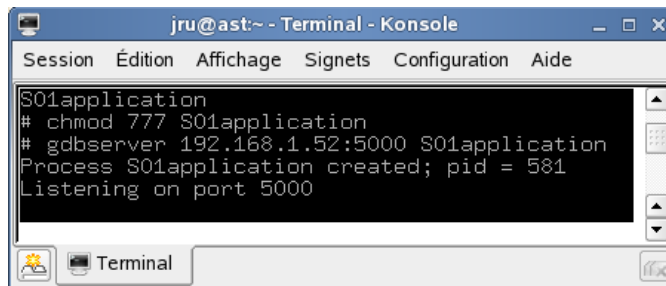
Via the terminal:

- Run the new file:  
# ./S01application -d
- List the active processes:  
# ps
- Verify that the new file is executed.

Method #2: Booting with gdb

Via the terminal:

- Run gdbserver on the LT200: The IP address for the development workstation is 192.168.1.152, and the open TCP port number for that PC is 5000.  
# gdbserver 192.168.1.152:5000 home/apps/ S01application
- The following message should appear:



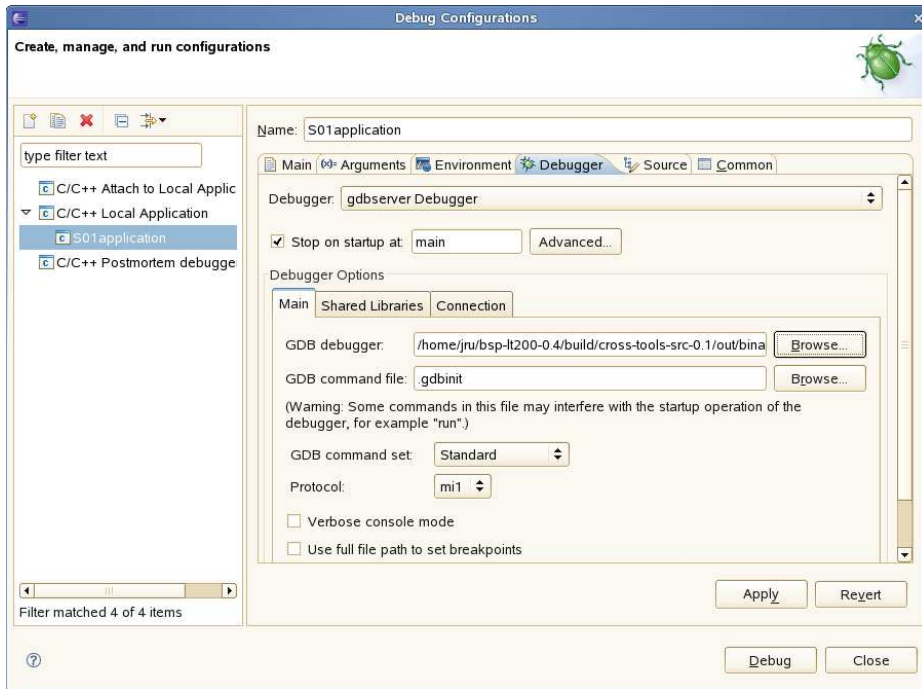
```

jru@ast:~ - Terminal - Konsole
Session  Édition  Affichage  Signets  Configuration  Aide
S01application
# chmod 777 S01application
# gdbserver 192.168.1.152:5000 S01application
Process S01application created; pid = 581
Listening on port 5000

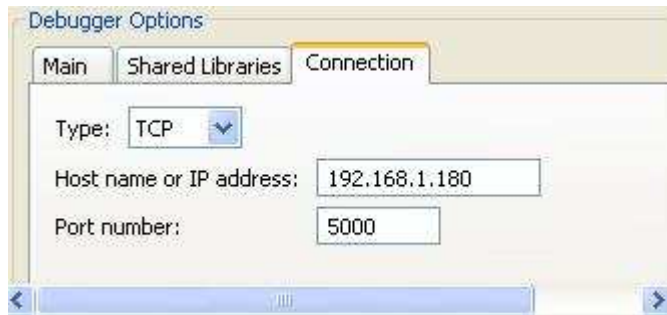
```

On your development machine,

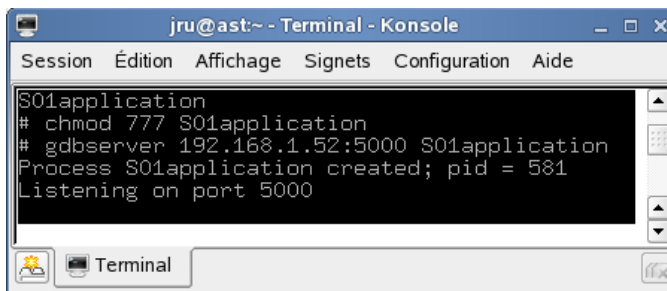
- Change the debug options in Eclipse. Open the Debug window from the "Open debug dialog" window, and go to the "Debugger" tab. In the "GDB debugger" field, input the path and "arm-none-linux-gnueabi-gdb.exe" executable.



- In the "Connection" tab, input the LT200's IP address (192.168.1.180 here) and the TCP port number to use:



- Click the "Debug" button. The debug process starts in Eclipse, and the following messages appears in the console:



- All of the Eclipse debug functions are available when running your program.

# Advanced Linux Programming

## Introduction

This chapter describes the exact steps for recompiling the kernel or part of it and updating the LT200.

In this chapter, we describe the following steps:

- Installing and configuring the development workstation
- LT200 operating modes
- Development environment
- Debugging in Appl mode
- Reloading the u-boot file
- Reloading the uImage, initrd, and jffs2 files on the LT200
- Modifying the library
- Adding a driver

## Installing and Configuring the Development Workstation

### Configuring the NFS server:

You must have administrator rights (system *root* user) to perform the operations in this section.

- Add the following line to the `"/etc/exports"` file (create the file if it does not exist):

```
/nfsroot *(rw,no_root_squash)
```

- Create a `"/nfsroot"` directory and modify its access rights:

```
# mkdir /nfsroot
```

```
# chmod 777 /nfsroot
```

- Restart the `"nfsserver"` service:

```
# /etc/init.d/nfsserver restart
```

### Configuring the DHCP server

This section describes the minimal configuration for operating the LT200. The server-specific settings are explained in the section on operating the LT200's DHCP client.

- Install the DHCP server.
- Add the following lines to the configuration file (generally `"/etc/dhcpd.conf"`):

```
host produit {  
  option host-name "produit" ;  
  option root-path "/nfsroot" ;  
  next-server XXX.XXX.XXX.XXX ;  
  hardware ethernet YY:YY:YY:YY:YY:YY ;  
  fixed-address AAA.AAA.AAA.AAA ;  
  filename "ulmage-lt200" ;  
  default-lease-time 172800 ;  
}
```

- XXX.XXX.XXX.XXX is the IP address for the NFS server.
- YY:YY:YY:YY:YY:YY is the MAC address for the LT200.

- AAA.AAA.AAA.AAA is the desired IP address for the LT200.
  - Restart the "dhcp" service:

```
# /etc/init.d/dhcpd restart
```

Configuring the tftp server

- Install the tftp package on the distribution. For the "tftp-0.38-2.2.i586.rpm" package, enter the following in a console:

```
# rpm -i tftp-0.38-2.2.i586.rpm
```

- After the installation, create the "/etc/xinetd.d/tftp" configuration file as shown below:

```
service tftp
{
    socket_type    = dgram
    protocol      = udp
    wait          = yes
    user          = root
    server        = /usr/sbin/in.tftpd
    server_args   = -s /tftpboot
    disable       = no
}
```

The files to be downloaded onto the LT200 (ex. in "prod" mode) should be in the "/tftpboot" directory.

- Restart the "xinetd" service:

```
# /etc/init.d/xinetd restart
```

LT200 Operating Modes

The LT200 has four operating modes, allowing for different uses:

- Operational (oper),
- Application development (appl),
- Kernel development (krnl),
- Production (prod).

Starting with version 1.3 of u-boot, new operating modes have been added:

- Production and set in operational (prodoper),
- updating of u-boot (updu),
- updating of initial ramdisk (updi),
- updating of jffs2 system file (updj).

The operating modes change how the LT200 behaves with respect to its software:

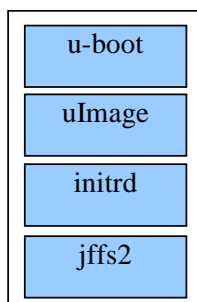
- u-boot-lt200.bin : Binary file of u-boot
- uImage-lt200: Binary file of the Linux kernel
- initrd-lt200.img: Initial RAM disk file for the LT200
- jffs2-lt200.img: File system residing in flash memory

Operational mode

This is the mode in which the LT200 is delivered. The LT200 uses the uImage-lt200.bin, initrd-lt200.img and jffs2-lt200.img files, which are written to its flash memory. This operating mode is the only mode in which the LT200 does have to be connected to the development workstation in order to operate. The LT200 is autonomous.

The Windows programming will only use this operating mode.

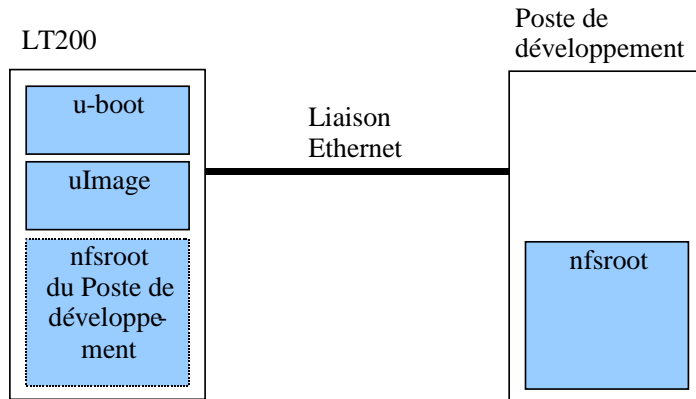
LT200



## Application mode

For this operating mode, the LT200 uses the "uImage-lt200" file, which is written to the flash memory. The "initrd-lt200.img" and "jffs2-lt200.img" files are replaced by their NFS equivalent on the development machine (/nfsroot directory).

This operating mode does not change the content of the flash memory. Going from application mode to operational mode does not involve the "nfsroot" file system used during application mode, but the "initrd-lt200.img" and "jffs2-lt200.img" files written the last time it was in production mode.



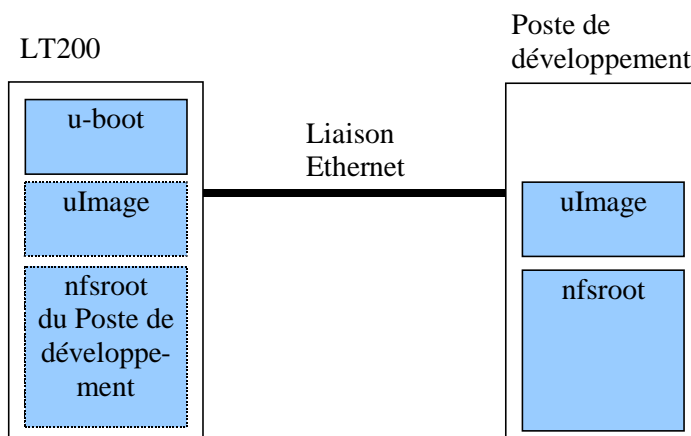
To use the "nfsroot" file system in operational mode, do the following:

- Generate the "initrd-lt200.img" and "jffs2-lt200.img" files from the nfsroot. See the Advanced Linux Implementation section.
- Put the LT200 into production mode in order for it to work with the previously generated files.

## Kernel mode

In this operating mode, the LT200 does not use any files in the flash memory. At startup, the LT200 loads the kernel (uImage-lt200) by tftp, and the file system by NFS (like in application mode).

This operating mode does not change the content of the flash memory. Going from kernel mode to operational mode does not involve the "nfsroot" file system and kernel used during kernel mode, but the "initrd-lt200.img", "jffs2-lt200.img", and kernel written the last time it was in production mode.



To use the "nfsroot" file system and kernel in operational mode, do the following:

- Generate the "initrd-lt200.img" and "jffs2-lt200.img" files from the nfsroot.
- Put the card in production mode so that it works with the previously generated files and the kernel used in kernel mode.

## Production mode

This is the only operating mode that allows the “uImage-lt200”, “initrd-lt200.img”, and “jffs2-lt200.img” files, written to the LT200’s flash memory, to be modified. When the LT200 starts up in production mode, the u-boot bootloader updates the software. The update steps are as follows:

- Request an IP address for the DHCP server
- Download the binary files and embedded file systems (uImage-lt200, initrd-lt200.img, and jffs2-lt200.img) by TFTP.
- Write the binary files and downloaded file systems to flash
- Verifying the binary files and file systems using checksum
- Doing a warm restart of the target in application mode

## How to change modes

The LT200 has an embedded Linux application “fw\_changemode” for changing the operating mode. For help with this application, type the command line below:

```
# fw_changemode --help
```

**WARNING:** When changing the mode, the bootloader’s configuration sector is modified. If the power is cut during this time, it is possible that the device will no longer operate correctly.

**Note:** If it is no longer possible to modify the operating mode from the “fw\_changemode” application, you can change it through the backup console.

Another possible method is to change the operating model when starting the LT200.

- Use a serial cable to connect the target and the development PC. Open a terminal, and establish a connection between the LT200 and the PC.
- Turn on the LT200, and press the “h”, “a”, “l”, and “t” to stop the bootloader when the prompt displays on the console.

```

jru@ast:-- Terminal - Konsole
Session  Edition  Affichage  Signets  Configuration  Aide
U-Boot 1.1.4-0.5Lai (Jun 25 2008 - 11:27:49)
U-Boot code: A0FE0000 -> A0FFBD74 BSS: -> A1000210
RAM Configuration:
Bank #0: a0000000 32 MB
Flash: 16 MB
In: serial
Out: serial
Err: serial

Overriding MAC address...
SMC9111: PHY auto-negotiate timed out
Using MAC address 12:34:56:78:9A:BC
Autobooting in 5 seconds, press 'h a l t' to stop
LT200-CPU610>

```

- Configure the bootloader in production mode:

```
LT200-CPU610> setenv mode prod
```

- Configure the bootloader in kernel mode:

```
LT200-CPU610> setenv mode krnl
```

- Configure the bootloader in application mode:

```
LT200-CPU610> setenv mode appl
```

- Configure the bootloader in operational mode:

```
LT200-CPU610> setenv mode oper
```

- Once the mode has been programmed, save the u-boot environment variables, and then restart the device:

```
LT200-CPU610> saveenv
```

```
LT200-CPU610> reset
```

## Development Environment

### BSP project description

Project structure:

You can regenerate each of the projects on the LT200. To do this:

- Go to the appropriate project scripts directory:
- ```
# cd projet/scripts
```
- Run the "build.sh" script, specifying the cross-compiler path with the "ARM\_COMPILER" shell variable:
- ```
#ARM_COMPILER=/home/lt200-v0.1/build/cross-tools-src-0.1/trash/arm-compiler ./build.sh
```

### Cross tools generation

By default, the "build.sh" script generates products in the "out" directory. You can change this behavior through the "BINDIR" shell variable.

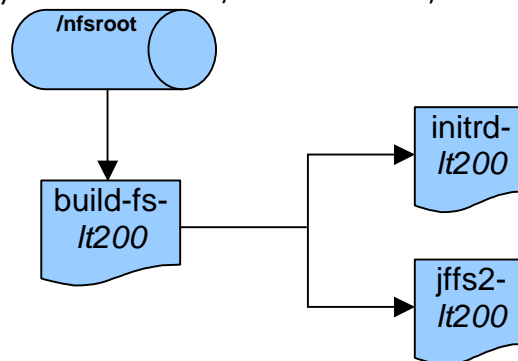
With the command line below, the project generates products directly on the file system, which is located in the "nfsroot" directory:

```
# BINDIR=/nfsroot ARM_COMPILER=/home/lt200-v0.1/build/cross-tools-src-0.1/trash/arm-compiler ./build.sh
```

### Generating embedded file systems

The "nfsroot" file system is located in the "/nfsroot" folder. This file system is the basis for the user's workspace. It can be modified to better fit the user's needs. Once the user has set up the file system (NFS), they can transform it into embedded file systems (initial ramdisk and jffs2) using the "build-fs-lt200.sh" script.

If the base NFS file system is in the "/nfsroot" folder, the script should be called using the



command line below:

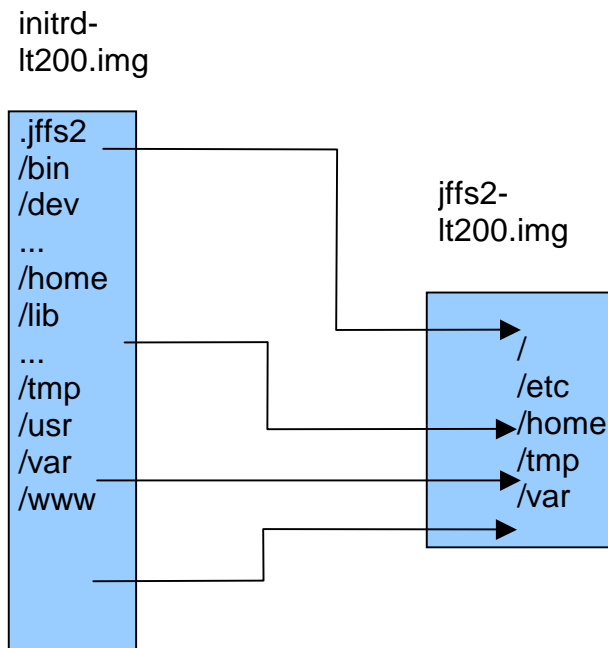
```
# ROOTFSDIR=/nfsroot ./build-fs-lt200.sh
```

The two generated files are in the "/home/lt200-v0.1/bin/binaries" directory.

**Note:** The "build-fs-lt200.sh" script prompts twice for the superuser password.

#### "initrd" and "jffs2" file contents

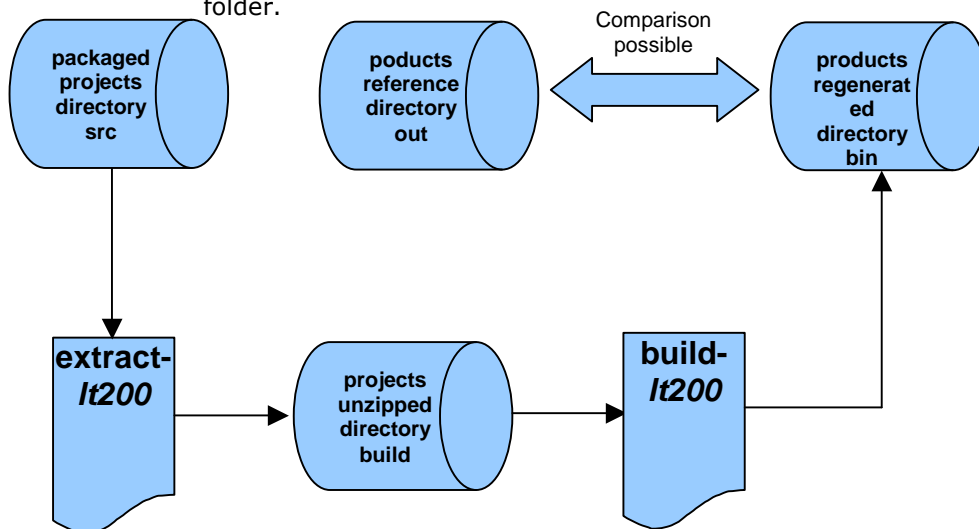
The embedded file systems are in "initrd-lt200.img" and "jffs2-lt200.img". They are generated from the "nfsroot" file system.



Total regeneration of the target

This use for the scripts is not essential to using the product. Its purpose is to regenerate the key products found in the "out" directory. The steps are as follows:

- Call the "extract-product.sh" extract script to extract the projects from the "src" directory into an intermediary folder (build).
- Call the "build-product.sh" script, which regenerates the binary files in the "bin" folder.



Notes: The "extract-It200.sh" script should only be executed once. The "build-It200.sh" script prompts three times for the superuser password.

Debugging in Application Mode

You can test your application in application mode. To do this, you must:

- Put the LT200 into application mode ("fw\_changemode" command).
- Configure your DHCP server with the right MAC address, and restart, if necessary.
- Have the DHCP, NFS, and TFTP services restarted.

You can copy your application to be debugged to the "/nfsroot/.jffs2/home" directory on your development machine.

The LT200 will start up as normal. You can start your application from the terminal or a Telnet connection, as described for implementing your Linux platform. The difference is that the LT200 file systems are on your development platform. You no longer have to use the FTP server to transfer your application.

## Reloading the u-boot File

As part of a change (boot, kernel), it may be necessary to reload the system startup file into the LT200's flash memory.

Warning: If an error occurs during this operation, the system might not restart and may need to be returned to the factory to be reconfigured.

Do the following operations:

- Start up the TFTP server on your development machine and copy the "u-boot-lt200.bin" u-boot image file from the "/tftpboot" directory to its directory.
- Start the LT200 and stop loading u-boot ("HALT" command). Verify that the TFTP server address and the IP for the device (the U-boot "serverip" and "ipaddr" variables, respectively) are correct, using the command:

```
LT200-CPU610>printenv
```

If necessary, change them using the "setenv" command:

```
LT200-CPU610>setenv serverip XXX.XXX.XXX.XXX
```

```
LT200-CPU610>setenv ipaddr AAA.AAA.AAA.AAA
```

➤ XXX.XXX.XXX.XXX is the IP address for the TFTP server.

➤ AAA.AAA.AAA.AAA is the desired IP address for the LT200.

Save the changes and restart:

```
LT200-CPU610>saveenv ;reset
```

Again, stop u-boot from loading on the LT200.

- Download the image from the TFTP server into RAM, using an unused address (0xa0600000):

```
LT200-CPU610>tftp 0xa0600000 u-boot-lt200.bin
```

- Next, unprotect the flash memory and delete the u-boot image located there:

```
LT200-CPU610>setenv bootstart 0x00000000
```

```
LT200-CPU610>setenv bootsize 0x00040000
```

```
LT200-CPU610>protect off $(bootstart) +$(bootsize)
```

"Un-Protected 1 sectors" message

```
LT200-CPU610>erase $(bootstart) +$(bootsize)
```

"Erased 1 sectors" message

- Then copy the image you just downloaded. (The "fileaddr" and "filesize" variables were automatically assigned during the download.)

```
LT200-CPU610>cp.b $(fileaddr) $(bootstart) $(filesize)
```

"Copy to Flash... done" message

- Protect the flash sector again.

```
LT200-CPU610>protect on $(bootstart) +$(bootsize)
```

"Protected 1 sectors" message

- Restart the LT200. In your console, you should see the number of the new version you downloaded.

## Reloading the uImage, initrd, and jffs2 Files

If any of these three files are changed, it may be necessary to reload them into the LT200's flash memory.

Do the following operations:

- Start up the tftp server on your development workstation and copy the files "uImage-lt200", "initrd-lt200.bin", and "jffs2-lt200.bin" to the "/tftpboot" directory.
- Start the LT200 and stop loading u-boot ("HALT" command). Switch to production mode:

```
LT200-CPU610>setenv mode prod
```

- Save the changes and restart:

```
LT200-CPU610>saveenv;reset
```

- The LT200 reboots in prod mode and automatically reloads its files. It then reboots into application mode.

- Warning : If the "initrd.bin" image and file size have changed, the "ramdisksize" environment variable must be modified for the new value:

```
LT200-CPU610>setenv ramdisksize xxxxx
```

xxxxx is the number appearing in the "initrd.log" file, corresponding to the size of the "initrd" file in the LT200's RAM memory once decompressed (default value: 12488 KB).

## Library Modifications

Do the following operations:

- In the "/nfsroot/lib" directory, replace the existing library with the new library.
- Regenerate "initrd-lt200.img" and "jffs2-lt200.img". To do this, execute the script "build-fs\_lt200.sh" with the following command line: # ROOTFSDIR=/nfsroot ./build-fs-lt200.sh
- Copy these files to the "/tftpboot" directory.
- Put the LT200 into Production ("prod") mode.
- Restart the LT200.
- After reloading the LT200, switch to operational ("oper") mode.
- Restart the LT200. Your application should start automatically.

## Adding a Driver

Do the following operations:

- In the "/nfsroot/lib/modules/2.6.12/kernel/drivers/" directory, create the directory that corresponds to your driver, and copy your "mydriver.ko" to there.
- Edit the "sysinit" file, located in the "nfs/root/etc" directory, and change it only by adding this driver. Save the file.
- Regenerate "initrd-lt200.img" and "jffs2-lt200.img". To do this, execute the script "build-fs\_lt200.sh" with the following command line: # ROOTFSDIR=/nfsroot ./build-fs-lt200.sh
- Copy these files to the "/tftpboot" directory.
- Put the LT200 into Production ("prod") mode.
- Restart the LT200.
- After reloading the LT200, switch to operational ("oper") mode.
- Restart the LT200. Your application should start automatically.

# LT200 Configuration

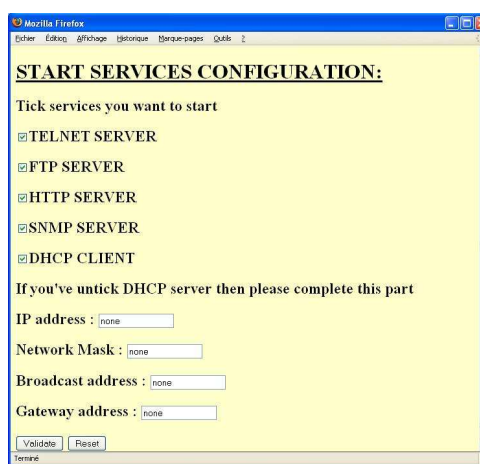
## Introduction

The user can easily modify how the LT200 behaves using Ethernet. This chapter covers how to configure the LT200's Ethernet services:

- Ethernet configuration and daemon activation
- DHCP daemon
- SNMP daemon
- FTP daemon
- HTTP daemon

## Ethernet Configuration and Daemon Activation

The LT200 comes with an embedded web page for modifying the Ethernet settings and activating/deactivating the various services.



**Note:** If the LT200 is no longer available by Ethernet, you can use the backup console to modify its settings.

- Allow the kernel to start until the welcome message and Linux kernel console prompt appear.
- Modify the product's Ethernet settings using the regular Linux commands:  
# ifconfig eth0 AAA.AAA.AAA.AAA netmask MMM.MMM.MMM.MMM broadcast BBB.BBB.BBB.BBB

## Daemon Configuration

The daemons may be configured through the `.jffs2/etc/user_services` file:

```
# variables changes
USER_FTP_ROOT_CUSTOM="NULL"
USER_HTTP_ROOT_CUSTOM="NULL"
USER_SNMP_CONF_CUSTOM="NULL"
```

```

USER_DHCP_BLOCKING="TRUE"
# export variables
export USER_FTP_ROOT_CUSTOM
export USER_HTTP_ROOT_CUSTOM
export USER_SNMP_CONF_CUSTOM
export USER_DHCP_BLOCKING
    
```

## DHCP Daemon

### Introduction

The DHCP client has two operating modes.

- To put the DHCP in blocking mode, the following must appear in the .jffs2/etc/user\_services file:

```

USER_DHCP_BLOCKING="TRUE"
    
```

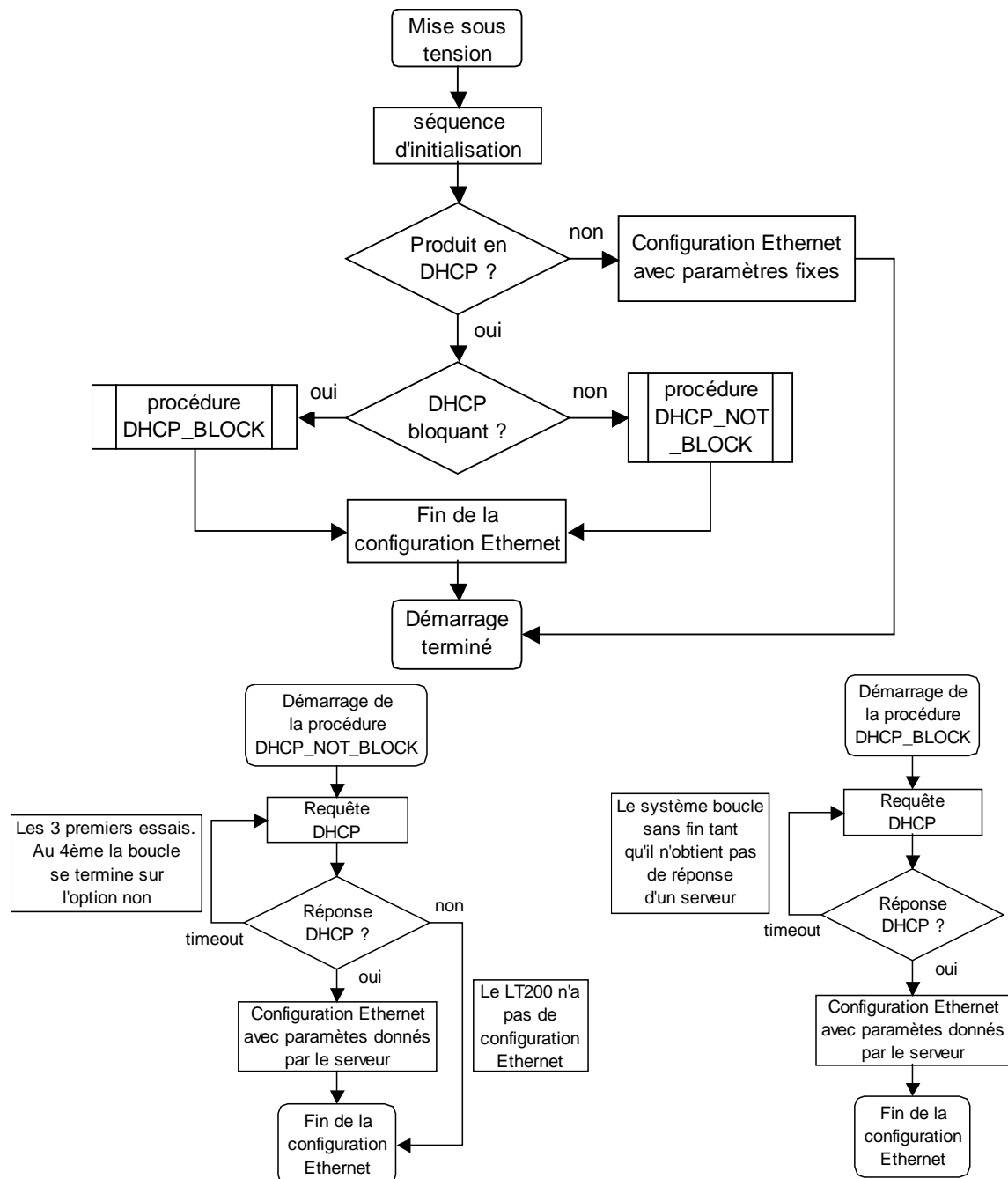
With this setting, the daemon will execute the DHCP\_BLOCK procedure.

- To put the DHCP in non-blocking mode, the following must appear in the .jffs2/etc/user\_services file:

```

USER_DHCP_BLOCKING="FALSE"
    
```

With this setting, the daemon will execute the DHCP\_NOT\_BLOCK procedure.



## SNMP Daemon

It is possible to modify the configuration file used by the SNMP daemon.

By default, the configuration file is `/etc/snmpd.conf`. To change the configuration file (ex. to `/home/snmpd/snmpd.conf`), replace the following line:

```
USER_SNMP_CONF_CUSTOM="NULL"
```

with the line:

```
USER_SNMP_CONF_CUSTOM="/home/snmpd/snmpd.conf"
```

and then restart the product.

### SNMP daemon modifications

The customer can add their own mibs to the SNMP daemon in \*.c format:

- Go to the `lt200-vX.Y/build/net-snmp-5.3.0.1-X.YLai/` directory.
- ```
cd produit-vX.Y/build/net-snmp-5.3.0.1-X.YLai/
```
- Create the "mibs" directory:
- ```
mkdir files/mibs
```
- Copy the individual mibs, in \*.c format, to the "files/mibs" directory:
  - Go to the "scripts" directory:
- ```
cd scripts
```
- Edit the "build.sh" script.
  - Go to the "NET-SNMP specific initialization" section.

**Note:** Only the variables in this section should be modified. Modifying a variable outside of this section can reduce the performance of the "build.sh" script or the SNMP daemon.

- Modify the variables in this section:
- `NET_SNMP_SYS_CONTACT`: Allows a default value to be assigned to the `SNMPv2-MIB::SysContact.0` OID

**Note:** This variable's value will be erased if a configuration is made in the `snmpd.conf` file.

- `NET_SNMP_SYS_LOCATION`: Allows a default value to be assigned to the `SNMPv2-MIB::SysLocation.0` OID

**Note:** This variable's value will be erased if a configuration is made in the `snmpd.conf` file.

- `NET_SNMP_ADD_MIBS`: Allows specific mibs from `./files/mibs/` to be included in the compilation. A mib added with this variable must be prefixed by `"$MIBOUTDIR/"` so that it will be properly included.

**Example:** To add the files in the `leroyio` mib to the mibs file, we have to add `"$MIBOUTDIR/leroyio"` to the `NET_SNMP_ADD_MIBS` variable.

- Modify the `./files/snmpd.conf` file.
- Execute the "build.sh" script, specifying the path to the cross-compiler and the target system file:

```
# BINDIR=/nfsroot ARM_COMPILER=/home/projet/build/cross-tools/trash/arm-compiler ./build.sh
```

- The SNMP daemon is regenerated in the given directory by the "BINDIR" shell variable.

It is possible to query the LT200's SNMP daemon using Net-SNMP tools. If the mibs with an OID of 4273 are added to the company branch, you can scan through them using the command below:

```
snmpwalk AAA.AAA.AAA.AA -v 2c -c leroy-public 1.3.6.1.4.1.4273
```

## FTP Daemon

You can change the root of the FTP server. By default, the root of the FTP server is the `/home/anonymous` directory. To change the root directory (ex. to `/home/www`), replace the following line in the "user\_services" file:

```
USER_FTP_ROOT_CUSTOM="NULL"
```

with the line:

```
USER_FTP_ROOT_CUSTOM="/home/www"
```

and then restart the LT200.

## HTTP Daemon

You can change the root of the HTTP server. By default, the root of the HTTP server is the "/www" directory. To change the root directory (ex. to "/home/www"), replace the following line in the "user\_services" file:

```
USER_HTTP_ROOT_CUSTOM="NULL"
```

with the line:

```
USER_HTTP_ROOT_CUSTOM="/home/www"
```

and then restart the LT200.

# Programming Your Application

## Introduction

This chapter describes the structure of a user application and the main functions of the LT200. Their implementation is described using basic programs. Please refer to the online documentation for details on the libraries or drivers used.

C programming on Linux offers many other possibilities, such as adding communication protocols and multithreading, which are not detailed here.

In this chapter, we describe the following functionalities:

- Starting and stopping applications
- Managing TOR and analog input/output cards
- Activating the watchdog (WdG)
- Using the millisecond counter
- Managing the LEDs on the CPU card
- Using the modbus/TCP slave protocol
- Using the modbus asynchronous master protocol
- Modifying the LT200's IP settings from a modbus serial master

In these examples, we will not show the traditional C header files, which obviously must be included:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <string.h>
```

The main program is described in Chapter 2. It is simply modified to run a particular function. The modifications are detailed in each of the following examples.

## User Applications: Starting and Stopping

LT200 Linux users develop applications that run on the Linux kernel.

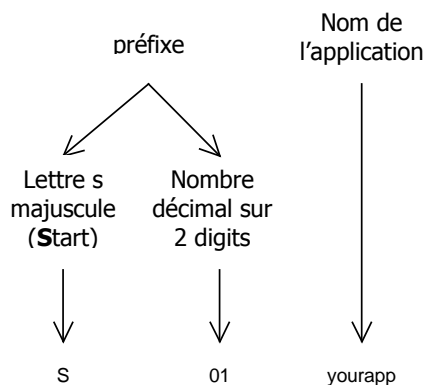
When these applications are finished being developed, they are permanently embedded and then execute when the product starts and stop when the product shuts down.

### General description

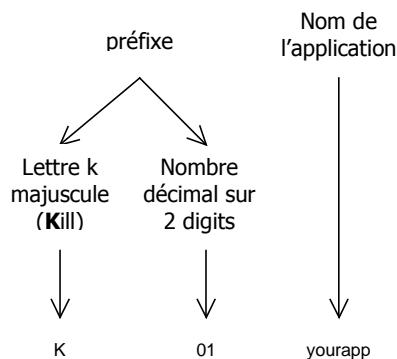
The start/stop system for user applications is based on the traditional daemon system in a Unix system. In this system, software scans a particular directory that contains the files to be executed on startup and the files to be executed on shutdown.

The default directory for user applications is the "/home/apps" directory.

In order for files to be executed at startup, their name should follow the naming convention illustrated below:



In order for files to be executed at shutdown, their name should follow the naming convention illustrated below:



The **SXX**daemon and **KXX**daemon are launched in order of their **XX** number.

### Startup

The user applications are launched at the end of the Linux environment initialization phase.

### Environment variables

There are environment variables that users may modify to affect the start/stop of user applications.

- **USER\_APPS\_ROOT\_CUSTOM** variable: This variable is used for modifying the **default** directory scanned by the application start/stop script.
- **USER\_PARAM\_LAUNCH\_APPS** variable: This variable allows the user to control the start and stop of user applications across the board. If

USER\_PARAM\_LAUNCH\_APPS is equal to 1, the script is executed; otherwise, it is not executed.

### Launching/stopping a user application:

The user application start/stop script detects binary files and SH script files in the file list based on the SXXapp or KXXapp pattern in the "/home/apps" directory. To be able to distinguish between these two types of files, the only binary format accepted is the ELF format. The two file types are distinguished by reading the first 4 bytes of the file. If the second, third, and fourth byte of the file are "E", "L", and "F", respectively, the file is a binary file; otherwise, it is a script.

### Launching/stopping using a script

When the user application start/stop script launches a script at system startup, it passes in the "start" parameter. When the user application start/stop script launches a script at system shutdown, it passes in the "stop" parameter. The code below is the recommended skeleton for user scripts:

```
#!/bin/sh
#
case $1 in
start)
# Add your code here to launch your application.
;;
stop)
# Add your code here to stop your application.
;;
*)
echo "Usage: $0 {start|stop}"
exit 1
esac
exit 0
```

The SXXapp and KXXapp scripts called at startup and shutdown are actually only symbolic links to a single script that is structured as shown above. Symbolic links are created using the standard Linux command "ln".

Using an intermediary script to launch the user application allows arguments to be passed in dynamically at execution.

### Launching/stopping using a binary file

When the user application start/stop script launches a binary file directly at system startup or shutdown, it is done without passing in a parameter.

### Constraints on user applications

Applications launched by the user application start/stop script must be "daemonized". To "daemonize" an application, place the following code at the top of the user application.

The main.c program in your application is as follows:

```
#include <stdio.h> /* for EOF */
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <getopt.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>

void signal_handler ( int number ) {
    ...
}
```

```
int main(int argc, char **argv) {  
.....  
    /* INITIALIZE YOUR APPLICATION HERE */  
    while (!KillHim) {  
        /* YOUR APPLICATION CYCLE SHOULD BE HERE */  
        usleep(10000);  
    }  
    /* CLEAN UP AFTER YOUR APPLICATION HERE */  
    return 0;  
}
```

## Managing Input/Output Cards

**Type:** driver

**Header file:** drv2.h

The LT200's TOR and analog input/output cards are managed via a driver called "driver C2", where C2 is the generic name for input/output cards. The CPU and input/output cards communicated via a bus, called B2.

The LT200 can manage up to 15 input/output cards on the B2 bus, numbered 1 to 15.

The C2 driver is managed as if it were accessing a file. The driver is opened and closed using the regular file open and close functions (open and close).

All of the input/output functionalities work by means of an IOCTL. An IOCTL is a system call. The IOCTL allows the driver to provide a specific interface, all while relying on system calls.

The IOCTL system call function has the following prototype:

```
int ioctl(int fd, unsigned long cmd, ... )
```

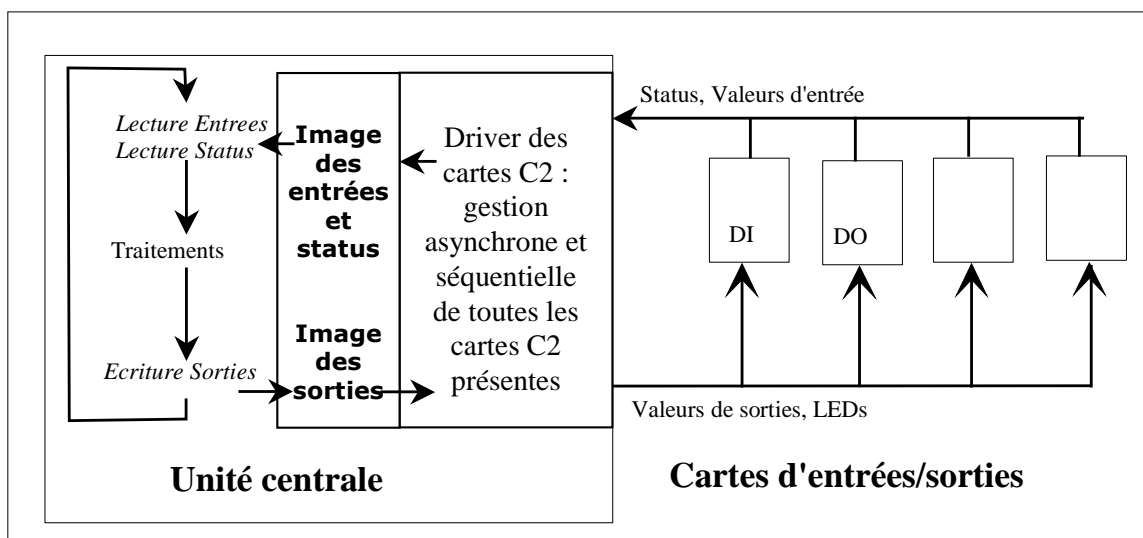
Here, "fd" represents the file descriptor obtained when opening the driver (open), and "cmd" represents the command associated with the IOCTL.

### Available IOCTL commands for managing the C2 driver:

- LT200C2\_IOC\_INSERT: declares a C2 card
- LT200C2\_IOC\_REMOVE: removes a C2 card
- LT200C2\_IOC\_DRV\_START: tells the driver to start managing cards
- LT200C2\_IOC\_DRV\_STOP: tells the driver to stop managing cards
- LT200C2\_IOC\_DRV\_STATUS: information on the driver's status
- LT200C2\_IOC\_GET\_INPUT: reads input from a C2 card
- LT200C2\_IOC\_SET\_OUTPUT: creates output from a C2 card
- LT200C2\_IOC\_GET\_STATUS: reads the status of a C2 card

The following shows the cycle to be programmed on the LT200 for your application to run safely. We recommend using the following order:

- Read input and status information from C2 cards
- Process the data
- Write the output



### Data structure for input/output cards

Each input/output card is associated with a working data structure, as defined in the drv2 file:

DI310 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- unsigned int Input: the card's 32 TOR input value

DO310 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- unsigned int Output: the card's 32 TOR output command

DI410 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- unsigned int Input[2]: table of the card's 64 TOR inputs

DIO210 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- unsigned int Input: the card's 16 TOR entries
- unsigned char Output: the card's 8 TOR output command

DI312 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- unsigned int Input: 32 input value
- unsigned int Fault: 32 input state
- unsigned int Mask: input state mask
- unsigned int RCNO: normally-open resistance sensor (in ohms)
- unsigned int RCNF: normally-closed resistance sensor (in ohms)
- unsigned int RL: line resistance value (in ohms)

AI110 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- short Input[8]: table of the card's 8 analog inputs
- unsigned char Led\_G: 8 green LEDs command
- unsigned char Led\_R: 8 red LEDs command

AI210 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- short Input[16]: table of the card's 16 analog inputs
- unsigned short Led\_G: 16 green LEDs command
- unsigned short Led\_R: 16 red LEDs command

AO121 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- short Output[8]: table of the card's 8 analog outputs
- unsigned char Led: 8 LEDs command

AIO320 card:

- unsigned char Type: card identification code
- unsigned char SlotAddr: card position on the B2 bus
- unsigned short Status: card status
- short Input[8]: the card's 8 analog input value
- short Output[4]: table of the card's 4 analog outputs
- unsigned char Leds: 8 LEDs command

Input/output card status

Cyclically reading the card status allows for an internal diagnostic to signal problems. The table below lists the status values that are read.

Meaning of the input/output card status bits:

|            |                                                                |
|------------|----------------------------------------------------------------|
| Bits 0-7   | Card code [0..FFh] Bit 6 is meaningless.                       |
| Bit 8      | 1 for proper internal power to the card                        |
| Bit 9      | See table below.                                               |
| Bit 10     | FAULT: 0 if the card is not refreshed by the CPU (FLT LED lit) |
| Bit 11     | See table below.                                               |
| Bits 12-15 | Card position on the bus [0..15]                               |

Input/output card codes

| Card   | Status Bit 11 | Status Bit 9 | Card Code [0..FFh]<br>Bit 6 Masked: BFh |
|--------|---------------|--------------|-----------------------------------------|
| DI310  | 1             | Ext Pwr      | 03h or 43h                              |
| DI312  | N/A           | Ext Pwr*     | 14h                                     |
| DI410  | 1             | Ext Pwr      | 06h                                     |
| DO310  | Overload      | Ext Pwr      | 05h or 45h                              |
| DIO210 | Monostable    | Vrel         | 16h                                     |
| AI110  | 0             | 0            | 80h                                     |
| AI210  | 0             | 0            | 81h                                     |
| AO121  | 0             | 0            | 88h                                     |
| AIO320 | Monostable    | 0            | 83h                                     |

Where:

- N/A: not applicable
- Ext pwr: 1 if there is external power to the terminals between ±20%.
- (\*) DI312 external power: External power between 24V±10%
- Overload: 0 if overloaded on a TOR output
- Monostable: 1 if the card has been properly refreshed
- VRel: 1 if the relays are properly powered

Example of managing input/output cards on the B2 bus:

The following operations should be performed:

The LT200 to be programmed has the following configuration:

| Desc                   | Power      | CPU Block: LUC4001 |                                | DI310 Block: LID16241 |                         |                         | DO310 Block: LOD16240 |                          |                          |
|------------------------|------------|--------------------|--------------------------------|-----------------------|-------------------------|-------------------------|-----------------------|--------------------------|--------------------------|
| Card name              | PSD33<br>1 | CPU610             | Com630                         | DI310                 | 16i24b                  | 16i24b                  | DO310                 | 16o24b                   | 16o24b                   |
| Terminal               | 24V<br>DC  | RS232,<br>USB      | Ethernet,<br>3 RS232<br>/RS485 |                       | 16 24 V<br>DC<br>inputs | 16 24 V<br>DC<br>inputs |                       | 16 24 V<br>DC<br>outputs | 16 24 V<br>DC<br>outputs |
| Position on the B2 bus | 0          | 0                  |                                | 1                     |                         |                         | 2                     |                          |                          |

The "lt200\_ms" sample program implements the three typical steps:

- Initialization:

```
// Open the input/output card driver
driver_c2 = open("/dev/drvc2", O_RDWR);
```

```
// Initialize and define the card working structure
memset(&Card1,0,sizeof(DRVC2_TS_DI310));
memset(&Card2,0,sizeof(DRVC2_TS_DO310));
Card1.Type=LT200C2_C_DI310;
Card2.Type=LT200C2_C_DO310;
```

```
// Card positions on the B2 bus
Card1.SlotAddr=1;
Card2.SlotAddr=2;
```

```
// Add the cards to the driver
retval=ioctl(driver_c2, LT200C2_IOC_INSERT, &Card1);
retval=ioctl(driver_c2, LT200C2_IOC_INSERT, &Card2);
```

```
// Start driver
retval=ioctl(driver_c2, LT200C2_IOC_DRV_START, NULL);
```

- Processing cycle:

```
// read input
retval=ioctl(*driver_IO, LT200C2_IOC_GET_INPUT, &Card1);
read card status
retval=ioctl(*driver_IO, LT200C2_IOC_GET_STATUS, &Card1);
retval=ioctl(*driver_IO, LT200C2_IOC_GET_STATUS, &Card2);
// processing example - copy input to output
Card2.Output=Card1.Input;
// write output
retval=ioctl(*driver_IO, LT200C2_IOC_SET_OUTPUT, &Card2)
```

- Stop:

```
retval=ioctl(*driver_IO, LT200C2_IOC_DRV_STOP, NULL);
printf("ioctl LT200 C2_IOC_DRV_STOP function returned %d.\n", retval);
retval=ioctl(*driver_IO, LT200C2_IOC_REMOVE, &Card1);
retval=ioctl(*driver_IO, LT200C2_IOC_REMOVE, &Card2);
retval= close(*driver_IO);
```

**Note: Handling Boolean data**

The C language has no Boolean type, but this type would be useful for us, as it corresponds to TOR input and output.

Define the Boolean type:

```
typedef enum
{
    false=(0!=0),
    true=(0==0)
}bool;
```

Declare the input variables:

```
Bool Input0;
Bool Input1;
Bool Input2;
```

You may also extra the TOR input from Card 1:

```
Input0 = (Card1.Input & (0x0001<<0))>0;
Input1 = (Card1.Input & (0x0001<<1))>0;
Input2 = (Card1.Input & (0x0001<<2))>0;
```

Likewise, to write TOR output to Card C2, you can use the following method before doing the "LT200C2\_IOC\_SET\_OUTPUT":

```
Card2.Output= (int)S0 + (int) S1 * 2 + (int)S2* 4 + (int)S3 * 8 + (int)S4 *
16 + (int)S5 * 32 +(int)S6 * 64 + (int)S7 * 128 + (int)S8 * 256 + (int)S9 *
512 + (int)S10 * 1024 + (int) S11 * 2048 + (int)S12 * 4096 + (int)S13 *
8192 + (int)S14 * 16384 +(int)S15 * 32768;
```

## Watchdog Management

**Type:** library: Libsoftwatch.so

**Header file:** softwatch\_client.h

The watchdog (WdG) monitors that your application is working properly.

Function: This daemon is active when starting up the device. When your application starts up, it must identify itself to the WdG daemon and then reconnect to the daemon. If this does not happen after a specified time while initializing the connection, the daemon will perform the requested actions during the initialization step.

### Example:

The "lt200\_wdg" sample program implements the above three steps:

- Initialization: Connects to the watchdog's internal server using the "watchdog\_register" function with the selected management type and timeout value, after which the LT200 reboots:
 

```
ret_code = watchdog_register(WDG_REBOOT_ACTION |
    WDG_WDGBUS_ACTION | WDG_LEDFAIL_ACTION, refresh_interval);
```
- Processing cycle: Refreshes during the watchdog cycle with the "watchdog\_refresh" function:
 

```
ret_code = watchdog_refresh();
```
- Stop: Disconnects from the WdG server to avoid rebooting during a kill:
 

```
ret_code = watchdog_unregister();
```

## Using the Millisecond Counter

**Type:** driver

**Header file:** none

A millisecond counter is useful for measuring the duration of a process, which is very common in automated devices: work time, rest, monostable, etc.

You can use a very precise counter that increments every millisecond (ms). This incrementation is done with a break, which makes it possible to have a more accurate reference point than the basic system time available in the user space.

It is represented as a Linux character driver.

The driver is opened and closed using the regular file open and close functions (open and close).

Of the traditional functions, only the read function is implemented. This function allows you to read the value of the counter.

There are actually two millisecond counters: a 16-bit counter and a 32-bit counter.

When reading 2 bytes using the read function, it reads the value of the 16-bit counter. By reading 4 bytes, the value of the 32-bit counter is returned.

The counters are coded with 16-bit and 32-bit unsigned integers.

### Example: Modifying the main program:

The "lt200\_ms" sample program implements the above three steps:

- Initialization: Opens the ms driver  
`CountMsFileDesc = open("/dev/drvcountms", O_RDWR);`
- Processing cycle: Reads the ms counter, 32-bit value.  
`read(CountMsFileDesc,&currentTime1,4);`
- Stop: closes the file descriptor  
`retval= close(CountMsFileDesc);`

## Managing the LEDs on the CPU Card

**Type:** driver

**Header file:** drvled.h

The LT200's motherboard has 6 LEDs on the front that can be controlled from a user application:

- led RUN
- led FAIL
- led I/O
- led PRM
- led F1
- led F2

The LED driver provides a simple interface to user applications for accessing the LEDs. It is managed like accessing a file. The driver is opened and closed using the regular file open and close functions (open and close).

All of the associated functionalities work by means of an IOCTL. An IOCTL is a system call. The IOCTL allows the driver to provide a specific interface, all while relying on system calls.

The IOCTL system call function has the following prototype:

```
int ioctl(int fd, unsigned long cmd, ... )
```

Here, "fd" represents the file descriptor obtained when opening the driver (open), and "cmd" represents the command associated with the IOCTL.

### Available IOCTL commands for managing the LED driver:

- LT200LED\_IOC\_SET: command for lighting the LEDs
- LT200LED\_IOC\_RESET: command for turning off the LEDs

### Example:

The "lt200\_leds" sample program implements the above three steps:

- Initialization: Opens the LED driver, obtains a file descriptor  
LedsFileDesc = **open**("/dev/drvled", O\_RDWR);
- Processing cycle: Directs the status of the LEDs via the IOCTL system call; parameters: file descriptor, command type (SET, RESET, etc.), LED to be controlled.  
retval=**ioctl**(\*LedsFileDesc, LT200LED\_IOC\_SET, LT200\_LED\_RUN);
- Stop: Closes the LED driver  
retval= **close**(LedsFileDesc);

### Example of managing flashing LEDs:

The "lt200\_leds\_ms" sample program shows a method for making the Run and IO LEDs blink on and off when your application is running.

main.c: main program

- Initialization: Initializes the milliseconds driver and LEDs
- Processing cycle: Calls the "cycle\_leds" program
- Stop: Closes the milliseconds driver and LEDs

leds.c:

- "Cycle\_leds" function: controls the RUN and IO LEDs at 2 frequencies; calculates based on the time, a milliseconds counter.

## Modbus Protocol

**Type:** libmodbusd.so

**Header file:** modbus\_api.h

Modbus is a communication protocol for exchanging data between several types of equipment. It is a master/slave protocol with either a serial (RS232, RS485, RS422) or Ethernet (100 MB) connection for physical support.

This protocol is described in several downloadable documents: <http://www.modbus.org/>

Implementing this protocol requires the use of the library specified above. There is a detailed description in RDL\_MUT\_XXXX\_A0.pdf, located in the library's documentation archive file.

The LT200 can simultaneously handle the following functionalities:

- On each of its 4 serial connections: master or slave serial modbus
- On its Ethernet connection: master and slave modbus/TCP

A data table can be associated with each slave.

The data is bit data and word data (16 bits), and the slave tables have a size expressed in words. The bit and word tables are combined.

Example modbus table:

This can be represented as a 16-column table, representing the 16 bits contains in a word, and x rows representing the words:

| Word Address | Bit Row | F | E | D | C | B | A | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Word Value                                 |
|--------------|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--------------------------------------------|
| 0            |         | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | =255                                       |
| 1            |         | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | =4                                         |
| 2            |         | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | =1*2 <sup>12</sup> +1*2 <sup>6</sup> =4160 |
| 3            |         |   |   |   |   |   |   |   | 1 |   |   |   |   |   |   |   |   |                                            |
| 4            |         |   |   |   |   | 0 |   | 1 |   | 1 |   |   |   |   |   |   |   |                                            |
| 5            |         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |                                            |
|              |         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |                                            |
|              |         |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |                                            |

Bit at address:  $16 * 2 + 12 = 75$

To find out the modbus address for a bit, you can do the following calculation:

bit address =  $16 * \text{word address} + \text{bit row}$

The total size of all of the slave tables should not exceed 4096 words.

Maximum number supported by the LT200 with TCP:

- Maximum number of modbus/TCP masters = 32
- Maximum number of modbus/TCP slaves = 32

The following modbus function codes are supported by the LT200 modbus library:

- 1: read\_coils: reads bits
- 2: read\_input\_discretes: reads input bits
- 3: read\_multiple\_registers: reads words
- 4: read\_input\_registers: reads input words
- 5: write\_coil: writes a bit
- 6: write\_single\_register: writes a word
- 7: read\_exception\_status: reads status information
- 8: return\_query\_data:
- 15: force\_multiple\_coils: writes bits
- 16: write\_multiple\_registers: writes words
- 22: mask\_write\_register: writes masked words
- 23: read\_write\_register: reads and writes words

Initialization and stop example:

The following operations must always be done performed one time when initializing the LT200. The settings are modified based on the expected functionalities before starting up the modbus service.

```
ModbusDaemonState state;
state.master=0;      // master subsystem activation: no.
state.nb_tcp_master=0; // Number of masters.
state.slave=1;      // slave subsystem activation: yes.
state.overlapping=1;// table of modbus bits, words, input bits combined.
state.size=4096;    // total size of all tables (overlap case)
ret=modbus_start(&state);
```

These settings are thus set as follows:

- master: 0: active; 1: inactive
- nb\_tcp\_master: must be >0 if the master service is active
- slave: 0: active; 1: inactive
- overlapping: leave as 1
- size: 1 to 4096

Service stop:

```
ret=modbus_stop(&state);
```

## Using the Modbus Slave Protocol

**Type:** libmodbusd.so

**Header file:** modbus\_api.h, bit.h

The LT200 serial modbus protocol allows a master to access the bits and words in the LT200 modbus.

The LT200 modbus/TCP slave protocol allows one or more modbus/TCP masters to establish a TCP connection with the LT200 on its TCP port 502 and to read or write bits and words using the modbus functions in its modbus table. The slave therefore behaves like a server, and the master behaves like a client.

The slave manages its connections using two functions:

- Function for filtering masters requesting a connection. The filtering is done by IP address.
- Function for monitoring connected masters.

### Example of implementing the RTU modbus slave protocol:

The "lt200\_modbusRTU\_slave" sample program shows a method for implementing the RTU modbus protocol.

main.c: main program

Do the following operations:

- Initialization:
  - Register the slave modbus/TCP: "modbus\_register\_slave" function
 

```
ModbusDaemonState state;
state.master=0;      // Master subsystem inactivate.
state.slave=1;      // Slave subsystem activate.
state.overlapping=1; // Modbus slave tables are overlapped.
state.size=4096;    // Size of all tables (overlap case)
ret=modbus_start(&state);

ModbusSlaveParameter slave_param;
int slave_handle=0;
reset_slave_param(slave_param); // Reset all members (security action)
slave_param.type=SLAVE_TCP_THREAD;
slave_param.slaveaddr=1;
slave_param.port=502;          // Default Modbus port.
ret=modbus_register_slave(&slave_param,&slave_handle) ;
```
  - Add a segment for mapping a defined variable to the modbus slave table. Here, it is the "seg[1024]" table, with 512 words.
 

```
ModbusTableSegment seg;
seg.size=1024;
seg.rw=seg_cb;
ret=modbus_add_segment(slave_handle,0,&seg);
```
- Processing cycle: No operation required
- Stop:
 

```
ret=modbus_unregister(slave_handle);
ret=modbus_stop(); // stop the modbus system
```

## Using the Modbus Master Protocol

**Type:** libmodbusd.so

**Header file:** modbus\_api.h, bit.h

Do the following operations:

Initialization:

- Configure and start the modbus service: "modbus\_start" function
- Configure and register the modbus master: "modbus\_register\_master" function
  - Type: MASTER\_RTU\_THREAD or MASTER\_TCP\_THREAD
  - targetname: 2 cases:
    - serial modbus: COM\_A ("/dev/tts/0"), COM\_B ("/dev/tts/1"), COM\_C ("/dev/tts/2")
    - modbus/TCP: the slave's IP address
  - specific parameters for the serial modbus:
    - baudrate=1200/2400/4800/9600/19200/38400 bauds;
    - databits=7/8: 7 or 8 bits
    - stopbits=1/2; 1 or 2 stop bits
    - parity=0/1/2: no parity, odd or even
    - mode= MODE\_RS232/MODE\_RS485/MODE\_RS422: communication RS232, RS485 or RS422
  - timeout=100; time-out in ms

There are two possible situations:

- A single request
- A periodic request

For each new periodic request, perform the following operations:

Initialization:

- Define a trigger with a type and send period.
- Add this trigger: "modbus\_add\_pool" function
- Define a periodic request: modbus function code, address for the data to exchange, called functions
- Add this request: "modbus\_add\_request" function

Processing cycle: No operation required

### Example: RTU modbus master:

The "lt200\_modbusRTU\_master" sample program implements the word reading function every 500 ms in an RTU modbus slave.

modbus.c:

Initialization: Function to initialize communication:

- Initialization of the modbus server:
 

```
ModbusDaemonState state;
state.master=1; // master subsystem activation.
state.nb_tcp_master=1; // Number of masters.
state.slave=0; // slave subsystem deactivation.
ret=modbus_start(&state); // start the modbus system
```
- Initialization of the RTU modbus master protocol:
 

```
ModbusMasterParameter master_rtu_param;
int master_rtu_handle=0;
reset_master_param(master_rtu_param); // Reset all members
master_rtu_param.type=MASTER_RTU_THREAD;
master_rtu_param.targetname="/dev/tts/2";
master_rtu_param.baudrate=19200;
master_rtu_param.databits=8;// 8 bits
master_rtu_param.stopbits=1;// 1 stop
master_rtu_param.parity=2;// even parity
master_rtu_param.mode=MODE_RS485;
master_rtu_param.timeout=100;// time-out in ms
```

```

        ret=modbus_register_master(&master_rtu_param,&master_rtu_handle)
    ;
- Creation of variables needed for the request
    int request_id;
    int pool_id;
    ModbusTrigger trig;
    ModbusRequest req;
    memset(&req,0,sizeof(ModbusRequest));
- Definition and addition of a trigger with a type and send period.
    trig.type=TRIGGER_PERIODIC;
    trig.periodic.time=500; // request sent every 500 ms
    pool_id=modbus_add_pool(master_rtu_handle, trig);
- Definition and addition of a periodic request: modbus function code, address for the data to
  exchange, called functions
    req.code=3; // modbus function: reads several words
    req.slave_addr=1; // slave number (0..255)
    req.read_ref=6; // data address in the slave
    req.read_cnt=1; // number of elements to read
    req.read=wcallback; // callback: function called with each read
    req.error=callback_er; // error callback: function called with each error
    request_id=modbus_add_request(master_rtu_handle,pool_id,req);
}
- Function called when reading in the slave:
    int wcallback ( int id, void* buffer,int size) {
        char* _temp;
        _temp=(char*)buffer;
        // copies info to the table @15
        memcpy(&(modbus_tab[30]),&(_temp[0]),2);
        return 0;
    }
- Diagnostics reading function
    int callback_er ( int id, void* buffer,int size) {
        ModbusRequestState* state;
        state =(ModbusRequestState*)buffer;
        // copies info to the modbus table @150
        memcpy(&(modbus_tab[300]),&(state->nb_transaction),4);
        memcpy(&(modbus_tab[304]),&(state->nb_error),4);
        memcpy(&(modbus_tab[308]),&(state->nb_exception),4);
        memcpy(&(modbus_tab[312]),&(state->error),4);
        memcpy(&(modbus_tab[316]),&(state->exception),4);
        printf("state: %d, %d, %d, %d, %d\r\n",
                state->nb_transaction,
                state->nb_error,
                state->nb_exception,
                state->error,
                state->exception);

        return 0;
    }
}

```

**Stop:**

- Deactivate the master communication:
 

```
ret=modbus_unregister(master_rtu_handle);
```
- Read the diagnostics
 

```
ret=modbus_stop();
```

## Modifying the LT200's IP Settings from a Modbus Serial Master

The purpose of this section is to show how to modify the LT200's IP settings from a simple modbus serial master.

The user "network" settings, which includes IP settings, are defined in a file called "user\_param" in the ".jffs2/etc" directory. It is a text file that can be read if you wish to use them, rewrite them, or change them.

The "lt200\_modbusRTU\_s\_ip" sample program implements the RTU modbus slave protocol and modifies its IP settings via the modbus table that is associated with the RTU modbus slave protocol.

Implemented function: The IP address is coded based on 4 modbus words (addresses 80 to 83) (IP address: AAA.BBB.DDD.DDD, 1st byte AAA in word 80, 2nd byte BBB in word 81, etc.). The subnet mask is also made up of 4 words (addresses 84-87), and the gateway has 4 words (addresses 88 to 91).

modbus.c:

Function for initializing the modbus slave serial port and the modbus table.

```
// initialization of the modbus table with the network settings at startup
void init_IP()

void init_modbus()
{
.....
    // initialization of the modbus table with the network settings
    init_IP();
}
```

The "userparam" file is updated only if it is requested by the modbus master. The "MAJ\_IP\_Tmodbus" program is called cyclically in the main program, looking for the modbus word at address 92. If the master writes the value 100, then the program reads the new network settings in the modbus table (addresses 80 to 91) and writes them via the "SetStaticEthAddr" function.

modbus.c:

```
void SetStaticEthAddr(char * interfaceName, char * NewIPAddress,
    char * NewBcastAddress, char * NewGateway, char * NewNetMask)

void MAJ_IP_Tmodbus()
{
.....
    SetStaticEthAddr("eth0", NewIPAddress, NewBrdCast, NewGateway,
NewMask);
.....
}
```

Main program

```
while (!KillHim) {
    /* YOUR APPLICATION CYCLE SHOULD BE HERE */
    // refresh the process every 5 ms
    usleep(5000);
    MAJ_IP_Tmodbus();
}
```

## Main Terminal Commands

The full documentation for BusyBox is located in the "doc" directory on the CD-ROM.

Common commands:

- **help**: lists all available commands
- **ls**: lists the contents of the current directory
- **ls -l**: lists the contents of the current directory along with their user rights
- **cd** + directoryname: changes from the current directory
- **mv** + filename + directoryname: moves a file into a new directory
- **cp** + filename + directoryname: copies a file into a new directory
- **rm** + filename: deletes a file
- **chmod** +rights+ filename: changes the rights on a file
- **ps**: lists the processes that are currently running
- **kill** + processnum: stops a process
- **ifconfig**: displays the network settings
- **ifconfig** eth0 AAA.AAA.AAA.AAA netmask MMM.MMM.MMM.MMM broadcast BBB.BBB.BBB.BBB: modifies the network settings, including the IP address, subnet mask, and broadcast address
- **netstat**: displays information about the current network connections
- **fw\_changemode** mode + modename: changes the LT200 startup mode. modename = "oper", "prod", "appl", or "krnl"
- **reboot**: stops and restarts the system
- **uptime**: displays how much time has elapsed since the last boot along with the average CPU load since startup

Advanced commands:

- **lsmod**: lists the loaded modules
- **df**: lists the used and available disk space on the file system
- **free**: displays how much RAM is used and available
- **top**: lists the current process, their CPU usage rate, and their RAM usage rate
- **nice**: executes a program with modified execution priorities

# GLOSSARY

- BSP: [Board Support Package](#), low-level software for motherboard support
- IDE: [Integrated Development Environment](#): Integrated Development Environment
- daemon: Disk And Execution MONitor; designates a type of [program](#) or [process](#) that runs in the background, rather than under the direct control of a user. Daemons are often started when loading the operating system. In [Microsoft Windows](#), these functions are executed by programs called "services".
- SDK: Software Development Kit
- CDT: C/C++ Development Toolkit by [Eclipse](#)
- DHCP: Dynamic Host Configuration Protocol: [network protocol](#) for automatically configuring IP settings for a [workstation](#), particularly by automatically assigning it an [IP address](#) and a [subnet mask](#).
- FTP: File Transfer Protocol: network protocol for TCP-based file transfer. The client initializes the connection to the server, generally using port 21. Most current browsers support FTP by using a [uniform resource identifier](#), such as ftp://username:password@server\_name\_or\_address:ftp\_port
- TFTP: Trivial File Transfer Protocol: a simplified file transfer protocol. It works with [User Datagram Protocol](#) on port 69, as opposed to [File Transfer Protocol](#), which uses [Transmission Control Protocol](#).
- Telnet: TErminAl NETwork. A [client-server](#) protocol based on TCP. Clients generally connect to the server's [port](#) 23.
- NFS: Network File System. NFS is a protocol developed by [Sun Microsystems](#), which allows a computer to access [files](#) over a network.

# INDEX

|                             |    |                    |    |
|-----------------------------|----|--------------------|----|
| application mode            | 19 | modbus             | 40 |
| daemon configuration        | 25 | modbus master      | 43 |
| DHCP daemon                 | 26 | modbus slave       | 42 |
| DHCP server                 | 17 | NFS server         | 17 |
| embedded Linux file system  | 2  | operational mode   | 18 |
| flash memory                | 3  | Physical resources | 1  |
| FTP daemon                  | 27 | production mode    | 20 |
| HTTP daemon                 | 28 | SNMP daemon        | 27 |
| Initrd                      | 3  | tftp server        | 18 |
| jffs2                       | 3  | u-boot             | 3  |
| kernel mode                 | 19 | ulmage             | 3  |
| managing input/output cards | 33 | user applications  | 30 |
| milliseconds                | 38 | WdG                | 37 |