

emDist 2.4.1

User Manual

Rev8 / 18.10.2011

© Copyright 2010 **emtrion GmbH**

All rights reserved. This documentation may not be photocopied or recorded on any electronic media without written approval. The information contained in this documentation is subject to change without prior notice. We assume no liability for erroneous information or its consequences. Trademarks used from other companies refer exclusively to the products of those companies.

Author: Martin Nylund, emtrion GmbH
Revision: **8 / 18.10.2011**

Rev	Date/Signature	Changes
2	03.04.2010/Ny	First release for the completely revised emDist documentation for emDist-2.3
3	22.06.2010/Ny	Typo fixes in the "download and run prepare-emdist..." chapter.
4	29.06.2010/Pk	Update for emdist 2.3.1 release
5	01.02.2011/Du	Update for emdist 2.3.2 release
6	31.03.2011/Sz	Update for emdist 2.3.3 release Adapted emdist-prepare chapter to new concept
7	10.06.2011/Sz	Update for emdist 2.4.0 release
8	18.10.2011/Du	Update for emdist 2.4.1 release

1 Content

3	Some conventions in this manual	5
4	emDist Support from emtrion GmbH	6
5	Introduction.....	7
5.1	Hardware specific documentation (interfaces, drivers etc.)	8
6	emDistVM Virtual machine installation and usage	9
6.1	System Requirements	9
6.2	Performance issues	9
6.3	Installing VMWarePlayer	9
6.4	Installing emDistVM.....	10
6.5	Starting emDistVM	11
6.6	Exchanging data with the virtual machine.....	12
6.6.1	Copy and Paste	12
6.6.2	Samba.....	12
6.6.3	NFS.....	12
6.6.4	Other Methods.....	13
6.7	Changing the virtual machine settings.....	14
6.7.1	RAM allocation	14
6.7.2	Change screen resolution	14
6.7.3	Change serial port.....	15
7	First steps with emDist and your target board	16
7.1	Configure the starterkit to use the virtual machine as NFS/TFTP server	16
7.2	Update images on the target boards flash	17
7.3	Run Linux kernel with NFS root mount.....	18
7.4	Building custom target file-system image.....	18
7.5	A word on the Development model	19
8	Installing emDist on Ubuntu Linux	20
8.1	Configure sudo	21
8.2	Configure BASH to be the standard shell.....	21
8.3	Download and run <code>prepare-emdist</code>	22
8.4	Start the CrossTarget web interface.....	22
8.5	Configure NFS server	23
8.5.1	Testing if NFS share is working.....	24
8.6	Installing emDist on other Linux distributions.....	24
9	CrossTarget and CtScripts	25
9.1	Starting the CrossTarget server	25
9.2	Running CtScripts from the web interface	26
9.3	Running CtScripts from command line	27
9.4	Bash command line completion for CtScripts.....	27
9.5	CrossTarget files and directories	28
9.5.1	Static files and directories	28
9.5.2	Dynamic files and directories.....	29
9.6	System build – the big picture	30
10	Target configuration files	31

10.1	Where target configuration files are kept?	31
10.2	Main target configuration files.....	31
10.3	Adaptation layers (root file-system variants).....	32
10.4	Creating custom target file-system variants	33
10.5	Adding packages to your configuration.....	34
10.6	Package dependencies	34
11	Package configuration files.....	35
11.1	A word on cross compiling open source software	36
11.2	CrossTarget package configuration files.....	38
11.3	A simple local package configuration file.....	39
11.4	A more complex example.....	40
11.5	Package parameters	42
11.6	Commonly used Macros.....	44
11.7	Standard Package Methods (prepare, configure, etc.)	45
11.8	Optional/Custom package methods	47
11.9	Patching package sources	48
11.9.1	Keeping a patch in the package configuration file (rfile).....	48
11.9.2	Using an external patch file without macros	48
11.9.3	Using an external patch file with macros (rfile).....	49
11.10	Filtering directories/files from the flash image (PATH_FILTERING).....	49
12	CrossTarget Configuration file syntax.....	50
12.1	Nodes and nodepaths.....	50
12.2	Comments (# ..).....	50
12.3	Simple value assignments (node = value)	51
12.4	Multiline values (node = ""value"").....	52
12.5	Enumerating node names (% operator)	52
12.6	Absolute section Headers ([node])	53
12.7	Relative section Headers ([.node])	53
12.8	Macros (@(node))	54
12.9	Including configuration files (node < foo.cfg).....	55
12.10	Reading content from external files (node < !myfile.txt).....	56
12.11	Debugging Ct configuration	56
13	Using the tool-chain	57
13.1	Tool-chain location and the PATH variable.....	57
13.2	Cross-compiling and running a simple application	57
14	Cross-Debugging applications	58
14.1	Remote debugging with plain GDB	58
14.2	Remote debugging with DDD	59

3 Some conventions in this manual

For the sake of clarity, there are some placeholders and dummy values used in this manual, which you need to adapt for your purposes.

Place-holder	Description	Example of a real value
CORE_MODULE	target board code, which should correspond	dimm-sh7724
\$CT, @(CT)	Top level directory of the emDist/CrossTarget installation (CT is abbreviation of CrossTarget)	/home/hico/emdist-2.3.2
\$ROOTFS, @(ROOTFS)	File-system tree for the target board	/home/hico/emdist-2.3.2/CORE_MODULE/rootfs
Hico	General username	-

4 emDist Support from emtrion GmbH

The support emtrion GmbH provide for emDist is divided into three categories:

Generally Supported:

- Problems or general questions on getting started with the delivered Virtual Machine and development environment.
- Problems or general questions on the target board hardware interfaces and device drivers (USB, Ethernet, CAN, etc.).

Supported to some extent ⁽¹⁾:

- Getting started with the delivered BSP on another Linux distribution. The BSP is designed to be distribution independent. We cannot, however, guarantee that it will work out-of-the-box on any distribution or development host configuration.
- Customer specific configuration and modifications on the Linux kernel, build tools, user-space configuration (e.g. flash partitioning, boot method, etc), and open source packages (Busy box, Qt, etc.).

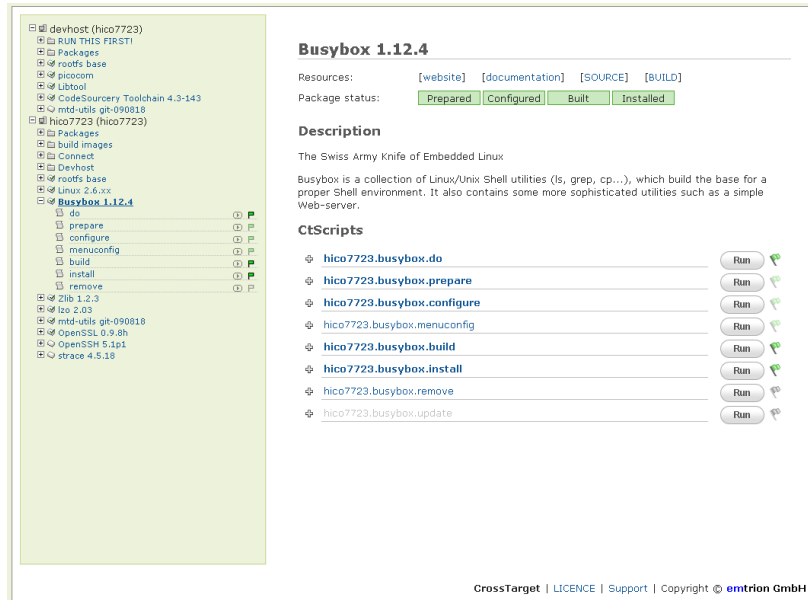
Supported only with extra support contract:

- Application development
- Issues on customer specific hardware
- Customer specific change/modification demanding notable effort.

⁽¹⁾ "Supported to some extent" means that we are willing to give tips and guidance, but are not bound to deliver any results.

5 Introduction

emDist (emtrion Linux distribution) is an embedded Linux distribution developed at emtrion GmbH. The distribution provides a comfortable web interface for building software packages and target images. It also provides a command line interface for developers who prefer to do the things on the console screen.



CrossTarget Web-interface

Main emDist components are:

- For the target board configured embedded Linux kernel and a large collection of user-space software packages; C/C++ libraries, Busybox, Qt embedded, Python, Apache web-server, PHP, GTK, SSH/SSL libraries and many more.
- Preconfigured root file system, with well commented start-up scripts and configuration files.
- GNU cross compiling tool-chain and GNU debugger (*gdb*) configured for cross-debugging with the target.
- A Web-browser based tool (CrossTarget web-interface) to build the packages and to keep an overviews of the system.
- Command line tools (CtScripts) which can be used as an alternative to the Web-interface.

Emtrion starter-kits are delivered on VMware virtual machines (**emDistVM**), which contain all this in a ready configured Linux environment (Ubuntu). You just need to install *VMwarePlayer* to run the environment on your Windows PC.

With the delivered emDistVM Virtual Machine you can start development immediately on your Windows host. If you already have your choice of Linux, you can also install emDist on it. See the support terms first, however, if you want to use CrossTarget on some other Linux distribution.

5.1 Hardware specific documentation (interfaces, drivers etc.)

This manual only describes usage of the emDist tools, not the target specific drivers and interfaces. Drivers and interfaces are documented on our support pages at www.support.emtrion.de. The boot-loader is also documented in a manual of its own.

6 emDistVM Virtual machine installation and usage

emDistVM is a VMware virtual machine with preinstalled Linux distribution (Ubuntu) with all required packages for the BSP and the BSP itself. You can start the VM on your Windows PC with the delivered VMware player. The purpose of the VM is to:

1. Serve as a development host for people who don't want to have an extra Linux PC next to their Windows PC. With the delivered virtual machine you can comfortably have both operating systems running on the same machine.
2. Serve as a reference installation, for people who want to use their own Linux installation.
3. Give a quick start for anyone who doesn't want to spend time on configuring the development environment.

Note: If you already have a Linux installation with which you are satisfied, you don't have to use the virtual machine. You can install emDist/CrossTarget on your development host as well. This is, however, not recommended for Linux beginners.

6.1 System Requirements

In order to develop efficiently on the virtual machine it is recommended to have a development PC with at least a 'Pentium 4 class' processor and 1GB memory. For development in the long term it is recommended to have 2GB of memory and set the Virtual Machine RAM to 1GB (see next chapter). The VM zip file is about 1.5G and after unzipping it requires 10G disk space.

6.2 Performance issues

If you are going to use the Virtual machine as your Development environment it is recommended that you give it more than 512MB of RAM. With the default 512MB the VM is not well suited for running large graphical programs like IDEs. You can change the amount of RAM as described in Chapter [RAM allocation](#).

6.3 Installing VMWarePlayer

Insert the emDist VM DVD and open the start site `start_here.html`. From here you can copy the VMware-Player installation program onto your host and run it. (You can also download it from [VMware web site](#)). The installation should be straight forward.

NOTE: In case you already have VMware workstation older than version 6.5, the delivered VM cannot be used with it. You need to upgrade your VMware software to 6.5 or later (or use VMware player).

6.4 Installing emDistVM

Unzip the `emdistvm.zip` from the delivered DVD into a local working directory (For example “My Files\My Virtual Machines”). The location doesn’t matter as long as it’s not a network directory

Start VMwarePlayer and open the emDistVM configuration file from the location where you unpacked the zip file (VMware Player will ask you for the location when you start it). After opening the file with VMwarePlayer you should see the virtual machine booting. After logging in, you will get to the emDistVM workspace.

Note - Some packaging programs have problems with very large (>1GB) zip archives. Please use [7-zip](#) if you encounter any problems.

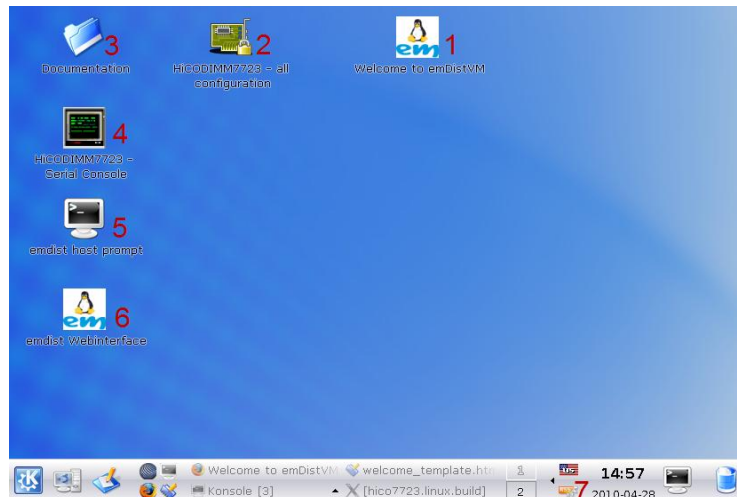
User account and host settings are as follows:

parameter	value
username	hico
user password	hico
root password	hico
hostname	emdistvm23
workgroup	emdistvm
machine name	emdistvm23
serial port	COM1 / ttyS0
Network configuration (IP address)	Assigned by DHCP

NOTE: You can change the used serial port (default COM1) in the configuration file `emdistvm23_hardy.vmx` located in the virtual machine directory. See chapter [Change serial port](#).

6.5 Starting emDistVM

After clicking on `emdistvm23_hardy.vmx`, the virtual machine is opened and the empty desktop appears.



Icons on the desktop provide following actions:

1. **Welcome to emdistVM:** This opens firefox and displays a small page describing the board and your IP settings, so you are able to configure the target to work with the virtual machine. It can safely be deleted when you are familiar with the environment.
2. Starts the CrossTarget server and opens the Ct web-interface with Firefox (Note that this icon uses the `all.cfg` configuration – you will need to modify the icon properties to start any other configuration)
3. The documentation folder contains documentation and prebuilt linux and rootfs flash images.
4. The serial console is the default interface to the target, e.g. for launching applications or configuration the target's network settings.
5. Opens a command console at the emDist top level directory.
6. If Firefox has been closed accidentally, this icon starts Firefox and connects it to CrossTarget.
7. Emtrion informs about new releases and updates via RSS feeds. A tool (`akregator`) is running on the emDistVM that will inform you about these news

The virtual machine is shipped with a pre-built NFS rootfs. Therefore it's possible to connect the target to the virtual machine and boot it from network without any further actions on the emDistVM.

In order to rebuild the NFS rootfs, all applications need to be recompiled as well as all object files have been removed from the emDistVM to make it fit on a DVD. Emdist will do it automatically for you when necessary. This may take up to a few hours when done the first time.

6.6 Exchanging data with the virtual machine

6.6.1 Copy and Paste

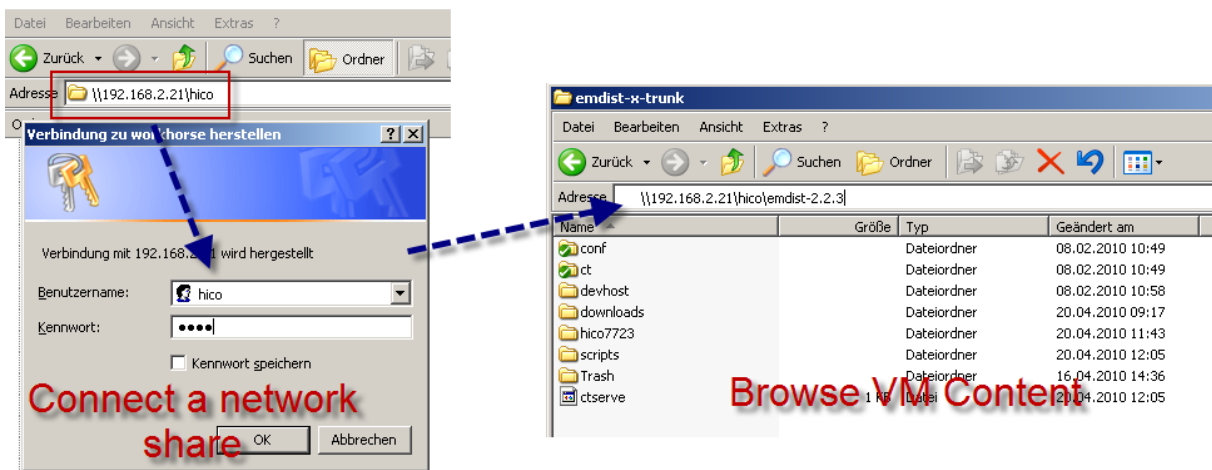
Clipboard function between the virtual machine and your Windows host requires the vmware-toolbox to be running on the VM. By default, vmware-toolbox is started at the VM boot time. If you happen to kill it you can start it again from a terminal window:

```
hico@emdistvm:~/ $ vmware-toolbox &
```

In order for the clipboard function to work, you need to start vmware-toolbox as a background process like above.

6.6.2 Samba

The VM uses samba to share the home directory of user `hico`, where the BSP is installed. You can open and explore all the samba shares by simply giving IP address of the virtual machine into the windows explorer (not internet explorer) address bar. The IP address of the virtual machine can be retrieved either by the command line tool `ifconfig` or by clicking on the "Welcome to emDistVM" icon.



You are prompted for username and password. Both are `hico`.

6.6.3 NFS

emDistVM has an NFS server enabled in first place for the NFS root mount (i.e. the target board mounts the root file-system over network). Windows doesn't have an NFS client included by default, there are, however, a number of commercial software for accessing NFS shares on windows. By default, only directory `/nfsroot/CORE_MODULE/rootfs` is exported and this is a link to the current target root file system of `$CT/CORE_MODULE/rootfs`. If you want to add more exports, you will need to add them to your `/etc/exports` manually.

6.6.4 Other Methods

- VMWare offers a method to share a folder between a Linux client and windows host. Configuration is not supported in the VMWarePlayer though. You will need to purchase VMWare Workstation or edit the VM configuration file manually. A tutorial how to do this you can find under address <http://www.visoracle.com/vm/debian40/sharedfolders.html>
- On the virtual machine you can of course start any file sharing server you wish (e.g. ftp, sftp) how to do this is out of the scope of this document.

6.7 Changing the virtual machine settings

If you use the free VMWarePlayer, you can change the image configuration by editing the main configuration file with a text editor. VMWarePlayer doesn't provide many tools to configure the virtual machine image. If you want to do the configuration more conveniently you will need to purchase VMWare Workstation.

6.7.1 RAM allocation

With the default 512MB of RAM, the VM is not well suited for heavy development applications like IDEs. In the VMware Player, you can change the amount of RAM for the virtual machine in menu "**Player -> Troubleshoot -> Change Memory Allocation**". Just drag the slide to the desired value and reboot the virtual machine. After changing the settings, but shut it down the virtual machine, close VMWarePlayer and start it again.

6.7.2 Change screen resolution

By default, the screen resolution is 1024x768. If you want to work with the VM in full screen mode, you probably want to change the resolution to fit your monitor resolution. In order to change resolution, proceed with following steps:

1. Open a Terminal on the Virtual machine and run command `vmware-config-tools.pl` as root (or with `sudo`):

```
hico@emdistvm:~$ sudo vmware-config-tools.pl
```

2. The script will ask you if you want to change the screen resolution. Answer 'yes' and select the desired resolution. To all the other questions the script asks; just give the default answer by pressing enter.

```
Do you want to change your guest X resolution? (yes/no) yes

Please choose one of the following display sizes (1 - 13):

[1]  "640x480"
[2]  "800x600"
.
.
[13] "2364x1773"
Please enter a number between 1 and 13:
```

3. After the script has finished, reboot the VM (System->Log Out->Restart the computer, or give command `sudo reboot`)

6.7.3 Change serial port

In order for the software in the VM to be able to use a serial port on your PC, make sure that you have the correct COM port in your VM configuration (*.vmx file in the VM directory). By default this is COM1.

```
serial0.fileName = "COM1"
```

7 First steps with emDist and your target board

Linux starter-kit boards are shipped with Linux installed in the flash. When powered on, they launch a Qt based demo application and are independent of the Virtual machine. In order to NFS boot the target or update kernel and file-system images, you need to set some boot-loader parameters and connect the board to the development host with an Ethernet cable and RS232 serial cable (null modem). Use the DSUB9 interface on the base board for the serial console connection. Serial port settings are:

```
115200 baud, 8 data bits, no parity, no HW or SW flow control
```

By default, the board configures its networking by DHCP and the `serverip` (i.e. IP address of the development host) is `192.168.105.3`. You, of course need to change this to correspond yours.

After connecting the cables, you can start the terminal program as described in chapter [Starting emDistVM](#). Then reboot the target and interrupt the booting into flash with Ctrl-C.

If you are using the HiCO.DIMM-Base, you need to press the yellow button on the board once after powering the board.

7.1 Configure the starterkit to use the virtual machine as NFS/TFTP server

By default, the target bootloader configures the network via DHCP. In order to download resources from the development PC, the target needs to know its IP address (this is not resolved by DHCP):

1. First determine the IP address of the virtual machine. The IP address is displayed in the welcome page of [Starting emDistVM](#) or can be retrieved on the console with

```
hico@emdistvm:~ $ ifconfig
eth0  Link encap:Ethernet  HWaddr 00:0c:29:3c:7c:f7
      inet addr:192.168.105.50  Bcast:192.168.105.255
```

2. On the target bootloader console, set the address to the `serverip` variable:

```
<uboot console>
CORE_MODULE # setenv serverip ip_of_the_vm
CORE_MODULE # saveenv
```

7.2 Update images on the target boards flash

Downloading and updating the system images is done from the boot-loader console. You can update the kernel image and root file-system image with commands `update linux nfs` and `update rootfs nfs`:

```
U-Boot 1.3.4em1-svn192 (Nov 12 2008 - 14:43:47)

[...]
Date: 1900-01-01 (Sunday) 05:13:21
Hit any key to stop autoboot: 0
CORE_MODULE# update linux nfs
BOOTP broadcast 1
DHCP client bound to address 192.168.105.25475
File transfer via NFS from server 192.168.105.50; our IP address is
192.168.105.25475
Filename '/nfsroot/CORE_MODULE/rootfs/boot/uImage-CORE_MODULE'.
Load address: 0x8e000000
Loading: .....
done
Bytes transferred = 1802828 (1b824c hex)
Calculated data checksum = 0xd5a39147
Erasing: complete
Writing: complete
Verifying: complete
Update successful
HICOXXXX # update rootfs nfs
BOOTP broadcast 1
DHCP client bound to address 192.168.105.25475
File transfer via NFS from server 192.168.105.50; our IP address is
192.168.105.25475
Filename '/nfsroot/CORE_MODULE/rootfs/boot/rootfs_CORE_MODULE_jffs2'.
Load address: 0x8e000000
Loading:
.
.
Bytes transferred = 2812900 (2aebe4 hex)
Calculated data checksum = 0x5b8f1d7e
```

As you can see from the boot-loader output, the images are fetched from `/nfsroot/CORE_MODULE`, which again, is a symbolic link to `$CT/CORE_MODULE` in the emDist installation. You can now start the system from flash with command:

```
CORE_MODULE# bootx linux
```

7.3 Run Linux kernel with NFS root mount

How the system boots – via network or from the boards flash – is defined by the bootlaoder environment variables:

Start system from flash:

```
CORE_MODULE # bootx linux
```

Start system via network (NFS root mount)

```
CORE_MODULE # bootx linux nfs
```

You can save the default method to the bootcmd variable:

```
CORE_MODULE # setenv bootcmd bootx linux nfs  
CORE_MODULE # saveenv
```

For more information on the boot-loader commands, please refer to the boot-loader manual.

7.4 Building custom target file-system image

In order to build a file-system image that can be downloaded into the boards' flash memory, use `CtScript CORE_MODULE.images.rootfs_(ubifs/jffs2)`.

You can answer yes when the system asks if the old (stripped) rootfs directory should be removed. After a successfull The root filesystem images is located in:

```
$CT/CORE_MODULE/rootfs/boot/rootfs_CORE_MODULE_(jffs2/ubifs)
```

Ubifs and jffs2 are Linux file-system types, of which ubifs is more efficient. Ubifs is default for emDist-2.3.1 and you should use it if it is enabled for your target.

7.5 A word on the Development model

A common problem in embedded Linux development is, that in the development phase you need much more flexibility in the data exchange between target and host, than the actual product requires when it's ready.

emDist is designed to use NFS mount for the development phase , which means that the target board and the development host use and 'see' the same root file-system directory (`CORE_MODULE/rootfs`). In the development phase you don't need to pay that much attention to the size of the root file-system because it is located on your host computer. No downloading or uploading with FTP or similar tools is required.

At some point when your application is ready and you are satisfied with the target system configuration, you want to build flashable file-system images and boot the system entirely from flash (transition from development into *stand alone*). emDist includes tools for conveniently turning the "development root file-system" into a lean flashable file-system.

8 Installing emDist on Ubuntu Linux

In this chapter we go through steps you need to take to get emDist/CrossTarget running on Ubuntu 8.04. Instructions how to install Ubuntu Linux (or any other Linux distribution) on your development PC is out of the scope of this document. Here are a few options you can use as starting point:

- Install Linux on a normal PC
- Use VMWare Workstation to install Linux on a virtual machine and run it on your Windows host.
- Download a ready configured VMWare Linux image (Appliance), run it with the free VMWarePlayer. You can find ready configured images at address <http://www.vmware.com/appliances/>.
- Use emDistVM where emDist is already built and configured. emDistVM is delivered only on DVDs from emtrion.

A note to emdistVM Users

If you are using the delivered virtual machine, all the necessary packages, the BSP and tool-chain are already installed and the NFS server is configured. In order to gain a better understanding of the system it is highly recommended though, that you read everything carefully.

8.1 Configure sudo

CrossTarget scripts use `sudo` in many places to run commands with root privileges. If you don't have `sudo` correctly configured on your system the build might fail or start to wait for root password at some point. On emDistVM, `/etc/sudoers` contains following lines, which enable users who belong to admin group to run root commands without a password:

```
# Members of the admin group may gain root privileges
%admin ALL=(root) NOPASSWD: ALL
```

We assume that you are not developing on a critical data server which is directly connected to internet. See manual pages for `sudo` for more information. Also see `/etc/sudoers` file in the emDist virtual machine.

NOTE: CrossTarget is not designed to be run as root! The CtScripts use `sudo` when necessary. Running everything as the root user is bad practice and may lead to unexpected behavior. Please configure `sudo` correctly instead.

8.2 Configure BASH to be the standard shell

By default, Ubuntu uses DASH shell as `/bin/sh`. DASH is a POSIX conform shell which more lightweight than BASH, which the default shell on most of the Linux distributions. There is one component in emDist which fails to install with DASH – the CodeSourcery toolchain for the target. Thus, you need to configure BASH to be `/bin/sh` (at least for the installation step). Use following command to do this and answer 'no' when `dpkg-reconfigure` asks if dash should be installed as standard shell:

```
$> sudo dpkg-reconfigure dash
```

If you want, you can switch back to DASH After you have installed emDist and the CodeSourcery toolchain. Use the same command and answer yes, when `dpkg-reconfigure` asks if dash should be configured as `/bin/sh`.

8.3 Download and run `prepare-emdist`

In order to checkout, and build emDist, give following commands on a console screen (you can skip this step if you are using the emDistVM virtual machine):

```
# Change to your home directory and download the prepare
# script and run it
$> cd ~
$> wget http://support.emtrion.de/emdist/prepare-emdist
$> sudo cp prepare-emdist /usr/local/bin
$> export EMTRION_SVN_USERNAME=XX
$> export EMTRION_SVN_PASSWORD=XX
$> mkdir work
$> cd work
$> prepare-emdist -t dimm-sh7724 -v tags/emdist-2.3.3
```

If you have a starter-kit DVD from emtrion, the SVN account information is printed on it. Contact emtrion if you don't have a starterkit DVD.

The script is designed to run through on an Ubuntu 8.04.2 „Hardy Heron“, Linux distribution. After the script has finished, you have a working emDist distribution in directory `emdist-2.3.3`.

8.4 Start the CrossTarget web interface

Script `$CT/ctserve` will generate the CtScripts and start the CrossTarget web interface.

```
$> cd emdist-2.3.1
$> export EMTRION_SVN_PASSWORD=XX
$> export EMTRION_SVN_USERNAME=XX
$> ./ctserve conf/CORE_MODULE/CORE_MODULE.cfg # or all.cfg
```

You can now open the web interface by giving address `http://localhost:8080` in your web-browser (in the virtual machine). You might want to add the `EMTRION_SVN*` variables to your `~/ .bashrc`, so that you don't need export them every time you start the server.

8.5 Configure NFS server

NFS root mount allows the Linux kernel on the target board to use `$CT/CORE_MODULE/rootfs` directly as its root file-system. This is a major advantage in development phase since you don't need to copy files to the target using protocols like ftp. Just copy any file into the `$CT/CORE_MODULE/rootfs` directory and the target board will "see it" immediately.

NFS is configured automatically by CtScript `CORE_MODULE.devhost.configure_nfs`. Before you can configure it, you need to have a NFS server installed on your system. How to setup the server depends on the Linux distribution you are using and you should consult your distributions documentation on this. For example, on SuSE you would run Yast (SuSE configuration tool), go to the "Network Services" and Click on the "NFS Server" icon. The configuration wizard asks for the exported directories and parameters.

The NFS configuration should look something like following in your `/etc/exports` file:

```
# This is the 'dirty' root filesystem with C headers,  
# manual pages etc.  
/nfsroot/CORE_MODULE/rootfs 0.0.0.0/0.0.0.0\  
    (rw,all_squash,anongid=1000,anonuid=1000,sync)  
  
# This is the cleaned up version. The downloadable target  
# system images is a copy of this directory (this share is optional)  
/nfsroot/CORE_MODULE/rootfs/boot/rootfs 0.0.0.0/0.0.0.0\  
    (rw,all_squash,anongid=1000,anonuid=1000,sync)
```

Note that here we use `all_squash`, `anongid` and `anonuid` to map all filesystem operations to user `hico`. Numerical user and group ID 1000 is user `hico` on the emDist virtual machine. If you are using some other user account, you need to replace these with your own user and group IDs (use commands `id -u` and `id -g` to find out your numeric gid and uid)

Note that you need to avoid creating any files/directories into the targets root file-system on your development host as the root user. To avoid problems, do not run CtServer or any CtScripts as root. `sudo` is used in the CtScripts only when it is absolutely necessary.

8.5.1 Testing if NFS share is working

You can make sure that your NFS server is working by mounting the exported directory locally. You should see the contents of `$CT/CORE_MODULE/rootfs` in the mount point:

```
$> cd ~
$> mkdir test
$> sudo mount -t nfs -o nolock localhost:/nfsroot/CORE_MODULE/rootfs test
$> ls test
.  .. bin dev etc lib mnt proc sbin sys tmp usr var
$> sudo umount test
```

Don't forget to check the system messages if you get stuck. A normal cause for the root mount to fail is the firewall or other security settings on the development host.

```
# Get last 5 system messages
$> dmesg|tail
```

8.6 Installing emDist on other Linux distributions

In order to install emDist/CrossTarget on another Linux distribution than Debian/Ubuntu you should have a basic knowledge of Linux and you should be able to install packages and do simple administrative tasks. EmDist is designed to work on any modern Linux distribution. **However, due to the great number of different Linux distributions out there, it cannot be assured that everything always will work "out of the box"**. Once you get the right packages installed from your Linux distribution, there shouldn't be any further problems though.

Script `prepare-emdist_2.3.1.sh` is designed for Debian base Linux distributions (such as Ubuntu). If you want to install emDist on other distributions (like SuSE or Redhat), you need to do the steps executed in `prepare-emdist_2.3.1.sh` manually. Use `prepare-emdist_2.3.1.sh` as reference, or modify the script to run on your system.

After you have checked out the emDist sources from emtrions subversion server, you will need to run `devhost.first_run.do`, `CORE_MODULE.devhost.configure_all`, `CORE_MODULE.packages.do_all`, In the given order to have everything built up and installed.

9 CrossTarget and CtScripts

CrossTarget is the engine behind emDist (emtrion Linux distributrion). The two main purpose of CrossTarget are to generate CtScripts and provide a comfortable Web-Interface for running them. CtScripts are not dependant on the web-interface. All information CrossTarget uses to build the CtScripts and the web interface, are stored in configuration files which have a simplistic and powerful syntax.

You could think of CrossTarget web-interface as an *interactive command reference*. It shows you all the CtScripts in an easily understandable fashion and lets you run the scripts directly without having to give any command line commands. The web interface also shows you the status of the software packages.

Packages are built and images are generated with CtScripts. CtScripts are BASH shell scripts, which are automatically generated from the content in the package and target configuration files. They are run under a special environment with strict error checking and helper functions to produce informative output. The scripts can be run from command console or from the CT web interface. All CtScripts are generated into `$CT/scripts` directory and named after the nodes *keypath* where the script was defined (e.g. `dim-sh7724.linux.build`, `dim-sh7724.linux.configure` etc.)

All the scripts you can run from the web command reference, you can also run directly from command console. Here are two examples how to run the same script `core_module.linux.do`, which will build and install the Linux kernel.

9.1 Starting the CrossTarget server

Cross target is always started with a main configuration file as argument. For example:

```
$> cd emdist-2.3.1
$> ./ctserve conf/hico7723/hico7723.cfg
```

The main configuration file pulls/includes all other configuration files which are needed for the target. After reading the data from the `*.cfg` files, it expands any macros in the values and creates a `node tree` of the data. This node tree is then used to generate the CtScripts into `$CT/scripts` directory. If `--cfg-only` is not given, CrossTarget will also start the web-interface server on port 8080.

9.2 Running CtScripts from the web interface

CtScripts which are started from the web-interface are executed in a xterm window.

Busybox 1.12.4

Resources: [website] [documentation] [SOURCE] [BUILD]

Package status: Prepared Configured Built Installed

Description

The Swiss Army Knife of Embedded Linux

Busybox is a collection of Linux/Unix Shell utilities (ls, grep, cp...), which build the base for a proper Shell environment. It also contains some more sophisticated utilities such as a simple Web-server.

CtScripts

- hico7723.busybox.do Run
- hico7723.busybox.prepare Run
- hico7723.busybox.install Run
- hico7723.busybox.headers_install Run
- hico7723.busybox.cscope Run
- hico7723.busybox.make_modules Run
- hico7723.busybox.remove Run
- hico7723.busybox.update Run

```

hico7723.linux.do
# Test that the package is built...
test -e /home/hico/emdist-x-trunk/hico7723/pkgs/linux-sh-2.6.xx-build/vmlinux || \
Abort "linux is not yet ...
# Install modules to the root filesystem
make modules_install \
ARCH=sh \
CROSS_COMPILE=sh-linux-gnu- \
INSTALL_MOD_PATH=/home/hico...
make -C /home/hico/emdist-x-trunk/hico7723/pkgs/linux-sh-2.6.xx-0=/home/hico/emdist-x-trunk/hico7723/pkgs/linux-sh-2.6.xx-build modules_install
INSTALL drivers/misc/itc2440_adc.ko
INSTALL drivers/scsi/scsi_wait_scan.ko
DEPMOD 2.6.33-rc1em0-svn902
# Copy the kernel image to the boot directory in the root
# filesystem, which is normally the place for kernel images
cp arch/sh/boot/uImage \
...
hico7723.linux.install OK
Run-once hico7723.linux.headers_install OK
Running hico7723.linux.headers_install
hico7723.linux.headers_install SKIPPED
hico7723.linux.do OK
Press 'q' to to exit or 'r' to repeat[]
  
```

CrossTarget | LICENCE | Support | Copyright © emtrion GmbH

After running a script, the user is asked if the script should be repeated. This is useful when making changes in a project or software package. For example, you can repeat the packages 'do' method with just one key press to have it built and installed every time you change something in the sources.

9.3 Running CtScripts from command line

All CtScripts are located in the `$CT/scripts` directory. The scripts are position independent and there are no command line parameters or required environment variables. Example:

```
$> ./scripts/CORE_MODULE.linux.do
```

On command line, you can easily run multiple ctscripts as batch by joining them with the `&&` operator. Say, you are making some changes in the Busybox source code, you could build, install and start a serial console to the target with following 'batch':

```
$> CORE_MODULE.busybox.build && \  
    CORE_MODULE.busybox.install && \  
    CORE_MODULE.connect.serial
```

9.4 Bash command line completion for CtScripts

If you want to use command line for executing scripts you might want to add `ct/util/ct-bash-helpers.sh` to your environment. After adding them you will have two helper functions with bash completion to start the CtServer (`CtServe`) and run CtScripts (`CtRun`). You can try them with following commands:

```
$> . $CT/ct/utlils/ct-bash-helpers.sh # add the helpers to env  
$> CtServe <tab>  
$> CtRun <tab>
```

You can run multiple scripts with one `CtRun` command:

```
$> CtRun CORE_MODULE.linux.build CORE_MODULE.linux.install
```

If any of the commands fail, the command chain is aborted.

9.5 CrossTarget files and directories

The directories can be divided into static directories (i.e. the directories which are there when you checkout a fresh version of the emDist sources) and dynamically generated directories (generated when emDist is run for the first time)

9.5.1 Static files and directories

Static files and directories are checked out from the emtrions subversion server.

Directory/File	Description
<code>\$CT/conf</code>	Contains all the configuration files (package configuration, target board configuration, build commands etc.)
<code>\$CT/conf/CORE_MODULE/CORE_MODULE.cfg</code>	Minimal default configuration for <code>CORE_MODULE</code>
<code>\$CT/conf/CORE_MODULE/all.cfg</code>	Adaptation layer for <code>CORE_MODULE.cfg</code> which contains most of the available packages.
<code>\$CT/ct</code>	CrossTarget source files. You should not change anything here.
<code>/nfsroot/CORE_MODULE</code>	This is the location from where the bootloader and Linux kernel expects to find all the target resources. <code>/nfsroot/CORE_MODULE</code> is a symbolic link to the <code>CORE_MODULE</code> directory in the emDist installation.

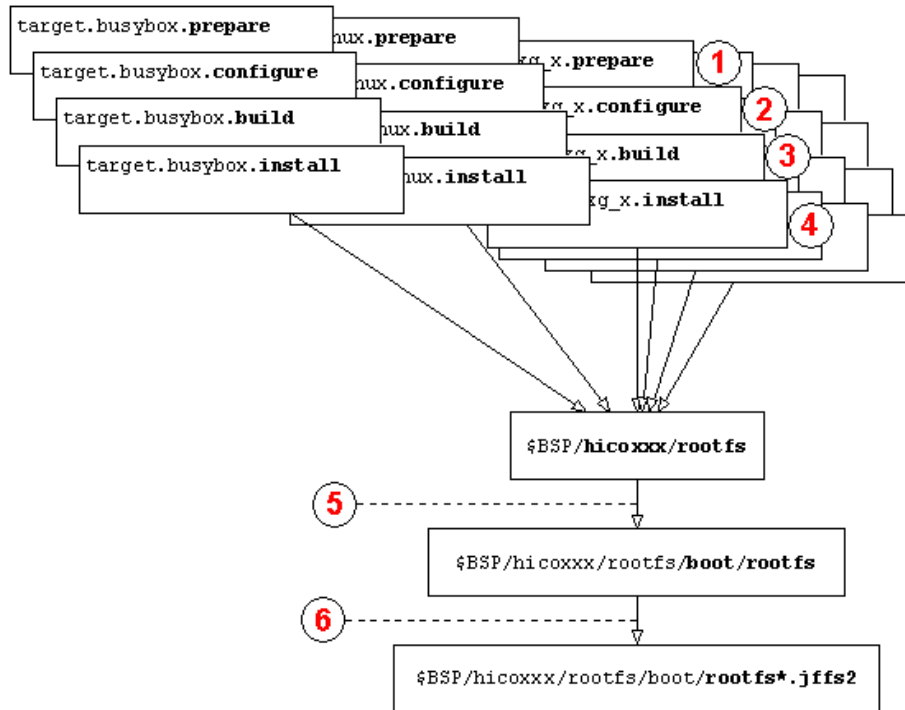
9.5.2 Dynamic files and directories

Dynamic files are created by CrossTarget at run-time. They are not under version control.

Directory/File	Description
<code>\$CT/CORE_MODULE/pkg</code> s	Package source directories for the target board
<code>\$CT/CORE_MODULE/rootfs</code>	NFS mountable root file-system for the target. All the packages are installed here
<code>\$CT/CORE_MODULE/rootfs/boot</code>	This is a special directory under <code>rootfs</code> , which is not included to the flash images. The flash file-system image and kernel images are installed here so that they are accessible from the target when using NFS mount.
<code>\$CT/CORE_MODULE/rootfs/boot/rootfs</code>	This is the stripped version of the root file-system. It contains only the run time files for the installed packages (i.e. no man pages, header files etc.)
<code>\$CT/scripts</code>	The CtScripts are generated into this directory
<code>\$CT/Trash</code>	Removed packages are moved here
<code>\$CT/downloads</code>	The package tarballs, patches and everything what is downloaded from internet comes here. TIP: if you have many emDist installations, it is better to have a global downloads directory somewhere and change <code>\$CT/downloads</code> to be a symbolic link which points there.

9.6 System build – the big picture

The following picture describes the system build from the individual source package into the file-system image file.



Every package has four compulsory commands; *prepare*, *configure*, *build* and *install*;

- (1) *prepare* – Downloads the package (normally a tar.gz file), unpacks it, applies any patches and does any other package specific preparation.
- (2) *configure* – This step normally includes running the packages ‘configure’ script with some target specific parameters (usually just the cross-compiler prefix and installation directory).
- (3) *build* – This command compiles/builds the package. Usually it just runs ‘make’
- (4) *install* – This command installs the resulting binaries into the targets root file-system. This is often a ‘make install’
- (5) After all the desired packages are installed, we have a working root file-system for the target board in the `$CT/CORE_MODULE/rootfs` directory, which can be NFS mounted by the target. To get a working flashable image we run the command `CORE_MODULE.images.rootfs_jffs2` which makes a copy of the `$CT/CORE_MODULE/rootfs` directory into `$CT/CORE_MODULE/rootfs/boot/rootfs`, leaving some unwanted directories and files on the way. Now we have a file-system which is still fully functional but reduced in size.
- (6) The final file-system image is made from directory `$CT/CORE_MODULE/rootfs/boot/rootfs`.

You might wonder why the copy is made inside the original `rootfs` directory tree (i.e. into `$CT/CORE_MODULE/rootfs/boot/rootfs`)? The reason is that this way we can access it from the target when the `$CT/CORE_MODULE/rootfs` is NFS mounted. You can, for example, chroot to the

stripped down `boot/rootfs` directory or use it as NFS root directory to check that everything works ok.

10 Target configuration files

A target configuration file is the main information source for your HW target. It defines the packages you want to include to the target file-system as well as the used cross compiling tool-chain.

10.1 Where target configuration files are kept?

Normally, all targets have their own directory under the `conf` directory (e.g. `emdist-2.3.1/conf/CORE_MODULE/`) and under the target directory are located one or more target configuration files, target specific package files, patches etc.). Here is an example of the target/board specific directory:

```
~/emdist-2.3.1$ ls conf/CORE_MODULE/  
all.cfg  CORE_MODULE.cfg  patches  pkgs
```

The `patches` directory contains some target specific modifications for the target packages and `pkgs` contains target specific packages. There are no official rules how you should keep your board specific files though.

10.2 Main target configuration files

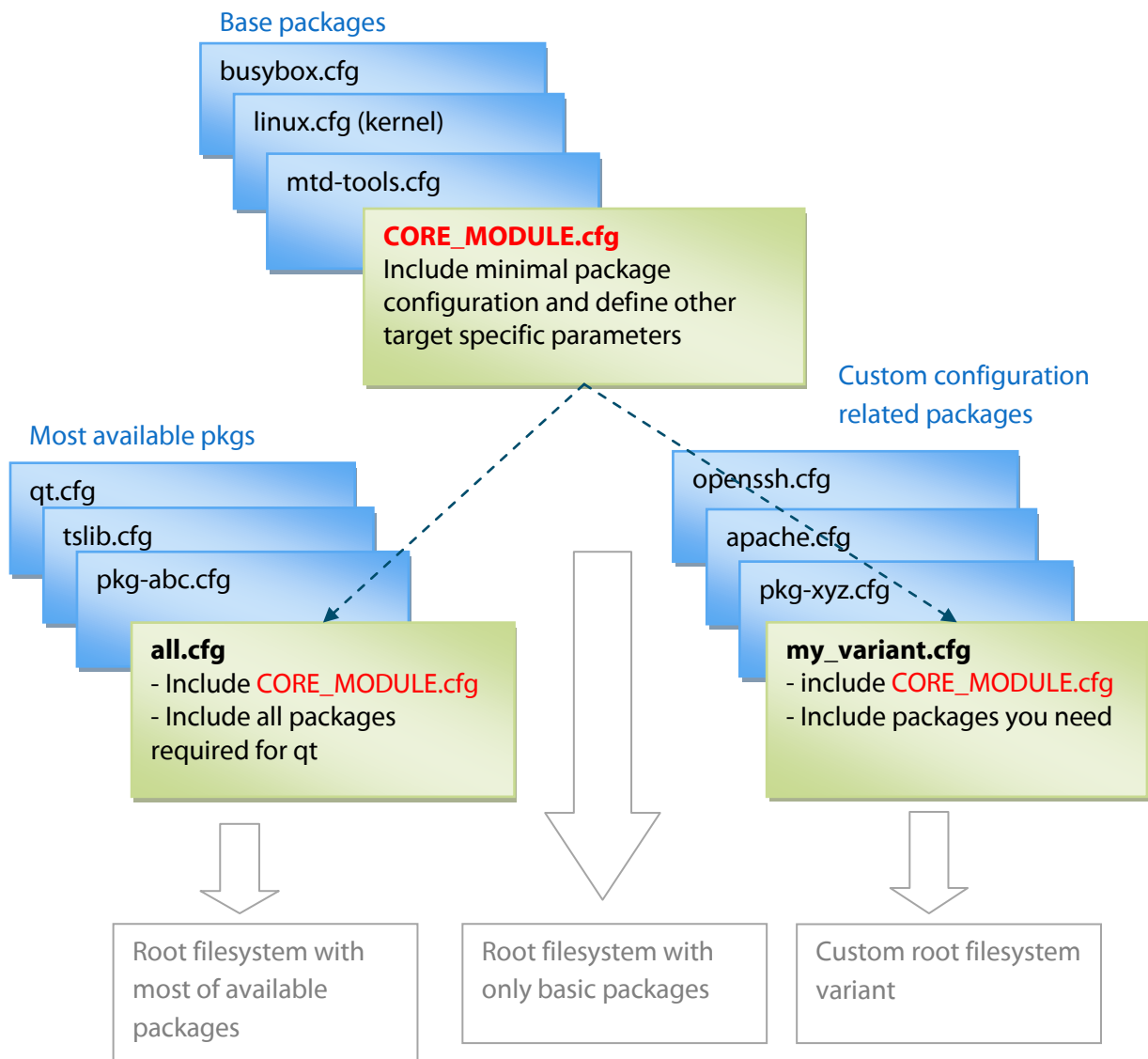
The main configuration file is always the `*.cfg` file named with the board code. For example `conf/CORE_MODULE/CORE_MODULE.cfg`.

Avoid changes values or configuration in the main target configuration file. Use an adaptation layer as described in the following chapters instead.

10.3 Adaptation layers (root file-system variants)

Thanks to flexible file inclusion, you can easily create diverse root file-system variants. For example, `$CT/conf/hico7723/all.cfg` and `$CT/conf/hico7723/enlightenment.cfg` are both adaptation layers that include the main target configuration file `hico7723.cfg` and then add some packages to it. This enables you to have many target root file-system variants.

Following diagram demonstrates how the variant configuration files are derived from the main target configuration file.



10.4 Creating custom target file-system variants

If you want to add some packages, you can create your own *adaptation layer*, include the main target file and add your adaptation there (you can also override the values defined in the main target configuration file). Here is an example of a simple adaptation layer (or variant) configuration.

```
# Include the main target cfg file
. < hico7723.cfg

[target.hico7723]

# Important! Set the variant string. This string is
# attached to the resulting root file-system (e.g.
# hico7723/rootfs.my_conf)
variant = my_conf

# Add some custom packages
[.pkg.mypkg < ../common/pkgs/target/mypkg.cfg ]
.
.
```

There are two approaches to create your own adaptation layer – either you create an empty layer like above or you make a copy of `all.cfg`, change `variant` from `all` to something else and start to reduce or add packages.

Good practice is to try to avoid making changes in the main target configuration file or the 'higher level' configuration files in general. You can always override all the values there in the adaptation layers.

10.5 Adding packages to your configuration

In the easiest case it is just a matter of including the corresponding file into your adaptation layer:

```
# Add a custom package  
[ .pkg.mypkg < ../common/pkgs/target/mypkg.cfg ]
```

In many cases, however, there are some parameters in the package which need some tweaking. This can be achieved by *overriding* the original values. Let's say that you want to include package libpng to your target, but you want to have a newer version of it. In that case you can override the pkg_url parameter of the package:

```
# Add a custom package  
[ .pkg.libpng < ../common/pkgs/target/libpng.cfg ]  
# Override the default package url  
pkg_url = http://prdownloads.sourceforge.net/libpng/libpng-1.2.32.tar.gz
```

You need to use the package configuration file as reference for the parameters, which you need to override.

10.6 Package dependencies

Packages are configured, built and installed in the order they are included in the main configuration file. You can see in `all.cfg` that the low level libraries like compression libraries are included first and the complex packages like Qt are included at the end of the file.

11 Package configuration files

The software packages in emDist can be roughly divided into two categories; *base packages* and *extension packages*. Base packages are the bare minimum to have a working Linux system. The base consists of root file-system skeleton with minimal set of target configuration files and scripts, GNU C/C++ run-time libraries, Linux Kernel and Busybox. All the rest of the packages can be considered as extension packages. They are optional and you can include them into your target system depending on your needs. Most commonly included extension packages are such like SSH/SSL for secure communication with the target, graphics library (e.g. Qt) and a web-server.

TIP: Always check if the required function or utility program is already included into busybox. For example, if all you need is a simple web-server you don't necessarily have to configure and cross-compile Apache for the task - you can use the one in

Before go through how the software packages are managed in emDist, it is important to know some theory on cross compiling open source software...

11.1 A word on cross compiling open source software

One important reason for using Linux on your embedded system is the vast resources of ready open source software. Using the open source software packages, however, is not a matter of just downloading them to the target – they need to be cross compiled for the target architecture and this is where things often get hairy. Most packages which comply to the GNU package convention (i.e. `./configure,make,make install`) are rather straightforward to cross compile. There are, however, many SW packages out there which were never meant to be cross compiled and thus getting them running on your embedded target can sometimes mean days of work.

Normally when you build GNU-compatible software on your development host, you just do the typical steps

```
$> configure
$> make
$> make install
```

after which the compiled application is ready and installed on your system (Normally you would use your distributions package manager of course, but sometimes you also need to compile something yourself)

When compiling for a different architecture, however, you have to take care that the cross compiler is used instead of your host compiler. For GNU compatible packages this means giving the machine code or compiler prefix with the `--host` option

```
$> ./configure --host=sh4-unknown-linux-gnu -prefix=/usr
```

From the `--host=sh4-linux` option, the configure script knows to use `sh4-linux-gcc` cross-compiler. Depending on your target, the prefix might also be named `arm-softfloat-linux-gnu`, `sh3-unknown-linux-gnu`, `sh4-linux` etc. The `--prefix` option tells where the resulting binaries are located in the target root file system, from the targets point of view (do not give an absolute path to the root file-system here. See side-box below).

We still have one problem, however. The configure script doesn't have a clue where to look for header files and libraries of any other software packages. For this, we need to pass the target specific header and library paths to the compiler. This is normally done with the `CFLAGS` and `LDFLAGS` environment variables. By adding these our configure command looks as follows:

```
$> LDFLAGS="-L$ROOTFS/usr/lib -Wl,--rpath-link -Wl,$ROOTFS/usr/lib" \  
CFLAGS="-I$ROOTFS/usr/include" \  
./configure --host=sh4-unknown-linux-gnu -prefix=/usr
```

With these flags we tell the compiler to look include files in `$ROOTFS/usr/include` and static/run-time libraries in `$ROOTFS/usr/lib`, where `$ROOTFS` is pointing to our targets root file-system. Note also that here we avoid exporting variables with the `export` command by giving the variables on the command line before the command call.

Ideally this is all you have to do. In reality, things are often not that simple. Depending on the complexity of the package, there are often other parameters to take into account as well. Very often you need to disable some features because of missing packages. For example, if you don't need X-server, but you need to cross-compile a package with some X specific features, you need to pass something like `--disable-x` to the configure script. In order to find out which options can be enabled/disabled you can run the packages configure script with `--help` option to get a list of all parameters to tweak.

```
./configure --help | less
```

If the package was configured properly, you normally just need to run `make` to cross compile it. For installing (`make install`), the `Makefile` normally has an environment variable which can be used to point the destination directory. By GNU convention, this is `DESTDIR`. In many cases however, you also need to look for a suitable variable or patch the `Makefile`.

```
$> make
$> make install DESTDIR=$ROOTFS/usr
```

For software packages which are not compliant with the GNU build system (i.e. `configure` and `make`), you normally have to modify the packages `Makefile` directly. It is impossible to say a general rule for all the software out there, but the idea is the same - you have to tell the build system that you are cross-compiling and that you want to install the resulting binaries into `$ROOTFS`

Correct usage of the `--prefix` argument

In many examples in internet you will see that the `--prefix` argument is set to an absolute path value on the development host (for example `--prefix=/home/foo/mylinux/rootfs/usr/`). This is, however not preferred since these absolute path values often get hardcoded into the compiled configuration files and binaries. As a result you get run time errors like `"file /home/foo/mylinux/rootfs/etc/hostname doesn't exist"` on the target.

If possible use a value for `--prefix` **that is meaningful for the target** and use the `DESTDIR` variable to tell where to install the resulting binaries (i.e. `make install DESTDIR=$ROOTFS`).

11.2 CrossTarget package configuration files

emDist/CrossTarget delivers a constantly growing selection of ready configured SW packages. The build *recipes* are stored in simple configuration files. Every package has its own *.cfg file and packages are added to the target build by including these files in the target configuration file.

While most of the packages are for the embedded target, some packages are only needed on the development host (such as the cross compiler). Thus, there are two kind of package configuration files, the ones for your development host (devhost), which are not cross compiled and the ones for the embedded target. Following table shows the location for the configuration files.

All configuration files are located under `$CT/conf` directory. Under `conf`, every target board has its own subdirectory, under which all the target specific patches and configuration files are located (for example `conf/hico7723`).

Directory	Description
<code>conf/common/pkgs/target/*.cfg</code>	Target package configuration files (i.e. packages to cross compile)
<code>conf/common/pkgs/devhost/*.cfg</code>	SW packages which are only meant to be used on the development host (for example cross compiling tool-chain and Gnu debugger GDB)
<code>CORE_MODULE/pkgs/*.cfg</code>	Packages which are only meant to be built for a specific target board (target specific packages)

11.3 A simple local package configuration file

Following example shows a minimalistic configuration for a local package `myapp` which is located in the users' home directory. The package directory has some source code files, `Makefile` to build the application and the resulting binary is called `myapp`.

```

title           = Myapp
desc           = My application
SOURCE         = /home/hico/myapp

flag.prepared  = @(SOURCE)/Makefile
flag.configured = @(SOURCE)/Makefile
flag.built     = @(BUILD)/myapp
flag.installed = @(ROOTFS)@(PREFIX)/bin/myapp

[.ctscript.do.cmd]
  common = @(do_common)

[.ctscript.build.cmd]
  common = @(build_common)
  % = make CC=@(CC)

[.ctscript.install.cmd]
  common = @(install_common)
  % = cp @(BUILD)/myapp @(ROOTFS)@(PREFIX)/bin/

```

- The `title` and `desc` fields should self explanatory – they are only informative parameters, which are not used in the build process.
- The flag files are used to deliver information on the packages status (i.e. is it built and installed etc.)
- We set the `SOURCE` to point to the top level directory of the SW project. If `BUILD` is not given, it is set automatically to same path as `SOURCE`.
- We use the `Makefile` as the prepared and configured flag. This means that if `@(SOURCE)/Makefile` exists, CrossTarget knows that the package is prepared, installed and ready to be built.
- We left `configure`, `prepare` and `remove` methods undefined since we don't need them for such a simple project. You will notice that the corresponding CtScripts are still created – they do nothing though. These scripts must exist but they don't necessarily need to do anything. Note that you always need to define the corresponding flag file for prepare, configure, build and install, even if you leave the method empty.

11.4 A more complex example

Let's look at one of the packages which include the `prepare` and `configure` methods as well. You need these methods if you want to add some open source package to emDist. Libxml is an example of a package which complies well with the GNU SW convention:

```
pkg_url      = ftp://xmlsoft.org/libxml2/libxml2-sources-2.7.2.tar.gz
website     = http://xmlsoft.org/index.html
title       = libxml @(version)

SOURCE      = @(PKGS)/libxml2-@(version)
BUILD       = @(SOURCE)

flag.prepared    = @(SOURCE)/configure
flag.configured  = @(BUILD)/Makefile
flag.built       = @(BUILD)/libxml2.la
flag.installed   = @(ROOTFS)@(PREFIX)/lib/libxml2.la

desc = XML Library

[.ctscript.do.cmd]
% = @(do_common)

[.ctscript.prepare.cmd]
common = @(prepare_common)

[.ctscript.configure.cmd]
common = @(configure_common)
% = """
CFLAGS="@(CFLAGS)" \
LDFLAGS="@(LDFLAGS)" \
./configure \
--host @(MACHINE) \
--prefix @(PREFIX) \
"""

[.ctscript.build.cmd]
common = @(build_common)
% = make

[.ctscript.install.cmd]
common = @(install_common)
% = make install prefix=@(ROOTFS)@(PREFIX)
% = fix-libtool-archive $(find @(ROOTFS)@(PREFIX)/ -name '*xml2*.la')

[.ctscript.remove.cmd]
common = @(remove_common)
```

Here are some remarks on the above configuration

- `Configure` script serves as the prepared flag. `Makefile` serves as the configured flag because `Makefile` is generated by `configure`. If the package is not based on GNU `autotools` you will need to find some other files for this purpose. Normally you would use the resulting application/library binary as built and installed flags.
- When configuring, we pass all the compiler and linker flags in the environment to the `configure` script. This is not always necessary, but recommended. `configure` tests whether certain header files etc. are present by compiling simple c files. If `CFLAGS` doesn't point to the include directories in `$ROOTFS`, `configure` assumes that the given header file or package is not installed and issues an error or configures to package to be built without the given feature.
- In many packages you will see in `install` script uses `fix-libtool-archive`. This is a CrossTarget tool to fix paths in the installed libtool archive files (`*.la`). In some references it is also called 'libtool slaying'.

In following chapters we look at the possible package parameters in more detail.

11.5 Package parameters

Here are listed the most important parameters which are used in the package configuration. Many of them are generated automatically, if not given explicitly.

Parameter/Macro	Explanation
<code>pkg_url</code>	This is the most important information for the package. The package files need to be <code>*.tar.gz</code> , <code>*.tar.bz2</code> for CrossTarget to be able to open them.
<code>pkg_name</code>	This is the name for the directory under which the package gets extracted. <u>Value is parsed from <code>pkg_url</code> if not given explicitly.</u>
<code>version</code>	If not given explicitly, the <u>value is parsed from the <code>pkg_url</code></u> . If the package file doesn't have a version number or if the version value is too exotic this might fail however. For now, the version number is only used for informative purposes.
<code>title</code>	Informative title for the package. <u>Generated automatically if not given explicitly.</u>
<code>desc</code>	A short description of the package.
<code>doc_url</code>	URL to the packages documentation. This can be a local file or directory (<code>file:///xxx</code>), or an external resource (<code>http://xxxx</code>).
<code>man</code>	If the package has a Unix man page, this is a local path to the manual page file.
<code>SOURCE</code>	Absolute path to the directory where the package sources are located. If not given explicitly <code>SOURCE</code> is automatically set to <code>@(CT)/CORE_MODULE/pkgs/@(pkg_name)</code> .
<code>BUILD</code>	Build directory for the package. If not explicitly given, <code>BUILD</code> is automatically set to <code>SOURCE</code> . Some packages, for example Linux kernel sources are built in a separate directory.
<code>flag.*</code>	These are pointing to flag files, which tell to CrossTarget something about the status of the package. Packages <code>prepare</code> , <code>configure</code> , <code>build</code> and <code>install</code> methods need to be run in the given order. Non-existence of one of these files will abort the process. Modification times of the flag files is also used to decide for example if the installed binary in the <code>ROOTFS</code> is up to date.
<code>flag.prepared</code>	If this file exists, CrossTarget assumes that the given package is prepared. You can normally use the <code>configure</code> file.

flag.configured	If this file exists, CrossTarget assumes that the given package is configured. Normally you can use <code>Makefile</code> of the, since <code>Makefile</code> is generated by the <code>configure</code> script.
flag.built	If this file exists, CrossTarget assumes that the given package is built. Normally you can use the resulting binary in the BUILD directory (i.e. <code>@(BUILD)/myapp</code>)
flag.installed	If this file exists, CrossTarget assumes that the given package is installed. Normally you can use the resulting binary in <code>@(ROOTFS)</code> . For example <code>@(ROOTFS)/bin/myapp</code> .
tfile.*	<code>tfile</code> is abbreviation of 'target file'. With <code>tfile</code> nodes you can add some configuration files and scripts to the root file-system. You also need to add <code>@(pkg:install_files)</code> macro to your install method.
tfile.*.path	This is the path for the file from the targets point of view (for example <code>/etc/myfile.conf</code>)
tfile.*.content	This will be the content of the generated file. If you add a shebang for any interpreter (i.e. <code>#!/bin/sh</code> etc.) the file will be marked as executable. Example: <pre>tfile.foo.path=/bin/myscript tfile.foo.content=""" #!/bin/sh echo "hello world" """</pre>

11.6 Commonly used Macros

A macro can be one of the above values, or a value which is defined in the target configuration. Here are the most common ones used

Parameter/Macro	Explanation
MACHINE	This is the machine string identifier, which is needed for cross compilation. This matches the string what gcc prints when given the '-dumpmachine' command line option. This is defined in the target configuration file.
CC	This expands to the gcc compiler command (for example <code>sh4-linux.gcc</code>). Note that <code>@(MACHINE)-gcc</code> this doesn't necessarily <code>@(CC)</code> . It is safer to use <code>CC</code> when calling the compiler.
CXX	Like above, but this expands to the C++ compiler.
LDD	Expands to the cross-linker tool. Using the linker directly is rarely required.
SAFE_CC	<code>SAFE_CC</code> is a wrapper for the default <code>CC</code> which verifies the path names given as command line argument for the compiler. It removes any include paths which point to the development hosts include/library directories (for example <code>-I/usr/include</code>) and outputs a warning when this is done. Generally, when cross compiling SW, <i>all</i> the header files and libraries should be searched <i>only</i> from the directories under the target root file-system <code>@(ROOTFS)</code> .
PREFIX	This is a common prefix for all the target packages (except for the most important ones, like busybox). By default it is set to <code>/usr</code> .
CFLAGS, LDFLAGS	These macros contain correct linker and compiler flags for cross-compiling for the target. These are not always needed – it is recommended to use them though.
ROOTFS	This expands to absolute path to the target root filesystem (for example <code>/home/ny/emdist-2.2.2/hico7723/rootfs</code>)
*_common	These are some predefined shell commands, which are used in all packages. In some (rare) cases it is necessary to override these, but in general you should add them to the package methods.

11.7 Standard Package Methods (prepare, configure, etc.)

Standard methods for packages are `do`, `prepare`, `configure`, `build`, `install` and `remove`.

Method	Description
<code>do</code>	This is a combination of <code>prepare</code> , <code>configure</code> , <code>build</code> and <code>install</code> macros. 'doing' a package means, taking care of everything from downloading from internet to installing the binaries to the target root file-system. The <code>do</code> method almost always consists of only one entry; <code>do_common</code> . If you look at the content of <code>do_common</code> in the web interface for any package, you will see that it takes care of the build logic. <code>do_common</code> makes use of the packages <code>flag.*</code> parameters to decide if a package needs to be built installed etc.
<code>prepare</code>	Download the target package and apply any emDist specific patches to it. You should normally add <code>prepare_common</code> macro, since it automatically handles the most common packages (downloading and unpacking <code>tar.gz</code> and <code>tar.bz2</code> files as well as checking out files from subversion and git repositories). If you need to do any patching for the package, this should be also done in the <code>prepare</code> method.
<code>configure</code>	Configure the package. This means normally running the GNU configure script (<code>./configure --hostXX</code>). On 'non gnu compatible' packages this method might include some other corresponding command or if the package doesn't have to be configured in any way this method can be left undefined.
<code>build</code>	Compilation of the source code (normally a <code>make</code> command).
<code>install</code>	Installation of the resulting binaries (normally a <code>make install</code> command)
<code>remove</code>	The package SRC directory will be moved to <code>emdist-x/Trash/</code> . CrossTarget doesn't actually remove the package source, it just moves it to <code>\$CT/Trash/</code> . In order to empty the Trash can, you need to remove the packages from Trash with command: <pre>cd \$CT; rm -rf Trash/*</pre>
<code>update</code>	This method is only used on packages which are checked out from version control repositories (subversion or git).

If you don't have a corresponding command for any of these methods you should leave them undefined. For example if the package is a binary package and there's nothing to build, you should not include a `build` method in the configuration file.

The content for all standard methods are normal shell commands. Commands can be easily added with the enumeration operator `%`. Here is an example of the install method.

```
[.ctscript.install.cmd]
# Include common routines
common = @(install_common)

# Add package specific install commands
% = make install DESTDIR=@(ROOTFS)
```

11.8 Optional/Custom package methods

In addition to the standard methods described above, packages can have any number of custom methods (an alternative install or build etc.). There are no limitations what kind of methods (or CtScripts) you add to the packages configuration. Here is an example of a dummy custom method

```
[pkg.foobar]
[.ctscript.my_method.cmd]
% = echo "hello my_method!"
% = ""
  cd @(SOURCE)
  echo "We are in @(pkg_name)'s source directory"
  ""
```

After reloading CrossTarget you will find the corresponding CtScript under `$CT/scripts`:

```
$> ~/emdist/scripts/CORE_MODULE.foobar.my_method
Running hico7723.strace.my_method
> echo "hello my_method!"
hello my_method!
> cd /home/hico/emdist/CORE_MODULE/pkgs/foobar-1.2.3
> echo "We are in strace-4.5.18's source directory"
>
We are in strace-4.5.18's source directory
hico7723.strace.my_method OK
.
```

11.9 Patching package sources

Many of the packages in emDist require some patching before they can be built. In emDist, Patching is done with normal diff patches and the patch utility. Patches can be added directly to the package configuration files (*.cfg) or they can be kept in separate files.

11.9.1 Keeping a patch in the package configuration file (rfile)

The patch has to be added as `rfile` (*resource file*) under the method where the patch is used. In most cases this is the prepare method. Following example shows how to include a patch directly to the package configuration file, and how to apply the patch in the packages prepare method:

```
[ctscript.prepare.cmd]
common = @(prepare_common)
% = cd @(SOURCE)
% = patch -p0 < @(rfile.libjpg_patch)

[ctscript.prepare.rfile]
libjpg_patch.content ="""
--- configure.orig      2008-09-08 15:07:42.000000000 +0200
+++ configure           2008-09-08 15:08:05.000000000 +0200
@@ -1559,7 +1559,7 @@
     if test "x$LTSTATIC" = xno; then
         disable_static="--disable-static"
     fi
- $srcdir/ltconfig $disable_shared $disable_static $srcdir/ltmain.sh
+ $srcdir/ltconfig $disable_shared $disable_static $srcdir/ltmain.sh $host
 fi

# Select memory manager depending on user input.
"""
```

11.9.2 Using an external patch file without macros

Using an external patch with no macro expansion doesn't require a resource file node (`rfile`). You just add a path to your patch in the prepare method.

```
[ctscript.prepare.cmd]
common = @(prepare_common)
% = cd @(SOURCE)
% = patch -p0 < @(CT)/conf/common/patches/mypatch.patch
```

11.9.3 Using an external patch file with macros (rfile)

Sometimes it is better to keep the patch as an external file because of its size, but still have all CT macros expanded in the content. Following example shows you how to get the patch content from an external patch file. Content for the `<patch_name>.content` is read with the '`<!`' operator from an external file.

```
[ctscript.prepare.cmd]
  common = @(prepare_common)
  % = cd @(SOURCE)
  % = patch -p1 < @(rfile.python26_patch)

[ctscript.prepare.rfile]
python26_patch.content <! ../../patches/Python-2.6.1.patch
```

Note that the given path cannot contain macros. If there is no leading `'/'` in the path, it is considered to be a relative path to the location of the including `cfg` file (i.e. with only a filename the file is searched from the same directory where the `cfg` file is located).

11.10 Filtering directories/files from the flash image (PATH_FILTERING)

As earlier discussed, the "un-cleaned" target root file-system (`@(ROOTFS)`) contains just about everything the packages want to install when running `make install`. This includes C-header files, manual pages, HTML documentation and other files which are not needed on the target image. The `PATH_FILTERING` parameter can be used to ignore paths and files from the resulting image. The `PATH_FILTERING` value is used only when running the image building scripts `CORE_MODULE.images.*`.

For anything non-trivial you will need to have some knowledge on regular expressions (`grep` syntax). Following path filtering is taken from the python package configuration file `python26.cfg`:

```
PATH_FILTERING = """
# let only the optimized file through
Ignore_paths 'lib/python2.6/*.*\.(pyc|py$)'
Ignore_paths 'lib/python2.6/*/test'
"""
```

In the first command we let only optimized python files through (`*.pyo`) – other formats are not necessary at run time. In the next command we ignore all test directories. For more examples, please look at the Qt configuration file.

12 CrossTarget Configuration file syntax

12.1 Nodes and nodepaths

CrossTarget stores data in a big data tree where data entries are called *data nodes*. A data node can have a value as well as child nodes. The initial data tree is built from the configuration files in `emdist-xx/conf/*`. Most of the paths in the configuration tree are logically structured in following manner:

```
<type of node>.<node name>.<type of node>.<node name>...
```

For example, following node path:

```
target.hico7722.pkg.busybox.cmd.install
```

Consists of two parts *typepath* and *keypath*, where `target.pkg.cmd` is the nodes *typepath* and `hico7722.busybox.install` is the *keypath*. All the CtScripts, for example, are named by using the nodes *keypath* (e.g. `$CT/scripts/hico7722.busybox.build`).

All parameters given in the configuration files consist of a nodepath and a value. Section headers (e.g. `[blah.foo.boo]`) are used to avoid having to write the whole path for every node.

12.2 Comments (# ...)

Comments start with a hash character (#) and the character must be the first non-whitespace character on the line. You cannot add a comment after a value assignment. Example

```
foo = bar      # I Am a comment!  
              # Am I a comment?  
              # No, You are part of the value!
```

12.3 Simple value assignments (node = value)

Simple values consist of the path part, where a dot ('.') separates the path nodes. Simple values need to be given on one line. Everything after the first '=' will be added to the value

```
my_param = 123  
my_param.description = My parameter
```

Do not use any quotes (' or "), unless you explicitly want that the value will have them too:

```
# Wrong:  
# value of my_param will be "foo bar" including the quotes  
my_param = "foo bar"  
  
# Correct:  
my_param = foo bar
```

12.4 Multiline values (node = ""value"")

Multiline values are given by giving triple quotes around the value

```
multiline_val = """
    blah blah blah...
    """
```

Whitespace on the last line before the closing triple quote will be removed on every line *if* the indentation is identical on every line. Example:

```
description = """
    This text describes something
    On multiple lines
    """
#^^^ Indentation on the last line is removed from every line.
```

With 'w' modifier extra whitespace can be stripped from the value. All whitespace sequence longer than 1, will be converted to one space:

```
# CFLAGS Will have value '-DBLAH=1 -DFOO=boo'
CFLAGS = w""" -DBLAH=1
           -DFOO=boo
           """

# Equals to
CFLAGS = -DBLAH=1 -DFOO=boo
```

12.5 Enumerating node names (% operator)

In some cases it is useful and more flexible if you don't need give an explicit name for a node. percent character will be expanded into the line number where the nodes was defined. For example:

```
script.% = cd $HOME
script.% = cp $something $somewhere
```

equals to something like:

```
script.3 = cd $HOME
script.4 = cp $something $somewhere
```

12.6 Absolute section Headers ([node])

Section headers set a prefix when adding data to the configuration tree, starting from the root of the tree.

```
[pkg.foo]                # absolute section header
bar = 15
mytext = Hello World!
```

Equals to:

```
pkg.foo.bar = 15
pkg.foo.mytext = Hello World
```

12.7 Relative section Headers ([.node])

Relative section headers add the given path which was previously defined with an absolute section header i.e.:

```
[pkg.foo]                # Absolute section header
  website = www.foo.com

[.ctscript.install.cmd]  # Relative section header
% = foo
% = bar
```

equals to:

```
pkg.foo.website = www.foo.com
pkg.foo.ctscript.install.cmd.1 = foo
pkg.foo.ctscript.install.cmd.2 = bar
```

12.8 Macros (@(node))

Like earlier discussed, the configuration data is read into a data-node tree and by using macros, you can access all the values inside the tree from any node. In order to access a value, you simply give the nodes name or path inside a macro definition, which is `@(<node-path>)`. Example:

```
[foo]
name = Foo-v2.5
path = /path/to/foo

# Here we use only values in the node
# foo, so we don't need to give the node path
description = """
    You can find the version @(version)
    of @(name) in @(path)
    """

[bar]
name = Bar-v1.2.3
path = /path/to/bar

# in addition to node 'bar' we also use values
# from node 'foo', for which we need to give the node path
description = """
    You can find @(name) from @(path),
    but you can also use @(foo.name),
    which you can find in @(foo.path)
    """
```

When you give a node name in a macro, the node will path will be traversed backwards and the first matching node will be taken.

Exported macros

Some of the macros, commonly used in the package and target configuration files are exported by CrossTarget in a manner that you don't need to give the node path in the macro calls. For example `@(ROOTFS)` will always expand to the targets root file-system path, even though the macros real nodepath is `@(CORE_MODULE.ROOTFS)`.

As a special rule, all uppercase nodes under `ctl` are exported globally (see `conf/common/ctl.cfg`)

12.9 Including configuration files (node < foo.cfg)

The configuration can be divided into many separate configuration files, and then use a *master configuration file*, to include them. The syntax is very simplistic, and you can include configuration files under certain nodes in the master file, and then do some adaptation to the original values.

```
# globals.cfg:

path = /to/something
ip    = 3.6.4.8
```

```
# foo_v12.cfg:

title = Foo version 12
path  = /path/to/foo
```

```
# main.cfg:

[globals < ../globals.cfg]    # include file globals.cfg
                               # under node globals..
ip = 1.2.3.4                  # ..and override value ip

[packages.foo < ../foo_v12.cfg] # include foo_v12.cfg under
                               # node packages.foo ..
path = /new/path/to/foo       # ..and override value 'path'
```

You can also include files without using a section header.

```
# Include to current node
. < foo.cfg
# Include file to node bar
bar < foo.cfg
```

Note that the inclusion is not a text inclusion or *preprocessing* action. The *nodetree* of the included file is added under the given node.

12.10 Reading content from external files (node < !myfile.txt)

You can also read content of any file into a node with the '<!' inclusion operator. Following example demonstrates this:

myconf.cfg:

```
foo.bar = Some file content
```

equals to:

myfile.txt:

```
Some file content
```

myconf.cfg:

```
foo.bar <! myfile.txt
```

Any macros in the included value are expanded as well. Note that you cannot use any macros in the file path. If relative path is given (i.e. no leading slash '/'), the path is assumed to be relative to the including configuration files location (i.e. location of `myconf.cfg` in the above example)

12.11 Debugging Ct configuration

You can browse any nodes value and child node values in the CrossTarget Web interface simply by changing the `.html` url ending to `.cfg`. For example `http://localhost:8080/target.cfg`.

13 Using the tool-chain

In its simplest form, cross compiling a program with the cross-compiling tool-chain doesn't really differ from compiling programs for your host. Instead of invoking `gcc` (which is a compiler for your development host), you invoke a compiler with a prefix. For example `'sh4-unknown-linux-gnu-gcc'`.

For anything more complex you also need to pass some include paths to the target root file-system, where the target specific header files and cross compiled libraries are located. This is discussed in chapter [Package configuration files](#) in more detail.

13.1 Tool-chain location and the PATH variable

You can access all the utilities installed for the development host (including the cross tool-chain) from the `$CT/devhost/rootfs/bin` directory, so exporting this directory is enough:

```
export PATH=$PATH:/home/hico/emdist/devhost/rootfs/bin
```

TIP: If you want to set the PATH variable permanently you can add following line in your `$HOME/.bash_profile`.

13.2 Cross-compiling and running a simple application

The following example shows the Makefile of the "Hello world" program.

```
CFLAGS+== -g

helloworld: helloworld.c
    $(CC) $(CFLAGS) $< -o $@

clean:
    rm -f helloworld
```

In order to build and test the program:

```
$> cd $CT/CORE_MODULE/pkgs/helloworld-2
$> CC=sh4-linux-gcc make
sh4-linux-gcc -g hello.c -o hello
$> cp helloworld $CT/CORE_MODULE/rootfs/usr/bin/
```

Change to the target serial console and run the program.

```
~ # helloworld
Hello World!
~ #
```

14 Cross-Debugging applications

The de facto debugger in the Linux is the GNU debugger `GDB`. There are also numerous graphical debuggers out there. They are, however, normally mere front-ends for `GDB`. One of the most popular graphical front-ends for `GDB` is `DDD`.

14.1 Remote debugging with plain GDB

Following example shows you how to start a remote debug session for a 'hello world' program. Note that in following examples the development host ip is 192.168.105.3 and `CORE_MODULE` ip 192.168.105.72. You need to change these to match your ip addresses.

1. Start `gdbserver` on the target

```
~ # gdbserver :4444 /usr/bin/helloworld
Process /bin/helloworld created; pid = 322
Listening on port 4444
```

2. Open a text editor and write `.gdbinit` file for helloworld program –

```
# $CT/CORE_MODULE/pkgs/helloworld/.gdbinit:

set solib-absolute-prefix /home/hico/emdist/CORE_MODULE/rootfs
file helloworld
target remote 192.168.105.72:4444
break main
continue
```

3. Start `gdb` in the same directory (export the toolchain directory first – see chapter Toolchain location):

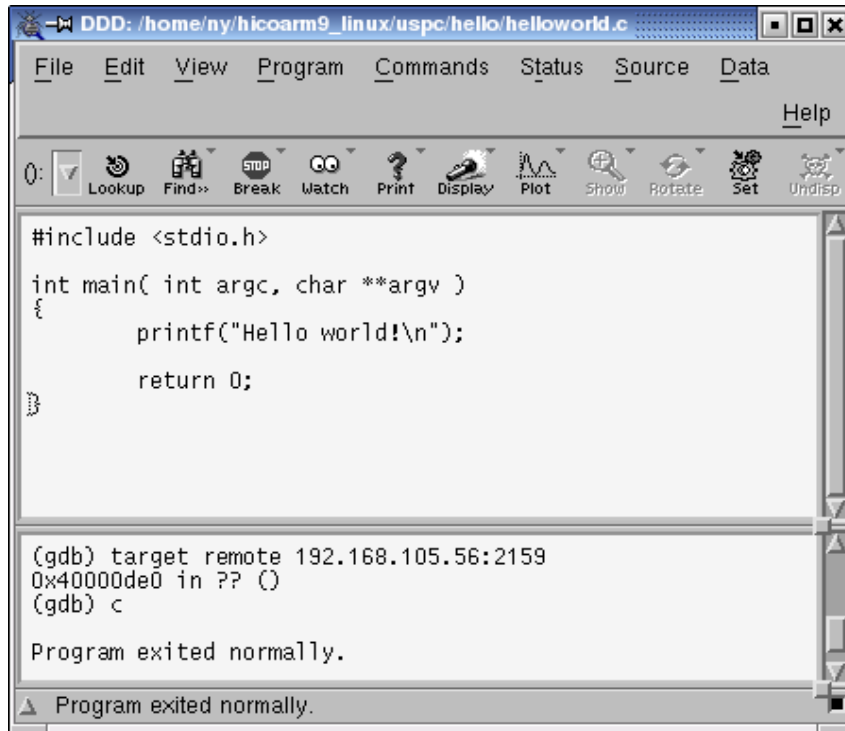
```
$> cd CORE_MODULE/pkgs/helloworld
$> sh4-linux-gdb
This GDB was configured as "--host=i686-pc-linux-gnu --target=sh4-linux".
0x40000bd0 in ?? ()
Breakpoint 1 at 0x84f8: file helloworld.c, line 8.

Breakpoint 1, main (argc=1, argv=0xbe997db4) at helloworld.c:8
8         printf("Hello world!\n");
(gdb)
```

For more information, see [GDB Website](#).

14.2 Remote debugging with DDD

DDD is a graphical front-end for `gdb` and in many cases more convenient to use than plain GDB. The underlying debug control still happens between `gdbserver` and `gdb`.



Starting a DDD debug session doesn't differ much from starting a session with `gdb`. You merely invoke `ddd` with the correct `gdb` command as parameter:

```
$> cd $CT/CORE_MODULE/pkgs/helloworld
$> ddd --debugger xxx-xxx-gdb
```

`xxx-xxx` in the command is a placeholder for the compiler prefix (e.g. `sh4-linux`) You need the same `.gdbinit` file as with using plain `gdb` in last chapter.

After starting `ddd` you should now be able to set breakpoints and control the program execution via the graphical interface. The `ddd` screenshot shows how the session should look like. For documentation and more information see DDD web site (<http://www.gnu.org/software/ddd>).