DINI GROUP

EMU Software Manual

Emulation Platform Controller

Contents

1. Introduction

Emu is both an end user application for interacting with Dini Group hardware as well as a development kit for extending Emu's capabilities or writing custom applications. Emu is designed to interface with any Dini Group board that comes with the Marvell processor. Emu compiles into both a command-line menu-system program (CMD version) and a graphical interface program (GUI version), supporting all functionality in both versions. Emu supports Windows and Linux platforms, using the QT windowing package for cross-platform support of native GUI interfaces. QT is freely available from the internet and is required for the GUI versions of Emu. It is recommended to use the QT environment in Emu development, but the CMD version can be built without it, using other standard environments such as gcc and MSVC.

The first part of this manual describes the basics of using Emu out of the box. It describes connecting to your Dini Group hardware, configuring FPGA's, setting up clocks, and transferring data between the host and the user FPGA's. The rest of the manual describes how to build the Emu application from source, and how to get started on your own Emu customizations or entirely new custom applications.

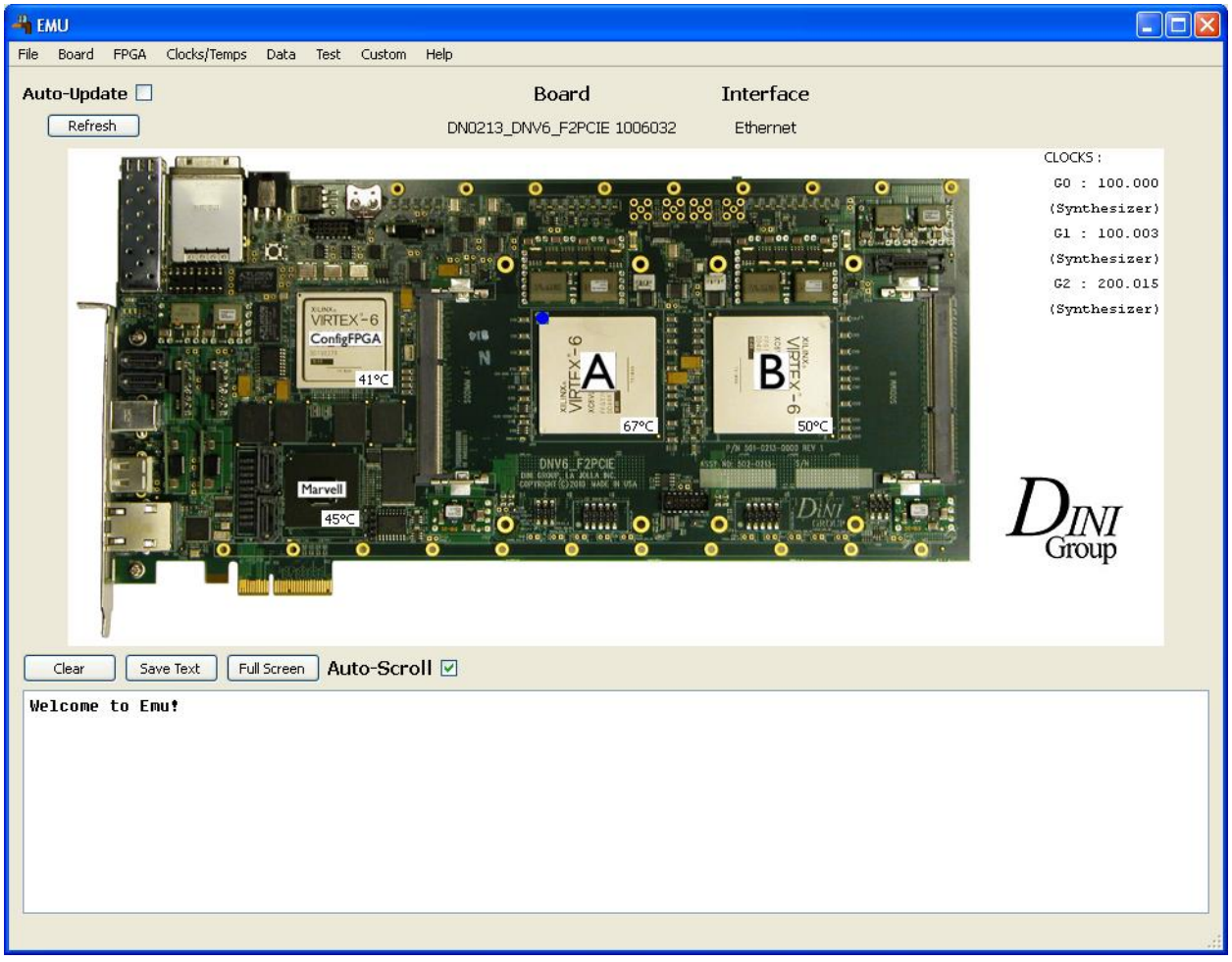

Figure 1-Emu screen shot, showing a DNV6_F2PCIE.

2. Using Emu

This chapter describes the basic use of Emu for tasks ranging from FPGA Configuration to Board Test.

I. Choosing A Binary

The first step to using Emu is finding the right executable, or building one if you are working in a Linux environment. The Emu release binaries are found in "App/out_release". The binaries are named clearly to reflect their usage:
emu_[gui/cmd]_[win32/linux]_[dbg].exe
For example, to run the GUI release version on a windows platform, use "emu_gui_win32.exe". The "release_notes.txt" file contains a list of changes made in each release. The binaries will match the latest release described in the "release_notes.txt" file.

The GUI version of Emu is the easiest to use for human interaction with Dini boards. It gives a visual representation of the board and lets you click on FPGA's to configure them, and provides a convenient menu system for all of the board features.

The CMD (command-line) version of EMU presents a text-based menu system and can do everything the GUI version can do, but without the pretty graphical interface. With the -c option the CMD version runs in command interpreter mode, taking commands and parameters from stdin, processing them, and producing output on stdout. This allows for advanced scripting of complex operations, which in many cases will allow users to automate complex tasks without touching a line of Emu code!

The "emu_mv" binary is Emu compiled for the onboard Marvell processor. This binary is preloaded on the Marvell filesystem to facilitate USB-Flash drive configuration and any other desired board interaction directly from the Marvell shell. See the chapter "Using Emu On The Marvell" for more information.

For Linux users, it is impossible to distribute binaries compatible with the wide range of systems out there, so it will be necessary to compile from source. Skip ahead to the chapter "Building Emu From Source" at this point, and return here once a working binary has been produced.

II. Device Drivers

This section describes how to install the device drivers that are required for Emu to communicate with a Dini Group board over certain interfaces. Source code is included for all device drivers, though it is unusual that anyone would want to modify or recompile the provided drivers. On Windows, drivers are required for PCIe and USB connectivity. On Linux, only PCIe requires a custom driver, but USB also has specific requirements- see below for details. Ethernet requires only that standard network drivers have been installed. If your computer can access an office network or internet connection, then it is ready to access a Dini Group board over Ethernet.

Windows: PCIe

Install the board into the PCIe slot, or connect the PCIe cable to the cable adapter card. Be sure that all required power connectors are connected (see board user manual for more information). Boot up the machine and let plug-and-play detect the board and prompt for a driver. Specify the driver location as:

[Customer Support Package]\Host_Software\emu\Drivers\windows_pci
Allow the driver installation to complete. The board is now ready for use with Emu.

Windows: USB

Boot up the windows machine without the board connected. Power on the Dini Group board and allow the processor to boot (wait about 1 minute). Connect the USB cable to the windows machine. Windows plug-and-play will detect the device and prompt for a driver. Specify the driver location as:
[Customer Support Package]\Host_Software\emu\Drivers\windows_usb
Allow the driver installation to complete. The board is now ready for use with Emu.

Linux: PCIe

Install the board into the PCIe slot, or connect the PCIe cable to the cable adapter card. Be sure that all required power connectors are connected (see board user manual for more information). Boot up the machine. Find the linux driver and source at:
[Customer Support Package]/Host_Software/emu/Drivers/linux_pci
There is a "README.txt" file there that describes how to build the driver for your linux distribution, and how to load and unload the driver once it is built. It is up to you to determine the appropriate place to source "dndev_load.sh" if you would like your system to automatically load the device driver on system startup.

Linux: USB

In Linux, Emu uses two mechanisms to attempt to connect to devices over USB. The first uses the "/proc/bus/usb" mount point of the "usbfs" (previously usbdevfs) file system driver. Until recently this was supported by most kernels. On these systems, the USB filesystem is mounted at "/proc/bus/usb", typically by adding a line to a startup configuration file, such as after the "proc" entry in "/etc/fstab":

```
none        /proc/bus/usb      usbfs      defaults      0      0
```

or by adding a line to "/etc/rc.sysinit":

```
mount –n –t usbfs /proc/bus/usb /proc/bus/usb
```

This last line can also be used from the command line to create the mount point after booting up.
To access the USB, Emu will have to be run as root unless the above mount command is altered so that usbfs is mounted with user permissions:

```
mount –n –t usbfs –o devmode=0666 /proc/bus/usb /proc/bus/usb
```

Emu expects the usb filesystem to be mounted at "/proc/bus/usb", and it is hard coded in "emulib_os_dep_linux.cpp". If usbfs is supported but the mount point is not standard, then edit this source file before compiling Emu for your system.
Recently, kernel distributions have stopped including /proc/bus/usb mounted usbfs support in their default configurations. In this case, Emu tries a second method, making a system call to the "usb-devices" program to query for usb devices, and opening devices for access under "/dev/bus/usb/". This method is compatible with the latest releases of most kernels.
Most systems will work without any modification to Emu or the kernel installation; simply connect the USB cable, open Emu, and select the board. If Emu cannot query the USB using either of the described methods it will display an error on startup. If you intend never to use USB and this error message is annoying, then edit the emu.ini file and set "discover_usb" to false, to prevent Emu from attempting to scan the USB bus.

III. Finding Hardware

When Emu starts it will attempt to reconnect to the last known board [This behavior can be suppressed with the –m option, or by editing the emu.ini file]. If the board was not found, or there is no last known board, it will open with no board selected. Choose "Board->Select Board" from the menus and Emu will scan Ethernet, PCIe, and USB for supported Dini Group hardware. You will then be prompted to select a board from the list of detected boards. Choose your board based on its type, serial number, and interface. Once a board is selected, all of Emu's features become available to set clocks, configure FPGA's, and transfer data.

Ethernet

The Emu "Select Board" option uses a broadcast command to detect boards on the local network. In some network environments Emu will be unable to find connected boards with its Broadcast command. In this case it will be necessary to select the board directly by its IP address. Use the "Board->Select by IP" menu option. By default the board acquires its IP address using DHCP. To determine the board's IP address, connect to its "marvell serial port" with a terminal program (19200bps, no parity, no flow control) and at the linux prompt type 'ifconfig' to discover what IP address your board was assigned. If your network does not support DHCP then you will need to set your board up for Static IP. Use the "Board->Set Board Info" option in Emu to do this. Be careful not to change the board type or any of the FPGA stuffing information in these dialogs! Uncheck the box for "Use DHCP", and set the Static IP, network mask, gateway IP, and DNS as appropriate for your network. Uncheck the box for setting the realtime clock, if it is checked. Accept all other defaults. Emu will disconnect from the board and the board will automatically reboot with the new settings. Once the board is booted you can check that it was configured with the new IP Address by using 'ifconfig' at the linux prompt as described above. Then use either the 'Board->Select Board', or 'Board->Select by IP" menu option to connect to the board in Emu.

USB

Currently the USB connection suffers from some connectivity limitations. That is, the board must be completely powered up and booted BEFORE the USB cable is connected to the board. If the board is to be power cycled, it must be powered down, USB cable removed, powered up, boot sequence completed, and THEN reconnect the USB cable. This limitation will be eliminated in a future firmware release, and this section will then be removed from the manual.

Multiple Boards

Emu is designed to work with one board at a time. If control of multiple boards is required, simply run multiple instances of Emu and select a different board with each. There is no hardware restriction to multiple board control within an application, so custom applications using the supplied EMULIB library code can be designed if this approach is desired.

IV. The Menu System

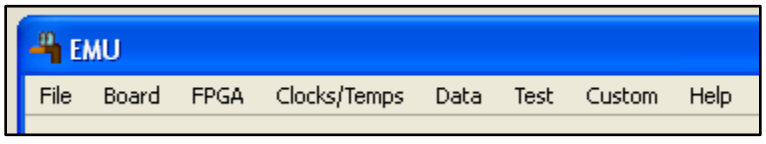

Figure 2-The Emu menu system as seen in the GUI build.

The menu system in Emu puts all of the common operation right at your fingertips.

The "File" menu (GUI only) gives you an "Exit" option and a "Kill Process" option. Every menu selection is run as a separate thread. In the case that something goes wrong and the thread is not returning control to the main program, this option can be used to kill the offending process and bring the software back to a normal state. This option may be useful while developing custom Emu functions to aid in debugging or to abort a long operation.

The "Board" menu provides options for selecting and deselecting Boards as described above in the section on "Finding Hardware". The "Display Board Info" option will show firmware revisions, fpga stuffing info, and other low level board data. When contacting Dini Group support (support@dinigroup.com) about a board, include this text in the message. "Set Board Info" is primarily for factory use, to set up board parameters before first-time use. It may be used to restore functionality in the event that the NAND Flash on the board is compromised. There is also a "system call" option here, which lets you execute shell commands on the onboard Marvell processor and see the returned text. Try issuing the "date" command or the "ls" command to see how it works. Remember that any command that blocks for user input will essentially lock up the board because the system call will never return- so be careful with this feature! If you lock up the system in this way, either power cycle the board or hit the "sys_reset" button and wait for the Marvell to reboot, then select "Board->Reconnect" to reestablish your connection.

The "FPGA" menu provides options for configuring and clearing FPGAs, as well as issuing logic resets to configured FPGAs.

The "Clocks/Temps" menu has options for displaying temperatures and reading and setting the boards configurable clock networks. Temperature sensors provide data on the user FPGAs, the configFPGA, and the Marvell processor. Every Dini Board has its own clocking topography, and Emu provides an interface for setting the clock muxes and the synthesizer frequencies all in one place. See the user manual for your specific Dini product for more information on the clocking topology of your board.

The "Data" menu is the gateway to the NMB bus, which provides a simple and efficient data transfer mechanism between hosts and user designs. Dini Group provides the NMB target module for users to include in their FPGA designs. Full source code and documentation is provided, as well as reference designs. Browse the NMB address space, transfer to/from binary files, or peek at specific addresses all from this menu. Other address spaces are also found in the "Data" menu, including the ConfigFPGA register space. The ConfigFPGA register space is primarily for diagnostic and debug purposes and will not be of much interest to most users.

The "Test" menu contains a variety of diagnostic tests that verify correct operation of the Dini Group hardware. See the section below on "Running Field Tests To Verify Board Operation" for more information on using these tests.

The "Custom" menu is provided entirely for customer use. Its functions are implemented in the "App/source/custom.cpp" file, and are well documented to get you started adding your own custom functionality to the software. In just minutes you could have a custom menu option to set global clocks, configure FPGA's, reset FPGA designs, and transfer a block of data over NMB!

The “Help” menu option (GUI only) contains the “about” option which will display the version and compile date of the application. It also provides a link to the online version of this document found on www.dinigroup.com.

V. Basic Operations: Set Clocks, Configure FPGAs

The basic steps to getting a design up and running include setting up the global clocks, configuring the FPGA’s, and optionally issuing a logic reset. There are several methods to accomplish these basic tasks in the Emu software which suit different purposes.

The simplest method is to use the GUI version of Emu. Simply click on each global clock on the GUI display, and set its Mux source and/or its synthesizer frequency. Then click on each FPGA and choose “configure”. When all FPGAs are configured with the target design, select “FPGA->Reset All FPGAs”.

The same operations can be done from the menu options, which is how they would be accomplished in the CMD version of Emu. Select “Clocks/Temps->Set Clock Mux” to set the clock sources and/or “Clocks/Temps->Set Clock” to set the synthesizer frequencies. Use the “FPGA->Configure FPGA From Host” option to configure each FPGA, and the “FPGA->Reset All FPGAs” option to issue a logic reset.

Most users will want to automate these basic tasks, since they will likely be done many times over the course of developing a design. There are two ways to accomplish this: using Emu in command-interpreter mode and writing a shell script, or by adding code to the “Custom” menu and rebuilding Emu from source to include your changes. Both methods have their advantages, and can even be combined: If you add functionality to the Custom menu, then your code can be run from a script by issuing the appropriate command such as “custom_1” to run the Custom 1 menu option. The shell script method can be very useful to configure a board automatically at power-up. Name the script “dini.sh” and placing it in the root of a USB flash-drive. Plug the flash-drive into the Dini board and the script will be executed each time the board boots. For more details on USB flash-drive usage, see the section in this document titled “Using Emu on the Marvell”. For more details on writing Emu shell scripts, see the section titled “Scripting with Emu”. For more details on adding code to the “Custom” menu or otherwise modifying the Emu software, see the section titled “Expanding Emu: Modifying The Source”.

VI. Running Field Tests To Verify Board Operation

The “Test” menu provides access to a test suite that can verify correct operation of the Dini product. Choose the “Field Test” option to automatically run the full test suite. The “Factory Test” option runs additional tests that require specialized hardware only available at the factory- this option is not intended to be run by customers in the field. The “Selected Tests” option can be used to choose specific tests to run, and additionally provides some useful debugging options such as ‘don’t set clocks’ and ‘automated mode off’.

The first time tests are run it will prompt you for the folder where you keep your Dini Group bitfiles. This is asking for the root folder of the bitfiles, so in the customer support package it would appear as: E:\FPGA Reference Designs\Programming Files

The chosen path is stored in the emu.ini file for future use. If it needs to be changed later the software should prompt you, but if problems are encountered you can edit the emu.ini file directly. See the section on “Emu.ini” for more information.

While tests are running you can press a key at any time to pause testing and get a list of options. By default tests will pause on error, but this behavior can be changed here. You can skip to the next test or abort testing entirely from this menu.

VII. NMB Memory Space and "MainRef" Design Examples

To begin exploring the NMB bus, the supplied "MainRef" reference design bitfiles can be used. Simply load the appropriate bitfile into the target FPGA (the bitfiles are found in the customer support package in the following folder:
FPGA_Reference_Designs\Programming_Files\[board name]\user_fpga\MAINTEST\[fpga type]\
Some boards require that the "G2" clock be set to 200Mhz in order for the NMB Bus to function properly. After FPGA's are configured and clocks are set, issue a user reset to make sure everything is in a good state: in Emu do "FPGA->Reset All FPGAs".

Once the design is loaded, the NMB bus functions under the "Data" menu can be used. The NMB memory browser allows display of NMB memory locations and makes it easy to write dwords or bytes and get immediate feedback. The file transfer functions can be used to transfer larger amounts of data. The read/write functions are good for quickly testing functionality of a memory range, such as writing data=address to a range and verifying with a read-back.

To use any of these functions requires an understanding of the NMB addressing scheme, and the specific address map of the MainRef design. The NMB bus uses 64-bit addresses. The upper 8 bits are decoded by the bus master in the configFPGA and represent the FPGA index (0=FPGA A, 1=FPGA B, 2=FPGA C, etc.). The remaining 56 bits are for the target FPGA to decode. The Dini Group reference designs by convention use the next 8 bits (55:48) as a function select decoded as follows:

```
// NMB SELECTS
#define NMB_SELECT_BLOCKRAM      (0x01)
#define NMB_SELECT_SODIMM        (0x02)
#define NMB_SELECT_INTERCON      (0x04)
#define NMB_SELECT_REGISTERS     (0x08)
```

These definitions and definitions of some of the register locations available in the MainRef designs can be found in the Emu source file "EMULIB/diniboard.h". For more detailed information about a specific board's memory map reference the verilog source for that board's MainRef reference design.

An example NMB address could look like this:
0x01020000_00001000
This address targets "FPGA B" (upper 8 bits = 0x01), DRAM (next 8 bits = 0x02), at byte offset 0x1000.

To start exploring the NMB space, try opening the NMB Memory Browser, and type "j", for "jump to address". Emu prompts for the upper 8 bytes of address, and then the lower 8 bytes of address. Try jumping to the start of the blockram space and experiment with reading and writing data. If DRAM modules are installed in the SODIMM sockets, then the NMB_SELECT_SODIMM can be used to access them as well.

It is worth mentioning that Emu will return an error message if an attempt is made to read or write to an NMB address which does not exist. For instance, on a 2 FPGA board, any address with the upper 8 bits

set to 0x2 or higher is invalid. The typical error in response to this is an NMB timeout, so if this is seen double check the address used to be sure it was correct.

3. Building Emu From Source

I. Finding the Source Code

The full source tree for each Emu release is provided in the "App/out_release" folder in a folder named for the Emu release number (such as "1.4.6"). Always use the latest Emu release as your starting point unless you have a good reason for using an older release. Note that the release binaries that are in the "out_release" folder were build from the source code in the source tree folder of the highest revision number. Inside the source release folder you will find a reproduction of the Software tree as it appears in the Dinigroup revision control system. To find the actual Emu source code traverse the tree:

App/out_release/X.X.X/Software/emu/App/source

The QT project file can be found here:

App/out_release/X.X.X/Software/emu/App/project_qt/emu.pro

II. The QT Environment

The QT development framework has been chosen for this project because of its robust cross-platform support, user friendly IDE environment, and its open source licensing. Anybody can download and install QT in windows or linux, and be developing in the Emu codebase in little time and with little effort. To get it, go to http://qt.nokia.com and get the SDK distribution for your platform. Be careful to get the "QT SDK for Open Source C++ development…", and not just the QT libraries or QT development tools, which can be downloaded separately. Get the LGPL licensed version unless you plan to redistribute your custom application as a closed-source product. The following sections describe how to set up QT for Emu development, and will jumpstart development on new projects.

III. The Project File (Switching between GUI and CMD builds)

The QT project file, located in "App/project_qt/emu.pro", is the key to building the emu application. It specifies all of the source files for each type of build (ie linux, windows, commandline, gui). It specifies where the binaries will go and what they will be called, and it specifies any external libraries that must be linked in. The only line most users are concerned about here is the first one, which specifies whether to build the GUI application or the CMD (command line) application. To switch between building the GUI and CMD apps, simply change the first line in the "App/project_qt/emu.pro" file. If set to GUI it builds the gui version, if set to CMD it builds the commandline version. After changing this setting do a 'make clean' from the build menu in QT Creator. If you forget to do this you'll get a bunch of weird errors when you try to compile. This is also a good time to switch your run configuration to run the executable you're about to build (see 'Run Configurations' section).

IV. Debug and Release Builds

In the 'Build' menu in QT Creator, there is a 'Set Build Configuration' option. For all development this should be set to "Debug", which will link to the QT and Compiler debug libraries providing full debug support. When building Release binaries, set this to "Release", which will link to the QT and Compiler release libraries [DO NOT BUILD RELEASE BINARIES UNTIL YOU HAVE READ THE SECTION ON STATIC LINKING!]. When the QT SDK is initially downloaded and installed, the Debug configuration is ready to

be used, and developers can get to work right away. The binaries will be dynamically linked to the debug libraries which is what you want for development.
For release builds, we will link statically to the QT libraries so that the distributed binaries don't depend on any non-standard .dll files or .so files. DO NOT COMMIT RELEASE BUILDS UNLESS THEY ARE STATICALLY LINKED! See the section below on setting up QT for static linking. Note that you will need two installs of QT, one for static linking and one for dynamic linking.

V. Compiling

By default, QT will use g++ for compiling and linking, and on Windows will use the minGW toolchain, which is installed by default along with the QT SDK. In order to be able to compile from the QT-Creator IDE, add the path to the minGW compiler to your PATH environment variable. The path is something like "C:\Qt\2010.02\mingw\bin". Microsoft Visual Studio CAN be used to compile and build QT applications, but this is highly discouraged due to the complexity in getting it set up.

VI. Run Configurations

Because we are building at least 4 versions of the program (or 5 if we build for marvell), we are not calling the output file emu.exe. Instead we have emu_win32_cmd_dbg.exe, and emu_linux_gui_dbg.exe, etc. These names and the "App/out_release" and "App/out_debug" folder locations that they reside in are specified in the project file "App/project_qt/emu.pro".

In order to have QT Creator run the application after building it (ie by hitting the green arrow button) you need to remove the default run configuration and add your own. This is done in the 'Projects' pane of QT Creator. Create a run configuration for each executable you plan to build.

To switch between run configurations, go to "Build->Set Run Configuration". Any time you switch between GUI and CMD or between Release and Debug you will need to switch the run configuration to run the appropriate executable. You will also need to run "Build->Clean All" and "Build->Run QMake" to complete the switch-over.

VII. Dynamic Linking and Missing DLL's

By default, QT applications dynamically link to the QT libraries. Developers should dynamically link to the debug libraries, which provides faster builds and full debugging support. When running the app from outside the QT-Creator IDE, windows will not find the required QT DLL files. To fix this, find the DLL's in the QT installation and copy them to the same folder as the executable. You will need QtCored4.dll and QtGuid4.dll. These are found in /QT/[date]/qt/bin. *NEVER* commit a Debug build to the cvs system- only Release builds will be committed and distributed to customers. We will be statically linking the Release builds so they can be distributed without any DLL files. Statically linking is more involved because it is not the default configuration. See below for instructions.

VIII. Plugins

QT also has "plugins", which are kind of like dll's. The "plugin" folder path is hard-coded in the qtcore4.dll. They recommend calling QApplication::addLibraryPaths() in your program to add more search paths to find plugins. We will avoid using plugins because they can't be used with statically linked programs. JPEG support is a plugin by default, and if the plugin is not found images simply aren't displayed!! We will be recompiling QT for static linking (see below). All images used in the program should be in PNG format, which is included in the qt library by default.

IX. Static Linking

We will statically link the QT libraries and compiler libraries for the Release builds of Emu, so we can distribute a single executable without worrying about dll's and plugins. [note: plugins cannot be used with statically linked programs] This requires recompiling QT on the target platform. The procedure goes like this:

Make sure the mingw32/bin folder is at the BEGINNING of your path environment variable. My build failed bizarrely because my mksnt folder came first in my path and its version of ar.exe did not support the command-line options used by mingw-make while building.

Open a command prompt and go to c:\QT\[date]\qt

> mingw32-make confclean
If you have not done a configure on this QT installation before, then this is not necessary, and will fail with a message like "no rule for confclean". Otherwise, wait for it to delete files. When it's done do:
> configure -release -static -opensource -no-qt3support -no-exceptions -no-sql-sqlite -no-libjpeg -no-libmng -no-opengl -no-gif -no-libtiff

Windows only: -no-dsp -no-vcproj -no-incredibuild-xge -no-dbus -no-phonon -no-phonon-backend

Note that without -no-exceptions there is no way to statically link required functions in MINGWM10.DLL. For this reason we do not use exceptions in Emu, and use assert() instead to halt program execution when something is broken.

Accept the licensing terms. We are using the opensource license, as specified on the command line above. Wait for a long while for this to complete [took 30 minutes for me]. Finally, do:
> mingw32-make sub-src

Now wait for a really long while for this to complete [took 2 hours for me].

QT is now configured for static linking. Because we specified -release on the configure command line this instance of QT should now be used only for making release builds (which are now statically linked). You cannot make debug builds with this QT instance anymore. To go back to dynamic linking debug versions, install a second instance of QT in a different folder. Then, to switch between dynamic linking and static linking, in QT Creator go to:
Tools->Options->Qt4->Qt Versions
You should see both installs in the box. If not, click the + on the right and manually add each install, giving each version a descriptive "Version Name" such as "STATIC 4.6.2" and "DYNAMIC 4.6.2".
In the "Default Qt Version" box at the bottom, select whichever version you want to be using and hit "OK". Be sure to go to "Build->Set Build Configuration" and choose "Release" if you are statically linking or "Debug" if you are dynamically linking.
Always do a "Build->Clean All" and "Build->Run QMake" after switching this stuff around.

Note, the compiler libraries are still linked dynamically by default, which would require us to distribute dll's like MINGWM10.DLL and LIBCC_S_DW2-1.DLL. We add -static to the linker command line for

release builds to link to these libraries statically by adding this line to the .pro file:
CONFIG(release,debug|release):QMAKE_LFLAGS += -static

X. What Went Wrong?

Use the "depends" tool (freely available on the internet) to verify that the executable is depending on the expected DLL's and shared libraries. If the release version depends on anything in the QT folder then something is not right. If the debug executable is giant (>100MB) then you have probably linked statically to the debug libraries, which is not recommended! If mingwm10.dll is the only incorrect dependency it may be because you have used exceptions in your code, which prevents static linking of this library. It may be possible to overcome this, but that is beyond the scope of this document. Our solution is to not use exceptions in the application.

XI. Building in Visual Studio

We highly recommend using QT. It is free, easy to use, and everything is set up for you already. If you absolutely MUST use Visual Studio then this section may help you- but don't expect it to be an easy out-of-the box solution: that is what QT is for.

A few source code modifications must be made due to differences in the standard library implementations of gcc and Microsoft Visual Studio. The problems that are encountered and their solutions are listed below:

Problem: open(), close(), read(), and write() are not found in diniapi_direct.cpp.
Solution: Add **#include <io.h>** to the top of diniapi_direct.cpp.

Problem: mkdir() does not take 1 parameter in emulib_os_dep_win32.cpp
Solution: Comment out the entire **mkdir()** function- it is not needed for Visual Studio.

Problem: ***snprintf*** is not found in diniboard_id.cpp.
Solution: change all **snprintf** statements to ***_snprintf***

Problem: stdint.h is not found.
Solution: This file was added in Visual Studio 2010. For older versions, download **stdint.h** from the internet and put it in an appropriate location.

Problem: emulib_os_dep_linux.cpp gets many errors and warnings.
Solution: Delete this file, it is for Linux builds only and should not be included in a Windows build.

Compiling the Command-Line version without QT is not too difficult, because no QT classes have been used outside of the GUI implementation. This means after making the syntax fixes above the program should build without any other extra steps.

Compiling the GUI version is a bit more involved, as the QT libraries must be installed and linked in to the Visual Studio project. Basic instructions for this follow.

1. Download and install the Qt SDK for your windows machine:

Go to: http://qt.nokia.com/products
Click 'Download the Qt SDK'
Click 'Go LGPL'
Download Qt libraries 4.6.2 for Windows (VS 2008, 194 MB) (or similar)
Run this executable using default values. (You can change the install location, just make sure you know where the directory is)

2. Configure Qt to run with the Visual Studio compiler:

Run the visual studio command prompt to set environment variables needed by the qt configure executable. This can be found from the start menu:
start->All Programs->Microsoft Visual Studio->Visual Studio Tools->Visual Studio Command Prompt

Now go to the path of your installed qt:
e.g. "C:/Qt/4.6.2/qt"

And type

> configure

When asked for commercial or open source, type **'o'**
When asked if you read the documentation, type **'y'**
Now wait while Qt configures itself to the Visual Studio Environment
When completed type
> nmake
Wait an even longer time while Qt rebuilds itself for Visual Studio

3. Download and install the Qt add-in for visual studio:

Download from http://qt.nokia.com/downloads/visual-studio-add-in
Install the addin.

When the Install is completed, open Visual Studio
Go to the "QT" menu as follows: **Qt > Qt Options**
Make sure that there is a Qt version and it matches the one you installed and configured
If it does not match, click **'add'** and put the correct version number and path
e.g. version = **4.6.2**, and path = **C:/Qt/4.6.2**

Now you are ready to build the entire GUI application.

4. Expanding Emu: Modifying The Source

Before making any modification to the source code, read the previous chapter on compiling Emu from source and be sure you can build and run a working binary from the existing code. Once you have accomplished this, then read through this section for help in understanding the basic structure of the application and for tips on how to get started customizing the Emu application. Some customers will prefer dropping the high level Emu code and using the Emulib API library directly instead- see the chapter "EMULIB Library" for more information on this topic.

I. The "Custom" menu

The first step to modifying the emu source code is to make a small change to one of the custom menu options, rebuild the application, and see the change takes effect. The file to edit is "App/source/custom.cpp". This file contains basic examples of how to perform user I/O in the program and how to interact with the Dinigroup hardware. In the GUI build, the "Cusom" menu is found in the top menu bar as a drop-down menu. Try selecting the options to see the results. In the CMD build, the option is displayed in the top level menu. In many cases, customers can get all the custom functionality they need simply by adding their code to an option in the custom menu. Other customers will want to create an entirely new application, keeping only the hardware interface part of the provided source code (called "EMULIB"); tips for this type of customization are found in the chapter titled "EMULIB Library".

II. Basic Application Structure

The application is divided into five distinct units. At the bottom is "EMULIB", contained in the "source/EMULIB" folder. This piece provides a low level API for interacting with the Dini Group hardware. See the chapter "EMULIB Library" for more about the library and how to use it. At the top are the user I/O units, contained in the "GUI" and "CMD" folders. These implement the user I/O API that is documented in the "emu.h" file in the main folder. The GUI build uses the "GUI" implementation and the command line build uses the "CMD" implementation. The "TESTFUNCS" folder contains a unit dedicated to hardware verification called "Oneshot Test". This provides a suite of tests, some of which are run only at the factory and others that can be run in the field for customer verification. The "Test" menu in Emu contains all of the options implemented in this unit. And finally, the main program unit, which is comprised of the files in the "source" folder itself. This unit builds the menu system (in "menu_system.cpp") and provides the interface between the high level user I/O and the low level EMULIB API.

III. Emu I/O System

When working in the high level Emu code, "Emu.h" describes all of the available I/O functions for displaying text and for interacting with the Dini Group hardware. This is an abstraction layer above the EMULIB library that handles user I/O and error reporting. This abstraction allows the program to be compiled as a GUI or a Command-Line application without any changes to the source code at all.

Another useful tool is the ini_settings system, which generates and reads in the "emu.ini" file. This settings file always resides next to the Emu executable and contains settings for things like last-used FPGA programming files, window size and location, and auto-board selecting options. See the top of

"ini_settings.cpp" for the list of implemented settings. New settings can be added here as well to support new features.

IV. Adding One-Shot Tests To Emu

One of Emu's primary functions is to provide a testing platform that allows both factory and field testing of Dini boards. The goal is to provide a system that is easy to use, flexible, and provides clear feedback as to the status of the various tests. In order to provide these features certain rules must be followed when adding tests to the One-Shot system. These rules are outlined in this section.

a. Every test has the same function prototype:
   bool func_name(oneshot_state_t& state);
   The function returns true if it passed, or false if it failed. The function must not directly access any fields in the "state" parameter. This parameter is used to interact with the One-Shot support functions described below.
b. Declare the function in "TESTFUNCS/oneshot_tests.h" in the appropriate section: FACTORY TESTS or FIELD TESTS. The factory tests are tests that require special hardware that is not provided to the customer. Field tests are tests that customers can successfully run in the field with the standard equipment.
c. Add the test to the appropriate boards in the static "BOARD_INFO" structure in "EMULIB/diniboard.h". Search for "factory_tests" or "field_tests" to find the appropriate column where the tests are listed. Add an entry for the new test using the "EMU_TESTFUNC" macro. The order the tests are listed here determines the order in which they will run, so choose intelligently where in the list it goes to minimize how many times the FPGA's must be reconfigured with different designs (If the correct design is already loaded it will not reconfigure).
d. Add the definition of the new function to an appropriate .cpp file in the TESTFUNCS folder. Try to group similar tests into the same .cpp file so we can limit the number of files here to a manageable number. Only create a new .cpp file if your test really doesn't fit in any of the existing categories, and then create a file that is general enough that future tests of similar nature can be included in the new file.
e. Follow the example of existing test functions!
f. Every test function must check that "global_selected_board" is not NULL before doing anything, and print "NO BOARD SELECTED" and return false immediately if it is.
g. Every test function must call "oneshot_is_automated(state)" to determine if the test is running in automated mode or manual mode. In automated mode the test must not pause for user interruption, and must do the most exhaustive and complete test possible. This is the mode used when we do our factory test, so it better fully test anything this function is supposed to cover. If in manual mode, then the user should be given options to select specific FPGA's and other parameters to aid in debugging when things aren't working right.
h. Every test must call "oneshot_prepare_test(state,folder,design_type,fpgas,clock_settings)" to guarantee the correct settings will be used for the test. If the test was run with the "don't set clocks" option, then the oneshot_prepare_test() function will not change the clock settings and

will instead display the current clock settings. If this function returns false the test should report the error and return false.

i. Every test must report errors with the “oneshot_reporterror(state,message)” function. After reporting an error the test can either return false immediately, or check the return value of oneshot_reporterror() and return false if oneshot_reporterror() returned false, or continue testing if oneshot_reporterror() returned true. Only allow continued testing if it makes sense that someone may want to attempt continuing in that situation.

j. Every test must use oneshot_nmb_read() and oneshot_nmb_write() when accessing the board. This function handles error reporting in the event that the NMB transfer fails. The test should return false if the oneshot_nmb_read/write returns false, otherwise it can continue knowing the transfer succeeded.

k. Every test must call “oneshot_checkquit(state)” at least once every few seconds during testing, and must produce some kind of text output at least this often. Tests that sit around for longer than 5 seconds without any indication that something is happening will not be tolerated. “oneshot_checkquit(state)” checks to see if the user has pressed a key, and if they have it pauses the test and presents a menu with options like ‘quit all tests’, ‘toggle pause on error’, ‘skip to next test’, and so on. The oneshot_checkquit() function does not print anything if no keypress is detected, but this is often a convenient time for the test to print a dot or something to indicate to the user that progress is being made. Calling oneshot_checkquit(state) at least once every few seconds is paramount to making the oneshot test system successful. It gives the user responsive control over the test flow at any time. Tests that do not call this function frequently will not be tolerated.

Following these rules will keep the oneshot test system running smoothly and everybody will enjoy using it. Thank You!!!

5. EMULIB Library

Some customers already have their own applications and prefer to make library calls to access the Dini Group hardware without the need to merge the high level Emu code into their app. By simply compiling the "EMULIB" folder into a standalone library this is accomplished.

I. Methods of using the EMULIB library

There are two options: compile the EMULIB source directly into your application (easy), or compile the EMULIB source into a library, and then link your application to the library (a little harder).

The code in the EMULIB folder is independent from the rest of the Emu program, except for the list of Oneshot Test functions for each product. In order to remove this dependency a compiler constant must be defined when compiling the library:

```
#define EMU_STANDALONE_LIBRARY
```

The library therefore does not support running the pre-written self-test diagnostics on the hardware using the Oneshot Test System- that is what the Emu program is for. Customers are welcome to write their own quick diagnostics using the API functions provided if this type of sanity checking is desired.

II. Compiling EMULIB directly into an application

Include all files in the "emu/App/source/EMULIB" folder in the project you are working with.
Include the file "emu/App/project_qt/EMU_QT_VERSION.cpp" in your project.

If you are using a windows machine, do not include **emulib_os_dep_linux.cpp**
If you are using a linux machine, do not include **emulib_os_dep_win32.cpp**

Define the compiler constant EMU_STANDALONE_LIBRARY either in your makefile or project file or at the top of "diniboard.h" with a #define directive.

If building on windows, two external Pcie driver headers are required: "Drivers/windows_pci/GUIDs.h" and "Drivers/windows_pci/Ioctl.h". These are located in the "emu/Drivers" folder. If this folder structure is undesirable for your project, copy these headers into the EMULIB folder and change the #include lines in "emulib_os_dep_win32.cpp" to the correct path.

The file "DiniCmos_interface.h" is the only other external file dependency (applies to both windows and linux). This file is located in "Software/Marvell/DiniCmos/includes". If this structure is undesirable to your project, copy the header file into the EMULIB folder, and change the #include line in "diniboard.h". This is the only file that references this external dependency.

For windows builds, there are two library dependencies that must be added to your project if they are not already used: ws2-32 and setupapi. If you are using QT, simply add these to your '.pro' project file:

```
win32:LIBS += -lws2_32 –lsetupapi
```

The header file "diniboard.h" must be included in your application to declare all of the API functions and other features of the library. This file is found at "App/source/EMULIB/diniboard.h".

III. Building EMULIB into a standalone library

Windows

A QT project is provided at "App/project_emulib/emulib.pro". Open the project in QT Creator and build it. It will produce a file called "libemulib_win32_dbg.a" which can then be linked to from your Windows applications. Note that QT uses the MinGW compile environment which uses gcc, therefore the library is in the libx.a format. The x.lib format is Microsoft specific and is only produced if you are using Microsoft Visual Studio.

QT is not needed to build the library. If gcc is to be used, simply build with the Makefile provided at "App/project_emulib/Makefile.gcc". The MinGW distribution of the GNU tools works great and is free. Use the command "make –f Makefile.gcc".

For Visual Studio, see the section in the "Building Emu" chapter about building in Visual Studio and make the changes discussed there first. Once that is complete, use "App/project_emulib/Makefile.nmake" to build the "emulib.lib" library file. Use the command "nmake /F Makefile.nmake".

Linux

The QT project in "App/project_emulib/emulib.pro" works for building the library both in Windows and linux. Simply open the project file in QT and build it. It will produce "libemulib_linux_dbg.a" which can then be linked to from your Linux application in the standard way (ie –lemulib_linux_dbg).

QT is not required to build the library. The Makefile provided at "App/project_emulib/Makefile.gcc" will build the library with gcc and ar in the standard way. Use the command "make –f Makefile.gcc".

You can include the library in your application by linking to it in the standard way: -lemulib_linux_dbg. You may also need to add a library search path depending on where you place the libemulib_linux_dbg.a file, which is done with "-L[folder_name]". Use "-L." to include the current directory in the library search path.

IV. Using the EMULIB API

There are two main sources of documentation for using the API functions. First, start with the sample application located at:

App/project_emulib/sample_app

The simple example in "main.cpp" shows all the basics of detecting hardware, connecting to a board, setting clocks, configuring FPGA's, communicating with the NMB bus, and handling API errors.

The second source is the main API header file which is located at:

App/source/EMULIB/diniboard.h

Start at the BOTTOM of this file, where all of the API functions are listed, with comments describing their usage.

A brief description of how the EMULIB API is to be used follows:

In the file in which you will be calling board functions, #include "diniboard.h".

EmuLib provides several global variables for convenience. Use these variables to manage the Diniboard_id and Diniboard objects that are returned by the various API functions:

String EMU_VERSION; // For reporting the version of Emulib
String EMU_DATE; // For reporting the release date of Emulib
Diniapi* global_usb_api; // For discovering boards on USB
Diniapi* global_pcie_api; // For discovering boards on PCIe
Diniapi* global_ethernet_api; // For discovering boards on Ethernet
list<Diniboard_id> global_board_list; // For holding information about discovered boards
Diniboard* global_selected_board; // For holding the pointer to the opened board

Emu supports multiple interfaces. At the time of this writing it supports USB, Ethernet, and PCIe, but the framework is extensible and new interfaces may be added in the future. Interfaces are implemented through a base class called Diniapi. Each interface implements a subclass of Diniapi. For example, the class Diniapi_ethernet is derived from Diniapi and implements the Ethernet interface. A global object of each interface is created at program start, and global pointers are provided to the user to access the API's. For example, to find any boards connected to the USB interface an application would call:
global_usb_api->emu_discover(global_board_list);
A list of Diniboard_id structures is returned describing the available boards. To connect to a board, the application calls the connect function from the supplied Diniboard_id:
board_id->api->emu_connect(board_id,global_selected_board);
A pointer to the board object is placed in global_selected_board, and all future operation are done through this pointer.

To discover boards:

```
global_board_list.clear();
global_usb_api->emu_discover(global_board_list);
global_pcie_api->emu_discover(global_board_list);
global_ethernet_api->emu_discover(global_board_list);
```

//global_board_list is a global list of diniboard_id's which now contains all found board id's.

To select a specific board:
//only after discovering boards

```
list<Diniboard_id>::iterator iter;
DINIAPI_STATUS apistatus = DINIAPI_STATUS_ BOARD_NOT_FOUND;
if(global_board_list.size() > 0)
  for(iter = global_board_list.begin(); iter != global_board_list.end(); iter++){
    if((*iter).serial_number == ########)
      apistatus = (*iter).api->emu_connect(*iter, global_selected_board);
  }
```

// ######## = serial number of the board you want to interact with

To interact with the selected board:
// only after connecting to a board
Now, assuming **apistatus==DINIAPI_STATUS_SUCCESS**, global_selected_board points to the board which you selected (with the serial number #######)
To interact with the board, just call functions from the board
uint32_t value;
global_selected_board->emu_set_clock("G0","200.0"); // Set clock G0 to 200Mhz

All available functions to be called from **global_selected_board** are located at the bottom of **"Diniboard.h"**

V. EMULIB API Error Reporting

If something goes so terribly wrong that there is nothing to do but give up, then the assert(false) call is used to force an abnormal abort and give the user a chance to debug the problem. The 'NDEBUG' constant is never defined, even in release builds, so that assert() will always cause program termination.

For errors that are slightly less offensive, there is an error reporting mechanism that uses error return codes supplemented with a text buffer for extra information. This is similar to the "getlasterror()" mechanism that many programmers are familiar with. Every API function that interacts with the hardware returns an error code of type DINIAPI_STATUS. Callers should check this status by comparing it to DINIAPI_STATUS_SUCCESS (which is defined to be zero). The macro DINIAPI_STATUS_STRINGS() can be used to get a static string describing any error code.

EMULIB code adds extra error information to a buffer using the following functions:
void emulib_errorput(const string& errortext);
void emulib_errorputf(const char* format, ...);

Higher level code accesses this information by calling:
void emulib_errorget(string& errortext);

Putting this to work, an EMULIB API call might look something like this:

```
DINIAPI_STATUS apistatus = myboard->emu_clear_fpga(fpga_bitfield);
if (!apistatus) {
   cout << "ERROR: " << DINIAPI_STATUS_STRINGS(apistatus) << "\n";
   emulib_errorget(errortext);
   cout << errortext;
   return 1;
}
```

Because the above code is so common, it is a basic requirement to define a function for it. In the Emu application it is declared as follows in "emu.h". If using EMULIB as a standalone library, you can use the version supplied in the sample_app "main.cpp" file, or write your own.

void emu_report_systemerror(DINIAPI_STATUS apistatus);

Any place in the application where a function that returns type DINIAPI_STATUS is called, it is followed by a check of the return status and a call to emu_report_systemerror() if the status is not DINIAPI_STATUS_SUCCESS (ie non-zero). For example:

```
apistatus = global_selected_board->emu_clear_fpga(fpga_bitfield);
if (!apistatus) { emu_report_systemerror(apistatus); return 1; }
```

6. Command Line Emu and Scripting

In addition to being an interactive menu application, the CMD version of Emu can be used to automate common tasks in scripts or to run simple operations from the command line.

I. Command Line Options

The examples below show emu running in linux, but the same operations can be done from the command shell in windows. First, run emu with the –h flag to see the help info:

```
> emu_cmd_linux -h

Emu Version 1.0.9, compiled May 24 2010
Usage: emu_cmd_linux [-<option>+]

Where <option> can be one of the following:
        h: print this message
        m: force manual board selection, overrides .ini setting
        c: enter command interpreter after processing options

Command Interpreter accepts commands on stdin.
Parameters match what is asked for if no parameters are given.
When a parameter is a list option, use the text of the option or its number.
Replace spaces with underscores when specifying the text of a list option.
The following commands are supported:

        select_board
        specify_board
        select_by_ip
        display_board_info
        set_board_info
        system_call
        reconnect
        disconnect
        configure_fpga_from_host
        configure_fpga_from_marvell
        show_stuffed_fpgas
        show_configured_fpgas
        reset_fpga
        reset_all_fpgas
        clear_fpga
        clear_all_fpgas
        set_clock
        set_clock_mux
        display_clocks
        display_fpga_temps
        display_fan_speeds
        configfpga_memory_browser
        configfpga_reg_read
        configfpga_reg_write
        nmb_memory_browser
        nmb_read
        nmb_write
        nmb_write_from_file
        nmb_read_to_file
        factory_test
        field_test
        selected_tests
        quit
```

The command line options are described in the screenshot above. The command interpreter mode gives a prompt and accepts commands on stdin rather than displaying a menu system. This is the mode that allows the program to be used in scripting or right on the command line, as follows:

echo “set_clock G0 200” | emu_cmd_linux –c

Note that this will only work if Emu has previously been run and connected to the target board, thus saving the target board’s information in the emu.ini file for auto-connection when Emu starts up. Otherwise the board to connect to would need to be specified before any commands are given:

echo “select_by_ip 192.168.1.6” | emu_cmd_linux -c
echo “set_clock G0 200” | emu_cmd_linux –c

In the example above, the first line connects to the board and saves it in the emu.ini file and then the second line autoconnects back to the board and sets the clock.

To disable the autoconnect feature, use the –m option. This eliminates the pause while Emu scans the various interfaces for boards at startup, and prevents Emu from connecting to anything.

Emu_cmd_linux –m

The above command will display the main menu without trying to connect to a board.

7. Using Emu On The Marvell

The onboard Marvell processor runs an embedded linux operating system, which takes about 20 seconds to boot up. This is why it takes about 20 seconds after the board is powered on before Emu can detect it. By connecting a serial port cable to the "RS232 CPU" port on the board, a terminal program can be connected to get a linux shell. (We recommend Putty which is freely downloadable. Use terminal settings: 19200bps, no parity, no flow control). If Ethernet is connected to the board you can also telnet in once the board has acquired an address over DHCP (or by using the static IP address if this option has been enabled). If using DHCP, Emu will tell you the IP address that the board got when you use the "Select Board" option. The boot partition is set to "read-only" mode, so that nothing gets corrupted when users turn off the board without properly "shutting down" linux. Contact support@dinigroup.com if you wish to do development on the Marvell and need more information about the kernel installation and the uboot boot-loader.

I. Configuring from a USB Flash-Drive

The CMD version of EMU compiles for the Marvell CPU onboard the supported Dini products, and comes preloaded on the NAND filesystem. This provides the flexibility for the Marvell processor to interact with the board without intervention from a host machine. The most common use for this feature is for setting up clocks and configuring FPGA's at startup from data stored on a USB flash-drive.

When a USB flash-drive is connected to one of the USB "A" type connectors on supported Dini boards, the Marvell processor automatically mounts the drive and, if present, executes the shell script "dini.sh" from the root of the flash-drive. Users are free to modify this script to do whatever they like, but the default version will be fine for all but the most advanced users. Most users will only want to edit the "config.txt" file, which is discussed below, but first take a look at the provided sample dini.sh script:

```
Software/emu/Sample_Scripts/dini.sh
```

The script attempts to connect to the local board, and if it can't it waits a bit before continuing. This protects against the case where the USB flash drive came up before the DiniCmos software was launched on the board. The emu_mv installed on the Marvell will connect to the local board using the Ethernet loopback adapter (address 127.0.0.1) by default when it is started (ie as long as the –m flag is not passed). Although it is using the Ethernet interface, data will not go out of the Ethernet port when we communicate with the local board.

We then send the contents of a text file called "config.txt" to emu. Config.txt is simply a list of emu commands and parameters. Emu will quit when it receives an EOF on stdin, so explicitly sending the "q" command is not necessary, but you may, if you wish, include the "q" command at the end of the config.txt file to explicitly quit emu. Note the use of dos2unix to force unix style newlines. Finally, the script scans the output of emu for errors and reports any that it finds. This provides a basic example of how to interact with emu in a script- pipe in commands, capture and parse the output.

Most users will only edit the config.txt file to get the clock settings and other features that they want. However, advanced users will find that they can create dini.sh shell scripts that automate their entire test suites doing everything from configuring FPGA's to writing and reading onboard DRAM.

Now take a look at the “config.txt” file:

```
Software/emu/Sample_Scripts/config.txt
```

This example shows only the most basic emu commands. You can explore the available commands by running “emu -h” at the command-line. To discover the parameters taken by each available command, experiment by running “emu -c” to enter command interpreter mode. You can then enter any available command and see what parameters it asks for. Note that when selecting an item from a list, you may use the list number OR the actual text of the list item, which may be preferred in the case where the list numbers may change across boards with different stuffing options. If list options contain spaces, then replace the spaces with underscores in your parameter text.

Most Dini Group boards ship with a USB flash-drive that already contains these sample scripts in the root folder, ready for your use. If not, simply copy the samples from the locations mentioned above into the root of the provided USB flash-drive and you’re ready to go.

NOTE: A link to Emu compiled for the Marvell (called “emu_mv”) is in the /bin folder on the NAND filesystem. This guarantees it will always be in your path. If you wish to use your own version of emu_mv, you can place the binary on the flash drive and in dini.sh change the references to “emu_mv” to “./emu_mv”, which will force it to run the local copy instead of the copy installed on the system.

II. Fixing An Unreadable USB Flash Drive

Some USB Flash Drives come formatted in a way that the Marvell linux distribution can’t read. The provided Dini Group USB Flash Drives will always work, so try these first if your own flash drive is not working. To “fix” an uncooperative 3rd party USB flash drive follow these steps:

1. Connect a terminal to the Marvell RS232 port (19200bps, no parity, no flow control)
2. Boot up the board and when it’s done press enter on the terminal to get the linux prompt.
3. Connect the USB flash drive to the board. If the board has more than one USB port either one is fine.
4. Type the command “cat /dev/zero > /dev/sda” (Erases all contents of the drive)
5. Wait until it says “cat: write error: No space left on device” (30sec to 5min depending on size of drive)
6. Remove the USB flash drive from the board and connect it to a Windows PC
7. Format the USB flash drive from the Windows PC as “FAT32”

III. Debugging Flash-Drive Configuration

If flash-drive configuration fails, there is no immediate feedback which can leave the user feeling helpless. Here is a list of things to do before contacting support@dinigroup.com:

1. Make all filenames lowercase, and make sure the commands in config.txt reference the filenames as all lowercase. Linux is case sensitive, while Windows is not, and sometimes filenames that have uppercase letters end up lowercase when copied to the flash-drive on a Windows machine. We therefore highly recommend to use only lowercase letters in filenames. In addition, ‘dini.sh’ and ‘config.txt’ must be all lowercase or they won’t be found.

2. Verify Unix-style line endings in all text files (ie dini.sh and config.txt). The Marvell attempts to run 'dos2unix' on all of the files before using them to avoid newline conflicts, but in the event this is not working properly it doesn't hurt to convert line endings manually.
3. Connect a terminal to the Marvell RS232 port (19200bps, no parity, no flow control). With the terminal connected you have a linux shell to the Marvell processor. Plug in the USB flash-drive and observe the text that is displayed. Look for the text "*** Running dini.sh ***", to see if it gets to the point of running the shell script or not.
4. The output of dini.sh is logged to a file called "dini.out" on the USB flash-drive. If this file was not created, then dini.sh did not run and there is a problem with the Marvell USB hotplug system. If it was created, it will likely contain error messages pointing to the problem.

If these steps do not solve the problem, then contact support@dinigroup.com and include the text that displayed on the Marvell RS232 port when the USB flash-drive was connected, along with the dini.sh and config.txt files that are on the USB flash-drive.

8. Upgrading Software and Firmware

We are constantly working to improve the performance and reliability of the software and firmware that is provided with the Dini Group hardware products. New versions of Emu are periodically released, which sometimes include the requirement for a firmware upgrade.

I. Software Updates

The primary source for software releases is from the Dini Group website at:
http://www.dinigroup.com/files/web_packs/emu.zip

The customer support package for your specific product will also be updated with the new software release. The latest customer support packages can be downloaded at:
http://www.dinigroup.com/files/cust_cd

We have a mailing list that sends a notice out each time a new version of Emu is released. The notice contains the release notes and the version number of the new release. To add your email address to this list, go to:
http://www.dinigroup.com/mailman/listinfo/emu-update
Note that posts from list subscribers are not allowed on this list. The only emails that will ever go out on this list are Emu update announcements. Please direct all questions and comments to support@dinigroup.com.

II. Device Driver Updates

Updates to the device drivers are included in software releases. Check the release notes to see if updates have been made to any of the device drivers, which would then require that the updated drivers be installed on your system (and recompiled if you are using linux). This is only an issue for PCI Express users and USB users on Windows. Ethernet and USB on linux do not require a device driver.

III. Firmware Updates

The Emu software will refuse to connect to boards that have out-dated firmware. If the Emu software is updated and requires a more recent firmware version than is present on your hardware you will receive an error message describing the version conflict when you try to connect to the board. The Emu release notes indicate the minimum acceptable firmware version.

There are two ways to upgrade the firmware on Dini Group Marvell processor based boards. The first is to do it directly over the network from the board itself, which requires the board to have an Ethernet connection that has internet access and for the board to be configured with an IP address (it uses DHCP by default, but a static IP address can be assigned if DHCP is not available). The second method is to download the upgrade package manually, place it on a USB Flash Drive, and connect the USB Flash Drive to the board for the upgrade process. Both methods are described in detail below.

DIRECT NETWORK FIRMWARE UPDATE PROCEDURE

1. Connect the board to Ethernet and remove any connected USB Flash Drives.
2. Connect an RS232 terminal to the “CPU RS232” port, sometimes labeled as “Marvell RS232”. This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no parity, no flow control. See the User Manual

for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.

3. Power on the board and break into U-boot. Text will be displayed on the terminal when the board is powered on, and within a few seconds the following message will be displayed:
   Hit any key to stop autoboot
   Hit a key at this point, you will have 3 seconds to do so before the normal boot process will begin. If you miss it, turn the board off and try again.
4. At the U-boot prompt, type the following command:
   run spi_boot_recoveryfs
   The board will now boot into the recovery filesystem.
5. When the boot is complete you will be in a linux command shell. Enter this command:
   sh root/recover.sh
   This will download the latest firmware package from Dini Group and install it. These packages are quite large and will take some time to download and uncompress. Expect about an hour to complete this process.
6. When the upgrade is complete the following message is displayed:
   done with recovery procedure
   Type "halt" and press enter. This will flush filesystem writes and safely shut down the os.
   Power cycle the system to boot into the updated firmware.

   If any problems occur contact support@dinigroup.com and provide the exact steps that you took and the messages displayed on the terminal. If Ethernet connectivity is a possible problem, then try the USB Flash Drive procedure instead.

USB FLASH DRIVE FIRMWARE UPDATE PROCEDURE

NOTE: Some USB Flash Drives come formatted in a way that the Marvell linux distribution can't read. If this seems to be affecting you please see the section in this manual titled 'Fixing An Unreadable USB Flash Drive'.

1. Download the following files from Dini Group:
   http://www.dinigroup.com/marvellfiles/rootfs.tar.bz2
   http://www.dinigroup.com/marvellfiles/recover.sh
2. Put the files into the root of a USB Flash Drive. A USB Flash Drive is shipped with Dini Group boards and may be used for this purpose. If any Flash Drives are connected to the Dini Group board, remove them at this time.
3. Connect an RS232 terminal to the "CPU RS232" port on the Dini Group board, sometimes labeled as "Marvell RS232". This is a 10-pin header for which a serial cable adapter is supplied with the board. Connect the serial cable to a PC and open a terminal program at 19200bps, no parity, no flow control. See the User Manual for your specific product for help in making this connection, or contact support@dinigroup.com if you are stuck.
4. Power on the board and break into U-boot. Text will be displayed on the terminal when the board is powered on, and within a few seconds the following message will be displayed:
   Hit any key to stop autoboot
   Hit a key at this point, you will have 3 seconds to do so before the normal boot process will begin. If you miss it, turn the board off and try again.

5. At the U-boot prompt, type the following command:
   run spi_boot_recoveryfs
   The board will now boot into the recovery filesystem.
6. When the boot is complete you will be in a linux command shell. When you see the linux prompt, then insert the USB Flash Drive into the Dini Group Board. If your board has more than one USB port, either port may be used. Wait a few seconds for linux to enumerate the USB device.
7. Enter the following command:
   sh mnt/sda/recover.sh
   This will uncompress the firmware package and install it. These packages are quite large and will take some time to install, expect about 10 minutes to complete this process.
8. When the upgrade is complete the following message is displayed:
   done with recovery procedure
   Type "halt" and press enter. This will flush filesystem writes and safely shut down the os.
   Power cycle the system to boot into the updated firmware.

   If any problems occur contact support@dinigroup.com and provide the exact steps that you took and the messages displayed on the terminal.

FILESYSTEM AND CONFIGFPGA UPDATES

Sometimes changes are made to the firmware outside of the DiniCmos program. This includes upgrades to the configFPGA bitfile, or changes to the linux distribution installed on the NAND filesystem. When the firmware upgrade procedure is performed (described above), the entire NAND filesystem is upgraded, including the configFPGA bitfile images. Changes of this type will be communicated to the users by doing a new Emu software release that requires a new DiniCmos version. This will enforce the firmware upgrade that will include any and all changes to the filesystem or configFPGA files. The Emu release notes will reflect what low level firmware changes were made.

9. Known Issues

I. Ethernet hostname registration in Windows

If your board's hostname gets a different IP address, and windows has cached the old one, then finding the board using the hostname may fail. In this case, running "ipconfig /flushdns" will flush the windows DNS Resolver cache and force it to re-resolve the hostname to get the new IP address. This does not seem to be a problem in linux, which is better about noticing when hostnames change IP addresses.

II. Hotplugging

Connecting: You may connect a board to the system at any time on Ethernet or USB. Hotplugging PCIe is not supported (most modern operating systems do not support this cleanly). On Ethernet, it may take up to a few minutes for the board to configure itself using DHCP after it is connected, and Emu will be unable to detect the board until this completes. The Marvell Serial Port can be monitored to diagnose problems with Ethernet configuration. If a static IP is being used, the board will be available almost immediately after the Ethernet cable is connected. On USB, the board will be detectable within a few seconds of connecting the cable.

Disconnecting: Emu will not detect when a board is suddenly disconnected from the system. On Ethernet, if the cable is unplugged or the board powered down while Emu is connected, subsequent interaction with the board will fail with error messages that may not directly indicate that the board has been removed from the system. On USB, suddenly unplugging the cable may cause unstable behavior, possibly even crashing the software. Hotplugging on PCIe is not supported and will result in unstable behavior on most modern operating systems. It is recommended to always disconnect from the board in Emu (or close the Emu software) before powering down or disconnecting the board.

III. PCIe Connectivity and Board Reset

When connecting to Dini Group boards over the PCI Express interface, enumeration occurs at the time the host system boots. A subsequent reset of the Dini Group board will erase the PCI Configuration Space registers and cause the host system to become unstable. This is typical behavior for PCI Express and is not considered a bug. Note that FPGA resets do not affect the board firmware and are expected during normal board use. Things that trigger a total board reset include pressing the "SYS_RST" button on the circuit board, issuing the "reboot" command from the Marvell linux terminal, or running the "Update Board Information" option in the Emu "Board" menu. If any of these events occur it is best to immediately reboot the host system before anything bad happens.