DICE Firmware SDK

User Guide

Revision 3.0.x

Doc Rev 0.9

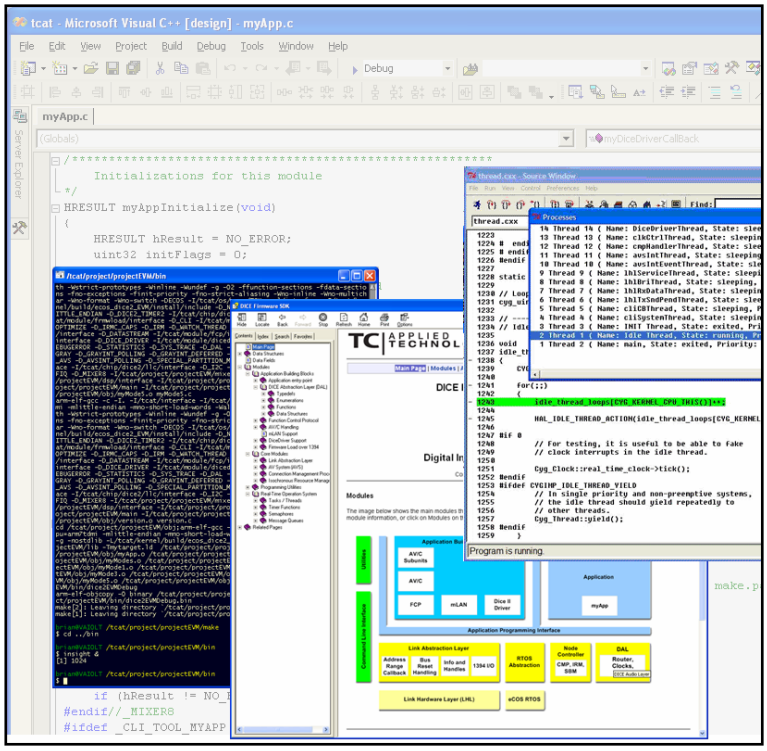
11/17/2006

TC APPLIED TECHNOLOGIES

CONTENTS

1. Overview

This document describes the DICE Firmware Software Development Kit components and how to use them for developing embedded firmware for devices based on the DICE chip family.



The Firmware SDK provides all software tools and source code needed to develop applications based on the *Digital Interface Communications Engine* family of chips.

Details on DICE EVM's, Drivers, and specific chip interfaces and registers are given in separate documents.

Please check the TC Applied Technologies website occasionally for updates, as new Documentation and Application Notes are added.

2. Audience

This User Guide is written primarily for the Embedded Software developer. Project Managers will also find this document useful for the purpose of evaluating the technologies.

Familiarity with Linux or Windows cross-development for Linux embedded targets, and knowledge of 1394 technologies, is useful but not essential. Familiarity with either Windows or a Unix variant is assumed.

The Developer should have DICE target hardware available such as the DICE II EVM.

It is assumed that you are reading this after you've installed the SDK, or have read the *DICE Firmware SDK Installation Guide*.

3. Document Roadmap

Evaluating the DICE and Tools

The *DICE Firmware Installation Guide* is a useful guide for understanding the Firmware SDK Environment and options.

If you are evaluating the DICE chip family for use in your products, use this document with a DICE EVM.

The EVM Documentation has information on how to connect to it and configure it for use. Design documents are also included with the EVM.

The *DICE Firmware CLI Reference* is a good reference for familiarizing with the functions of the EVM using a serial terminal connection.

Consult the *DICE II User Guide* for details about the DICE chip components and control and status registers.

For Host interfacing, see the relevant documentation regarding the driver architectures and available Host Software development code for Control Panel and other GUI Applications.

Project Managers and Engineers will find sections 5, 6, 7 most useful and a scan of the rest of this Guide informative.

Developing Products

A quick read of this entire Document is recommended including information about using the build system and finding your way around the firmware, then start with the section *Application Development Quickstart* to get started with firmware development.

Users of version 2.x of the SDK should consult the *DICE Firmware Migration Guide* for information about the differences between these two major versions, and for details on how to install the new SDK in an existing SDK.

4. Document Conventions

Terms

The **Host** refers to a computer that runs device drivers and software applications that interact with DICE devices.

Application generally refers to the firmware implementation running on a DICE-based device.

Notation

Notes

> **Note**
>
> Information of particular importance is highlighted in this way, including the parts of the SDK source code that are most relevant to the developer.
>
> The firmware is structured so that the great majority of tasks required to implement a device are abstracted in a small set of interfaces for the developer.

Paths

The Firmware SDK is a Windows-based cross development system, using the Cygwin Unix-emulation layer. Paths and command line examples are given in bold text. Windows paths and Bash path examples are self-evident from the direction of the slashes.

If you have installed the SDK in **c:\cygwin**, then this is the 'root' directory in the bash environment. For example, in a DOS box or in Windows Explorer the **firmware** source code directory corresponds to:

c:\cygwin**\firmware**

In a bash shell window, this corresponds to

/firmware

Incidentally, the equivalent complete path in the Cygwin environment is:

/cygdrive/c/cygwin**/firmware**

Command lines

Command line examples are given for either a bash prompt or the Command Line Interpreter (CLI) which is used via a serial connection to your DICE-based hardware. In short, if you see a '**$**' prompt it is a bash example, if you see an angle bracket '**>**' in the prompt then it's a CLI command.

Bash prompt

Bash prompt examples show the path and command as they appear in the shell window, followed by the command(s) that are to be executed. For example the first line indicates the path from which the command is executed, followed by the command line itself in bold text:

```
user@computername /firmware/project/make
$ ls
```

CLI prompts

RedBoot CLI prompts are indicated with:

```
RedBoot> version
```

Firmware Application prompts are shown as follows:

```
> splash
```

5. Checklist

Upgrading from SDK versions 2.x

☐ There is no need to reinstall the entire SDK, you will only need to install the new firmware source tree, and update the **TC_DIR** bash environment variable to **/firmware** in your **.bashrc** file. See the *DICE Firmware SDK Migration Guide* for more information.

Evaluating the DICE chip, tools and software

☐ Install the DICE Firmware SDK. Please consult the *DICE Firmware SDK Installation Guide* for useful information.

☐ EVM Users, consult the *DICE II EVM Users Guide* for information about preparing the board for use.

☐ Familiarize with the data sheet for the DICE chip and EVM you are using.

☐ Review this document for information about the rest of the checklist

---

☐ Build the EVM project. Compare your result with the included pre-built binary file to verify your source code configuration.

☐ Run the built image on an EVM to verify your tool-chain configuration. Debugging is supported using the serial cable provided with the EVM, and JTAG debugging is also supported.

☐ Experiment with the CLI commands, Host interfacing, and experiment with the tool-chain and source code. See the *DICE Firmware CLI Reference* for details.

Application Developers

☐ If you have done the above steps, **$ make cleanAll** and Commit the **/firmware** directory to your revision control system

☐ Create a new custom project from the project Template and run **$ make install** from the new project's **make** directory

☐ Make changes to the firmware that implement your custom application

6. SDK Orientation

The *DICE Firmware SDK Installation Guide* provides an overview of the various components included in the SDK. More detail is given here to outline how the parts all fit together.

Note

The DICE Firmware SDK Installer is implemented in a way that makes it as straight-forward as possible to setup a DICE development environment. The components provided are a version-stabilized collection of tools, which saves the Developer the trouble of finding the correct combinations and versions of the tools themselves. All of the components can be updated or reinstalled separately, however the DICE code and some of the various components are under parallel development as separate projects in the Open Source Community, and you may find that updating one or more of these components from on-line sources may make them incompatible with the rest.

We have made every effort to provide a sufficient set of tools and utilities to allow you to set up a stable environment and quickly get to the point where you are developing code for DICE target hardware.

Application code

The DICE Application firmware consists of an embedded 1394 protocol stack, support for controlling and monitoring audio routing, processing and synchronization, and other functions built into the DICE chip family. All DICE Application code is located in the **/firmware** directory.

Real-time OS

In this distribution, the code uses RTOS services provided by a port of the eCos operating system, and the RedBoot debug ROM monitor and bootstrap. eCos and RedBoot are based on the GNU development tools.

Compiler, Debugger, Utilities, etc.

DICE applications running on top of eCos are built and debugged using an ARM implementation of the GNU open source toolchain, which supports the ARM processor core used in the DICE chip. eCos/RedBoot drivers are implemented for support of all of the hardware peripherals on the chip, such as UART, Timers, Memory controller and flash image management, etc.

Unix-like environment

DICE applications are built using Windows cross development with the GNU toolchain. The Cygwin Unix Emulation Environment is chosen for development

with the GNU-based tools in Windows. Developers will not need a deep understanding of the Unix environment to develop firmware. Necessary working knowledge can be acquired in very short time, and the general concepts required are covered in this guide. See the *Resources* section for a list of useful additional references.

Editors

If the developer does not already have a preference for editors, several options are available for editing and debugging the DICE code. For *nix developers who are used to Emacs, GNU Windows Emacs is provided with default configurations to work within Cygwin, including gdb. Users who wish to use the Cygwin X Window system, the X version of Emacs, and other programs can update the Firmware SDK Cygwin distribution to do so. However, we have provided a xterm-like shell and a full emacs installation here, so it is not likely that you'll need CygwinX. See the *DICE Firmware SDK Installation Guide* for instructions.

Additionally, users who are more used to using IDE's such as Visual C++ .NET for their embedded development front-end (excluding debugging) can find Visual C++ .NET project file in the DICE distribution that aids in browsing the DICE code files. The developer can edit and build the DICE application, double click on errors in the Output window to go to the errant source file and line, and start the Insight debugger within the IDE.

There are a few things to consider when choosing an editor. First, it should be able to open and save file with names such as `.bashrc` where there are no characters before the period. Also it should be able to handle newline correctly for both Windows (CR+LF) and *nix (LF). For example WordPad handles this, but Notepad does not.

GDB Debugger

Several options are also provided for debugging your application code. These include command-line **gdb**, gdb in **Emacs** and the **Insight** graphical front-end for gdb. These all have support for serial debugging of DICE targets. JTAG debugging is also supported.

CLI

Developers can also use the comprehensive, extensible built-in Command Line Interface via a serial port on the target hardware. The CLI has commands for every key API in the DICE code, allowing the developer to interactively test-run configuration sequences of the DICE chip functions to reduce edit/compile/debug cycles within the source code.

The CLI also supports a number of 1394 transaction and bus management capabilities that allow the developer to set up complex device and bus configurations for prototyping, test, experimentation, etc.

When development is complete, the CLI can be commented out to reduce memory requirements. See the section *More on DICE Applications* regarding "`Make.params`" for details.

Automatically generated documentation

The DICE application source code is self-documenting using **doxygen** tags in the source comments. In this case, doxygen produces indexed and hyperlinked HTML documentation for viewing on any platform. The help can be updated or rebuilt at any time using the included tools and project.

The HTML can also be converted to Windows Help, which allows browsing and searching of the API's and CLI commands. If you download the HTML Help Workshop, and install it in it's default directory, the Help file will be automatically generated from the HTML each time you build the documents. See the *Self Documenting Code and Doxygen* section for more info.

As new changes, patches and files are added, the documentation can be easily rebuilt to stay current.

Hardware Targets

Developing on DICE target hardware requires only a serial port for the debug channel and an additional serial port is recommended to access the CLI. JTAG debugging is also supported.

PC/Mac Drivers

At the time of this writing, Host Drivers are available from TCAT which support ASIO and WDM for WindowsXP, and CoreAudio for Macintosh OSX.

This release also includes an implementation of AV/C Class Compliance with Mac OSX.

Driver support for Windows (including Vista) and Macintosh OSX workstations is evolving rapidly. Contact TC Applied Technologies for the latest drivers and for more information about using your device with various platforms and operating systems.

Driver Model

Firmware support for Host drivers is referred to as the Driver Model. The driver model is selected by making defines in the build system. Firmware coding requirements differ depending on the Host driver you choose to interoperate with.

7. Application Development Quickstart

Overview

This section gives an introduction to development with the Firmware SDK. The Developer will create a new project based on the Template, and make changes mainly in the new project directory.

Start Here

This document covers the SDK and firmware sources in some detail in the later sections; however the Developer doesn't need to be familiar with most of the source code in order to develop applications.

Most of the changes necessary to customize your firmware for your Applications will happen in files collected in the project directory and are centered on **myApp.c** and a few other files in the project directory. This file is a good starting point to work in, and then work your way out from there.

As described below, you will make a new project from the Template using the shell script provided: **/firmware/project/newproj.sh**. Run '**make install**,' then start your edit/compile/debug routine. You'll run '**make**' from then on, unless you change dependencies or add new files.

> **Note**
>
> If you have Visual C++ .NET available to use as your IDE, you'll find that browsing the code is very straight-forward, as well as building and launching the debugger. This IDE also makes it convenient to open the EVM project or your custom projects as required to make comparisons.

The EVM project, in **/firmware/project/projectEVM**, provides a lot of examples that can be brought into your custom Application as needed.

Main Areas of Customization

The majority of the changes necessary fall into a few categories.

Board-Specific configuration

Your hardware will typically use GPIO lines and set up the use of multi-function pins. This is done in the project's **target** directory in **targetboard.c**

Device Discovery

The device identifies itself on the 1394 bus by adding entries in its Configuration ROM. This includes vendor identification, and specification of the protocols it uses for communications.

These changes are made in the project's **target** directory, in **targetVendorDefs.h**

Audio Routing and Streaming Configuration

The DICE Router and Audio Interfaces are set up using the DICE Abstraction Layer (**DAL**). This is the central module for the DICE. The **dal** manages the audio interfaces which are used for input and output, how audio is routed between the inputs and outputs, and the properties of the interfaces during the operation of the device. Tables are used to describe these configurations for the Sample Rate ranges, channel names, etc. that the device shall support.

The initial Template application is set up for you in the project's **main** directory, in the file **myApp.c**. The template uses the **DiceDriver** model, which interfaces with Host drivers provided by TCAT. Here, the tables which describe the Router configuration for each sample rate range, which events the firmware is interested in, which events the Host driver shall be notified about, channel naming, etc.

Communication with Inter IC peripherals

The **i2c** module provides an API for communicating with **i2c** peripherals such as nonvolatile storage, CODEC's, etc. The EVM project includes an example for reading and writing the EPROM, and the **axm20** (the module that supports the EVM i/o expander board) shows an example for configuration of the CODEC's.

Note

The i2c module operates as an interrupt-driven interface by default. However, any Read or Write calls to the i2c module before **TCTaskStart()** is called will use a polled mode.

Other Areas of Interest

Gray rotary encoder interface

The DICE has two Gray encoder interfaces built-in. The **axm20** module also provides an example for using them, as well as a demonstration of using an Application thread to poll them, and finally a demo of how to notify a Host computer that a Gray event has happened.

DSP

The EVM project contains a sample **DSP** application. This application routes audio in and out of the DSAI port on the DICE, and implements a communication mechanism between the DICE and the DSP for changing DSP parameters.

CLI

The firmware implements a large number of **CLI** commands for most of the relevant API's in the source code. See the *DICE Firmware CLI Reference* for

descriptions. Developers may find it useful to add additional commands as they go. The process for adding new commands is simple. Look at **/firmware/module/cli/interface/cli.h** for instructions.

Timer2

In addition to the timer that uses a resolution of timer ticks (approximately 10ms by default), this module provides an additional fine timer in system clock ticks (1/49152000 sec).

AML

The Abstract MIDI Layer module abstracts MIDI handling, between UART ports and 1394 AVS ports, into a simple API. The AML provides CLI commands for monitoring MIDI traffic and sending and receiving test bytes.

Firmware update over 1394

The **frmwload** module implements a method for using a Host computer to manage firmware files on the DICE device. The developer need not make changes to this module, but it is an example of a way to implement a protocol for exchanging data and status information with a Host.

Mixer8

Since the router is implemented in hardware, the ARM doesn't have to directly process audio data, and has a great deal of compute cycles available. This can be used for example to mix audio from the various router interfaces. Mixer8 provides an example of how to route audio between the DICE Router and the embedded ARM core, and for doing processing in the ARM. Note that this is also an implementation of a fast-interrupt (FIQ) handler in eCos.

This module also provides a Control and Monitoring interface so that mixing parameters can be viewed and set from a Host user interface.

Control and Monitoring

As you have seen, there are several ways to transfer data to and from a Host computer. The **frmwload** and **mixer8** modules show an example of how to use a memory mapped method. The **axm20** and **dicedriver** modules show how to use notifications. Additionally the 1394avc module uses the methods outlined in the standards to communicate information and control settings. The Open Generic Transporter (**OGT**) implementation implements this standard as well. Contact TCAT for more information about OGT.

Kernel

TCAT has ported eCos for use with the embedded ARM on the DICE. This includes kernel configuration files that select relevant drivers for on-chip peripherals and for off-chip parts on the EVM's. The **TCKernel** module provides an abstraction layer that exposes the necessary API's to the firmware OS abstraction. This also includes a number of custom support files in the eCos sources.

In general, the eCos **kernel** will need to be changed if you are using a different hardware configuration than the DICE II EVM, typically if it involves different memory peripherals. Many implementations will not need changes, however. If you are making changes, read the following overview of how the kernel is used.

For those not familiar with eCos, see the *References* section for other sources of information. Also, if necessary contact TCAT if you require assistance with customizing the kernel for your uses.

The eCos kernel configuration is kept in a configuration file with a **.ecc** file extension. The various kernels supplied with the SDK are in

/firmware/kernel/configurations

If you wish to modify a kernel, for example the EVM kernel, you can open its configuration file in the eCos Configuration Tool, **configtool.exe**, located in

/firmware/os/ecos/src/tools/bin

This is a graphical configuration tool that allows you to specify drivers for various flash memories and RAM, configure serial ports, etc. You can save this as a new .ecc file and modify the **Makefile** to produce a make target for this kernel.

Look in Makefile to see examples of how these files are included in the build.

Once you have a correct .ecc file and requisite support files, the **ecosconfig.exe** utility is used to copy the corresponding eCos sources into the kernel directory and then build the tree into a kernel.

8. Directory Structure

The Cygwin "root" directory contains the entire SDK installation. Paths described in this document are based on this root. In many cases you will use a **bash** shell to navigate the directories and to use various tools. The file system is slightly different between bash and DOS. For example, if you have installed the SDK in **c:\cygwin**, then in a DOS box or Windows Explorer the **firmware** source code directory corresponds to:

c:\cygwin**\firmware**

In a bash shell window, this corresponds to

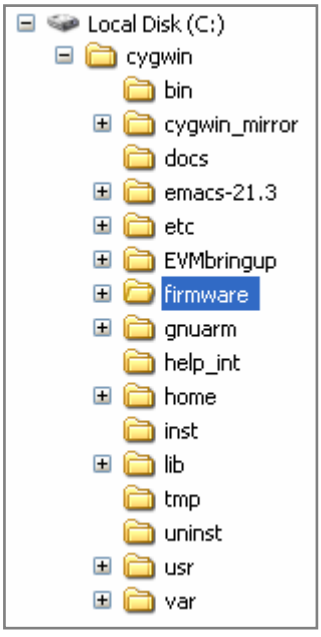
/firmware

Incidentally, the equivalent complete path in the Cygwin environment is:

/cygdrive/c/cygwin**/firmware**

Cygwin root installation directory

If you have installed the Firmware SDK in the default **c:\cygwin** location, you will see the following structure on disk.



/bin - binaries used by the Cygwin distribution
/cygwin_mirror - the Cygwin installation files
/docs - Firmware SDK documents
/emacs-21.3 – if the Emacs installation option was chosen
/etc - etc directory for Cygwin
/EVMbringup - pre-built images for the DICE EVM's
/gnuarm – the GNUARM compiler, buinutils, etc.
/help_int – utility for integrating Help into VC 6.0
/home – home directories for users
/inst – backup initialization files
/lib – Cygwin lib directory
/firmware – the DICE firmware distribution
/uninst – Firmware SDK uninstaller
/usr – Cygwin usr directory
/var – Cygwin var directory

Figure 8.1:
Root installation directory

Cygwin-related directories

/bin

Most of the executables and DLL's from the Cygwin packages live here. These files support the Unix emulation environment and its common utilities on Windows.

> **Note**
>
> Regarding the cygwin1.dll, which provides the Linux API emulation layer for Windows programs which use the Cygwin environment. This DLL is used by a number of other programs, and there are a number of installers that add this DLL to your system path.
>
> One important situation comes about if you are using the MAJIC JTAG probe from EPI Tools (Mentor Graphics). The EPI software also installs this DLL, which will conflict with the Cygwin distribution used in this SDK. To fix this, rename the cygwin1.dll file that comes with the EPI tools to something else. EPI Tools will work with the DLL used by this SDK.

/cygwin_mirror

Contains the Cygwin setup program and package mirror archives used for the base Cygwin installation in this SDK. If you wish to add new packages to your Cygwin environment, you will work from here.

/etc

Various default and post-installation configuration files are stored here and automatically copied to your home folder. If you accidentally overwrite or delete a configuration file, you can find versions here with strong defaults.

/home

When you first use a bash shell in your Windows login account, a folder for you will be placed here and default initialization and configuration files will be copied for you. Edit these for your preferences. If you wish to locate your home directory elsewhere, see the Installation Guide for instructions.

/lib /usr /tmp /var

Typical Unix directories used by Cygwin.

GNUARM

/gnuarm

Contains the GNUARM tools, including binutils, ARM compiler, assembler, linker, debugger, and Insight GUI debugger. Look in the **/gunarm/bin** directory for useful tools for manipulating binary files.

Utilities

/emacs-21.3

Contains the GNU Windows Emacs binaries. Emacs customizations will generally go in the lisp directories below, and in your **~/.emacs** file.

/help_int

Includes a utility for integrating compiled HTML Help (CHM) files into Visual Studio 6.x. Visual Studio.NET Help Integration is done using a different method. See the section *Self Documenting Code and Doxygen*.

/uninst

The Firmware SDK Uninstaller.

SDK Documentation

/docs

PDF files, Precompiled searchable DICE Source Code Help file, Support URL.

Pre-built Binaries

/EVMbringup/

Several pre-built images are copied here by the Installer to allow bring-up of DICEII EVM's and other DICE targets. These are not overwritten when you build the DICE code, and so can also be used as fallback images for disaster recovery or by comparison to aid in troubleshooting code under development.

DICE Firmware directory

DICE Firmware is all contained in the **/firmware** directory of your root installation, see figure 8.2.

This structure implements a separation of files into logical groupings that allow common firmware application code to run on multiple versions of the DICE chip family. Development of multiple application projects at the same time is straight-forward. Application code for each project is contained in a project folder for each application. A project template is provided for easily starting new firmware application development. In most cases, developers will be working with files in the **main** and **target** directories in each project.

> **Note**
>
> When you use search tools within your favorite editor or from a command line too such as **find**, it will save a lot of time if you narrow the search term to include only ***.c** and ***.h** file types and exclude the **/firmware/os/ecos/src** and **/firmware/kernel** directories, and, since these directories contain a great deal of files that are not relevant to day-to-day development.

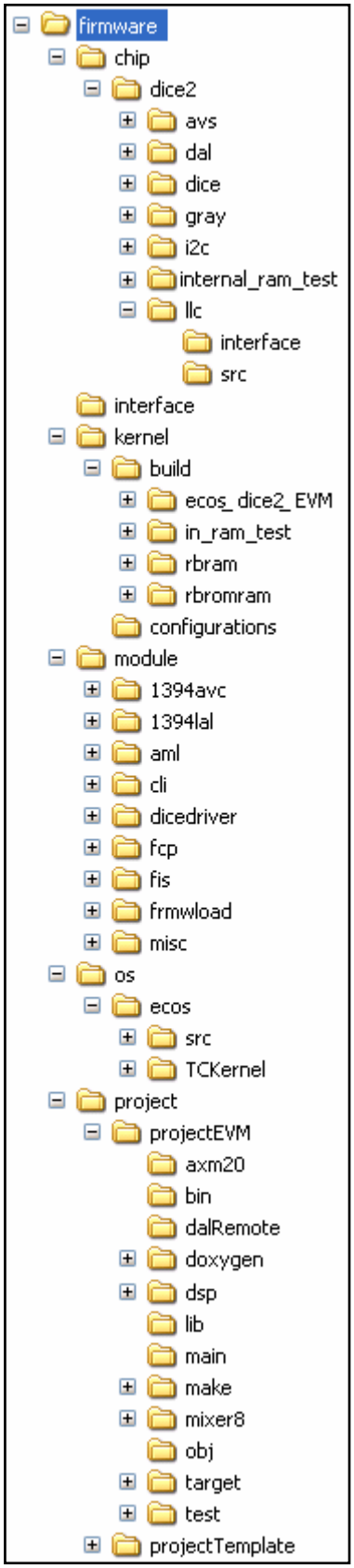


Figure 8.2:
Firmware directory structure

chip – groups all DICE chip-specific modules
dice2 – modules for the DICEII STD chip
avs – Audio Video System (1394 streaming)
dal – DICE Abstraction Layer (Routing)
dice – Audio Interfaces
gray – Gray Coder/Decoder
i2c – IIC interface
internal_ram_test – memory test utility
llc – 1394 Link Layer Controller
interface – header files, each module has this
src – C files, each module has this
interface – groups the main global headers
kernel – groups the RedBoot and Application kernels
build – built images, created by make
ecos_dice2_EVM – the DICEII EVM Kernel
in_ram_test – kernel library for memory test utility
rbram – RAM loadable RedBoot for debugging
rbrom – ROM loadable RedBoot for flash bootstrap
configurations – ecosconfig files for the above
module – groups core platform independent modules
1394avc – AV/C implementation
1394lal – Link Abstraction Layer
aml – Abstract MIDI Layer
cli – Command Line Interface
dicedriver – DiceDriver implementation
fcp – Function Control Protocol
fis – Flash File System
frmwload – Firmware Loader interface (via 1394)
misc – low level utilities
os – groups operating system files
ecos – Operating system files
src – eCos base distribution
TCKernel – Operating System abstraction
project – principle app development directories
projectEVM – the EVM project
axm20 – support for the i/o expander board
bin – built application binaries, created by make
dalRemote – remote calls to the dal module
doxygen – source code documentation system
dsp – support for the EVM companion DSP
lib – intermediate libraries, created by make
main – most files modified for applications
make – the build system and VC project files
mixer8 –software mixing in the embedded ARM
obj – all intermediate objects, created by make
target – hardware-specific initializations
test – test routines for the EVM
projectTemplate – for new custom target projects

9. DICE Firmware Directory details

Directory groupings

The application firmware files are separated into groupings that collect chip-specific, operating system-specific, core, and project-specific source code files.

This allows the various components to be updated more independently, allows the developer to maintain several separate concurrent projects at once, and provides a clean path for wider use of library-based development.

Chip-specific

/firmware/chip
Files that support the particular peripherals, such as the LLC, AVS, IIC, and media interfaces for each chip in the DICE family are kept here. Also, a minimal application is provided which runs entirely in internal on-chip SRAM, allowing developers to do initial testing with their custom hardware designs.

Globals

/firmware/interface
Global header files. This is similar to a global /include directory.

Kernels

/firmware/kernel
Code that implements the operating system services. Several configurations are provided for RedBoot RAM and ROM images, minimal memory test Application, and the DICE Application kernel library. When the kernels are built, their source trees and binaries are stored here as well.

Core generic code

/firmware/module
Code that implements the 1394 stack and other services.

Operating System

/firmware/os
Contains the base eCos distribution and TC OS Abstraction.

Project

/firmware/project
DICE EVM project, Customer projects, and a project template. Allows concurrent development of multiple Applications.

Developers will mainly make changes to code in this directory. See below.

Chip-specific Directories

/firmware/chip/dice2/avs
Audio Video System (AVS) manages 1394 audio streaming. Files in this directory will not normally be modified by the developer. The Audio Video System manages media streaming through the DICE router and is the interface for configuring 1394 bus formats and synchronization. This module includes code to configure and control the avs. Connection Management Procedures (CMP) is also in this directory.

/firmware/chip/dice2/dal
This directory includes code that implements the DICE Abstraction Layer. These functions encapsulate calls to the DICE router, AVS, Clocks and other functions into one simplified API. Your application will consist mostly of calls to this module and the dice module.

/firmware/chip/dice2/dice
This directory includes code to configure and control all audio interfaces, e.g. AES, ADAT, TDIF, DSAI, I2S. It also has files for configuring routers, and JetPLLs.

/firmware/chip/dice2/gray
Supports the gray encoder/decoders built into the DICE II chip.

/firmware/chip/dice2/i2c
Philips semiconductor's I2C-bus support

/firmware/chip/dice2/llc
Driver for the Samsung 1394 link layer controller in DICE II. Files in this directory will not normally be modified by the developer. This directory holds the hardware-specific code for Samsung 1394 link layer controller core.

/firmware/chip/dice2/internal_ram_test
Contains a minimal application that runs entirely in the on-chip RAM in the DICE II chip. This is useful for initial testing of new hardware.

Kernel

/firmware/kernel/configurations
Contains the utilities and configuration files that generate the RedBoot images and the Application kernel library. These files are used by the **ecosconfig** utility to create kernel source trees that are then built with make. You can uses these as a basis for custom hardware if your design diverges from the examples provided. Some ecosconfig files provided include:

kernel.ecc – ecosconfig file for Application kernel
redboot_ram.ecc – RAM loadable monitor for JTAG debugging, new board bring-up
redboot_romram.ecc – ROM loadable monitor. Flash resident boot image for serial debug and Application bootstrap

/firmware/kernel/build
The kernel files produced by ecosconfig during build for DICE EVM's and similar hardware. Customer hardware will typically require few changes to the kernel, such as for different RAM and flash memories. These changes are made in the configurations directory.

Module Directories

/firmware/module/1394avc
The 1394 Audio Video Control module implements the AV/C stack, including AV/C general and optional subunits, Connection Management, and Function Control Protocol (FCP). Files in this directory that are modified by the developer are usually related to the Subunit implementations. These files become part of the Application when it is defined in the build system, and define your device on the 1394 bus as an AV/C capable node.

/firmware/module/1394lal
The 1394 Link Abstraction Layer implements the 1394 stack. Files in this directory will not normally be modified by the developer. This directory has the implementation of the core 1394 services. It includes the standard 1394 architecture modules, including: the Link Abstraction Layer, which is a platform-independent API for application code; the Link Hardware Layer, which provides access to common Link and PHY layer configuration and diagnostics; Node Controller Interface functions, and the Isochronous Resource Manager

/firmware/module/aml
The Abstract MIDI Layer. This module abstracts MIDI handling, between UART ports and 1394 AVS ports, into a simple API. The AML provides CLI commands for monitoring MIDI traffic and sending and receiving test bytes.

/firmware/module/cli
The extensible Command Line Interface can be accessed via standard serial terminal. Files in this directory will not normally be modified by the developer. This directory provides the underlying CLI mechanism. CLI commands for each particular module are implemented in those modules.

/firmware/module/dicedriver
Implements the dicedriver model for communicating with TCAT Mac and Windows host drivers. Contains files that handle audio configuration transactions between the PC driver and the target DICE device. Files in this directory will not normally be modified by the developer.

/firmware/module/fcp
The Function Control Protocol asynchronous communication layer

/firmware/module/fis
Files in this directory will not normally be modified by the developer. Contains files used for managing flash file system files from the RedBoot prompt.

/firmware/module/frmwload
Files in this directory support loading of flash files from a host computer over the 1394 bus. Files in this directory will not normally be modified by the developer.

This module is a good example for developers who may wish to implement their own communications module for Host-Target control and monitoring.

/firmware/module/misc
Utility functions

Operating System Directories

/firmware/os/ecos/src
This is the official eCos 2.0 distribution with changes for the DICE chip. These files implement the DICE port of eCos

/firmware/ecos/TCKernel
The DICE Firmware is designed to be OS independent. Implements the operating system abstraction layer. Files in this directory will not normally be modified by the developer. This provides a TC Applied Technologies defined kernel API. It wraps eCos kernel functions. This makes porting to other platforms and RTOS easier.

Project Directories

The EVM Project

This project exercises most all of the functions of the DICE, including on-chip audio mixing, and contains complete support for the hardware configuration of the DICEII EVM, including the companion DSP, I2C bus, GRAY encoder/decoders, the AXM20 i/o expander board, Audio ports, GPIO, JTAG and RS232 interfaces.

It is a good starting point for evaluating the DICE, Tools, and source code. It also provides examples that can be carried over where appropriate to custom hardware designs.

The Template Project

This project is not meant to be used or modified by the developer. It is a starting point for creating new projects for starting custom product implementations. The Template project is similar to the EVM project, except that it does not include a number of modules that are specific to the DICEII EVM, such as **axm20** (expander board support) and **dsp**, which assumes a particular DSP part. These modules can still be a handy reference if your design is similar to the EVM, and can easily be copied over and integrated into your custom project.

More detail is given in the *Template Project* section below.

The EVM Project

/firmware/project/projectEVM
The project for the DICEII EVM.

/firmware/project/projectEVM/axm20
Files that support the AXM20 i/o expander board

/firmware/project/projectEVM/bin (created by make)
This directory contains the binary (ROM-able) and debug images for the application. For example of the DEBUG EVM project, they are named **dice2EVMDebug.bin** and **dice2EVMDebug** respectively.

/firmware/project/projectEVM/dalRemote
Supports remote Host calls to the DAL

/firmware/project/projectEVM/doxygen
This directory contains the framework HTML files and Doxygen project used for generated HTML documents from the tags in the dice source code. The generated documentation is place in the **GeneratedDocumentation/html** folder. View **index.html** for the top-level page. See the section *Self Documenting Code and Doxygen* for details.

/firmware/project/projectEVM/dsp
Code for the Motorola DSP chip included on the DICEII EVM.

/firmware/project/projectEVM/lib (created by make)
Intermediate libraries for the project Application

/firmware/project/projectEVM/main
Most files used by Application developers are kept here. This includes the application code entry point. See the **myApp** and **myMode** files, for setting up interfaces, routing and sample rate configurations.

/firmware/project/projectEVM/make
Makefiles, batch files, and VC++ project files

/firmware/project/projectEVM/mixer8
Implements software mixing in the embedded ARM processor

/firmware/project/projectEVM/obj (created by make)
All other intermediate objects for the project are kept here

/firmware/project/projectEVM/target
Developers set up the 1394 personality of the device here

/firmware/project/projectEVM/test
Several handy test utilities, including a A memtest module that can be used to verify the memory configurations.

The Template Project

/firmware/project/projectTemplate
Starting point for new firmware applications
Use the included shell script to create your new projects. See the section *Project Template* for details.

If a new project is created using

user@computername /firmware/project
$ newproj.sh ABC

the resulting project directories are created as follows:

/firmware/project/projectABC/bin (created by make)
This directory contains the binary (ROM-able) and debug images for the application. For examplehere, they are named **dice2ABCDebug.bin** and **dice2ABCDebug** respectively.

/firmware/project/projectABC/doxygen
This directory contains the framework HTML files and Doxygen project used for generated HTML documents from the tags in the dice source code. The generated documentation is place in the **GeneratedDocumentation/html** folder. View **index.html** for the top-level page. See the section *Self Documenting Code and Doxygen* for details.

/firmware/project/projectABC/lib (created by make)
Intermediate libraries for the project Application

/firmware/project/projectABC/main
Most files used by Application developers are kept here. This includes the application code entry point. See the **myApp** and **myMode** files, for setting up interfaces, routing and sample rate configurations.

/firmware/project/projectABC/make
Makefiles, batch files, and VC++ project files

/firmware/project/projectABC/mixer8
Implements software mixing in the embedded ARM processor

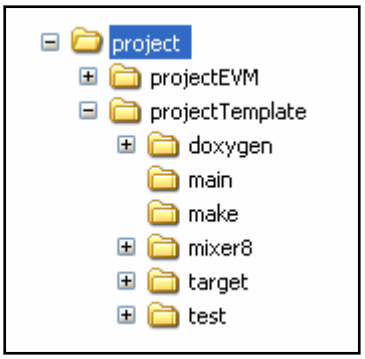
/firmware/project/projectABC/obj (created by make)
All other intermediate objects for the project are kept here

/firmware/project/projectABC/target
Developers set up the 1394 personality of the device here

/firmware/project/projectABC/test
Several handy test utilities, including a A memtest module that can be used to verify the memory configurations.

10. Project Template

Overview



The template project

/firmware/project/projectTemplate

contains a generic source code tree for creating custom implementations which are not based on the DICE EVM. The project will indeed run on the EVM, but does not make use of some of the features that are likely to be different for customer hardware configurations.

Figure 10.1: the Template project

The main differences between this project and projectEVM are that is does not contain **axm20** or **dsp** implementations, since these are particular to the EVM. The AXM20 is the i/o expander board for the EVM. The DSP implementation assumes a specific companion DSP.

If your implementation has a great deal of i/o using the I2S bus or the DSAI ports, you can browse the EVM project for source code that you may wish to reuse for your custom hardware.

Creating a new project from the Template

In your **/firmware/project** directory is a shell script called **newproj.sh**

Create a project using this template as follows:

user@computername /firmware/project
$ ./newproj.sh ABC

Where for example here, '**ABC**' is the suffix name to be given to the project. The shell script copies the projectTemplate directory to projectABC, then replaces all of the 'Template' references in the relevant files with 'ABC.' Your built binary files will have this suffix appended to their names.

The new project is then an independent working directory for that particular project.

To build the new project, go to the **.../make** directory and run **make install**.

All of your object and library files are maintained within this project directory. Your kernel will be built separately as well, and is kept in the **/kernel/build** directory in a subdirectory with the project suffix appended.

Read on in *The Build System* for more information.

11. The Build System

Overview

Each **project** directory contains a **make** directory, which is where the targets for the project are built from. This directory contains the top level **Makefile** and a build configuration file called **Make.params**. Also, a file called **Make.def** in this directory is used by makefiles within the various source subdirectories.

The resulting intermediate and executable files are kept within the project directory in the **bin**, **lib** and **obj** directories.

Make Hierarchy

Kernels, Boot monitors, Applications are all built using predefined targets the **Makefile**, which is in each project's **make** directory. The full source tree is built using this top-level Makefile and the hierarchy of makefiles in each source subdirectory. Options for controlling the builds are given in **Make.params** in the project make directory. Also, the file **Make.def** contains defines and directives that communicate necessary information and commands to the makefiles in the hierarchy.

The project Makefile provides targets for building the initial installation, various kernels, and for the typical situations of ongoing development. These include the usual cleanup, builds which accommodate changes to dependencies, and for building the normal application.

The build system is implemented so that make can be invoked from a bash command line, or from within Visual Studio, where the resulting output is redirected to the Output window using **awk** scripts that are created during the build.

Table 11.1 below summarizes the use of the various make targets.

Make from the bash command line

To build a target for the DICEII EVM, go to its project make directory and invoke **make** for the target from there.

```
user@computername ~
$ cd /firmware/project/projectEVM/make
```

The various targets are described below. Look at the **Makefile**s, **Make.dep** and **Make.params** to find out what's being done when these targets are used.

Make from Visual Studio

Visual studio has default Build options of Build, Rebuild and Clean. In the provided Visual C++ Project files, these are mapped to calls to **vc_make.bat** with arguments that call corresponding make targets.

For info about what is going on here, see the 'Properties' of the project itself, the files **vc_make.bat** and **BuildDiceFromMSVC.txt** in the project 'make' folder.

VC++ 'Build->Build projectXXX' calls vc_make.bat with 'dep'
VC++ 'Build->Rebuild projectXXX' calls vc_make.bat with 'install'
VC++ 'Build->Clean projectXXX' calls vc_make.bat with 'clean'

There are no other equivalents in the default VC++ Build Menu for the other make targets, but if you use them often then these can be added to the VC++ IDE as External Tool calls (in the same way as the Insight debugger is called up), as described in the section *Editors*.

Build-related Naming conventions

The defines in **Make.params** control the resulting file names of the executable files. The names are created by concatenating the CHIP and BOARD defines, and "Debug" is appended to the name if the build is not defined with _RELEASE in the CFLAGS definition. For example, the EVM project is defined by default as:

dice2EVMDebug	Contains debug symbols for debugging
dice2EVMDebug.bin	Application executable that can be written to flash

Note that the kernel source directory, that is created and built for each Application project, is named by concatenating the CHIP, OS and BOARD defines. For example, the EVM project creates the following kernel directory:

/firmware/kernel/build/ecos_dice2_EVM

Resulting Files

The various targets in the Makefile produce results in different directories, which are summarized below.

Compiled Application file locations

When an Application is built, all of its object, library, and resulting executable binaries are kept in the project's directory in the **/firmware/project/projectXXX/obj**, **lib** and **bin** directories.

The application's kernel library is kept in **/firmware/kernel/build/** directory in a subfolder that is given a name as described above in the *Build System* section.

Compiled Debug Monitor file locations

For the RedBoot RAM loadable monitor (for loading with JTAG) use the **make redboot_ram** target, and look for the result:

/firmware/kernel/build/rbram/install/bin/redboot.elf

For the RedBoot ROMable monitor (for writing to flash) use the **make redboot_romram** target, and look for the result:

/firmware/kernel/build/rbromram/install/bin/redboot.bin

Compiled Memory Test Utility file locations

The **make in_ram_test** make target is currently written for the DICEII chip. The build produces its result in a kernel library in **/firmware/kernel/build/in_ram_test** and a JTAG-loadable executable named **mini_test** in **/firmware/chip/dice2/internal_ram_test**

See the section *Internal RAM-Resident Memory Test Utility* for details.

Compiled Documentation

The **make doxygen** make target builds the html docs start page **index.html** in:

/firmware/project/projectXXX/GeneratedDocumentation/html

If you have installed the Microsoft Help Compiler, the help file will be in:

/firmware/project/projectXXX/GeneratedDocumentation/help

See the section *Self Documenting Code and Doxygen*.

When to use each make target

Scenario	Command(s)
When first installing the SDK, or when creating a new project from the project Template	Bash
	user@computername /firmware/project/projectXXX/make **$ make install**
	VC++
	Build->Rebuild projectXXX
After ongoing edits to the non-kernel source code	Bash
	user@computername /firmware/project/projectXXX/make **$ make** **or** **$ make dep**
	VC++
	Build->Build projectXXX
After changing dependencies: Changes in Make.params that add or remove modules, and changes to module makefiles	Bash
	user@computername /firmware/project/projectXXX/make **$ make clean** **$ make dep**
	VC++
	Build->Clean projectXXX Build->Build projectXXX
When making changes to the kernel	Bash
	user@computername /firmware/project/projectXXX/make **$ make cleanAll** **$ make install**
When making the flash-resident RedBoot boot monitor	Bash
	user@computername /firmware/project/projectEVM/make **$ make redboot_romram**
When making the RAM loadable RedBoot boot monitor	Bash
	user@computername /firmware/project/projectEVM/make **$ make redboot_ram**
When making the in-RAM test application	Bash
	user@computername /firmware/project/projectEVM/make **$ make in_ram_test**
When making the project's documentation	Bash
	user@computername /firmware/project/projectXXX/make **$ make doxygen**

Table 11.1 Make targets summary

12. More on DICE Applications

Firmware on a DICE device consists of three main parts. A *Boot Loader/ROM Monitor*, a *kernel*, and *Application* software.

Boot loader/RedBoot ROM Monitor

In a device that is running from flash memory, this is a file that is executed from the flash boot sector. The running image, as a debug Monitor, then provides debugger stubs that allow the **gdb** debugger to attach, or it looks at the flash setup file for indication that it should load an Application image file into RAM and transfer control to it. The SDK provides two boot loaders, one that is intended to reside in the flash boot sector (**redboot_romram**), and another that is intended to be loaded via JTAG into RAM (**redboot_ram**) for board bring-up or disaster recovery.

The boot loaders include a kernel with built in drivers and services for the embedded ARM processor to implement a CLI, manage flash memory files, and provide debug stubs that enable serial debugging. When using JTAG debugging, this ROM monitor is not needed, and if one is there it is ignored and superseded by the debugger in this case.

In a production device, the Boot loader acts as an initial bootstrap mechanism and can be configured to run any chosen Application image from files resident in flash memory.

Kernel

The kernel is built into the various images created by the SDK tools. For Boot Loaders, this file is built completely into a stand-alone executable image. In Applications, the make system creates a kernel library that is separately linked into the Application. Application kernels generally do not need to change during Application development, so compile time is greatly reduced by building the kernel library separately.

The eCos kernel includes drivers that manage various DICE peripherals and operating system services that provide tasking, interrupt, timer abstractions, etc. for use by application code above.

Application

An Application, in this document, is one part of the entire system of the embedded software that implements a DICE-based device. A DICE Application runs on top of the linked-in kernel as described above, and handles the control and monitoring of the dice Router, 1394 Streaming, implements a 1394 stack and various higher-level 1394 protocols, onboard software audio mixing, provides drivers for DICE peripherals such as the gray encoder/decoders, I2C interface, etc., and includes support for external companion DSP's.

In general, a DICE device works in combination with other devices on the bus and with host computers via a Host driver of some sort. TC Applied Technologies makes available this Firmware SDK, EVM boards, and drivers. Here we focus mostly on the firmware in the SDK, but it's helpful to describe how the firmware works with the Host computer.

Driver Model

Depending on the type of Host driver your device will interoperate with, you will select a driver model in the firmware. Currently, there are three driver models available, DiceDriver, AV/C and OGT. The driver model is selected by making the appropriate module defines in **Make.params** in your project's **make** directory. These driver module defines are fully documented in the file.

DiceDriver

DiceDriver is the architecture created by TCAT. Support for this model includes a firmware implementation as well as Host drivers for Windows and Mac OSX.

This model is the default used in the Firmware SDK. The projects provided are a complete implementation of the architecture. Contact TCAT to get the latest Host drivers.

This driver supports very low latency Audio and MIDI streaming, at every common sample rate up to 192KHz. The driver model supports stacking (channel aggregation) of multiple DICE peripherals, among other features. The Host drivers provide ASIO/WDM (Windows) and Core Audio (Mac OSX) interfaces to the Host, and Control Panels which provide a UI for configuring the devices from the Host.

In this architecture, the DICE device advertises its configuration to the Host driver, which then configures to support the device. For example, when the device is attached, the Host will enumerate it and load the DiceDriver which then discovers how many channels of audio i/o it uses, supported sample rates, sync sources, channel names, etc. The driver then exposes the device to the Host audio subsystems in the relevant formats, depending on the type of Host.

This communication is done using a memory mapped 1394 register space, and also includes a mechanism for updating the device firmware from the Host.

The device also provides a notification mechanism, so the host can be informed of status changes, such as sample rate change, so it can take the appropriate actions.

The DiceDriver implementation is described in detail below.

OGT

Currently, the firmware implementation of the Open Generic Transporter is not included in the DICE Firmware SDK. This is provided separately from TCAT as a drop-in addition at the project level of the firmware source tree.

AV/C

This SDK provides an initial implementation of an Audio Video Control class-compliant audio device. This is compatible with the current shipping Mac OSX drivers and requires no separate driver installation.

When you build the AV/C driver model for the DICE II EVM, the EVM will present itself to the Host as an 8-in, 8-out device which supports 44.1KHz, 48KHz, and 96KHz.

The main components of the AV/C Audio implementation are the Descriptors and Unit/Subunit commands. These use the core AV/C services which manage **fcp** (Function Control Protocol) basic transactions, descriptor services, **cmp** (Connection Management Protocol) for connection management and plug configuration, among others.

This implementation supports the Music Subunit, and Audio Subunit controls include Mute and Volume, where volume is implemented as an example (messages are relayed to the CLI only).

AV/C generally provides sufficient controls for communicating with a device using commands defined by the various AV/C subunit specifications from built-in Operating System interfaces (Audio MIDI Setup, API's provided to DAW Applications, etc.) However, if you require a richer set of controls using a Custom GUI, you can still implement a control and monitoring protocol using a method similar to the **mixer8**, **axm20** and **frmwload** methods.

Mac AV/C drivers are updated occasionally, and the firmware implementation will be updated to support new features as they are added. Also, AV/C support is in various stages of development on other Host platforms, and TCAT will verify class compliance with these as well.

Windows Host Interfacing Example (DiceDriver)

The default configuration in the firmware uses the DiceDriver model. Here we describe the Windows Host, although the firmware implementation is identical for both Mac OSX and Windows.

The AV/C architecture is developed per industry standards, and firmware support is ongoing as Host implementations progress. The OGT implementation is provided and documented separately, available from TCAT.

The DiceDriver implementation is a complete, low-latency, cross-platform method and is available today. Contact TCAT for Host Development Kits.

Figure 12.1 below illustrates the components used to make a Windows DICE device using the DiceDriver target.

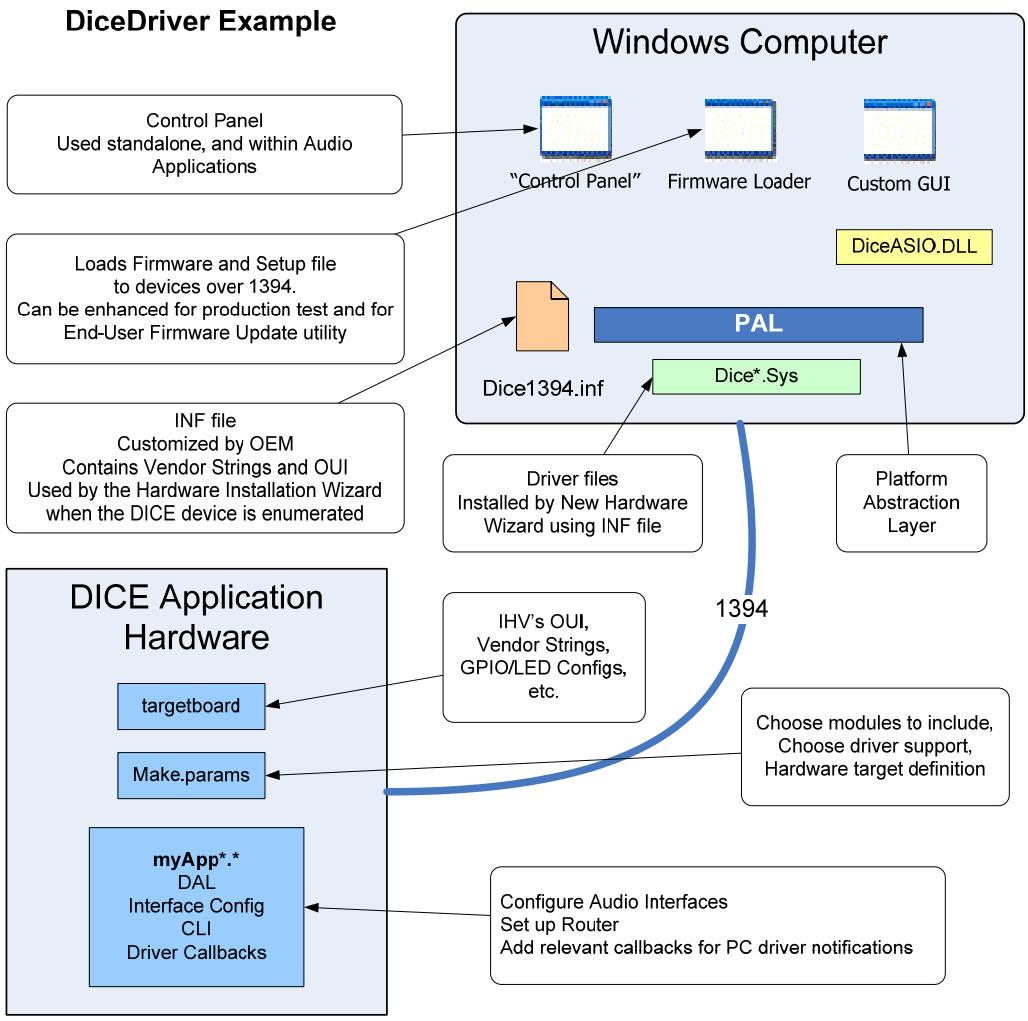


Figure 12.1: DICE hardware and Computer components

Host Computer

Device Discovery

When a DICE device is connected to the computer, the computer enumerates it by reading its Configuration ROM, which contains fields that help the host computer operating system install an appropriate driver. On Windows, this is accomplished in part by matching the Configuration ROM entries with values and strings present in the Dice1394.inf file. The **`inf`** file then installs the **`Dice*.sys`** files, the DLL that provides a User-mode API for Windows Applications, and Registry settings that allow applications to be aware of the presence of the device and its driver. This includes a Registry setting that tells ASIO Applications where

the Control Panel executable is for the device when it is to be invoked from within the ASIO Application.

Platform Abstraction Layer

Stream Management

The `Dice*.sys` drivers, in combination with Windows Applications manage Audio and MIDI streaming.

Device Control and Monitoring

In the case of ASIO, these Applications all use the DiceASIO API to communicate with the target hardware.

Control Panel is an application that is run stand-alone, or is invoked using the Control Panel API in most ASIO enabled applications.

Firmware Loader is just what it sounds like. This application can also be extended for any purpose, such as for production test and as a utility for end-user firmware updates. This method of communication can also be used as a basis for developing rich User Interfaces on the host for control and monitoring of device functions. See the **mixer8** module for an example of how this is done.

DICE Firmware

In the DICE Application Hardware, the firmware is constructed in a way that changes by the manufacturer are necessary in relatively few places.

Device Discovery

The firmware identifies itself on the bus using entries in Configuration ROM. These are configured in

`/firmware/project/projectXXX/target/src/targetVendorDefs.h`

See the comments in the file for more information.

The driver in the computer also looks at Unit Directories in the Configuration ROM to find out more information about the protocols the device supports, then knowing that information, it uses those protocols to find out everything else it needs to know.

The 1394 Audio protocols supported in the DICE firmware include DiceDriver, OGT (provided separately), and AV/C. The protocol supported in the firmware can be selected using defines in

`/firmware/project/projectXXX/make/Make.params`

You can see which driver the firmware is built for in the CLI splash screen. See Figure 16.1.

Stream Configuration

Stream configuration and DICE Audio Interface configurations are handled in `/firmware/project/projectXXX/myApp.c`

This includes setting up audio routing, clocks, and 1394-related formatting for isochronous transmit and receive. This is done using mainly calls to the **DAL** (Dice Abstraction Layer) and **AML** (Abstract MIDI Layer) modules. This file is explained in detail below.

Device Control and Monitoring

Also depending on the protocol used, device control is handled in the corresponding firmware module.

DiceDriver applications are handled in `/firmware/module/dicedriver`

AV/C interfacing is handled in `/firmware/module/1394avc`

OGT support is provided separately as a project level drop-in.

Build control

The **Make.params** file contains the compilation flags necessary for the entire project. All possible compilation flags can be found in the documents as comments (preceeded by a #) and listed under functional groupings. You may need to create your own flags and should make a set of comments showing all of your own possible flags as well.

A flag takes the form of **–D_FLAGNAME** where the **–D** is the C preprocessor switch to define a flag and **_FLAGNAME** is the flag name itself.

The actual parameters are defined in the lower section of the code in groups using the backslash to create a contiguous line of flags. As per usual with make, a linefeed is seen by the preprocessor as the end of the compiler statement and cannot be allowed, even at the end of the statement.

Parameters are grouped into functional groups then the groups are themselves expanded into the final statement **CFLAGS = -c** … Parameters are likely to be different between development and production versions of the firmware. For example, if you wish to reduce the size of the executable image in flash memory, you can remove the _CLI define and it will be compiled out of the Application.

targetBoard

The files found in **/firmware/project/projectXXX/target** must be customized for any target board other than the EVM. When you create your own target, **targetBoard.c** and **.h** will be where you configure the multifunction pins and LEDs if you have them connected directly to GPIO pins. If you do not have LEDs connected to GPIO pins, then removing the flag **–D_LEDS** from **Make.params** will set up the default (no-leds) condition.

These files are also used to configure the default frequency constant for the clock crystal used on your hardware.

myApp

myApp.c and myApp.h

As a starting point for developing with the DICE firmware, **myApp** is provided as a framework to get you started. The file

/firmware/project/projectXXX/main/myApp.c

collects all of the properties, initializations and event handling code for your streaming applications. This is where application threads are initialized, suspended and resumed. This is also where your application itself is initialized and the task "resumed" for the first time. A template thread, **myAppThread()**, is provided for the developer to add worker functions that should be called periodically, such as polling GPIO's, etc.

The framework, as it is distributed in the SDK, is already being initialized by the operating system (although is currently does nothing but "return"). Once you write your application and optionally add CLI commands into this framework and build, your application will become part of the running software: You do not need to go outside the myApp framework for task or CLI initialization.

This module makes use of the DAL (DICE Abstraction Layer) to configure the router and streaming configurations for the application.

DAL

The DAL collects all of the various firmware API's into a simplified API for setting up the DICE router, audio interfaces and streaming.

In general, the firmware will create a router interface, add input and output audio interfaces, add router entries between the inputs and outputs and start the router. This is done in **myApp.c** in **myAppCreateDAL()** and in the supporting data structures.

The router then operates independently from the firmware unless an event occurs that needs attention from the firmware. Events that are caused by the audio interfaces, such as lock state, sample slips and repeats, etc. are handled in **myApp.c** in **myDalCalback()**. Events that originate from the host driver, such as clock source changes, attach state and enable state are handled in **myApp.c** in **myDiceDriverCallback()**.

When a configuration is created, it specifies a sample rate range, or **Mode**, that will be supported. When the DICE must be operated outside that range, the interface is stopped and recreated using the new sample rate range. See the myApp files and the myModes files in the EVM Project for how multiple sample rate ranges are configured.

Defines

MY_NB_RX_ISOC_STREAMS
: This can be 1, 2, 3 or 4. For each stream, provide entries in the receive STREAM_CONFIG struct.

MY_NB_TX_ISOC_STREAMS
This can be 1,or 2. For each stream, provide entries in the transmit STREAM_CONFIG struct.

MY_DEVICE_NICK_NAME
This will be the default nickname provided to the host driver.

MY_INPUT_DEVICES
This is a bitmap that determines which audio interfaces will participate as inputs to the router.

MY_OUTPUT_DEVICES
This is a bitmap that determines which audio interfaces will participate as outputs from the router.

DAL_EVENTS
This is a bitmap that controls which audio interface events will be used to call the callback function, as installed in **dalInstallCallback()**.

Data Structures

STREAM_CONFIG
This struct is used to enumerate the number of audio and MIDI sequences (channels) in each isoc stream, and to name the channels within the isochronous streams. These configurations are advertised to the host driver which configures itself to match.

myDiceDriverCallback()

This is where events from the host computer will arrive. Currently, these events include clock source management, and indication of host driver attach and enable states.

myDalCallback()

This is where events from the DICE router and audio interfaces are handled, by using a status LED or by triggering event notifications back to the host driver.

myAppThread()

The template thread for your use. In the EVM Project, this thread polls the states of the switch on the EVM and the gray encoders for use with the expander board example application.

myAppInitializeTasks()

This is where tasks are initialized. Most importantly, this is where tasks are "created" by passing information about the task and given an ID by the operating system.

Note

You can see the thread list in gdb when using serial debugging, as this is supported in the RedBoot debug monitor. In gdb use the command

(gdb) info threads

In Insight you can see the list by choosing View->Thread List, or by entering the same gdb command as above in the console window.

When using JTAG debugging, the ROM resident debug monitor is bypassed, so you will not see the thread list when using JTAG.

myAppResumeTasks()

Tasks are created in the suspend state or may be suspended by the operating system, other tasks, or itself for various reasons. When the task is to be resumed, myAppResumeTasks is called.

For each task being resumed, the TCTaskResume function is called and passed the taskID of the task being resumed.

Have a look at **drdResumeTasks** in **drd.c**.

myAppSuspendTasks()

For one reason or another, your tasks may need to be suspended. This function suspends a task.

myApp CreateDAL()

This function creates the routing and streaming configuration for the application, using the defines and data structures described above, and specifies the event handler functions. Here an interface is created, which means that the router is configured with it's audio interfaces, sample rate range, routing and event callbacks. If all went well the interface is then started.

initializeInterfaces()

Here you have the opportunity to make changes to the individual audio interface configurations if their defaults are not appropriate.

myAppInitialize()

This is where your application is initialized at the appropriate time by the DICE firmware (see **cyg_main.c**). The functions described above are called in the correct order.

myAppCli

The Developer may want to add additional Command Line Interface (CLI) commands for debugging or prototyping purposes. The location provided for

those items are in the **myAppCli.c and myAppCli.h** files. This is typical of most modules in the firmware.

These files are found in the **/firmware/project/projectXXX/main** directory.

An example CLI structure is shown in the C file. Note that an "if" preprocessor directive is used to inhibit compilation of the example. The file "**cli.h**" describes the structure and sets forth rules for setting up CLI commands.

The function **cliInstallCLIDescriptor(myAppCli_Descriptor[])** installs the CLI descriptor in the operating system.

There are many examples of CLI commands throughout the dice code.

The following provides a brief overview of the framework and points you to a few examples already in the code.

myApp and myModes in the EVM Project

The template project is the basis for your applications; however the EVM Project contains a lot of informative code that can be used where it's appropriate in your projects.

In the EVM Project, the myApp.c file contains example for supporting multiple sample rate ranges, and specifying audio interface initializations other than the defaults, etc. The myMode files contain examples for using the EVM in various streaming configurations, such as number of channels and various combinations of audio interfaces, etc. New users will benefit from using the EVM Project along with a DICEII EVM for experimenting and prototyping firmware before your custom hardware design is complete.

13. Software Updates

Version control

Once you have installed the SDK and before making changes to the source code, it would be a good idea to commit the **/firmware** directory to your version control system. See the *DICE Firmware SDK Installation Guide* for details. As TC Applied Technologies continues to develop the firmware, we will be releasing updates as patches to the original SDK. We suggest that when you receive a patch from us, you run it onto a checked-out version of the current SDK (the "trunk"), commit the changes, then merge the changes with your own project residing as a "branch".

There are no special requirements for the type of Source Control software to use.

Firmware Updates

TCAT will make major releases available as full snapshots, i.e. from revision 3.x to 4.x. Minor releases will be provided in the form of patches. Also, visit the TCAT website occasionally for new information regarding updates. At some point, firmware updates may be made available directly from a revision control server using SVN.

14. Shell

The Cygwin component's setup program uses a 'DOS' window for the **bash** shell by default in the original distribution. This SDK replaces this default with **rxvt** which is more like **xterm**, allowing you to customize color coding, among many other preferences, and allows familiar copy/paste operations. You can customize rxvt by editing your **~/.Xdefaults** file.

For those unfamiliar with xterm, to paste a copy-buffer into rxvt:
a) Hold the Shift key and press Insert, or
b) Press the center button (or its equivalent if any) on your mouse

To copy within the rxvt window:
a) Select the text with the left mouse button, or
b) Click the left mouse button for the start selection point, then click the right mouse button to mark the end of the selection.

Copy/paste also works between native Windows applications and `rxvt`.

Note

1) The shell is launched using the Cygwin icon on your desktop. This launches the shell using the **cygwin.bat** file in your installation root directory. If you launch the shell and it immediately disappears, check your System Path variable to make sure that the proper path to Cygwin is present. Your Windows system PATH should include the correct paths to Cygwin and GNUARM bin directories, and should only appear once each.

A typical installation will have the following directories in the System PATH environment variable:

c:\cygwin\bin
c:\cygwin\gnuarm\bin

Also, while we're describing such things:

2) The Windows System environment variable CYGWIN_PATH should be set to the root installation directory. In a typical installation, this looks like:

CYGWIN_PATH=c:\cygwin

3) The Bash environment variable TC_DIR should be set to the **/firmware** directory. This is set in your **.bashrc** file in your home directory. If you have updated from an older version of the SDK, you will need to change this.

TC_DIR=/firmware
export TC_DIR

15. Editors

GNU Windows Emacs is included here for developers who prefer it. Emacs can be invoked from a bash command line using **emacs**, or from the Start Menu shortcut in the Start Menu/Programs/DICE Firmware SDK/Gnu Emacs folder shortcut. Lisp files and **~/.emacs** are configured with defaults for you in the Cygwin environment.

You can invoke **gdb** from within emacs. See 14 *Debugging* for details.

For developers who are used to using IDE's like Visual Studio for their editor, a workspace project is provided in the **/firmware/project/projectXXX** directory. This is useful for browsing the dice code, and also adds commands that build the dice code, redirecting output to the output window.

In Visual Studio .NET, use the Tools->External Tools… dialog to add the debug tool as shown below.

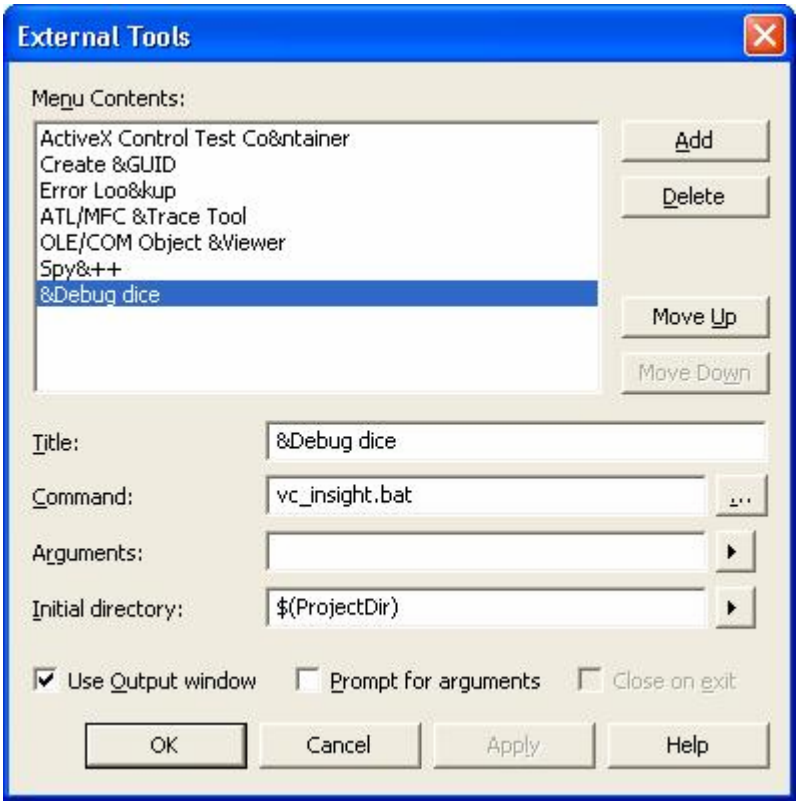


Figure 15.1: Adding a "Debug dice" tool to VS .NET

This tool will work on the currently active project, so you'll only need to create it once for all DICE projects used in the VC++ IDE.

16. Debugging

General Notes

GDB background

Source level debugging of DICE targets is done using the GNUARM port of the gdb debugger, **arm-elf-gdb.exe**. This is a separate debugger than the **gdb.exe** which is part of the Cygwin distribution, so those who have this built into their finger memory should take care to use the GNUARM version instead, although from here on we refer to arm-elf-gdb simply as 'gdb.' This SDK supports using gdb in a bash shell, or within the included GNU Windows Emacs, and with the included Insight graphical interface for gdb.

The examples here illustrate using gdb with a serial debug connecting to the target. If you are using JTAG for debugging, see the section *Bringing up New Hardware and Debugging with JTAG*, to see the connection settings for this.

Serial debugging

By default, the debug monitor firmware uses serial debugging on the secondary serial port, UART1. The steps for this are described later in this section. UART0 is used as the default CLI port.

The debug port and CLI port both use the following serial settings:

115200 baud, 8 Data bits, No parity, 1 Stop bit, No flow control.

Note

The default debug serial port on the DICEII EVM is the Secondary port, UART1. Make sure the jumper on the board is set for RS-232. The default CLI port is UART0, the Primary serial port. A custom serial cable is provided for use of the secondary serial port on the 3-pin header. Note that the arrow on the connector corresponds to pin1 of the 3-pin header (J15). As with all connectors on the board, pin 1 on the header is indicated by a square solder mask on the bottom of the board.

In order to tell the debug monitor that you wish to connect to UART1 for debugging, you must enter a '**$**' at the **RedBoot>** prompt. See the figure below. This allows the UART to be used for other purposes, for example as a MIDI port. See below regarding using MIDI on UART1.

Note

If you use a USB->Serial adapter on the serial port you're using for debugging, see the section below *About Non-Native Serial Ports* for an important note about brands with usable performance. Most any brand will suffice for the CLI port.

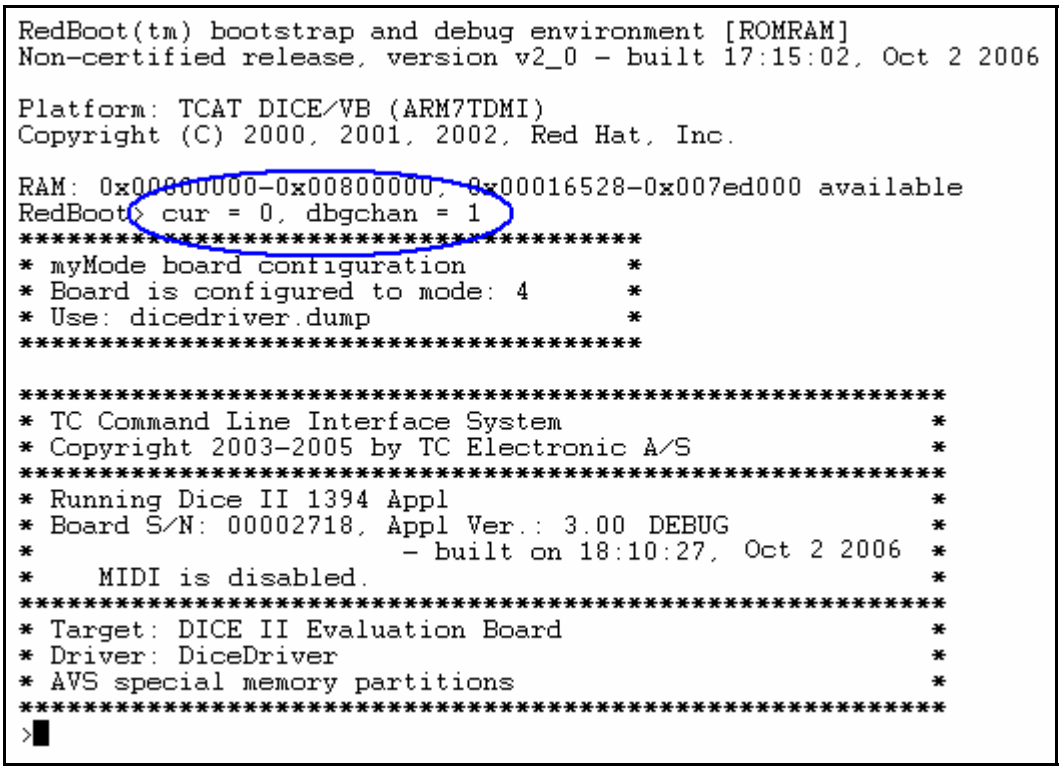


Figure 16.1: Successfully connected to the target (via serial port) and running the Application

Note that when you are ready to connect to the board with gdb, RedBoot will indicate the serial "channels" that the debugger and CLI are using. In figure 16.1 above, the developer has typed a '**$**' at the first **RedBoot>** prompt, and so the CLI is set to its default of channel 0 and the debugger is using channel 1.

Note

When using JTAG debugging this step is not necessary, as the debugger will reset the processor and attach regardless of the state of the firmware, bypassing the debug monitor.

If you wish to switch the functions of the EVM serial ports enter the following command at the RedBoot prompt using a serial terminal program.

RedBoot> channel -1

MIDI and UART1

Note in the figure above that the splash text indicates that MIDI is disabled. On the DICEII EVM this is controlled with switch 1 on the 'SW3' dip switch (located near the Word Clock connectors).

It's clear by now that if your application uses MIDI and you want to use serial debugging, then the CLI will not be available. UART1 is the default debug channel for DICE targets. If your application hardware will use UART1 for MIDI, the debug

channel will not be available on UART1 (secondary port on the EVM), and you can use UART0 (primary port).

The best alternative for debugging Applications that use MIDI and the CLI is to use JTAG. However, you may also download your compiled image into onboard flash and execute it from there. In that case, the CLI will be available and you will have printf-style debugging capabilities.

The SDK firmware distribution uses a GPIO pin to determine if MIDI is in use. On the DICE II EVM (Rev. 1.2 and up), this GPIO is attached to SW3. Switch 1 on SW3 should be set to "OFF" to enable MIDI in the firmware. Also, on the DICE II EVM, you must short pins 1-2 on J13 to use MIDI on UART1 (the secondary serial port).

The rule of thumb for SW3 is that to use MIDI, make sure the switch on SW3 closest to the Word Clock connector is set to the position toward the inside of the board.

Compiler Optimizations

The compile steps in the build system in the Firmware SDK use the debug **–g switch** *at the same time as the optimize* **–O2** switch. This means, of course, that your debug **dice** image has symbols in it that enable source level debugging, and *it also means that your debug image has optimizations that may cause the debugger to step to unexpected lines in the C source code files*, since optimizations are made by the compiler. If you are seeing that this makes debugging your code problematic, simply remove the –O2 optimization switch from the compiler flags and rebuild.

Compiler flags are found in the file **Make.params** in the project's make directory, for example **/firmware/project/projectEVM/make/Make.params** where **ECOS_GLOBAL_CFLAGS** specifies the compiler switches for the build system.

Once you have added or removed compiler switches, do the following to recreate the dependencies and rebuild the kernel library and the application image:

```
user@computername /firmware/project/projectEVM/make
$ make cleanAll
$ make install
```

Command-line debugging

While the developer may not typically use this method, it's useful to describe it here, since the other methods use gdb in the same way under the hood. Note that your target hardware must be at the **RedBoot>** prompt in order for the RedBoot debug stub to be available for gdb to attach to it.

To run gdb in a bash shell, the developer will generally go to the directory where the debug image resides and then start gdb. The build system in the SDK will place your debug image in the **/firmware/project/projectEVM/bin** directory, where the **dice2EVMDebug** image contains debug symbols for gdb (and the

dice2EVMDebug.bin image is a binary that can be downloaded to flash memory and run from there).

user@computername /firmware/project/projectEVM/bin
$ arm-elf-gdb.exe –nx dice

In gdb, you will set your com port settings, then connect, load, set breakpoints, step/continue. A typical command-line gdb session looks like this:

```
user@computername /firmware/project/projectEVM/bin
$ arm-elf-gdb.exe -nx dice2EVMDebug
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --target=arm-elf"...
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS3
Remote debugging using /dev/ttyS3
0x000054f8 in ?? ()
(gdb) load
Loading section .rom_vectors, size 0x40 lma 0x30000
Loading section .text, size 0x3caac lma 0x30040
Loading section .rodata, size 0xbb9c lma 0x6caec
Loading section .data, size 0xbc9c lma 0x78688
Loading section .sram, size 0xbe4 lma 0x84324
Start address 0x30040, load size 347912
Transfer rate: 61851 bits/sec, 319 bytes/write.
(gdb) b main
Breakpoint 1 at 0x38338: file
/firmware/os/ecos/src/packages/language/c/libc/startup/v2_0/src/main.cxx, line
110.
(gdb) c
Continuing
[New Thread 2]
[Switching to Thread 2]
Breakpoint 1, main (argc=0, argv=0x8917c) at
/firmware/os/ecos/src/packages/language/c/libc/startup/v2_0/src/main.cxx:110
110     } // main()
(gdb) n
cyg_libc_invoke_main () at
/firmware/os/ecos/src/packages/language/c/libc/startup/v2_0/src/invokemain.cx
x:123
123         exit(rc);
(gdb)
```

Figure 16.2: Running gdb from a bash shell

Serial vs. JTAG settings

Serial connection settings

The example above uses serial debugging, and uses COM4 on your PC. The **ttySn** setting is always equivalent to COM(**n**+1) on your computer.

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS3
```

JTAG connection settings

For JTAG, since you are connecting to the probe using Ethernet (in the case of the MAJIC-LT) you'll specify an IP address and a port number. Since you are communicating through the MDI server running on the same computer, the IP address is just localhost, or 127.0.0.1 and the port number is the default port number set up during the MAJIC Setup Wizard. If you specified a different port number to use in the setup wizard, then use that one here.

```
(gdb) target remote 127.0.0.127:2345
```

Emacs

You can invoke gdb using the emacs menu option: 'Tools->Debugger (GUD)…' and then typing "**arm-elf-gdb –nx dice2EVMDebug**" into the scratch buffer. If you haven't evoked emacs from the directory containing the debug image, then enter the full path to the file as well. Note that tab-completion works in the emacs scratch buffer. See figure 16.3.

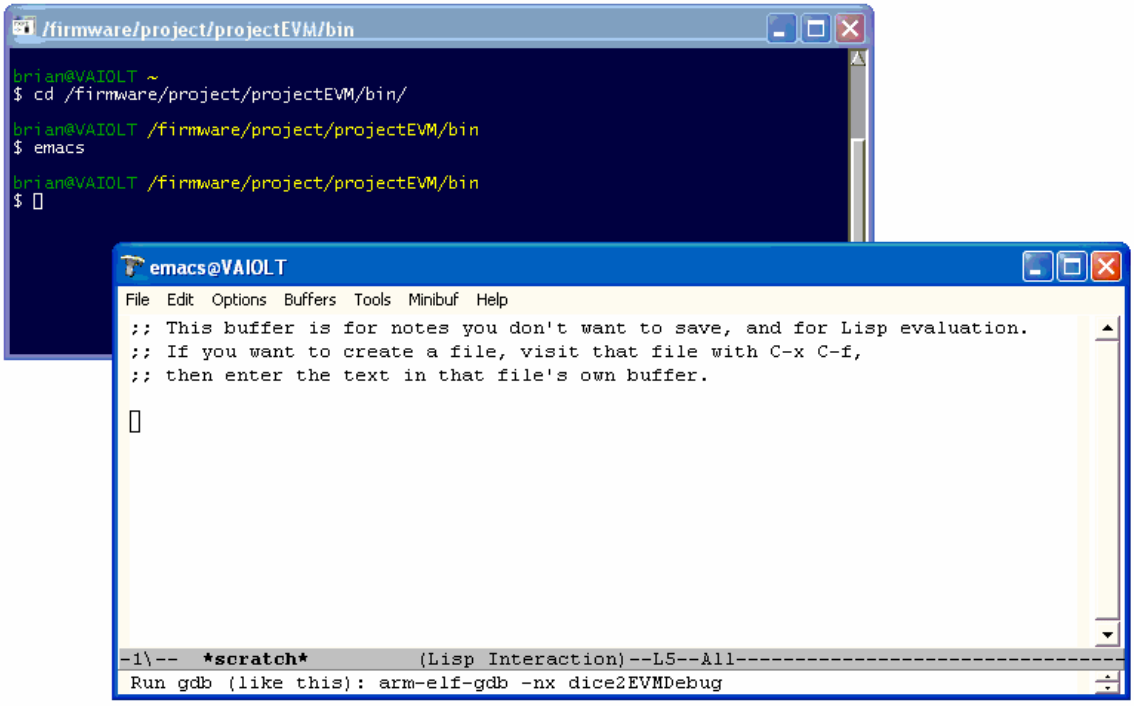


Figure 16.3: Using gdb with GNU Windows Emacs

From there, gdb works just as in the command line. You can use any available serial port on your computer for connecting to the target hardware. Note again that the tty setting in gdb within Emacs is one number less than the COM port number on the PC. So, **/dev/ttyS3** is **COM4**.

Again, before you start the debug session, make sure you have typed a '**$**' at the **RedBoot>** prompt to tell the debug monitor to expect a gdb connection.

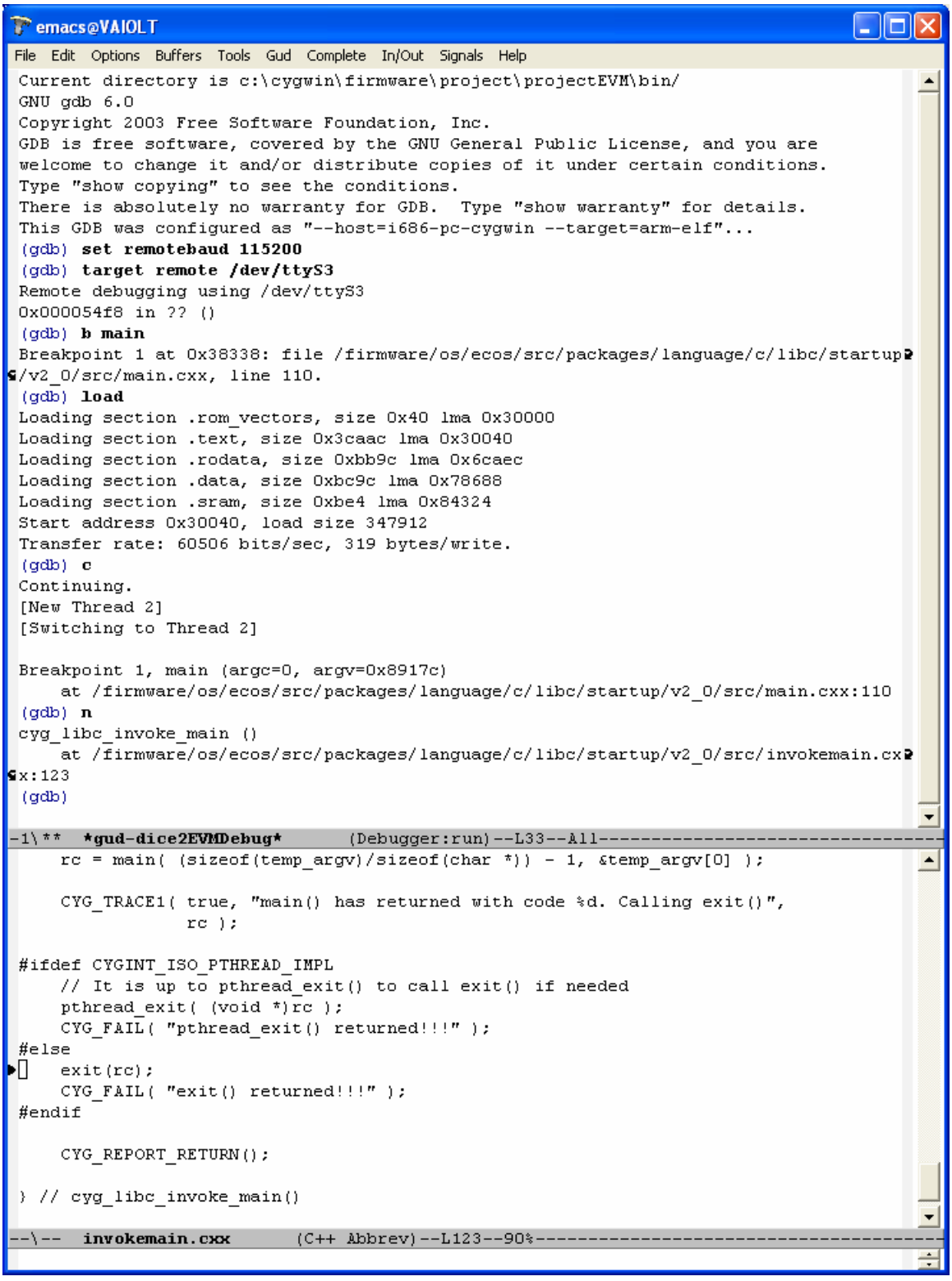


Figure 16.4: Loading, running and stepping though dice code with Emacs

Note

Before using a debugger: If you attach a serial terminal to the target hardware and you do not see a **RedBoot>** prompt, the board is either running an application already, or its flash has not been loaded with a working RedBoot image. If your board has un-initialized flash or has a corrupt boot image, consult your EVM manual or see section 16 *Bringing Up new Hardware with JTAG*.

Your EVM board will arrive configured to automatically run an application image, and you will see a plain prompt '**>**'

To disable automatically running the application do the following:

1) Reset the board and type Ctrl+C into the serial terminal to get to the RedBoot Prompt.
2) Then enter **fco** in Redboot and replace **true** with **false**:

RedBoot> fco
Run script at boot time? false
RedBoot> reset

You should now see a RedBoot prompt when the board is reset.

As in figure 16.4 above, when you "continue" in gdb, you'll see that the application splash screen has appeared, indicating that you are now running your code.

The Insight Debugger

Insight is a graphical interface for **gdb**, and as with the command-line or emacs debugging methods, can be used for either serial or JTAG debugging.

You can invoke the Insight graphical debugger from a bash prompt using:
$ arm-elf-insight &

If you've created a Tool in Visual C++, you can launch it for the built binary for the currently loaded project.

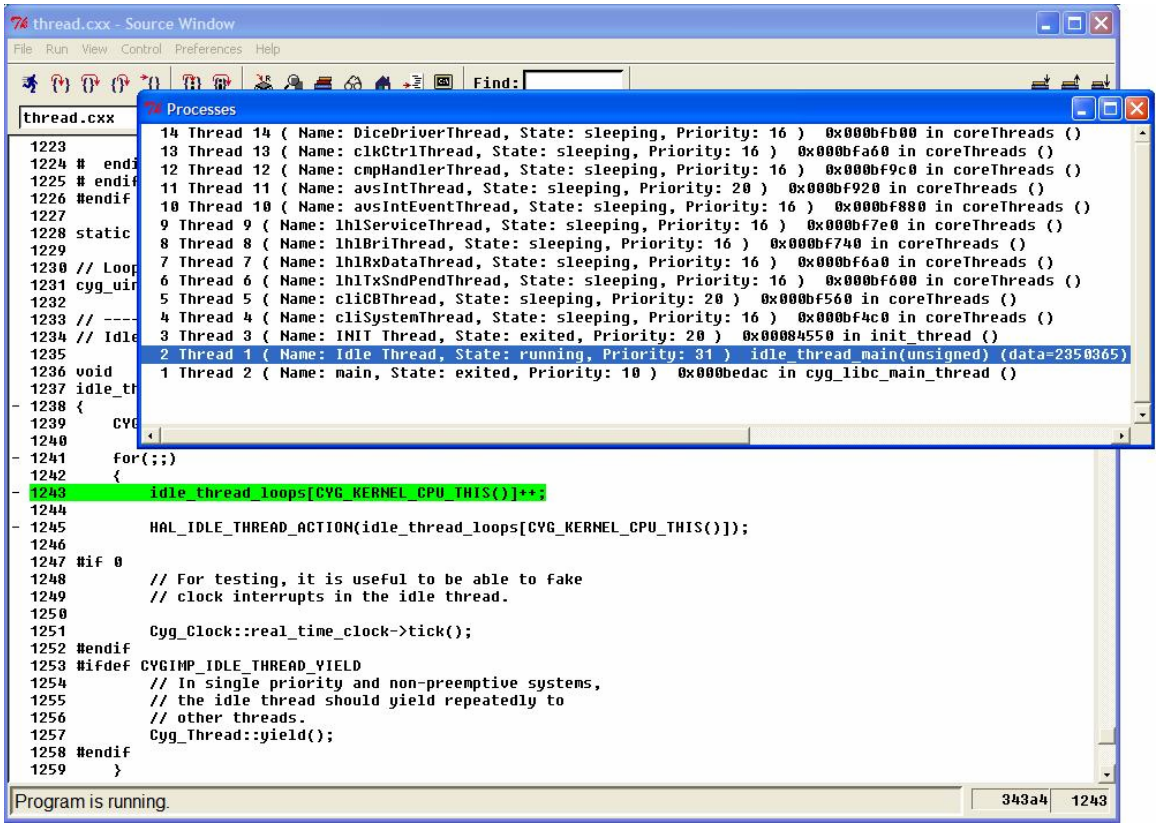


Figure 16.5: Insight debugger

The Target Settings Dialog

This dialog serves as a graphical layout of the debug session initialization, as it would happen in a command-line gdb session. When you load a debug image, Insight will associate these setting with the file being debugged, and use them again the next time you open the file. This is convenient, but make sure to double-check the Connection settings in this dialog if you happen to use Insight with that same filename later on a different serial port on your PC. When you click the Run button, the settings selected in the Target settings dialog will be transferred to gdb.

Target settings are configured in the File/Target Settings menu.

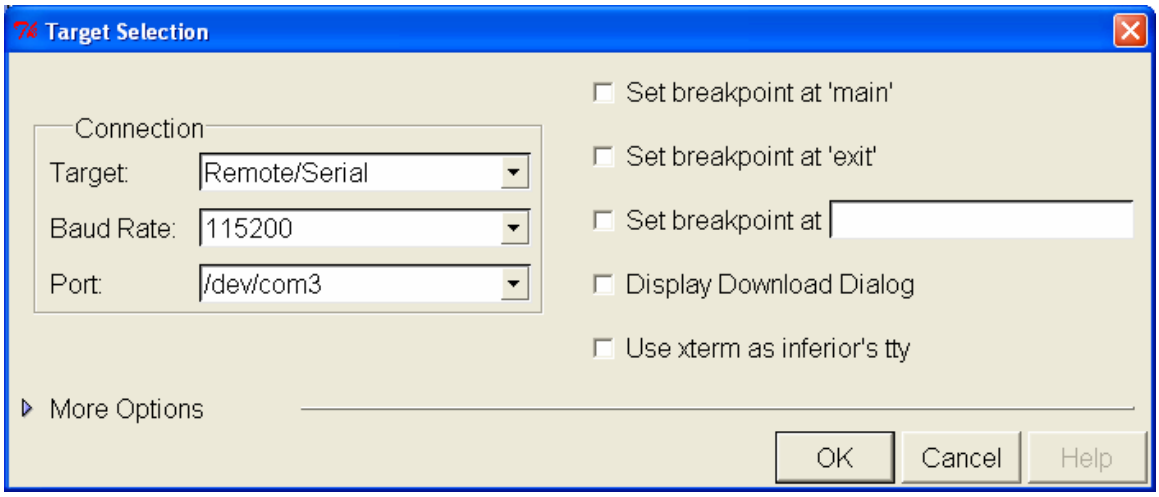


Figure 16.6: Setting up Target Settings in Insight for serial debugging

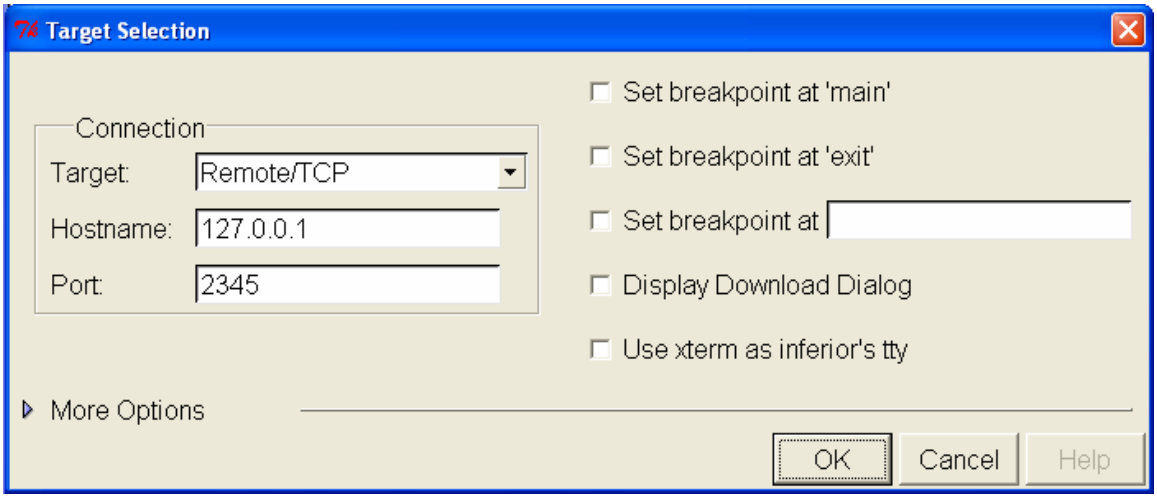


Figure 16.7: Setting up Target Settings in Insight for JTAG debugging

Insight COM port usage

Note

The Insight debugger looks for and uses the /dev/com1-4 devices only, corresponding to COM1-COM4 (and /dev/ttyS0-3). If you have serial ports that are set to higher COM numbers, remap at least one of them down into this range and use that one with Insight.

You can do this in the Device Manager as in figure 16.7, or use the utilities that came with your serial adapter hardware where appropriate.

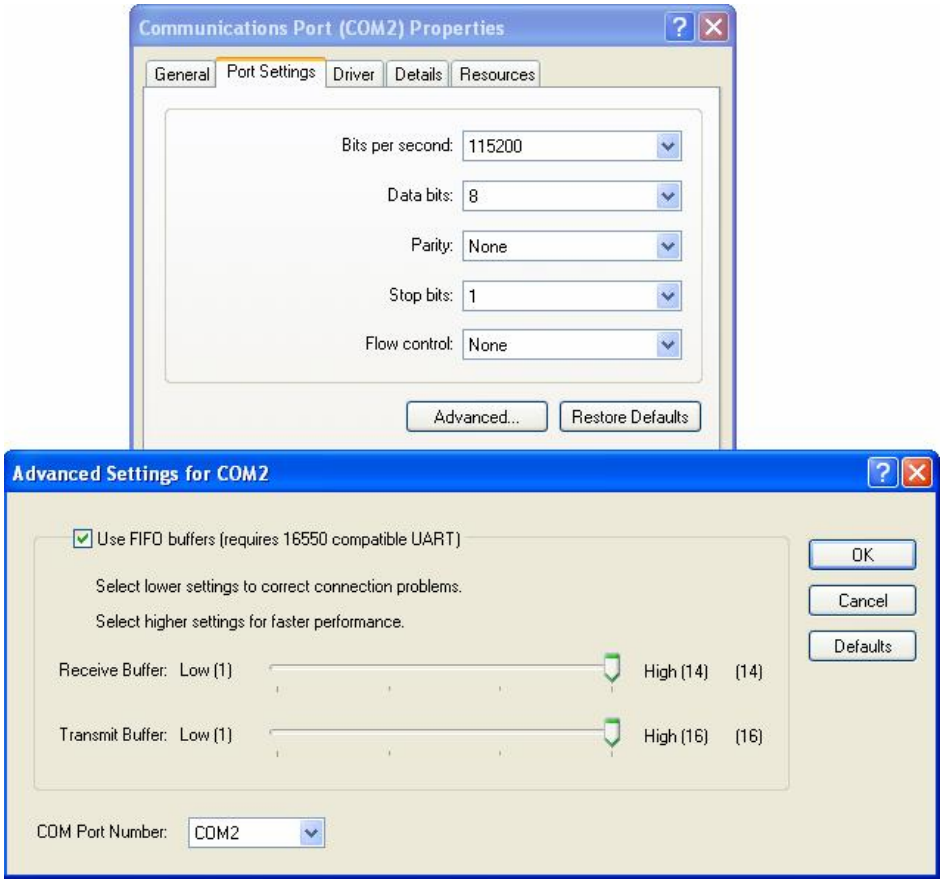


Figure 16.8: Remapping COM port numbers

Insight provides views for watching memory, variables, threads, source code, etc. and has keyboard and menu commands for most of the common gdb commands. You can also open a gdb Console view if you prefer to type in gdb commands directly.

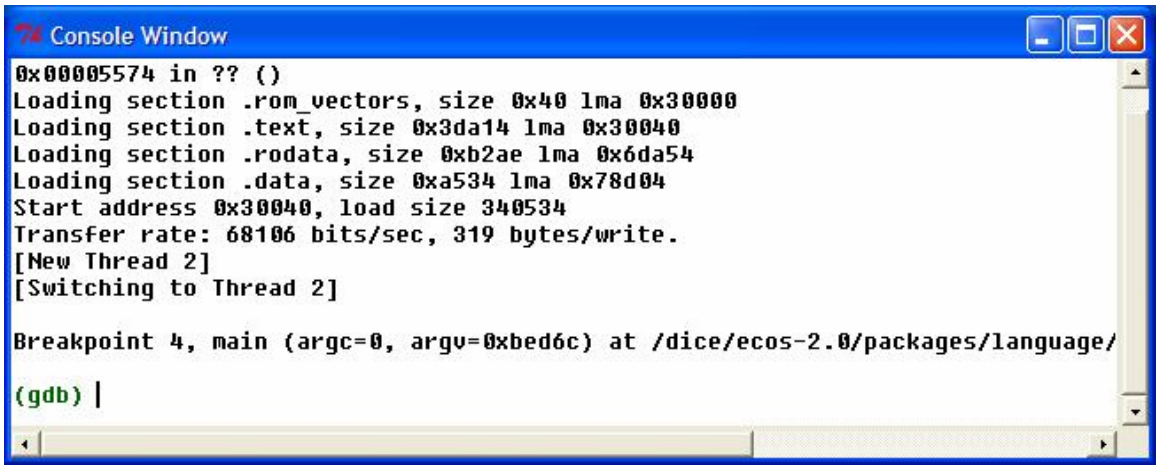


Figure 16.9: Insight gdb console

As before if you're debugging via serial port, make sure that you see a **RedBoot>** prompt, and type a '**$**' in the CLI terminal before connecting.

17. Command Line Interface

The built-in CLI in the dice code is a powerful tool for experimenting and prototyping with the dice API's. You can easily extend the CLI with your own commands as well. See the section *More on DICE Applications* for more information.

The mechanism for adding new CLI commands is documented in the source code. See the header files in the cli module for details.

For a complete reference on the CLI commands, and a detailed explanation for using the DAL commands, see the *DICE Firmware CLI Reference* document.

18. Bringing up Hardware and Debugging with JTAG

Under normal circumstances, you will develop application firmware with DICE target hardware using a serial connection. However, serial debugging requires a preloaded RedBoot ROM monitor to be running on the target hardware. During development, this can be done with a JTAG probe and gdb.

If you have a DICEII EVM, this is also described in the included documentation.

We provide an example here that uses the MAJIC LT probe from EPI Tools. This example can be adapted for your JTAG probe if you are using a different brand.

Sequence

The general sequence for loading un-initialized flash is as follows:

1) Load a RAM RedBoot image into RAM on your target hardware with gdb via JTAG.
2) Run the RAM image, which gives you a RedBoot prompt.
3) Using RedBoot, load a ROM image into RAM on the target via serial port.
4) Copy it to flash memory.
5) Disconnect JTAG, and reset the board to run the ROM image.
6) Using RedBoot again, load a setup file into flash.

The board is now ready to run application code using serial port debugging.

Below is the detailed description of the steps:

1) Install the SDK and configure your bash environment as described in the *DICE Firmware SDK Installation Guide*.
2) Build the relevant RedBoot and Application images.

For the EVM project, do this as follows:

```
user@computername ~
$ cd /firmware/project/projectEVM/make
user@computername /firmware/project/projectEVM/make
$ make install
$ make redboot_ram
$ make redboot_romram
```

3) Install the MAJIC software and configure it to use MDI.
4) Configure your probe with a static Ethernet address.
5) Connect to the MAJIC probe with Ethernet.
6) Connect the MAJIC JTAG cable to your hardware.
7) Connect a serial terminal program from the PC to the target hardware, primary serial port. Use 115200 baud, 8, N, 1, no handshaking
8) Power the target hardware then the MAJIC.
9) Run the MDI server on the PC.
10) In a bash shell, go to the directory containing the RAM image file

$ cd /firmware/kernel/build/rbram/install/bin

11) Run gdb

$ arm-elf-gdb.exe

12) Load the RAM RedBoot image and run it

(gdb) target remote 127.0.0.1:2345
(gdb) load redboot.elf
(gdb) c

You should now see a RedBoot prompt in the serial terminal.

13) Load the ROM boot image via Serial Terminal and write it to flash.
RedBoot> load –r –m ymodem –b 0x100000
14) From the serial terminal, start the transfer of the ROM version of RedBoot (using "xmodem 1k"). The file is:

/firmware/kernel/build/rbromram/install/bin/redboot.bin.

15) Write the image to flash

RedBoot> fis write –b 0x100000 –l 0x20000 –f 0x4000000

16) Quit gdb.
17) Power off the JTAG probe and remove it from the target hardware.
18) Reset the device to boot from flash, you should see a RedBoot prompt again.
19) Initialize the flash file system

RedBoot> fis init –f

The next steps are to load the setup file then the DICE application file. See the section below for creating the setup file and for instructions for loading application code into flash memory.

Note

Regarding the cygwin1.dll, which provides the Linux API emulation layer for Windows programs which use the Cygwin environment. This DLL is used by a number of other programs, and there are a number of installers that add this DLL to your system path.

One important situation comes about if you are using the MAJIC JTAG probe from EPI Tools (Mentor Graphics). The EPI software also installs this DLL, which will conflict with the Cygwin distribution used in this SDK. To fix this, rename the cygwin1.dll file that comes with the EPI tools to something else. EPI Tools will work with the DLL used by this SDK.

19. Managing Images in Flash Memory

Flash files

RedBoot organizes flash memories as a collection of files in a file system. RedBoot itself is written to flash as a raw write sequence to a well-known location so it is executed at boot. RedBoot contains file management code in the **fis** commands that initialize and manage the flash configuration, directory table, and the files themselves.

The DICE application relies on a number of files in the flash file system. The best way to organize your files for DICE Applications is to write the DICE Setup file first, and any Application files after that. When **setup** and application files (e.g. **dice**) have been loaded, the file system will look something like the following:

```
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x04000000  0x04000000  0x00020000  0x00000000
RedBoot config    0x041E0000  0x041E0000  0x00001000  0x00000000
FIS directory     0x041F0000  0x041F0000  0x00010000  0x00000000
setup             0x04020000  0x00000000  0x00010000  0x00000000
dice              0x04030000  0x00030000  0x00080000  0x00030040
RedBoot>
```

Listing 20.1: a complete flash file system

Note

This is a good time to check that your Setup file comes first in your flash file system, i.e. right after the **FIS directory**. This can be done either at the RedBoot prompt or at the Application prompt:

RedBoot> fis list

or

> fis.list

If it does not, then you may want to change this by deleting all of the files that follow FIS Directory, using

RedBoot> fis init -f

and following the steps below. Users were previously instructed to load an Application file, followed by the Setup file. In this case, the Application file does not have room to grow in size without running into the Setup file. Writing the setup file first resolves the issue.

If you have not yet set up your RedBoot file system or if it has become corrupted, consult your EVM documents, or the *Bringing Hardware and Debugging with JTAG* section for details on loading a RedBoot image into un-initialized or corrupted flash memory.

RedBoot

The ROM debug monitor and bootstrap. RedBoot RAM and ROMRAM images are built for you in the dice code. Pre-built images are also provided for DICE EVM's, for example in **/firmware/EVMbringup/DICEIIEVM/redboot**.

Setup

Contains the serial number of the device and initialization variables. This file should be created first, after creating the initial RedBoot file system, so that the Application can grow in size without needing to relocate the setup file.

Application

The binary image created when you build the DICE application. This binary is called something similar to **/project/projectXXX/bin/dice2XXXDebug.bin**. The important thing to note is that you are loading the **.bin** flash-loadable file and not the image intended for use with the debugger, which has no file extension.

Loading files

The following instructions apply to the DICEII EVM, but will apply to most any DICE target hardware. Information is repeated here from the EVM documentation, in case you are working with other hardware.

Otherwise, loading files is done via a serial terminal. As usual, use the following serial port settings: **115200, 8, N, 1, No HS**

Connect to the device using a serial terminal program and at a **RedBoot>** prompt follow the appropriate sequence for the file you wish to update:

Loading the Setup file

1) Edit **/firmware/EVMBringup/config.txt**
The file should look similar to this, *including a final carriage-return*:

```
SERIAL_NO=54321
HPLL_CLK=50000000
```

SERIAL_NO is the lower 5 digits from the yellow serial number sticker on the EVM. This is used to calculate the lower bytes of the device WWUID. See the section *1394 WWUID and Device Serial Number* regarding this value.

HPLL_CLK the JET-PLL is running off of the DICE crystal and in order for the sample counter to be correct it needs to know which crystal frequency is

used. The default is 25 MHz, but it can be overwritten with the HPLL_CLK parameter. The frequency should be two times the crystal used. In case of a 24.576MHz crystal it should be stated as: **HPLL_CLK=49152000**

2) **load -r –m ymodem –b 0x30000**
3) Transfer **config.txt** (using "1K Xmodem")
4) **RedBoot> fis create –b 0x30000 –l 0x40 -e 0x0 –r 0x0 setup**

The **–l** length parameter should be at least the length returned by the load command (end – start +1). **0x40** is sufficient here, and RedBoot will round up to the nearest sector size anyway. If your flash already has a file named "setup" confirm that you want to overwrite the file.

You can confirm that the setup section is written to flash by examining the flash memory. If you have gone through the steps above, the **setup** section will be located in flash memory at **0x04020000**, use fis list to verify this.

5) RedBoot> fis list
6) RedBoot> x –b 0x04020000 –l 0x40 -1

This will show something similar to this:

```
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x04000000  0x04000000  0x00020000  0x00000000
RedBoot config    0x041E0000  0x041E0000  0x00001000  0x00000000
FIS directory     0x041F0000  0x041F0000  0x00010000  0x00000000
setup             0x04020000  0x00000000  0x00010000  0x00000000
dice              0x04030000  0x00030000  0x00080000  0x00030040

RedBoot> x -b 0x04020000 -l 0x40 -1
04020000: 53 45 52 49 41 4C 5F 4E  4F 3D 35 34 33 32 31 0D  |SERIAL_NO=54321.|
04020010: 0A 48 50 4C 4C 5F 43 4C  4B 3D 35 30 30 30 30 30  |.HPLL_CLK=500000|
04020020: 30 30 0D 0A 54 01 03 00  78 01 03 00 A4 01 03 00  |00..T...x.......|
04020030: C0 01 03 00 00 00 00 00  C0 02 03 00 88 02 03 00  |................|
RedBoot>
```

Listing 20.1: location and contents of the setup file

Note that the setup file is 0x10000 bytes in length, when we only wrote 0x40 bytes. This is per usual with such memories, where this flash memory has a minimum segment size of 0x10000 bytes at that address range.

Loading your Application file

1) **load -r –m ymodem –b 0x30000**
2) Transfer the application file **\dice\diceApp\dice.bin**
   i. (using "1K Xmodem")
3) **fis create –b 0x30000 –l 0x80000 –e 0x30040 –r 0x30000 dice**
   If your flash already has a file named "**dice**" choose a different name or confirm that you want to overwrite the existing file. DICE EVM's have room for multiple Application images. You can load your preferred image, i.e. **mydice** with the following RedBoot commands:

RedBoot> fis load mydice
RedBoot> go

The flash on the EVM can store more than one Application image, and can be optionally configured to automatically run an image when the device boots.

Automatically running an Application image

1) At the **RedBoot>** prompt, type **fco**
2) Replace **false** with **true**
3) Enter the following two lines at the **>>** prompts, followed by a **return**:

   >> fis load dice
   >> go
   >>

4) Enter a value of **20** for the boot delay. This is equivalent to 2 seconds (the ms resolution indicated is not correct).
5) Enter '**y**' to confirm
6) Reset the board.

Verify the WWUID

The EVM should run **RedBoot** and then auto-run the **dice** application at this point. Look at the splash screen. The **Board S/N** should be the same serial number entered in the **setup** section, from **config.txt**. ***Note that this value is displayed in hexadecimal here.***

```
>splash

************************************************************
* TC Command Line Interface System                         *
* Copyright 2003-2005 by TC Electronic A/S                 *
************************************************************
* Running Dice II 1394 Appl                                *
* Board S/N: 000d431, Appl Ver.: 3.00 PRE-RELEASE          *
*                        - built on 14:29:57, Oct  4 2006 *
*    MIDI is disabled.                                     *
************************************************************
* Target: DICE II Evaluation Board                         *
* Driver: DiceDriver                                       *
* AVS special memory partitions                            *
************************************************************
>
```

The serial number will also become the lower bytes of the WWUID for the device. *The upper bytes in the EVM are set to a temporary number to be used only for evaluation, development and test. Shipping products must contain the OUI of the manufacturer.*

```
> lal.getwwuid
WWUID: 0x0013f004 0x004d0431
>
```

20. Initializing Flash with the Pre-Built Boot Loader

This section is provided for information for developers who use the MAJIC-LT JTAG probe. Normally you will use the boot loader that you have created yourself, in which case you can load them using normal JTAG debugging. This section describes how to load the boot loader directly to flash via the flash programming utility comes with the MAJIC probe.

Configure the MONICE Debugger

MONICE is a command-line debugger. It will be used to load the flash programming utility into the ARM.

1) Install the EPI Tools and run the MAJIC Setup Wizard

2) Choose 'Setup A Debug Environment,' using 'MONICE' as your debugger.

3) Give the project a unique name (i.e. other than your MDI project name), such as My_MONProject

4) Use Processor type ARM7TDMI, Little Endian, Intrusive Reset

5) Use the Ethernet Static IP address that was set up during the MDI Setup

   If you haven't set the probe's IP address yet, then back up and do this now.

6) Use Existing Startup File in
   C:\Program Files\EPITools\edta23b\targets\wavefront_dice

   The startice and dice.cmd files configure the the DICE registers correctly, and prepares it for running the flash programming utility.

7) Choose a unique directory for the project

   The default is fine if you have picked a unique Project name above.

8) Select the "Copy startup *.cmd files…" and "Create MON Shortcut on Desktop" checkboxes.

9) Click on the 'Perform Actions' button then quit the program

Run the flash programming utility, Write the Boot Loader to Flash

1) Use the MON Desktop icon to run MON, load the flash programming utility into the DICE embedded ARM processor, then write the boot loader to flash.

   The utility should show information as shown in the listing below:

```
EPI Symbolic Assembly Level Debug Monitor, version V6.11.3 - WIN/NT(386)
Copyright (c) 1987-2006 by Mentor Graphics Corporation - All Rights Reserved.
Processing register file: C:\Program Files\EPITools\edta23b\bin\arm/spaces.rd
Processing register file: C:\Program Files\EPITools\edta23b\bin\arm/majic.rd
Reading command history from: 'C:\Documents and Settings\brian\Application Data\
monice_ARM_startedb.hst'
Establishing communications with remote target via 192.168.16.101...
Connection verified
Target System:  EPI MAJIC Probe, Version: 3.7.0, S/N xxxxxxxx
Hardware Rev:   12:33:1:0
Target CPU:     ARM7TDMI
Ethernet:       at address 00:80:CF:00:1E:B5
IP address:     192.168.16.101, Subnet mask: 255.255.255.0
Trace Buffer:   None
Connected via:  Ethernet UDP/IP
Device name:    192.168.16.101
Target Endian:  little
Config Key( 3): xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reading commands from c:\majic\My_EDBProject\startice.cmd
MON> +q  // Enter quiet mode
Reading startice.cmd file
Notification from the target:
JTAG interface enabled at 3v level
Auto JTAG detection process detected 1 TAP
JTAG connection established
Reading dice.cmd
Initializing target...
81000054:    00000246
Target initialization completed.
Finished reading dice.cmd
Finished reading startice.cmd
MON>
```

Listing 21.1: MON output if configured properly

2) Load the appropriate flash programming utility, and verify it loaded ok:

```
MON> L "C:\Program Files\EPITools\edta23b\samples\le\ram_0x00008000\
epiflash.axf"
loading c:\Program Files\EPITools\edta23b\samples\le\ram_0x00008000\epiflash.axf

 section ER_RO    from 00008000 to 00017597
 section ER_RW    from 00017598 to 000182a7
 section ER_ZI    from 000182a8 to 00018b1b
Entry address set (pc): 00008000
$a:
00008000:     e28f8090    ADD    r8,pc,#0x90

MON> VL
checking c:\Program Files\EPITools\edta23b\samples\le\ram_0x00008000\epiflash.axf
 section ER_RO    from 00008000 to 00017597
 section ER_RW    from 00017598 to 000182a7
 section ER_ZI    from 000182a8 to 00018b1b
Verify succeeded.
MON>
```

Listing 21.2: flash programming utility successfully loaded

3) Run the utility:

```
MON> da epiflash
epiflash: "fr c dice.cmd 4  // EPIFLASH runs EPIFLASH script   (flash programming utility)"

MON> g

EPI Flash Programmer v3.1.5.4 - LE

Flash Device  : Am29DL640G   x16  8MEG
Devices       : 1-Series  1-Parallel  1-Total
Sector Groups :     3
Sector Count  :     8    126      8
Sector Size   :    8K    64K     8K (bytes)
Device Base   : 0x04000000
Device Offset : 0x00000000

Filename      : c:\cygwin\EVMBringup\redboot.bin
File Offset   :         0  (0x00000000)
File Length   :     71160  (0x000115F8)
Image Size    :   1048576  (0x00100000)
Image Address : 0x04000000 - 0x040FFFFF

Device Buffer :         0 (bytes, max = 0)
Image Buffer  : 0x00000000 (bytes, Disabled)
Image Buffer  : 0x00000000 (Addr)

  MAIN MENU
  1) Flash Type
  2) Image Filename
  3) Device Base Address
  4) Device Offset
  5) Image Size

  6) Erase Menu
  7) Program Menu
  8) Diagnostic Menu
  9) Flash Device Menu
 10) Configuration Menu

  0) QUIT Program

Select an Option:
```

Listing 21.3: Running the utility

4) Select "Image Filename" and enter **C:\cygwin\EVMbringup\redboot.bin**
5) Select "Device Base Address" and enter **0x04000000**
6) Select "Program Menu" then
7) Choose "Erase Image, Program Image, Verify" and continue
8) Exit MON and then reset the EVM

   You should now see a RedBoot> prompt on the EVM Primary serial port (using 115200, 8, n, 1, no flow control serial settings).

9) Initialize the flash file system at the prompt:

   RedBoot> fis init –f

You are now ready to load the setup and application files, see the section *Managing Images in Flash Memory* above.

21. Internal RAM-Resident Memory Test Utility

Board Verification

The internal RAM test is a minimal DICE application that runs entirely in the on-chip SRAM. This utility program can be used to verify your external memory interfaces during your board verification.

When this program is running on the DICE, it can be used to read and write memories connected to the DICE and can run comprehensive memory tests that can be used to verify your address and data buses.

RAM Footprint

This utility demonstrates the flexibility of eCos in that it can be configured to produce a very small kernel. Further, the test uses the standard library printf() function for it's CLI, which uses approximately 7-8kB of memory. The developer may write their own serial output function to save additional memory space which can be used for more comprehensive tests.

Note

This executable must be loaded and run via JTAG. When using the internal RAM test, you will need to configure the ARM Remap module to set the low portion of the address space to the internal RAM. This is done by writing a '1' to register 0xc0000008.

If you are using the EPI Tools MAJIC-LT JTAG probe, make an addition to the **dice.cmd** file that you are using with the MDI Server. You can find this file by getting Properties on the MDI server icon that was created by the MAJIC Setup Wizard, and looking in the 'Start in:' directory shown.

Add the following line to dice.cmd. Anywhere near the end of the file will work.

ew 0xc0000008 = 0x01

If you don't do this, the test will appear to load and run normally, but then will fail in the middle of the test, causing unnecessary hardware troubleshooting and head scratching.

Be sure to comment this out or otherwise undo the setting in your setup when debugging any other DICE application or boot loader.

Listing 1 below shows the output of a normal memory test, using '**m 1**'

```
TC Applied Technologies.
DICE EVM internal RAM tests:
   m count: Memory test
   x start_addr length_in_quadlet: display memory
   l/w/b address value: Set value
   ?/h: Help, this menu
$ m 1
m 1
memTestDataBus8: base:0x01000000
memTestDataBus8: wr/rd single address with single bit set values
memTestDataBus8: done
memTestDataBus16: base:0x01000000
memTestDataBus16: wr/rd single address with single bit set values
memTestDataBus16: done

data bus test ... done.
memTestAddrBus8: base:0x01000000, addrMask:0x007fffff
memTestAddrBus8: write pattern:0xaa to each of the power-of-two offsets
memTestAddrBus8: check for address bits stuck high with antiPattern:0x55
memTestAddrBus8: check for address bits stuck low or shorted:
memTestAddrBus8: done
memTestAddrBus16: base:0x01000000, addrMask:0x007fffff
memTestAddrBus16: write pattern:0xaaaa to each of the power-of-two offsets
memTestAddrBus16: check for address bits stuck high with antiPattern:0x5555
memTestAddrBus16: check for address bits stuck low or shorted:
memTestAddrBus16: done

address bus test ... done.
***********************************
This is      1th run of the memTest.
***********************************
memTestRdWrMem8: base:0x01000000, size:0x00800000, step:all(0x00000004)
memTestRdWrMem8: testing addresses:
memTestRdWrMem8: 0x01000000,0x01000001,0x01000002,0x01000003,0x01000004,
memTestRdWrMem8: 0x01000005,0x01000006,0x01000007,...
memTestRdWrMem8: fill memory with a pattern
memTestRdWrMem8: verify memory pattern and fill with antipattern
memTestRdWrMem8: verify memory antiPattern
memTestRdWrMem8: done
memTestRdWrMem16: base:0x01000000, size:0x00800000, step:all(0x00000004)
memTestRdWrMem16: testing addresses:
memTestRdWrMem16: 0x01000000,0x01000002,0x01000004,0x01000006,0x01000008,
memTestRdWrMem16: 0x0100000a,0x0100000c,0x0100000e,...
memTestRdWrMem16: fill memory with a pattern
memTestRdWrMem16: verify memory pattern and fill with antipattern
memTestRdWrMem16: verify memory antiPattern
memTestRdWrMem16: done
memTestRdWrMem32: base:0x01000000, size:0x00800000, step:all(0x00000004)
memTestRdWrMem32: testing addresses:
memTestRdWrMem32: 0x01000000,0x01000004,0x01000008,0x0100000c,0x01000010,
memTestRdWrMem32: 0x01000014,0x01000018,0x0100001c,...
memTestRdWrMem32: fill memory with a pattern
memTestRdWrMem32: verify memory pattern and fill with antipattern
memTestRdWrMem32: verify memory antiPattern
memTestRdWrMem32: done
Total number of errors is 0.
$
```

Listing 21.1: Output of memory test utility with 'm 1'

22. 1394 WWUID and Device Serial Numbers

Terminology

Each device on a 1394 bus must have a World-Wide Unique ID. A WWUID consists of a combination of an Organizational Unique Identifier (OUI), also known as a Vendor ID, and a unique number assigned by the vendor to the hardware device. This is similar to an Ethernet MAC address, in that you use a 3-byte OUI obtained from the IEEE in the first three octets, and your own numbering system for the rest of the address. In IEEE terms, a WWUID is an EUI-64.

Format

The 64-bit WWUID appears in the device's Config ROM as two 32-bit "quadlets."

In order to communicate properly on the 1394 bus with Asynchronous transactions, your device must have a WWUID of some kind. During your development, make sure that all devices on your test bus have unique ID's.

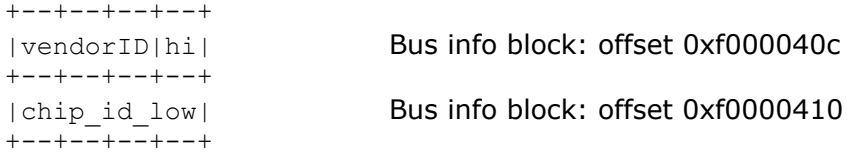
The DICE code uses a temporary OUI for your development. DICEII EVM's are preloaded with the PCB production serial number for use as the rest of the WWUID. This serial number is written in the **setup** flash file on DICEII EVM's.

Note

Your shipping products must have your own OUI and unique serial numbers as a valid WWUID. See the section *Resources* for information about where to obtain your own OUI.

As mentioned before, the Vendor ID is a 3-byte value stored as the most significant 3 bytes of the appropriate CSR in the Bus Info Block of the Configuration ROM. Together with chip_id_hi and chip_id_low, this constitutes the entire WWUID for the node. Chip_id_hi and chip_id_low must be unique for all devices shipped with the particular Vendor ID.

WWUID, as shown in Configuration ROM



The Dice firmware uses the chip ID as follows

Upper 32 bits of WWUID

```
+------------------------+--------+
|  24 bit OUI – 0x000166 | Cat    |
+------------------------+--------+
```

Category 8 bits: Use **0x04** for all DICE II devices

Lower 32 bits of WWUID

```
+---------------------------+----------------+
|  10 bit product identifier | 22 bit serial# |
+---------------------------+----------------+
```

The 24-bit OUI, Category, and 10-bit product identifier are coded into the firmware in **targetVendorDefs.h**

The remaining 22 bit serial number is read from the setup file in flash memory, as described in the section *Managing Images in Flash Memory*.

To personalize your device:

Edit **targetvendordefs.h**:

Change **THIS_VENDOR_ID**	to your OUI, as registered with the IEEE
Change **THIS_VENDOR_NAME**	to the name of your company (also as registered)
Change **THIS_PRODUCT_ID**	to the ID you selected for your product
Change **THIS_PRODUCT_NAME**	to the name of your product

23. Self Documenting Code and Doxygen

HTML Documents

Comments in the dice application source code are tagged with symbols that are recognized by Doxygen, a code documentation utility. Doxygen creates indexed and hyperlinked html files from source code and tags found within.

A framework project and html files are provided at the project level. You can build the HTML documentation at any time as follows:

user@computername /firmware/project/projectXXX/doxygen
$ doxygen

The resulting HTML start page is

/firmware/project/projectXXX/GeneratedDocumentaion/HTML/**index.html**

Windows Help (CHM)

HTML can be converted to Windows Help files using utilities such as HTML Help Workshop:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/hwMicrosoftHTMLHelpDownloads.asp

Note that if you download and install this program in its default directory, the doxygen command above will also compile the Windows Help file for you each time automatically. The resulting files (CHM and CHI) are place in the

/firmware/project/projectXXX/GeneratedDocumentaion/help/

directory as **SDKUserCompiledHelp.*** [Note that this Help Compiler will give the message "Error: failed to run html help compiler on index.hhp"] The error will appear even when the compiler succeeds, and of course we have no idea why the error happens or what it means…

A pre-built Help file from the Rev. 2.0 release version of the DICE code is included in the SDK, and a shortcut to it is provided in your DICEFIRMWARE SDK Program Group.

Converting HTML to Windows Help allows you to search the documentation as with any Help file. Also, If you use Visual Studio 6.0, you can integrate the help file into MSDN and use select-F1 context sensitive lookups within the Visual Studio IDE. See below for .NET integration.

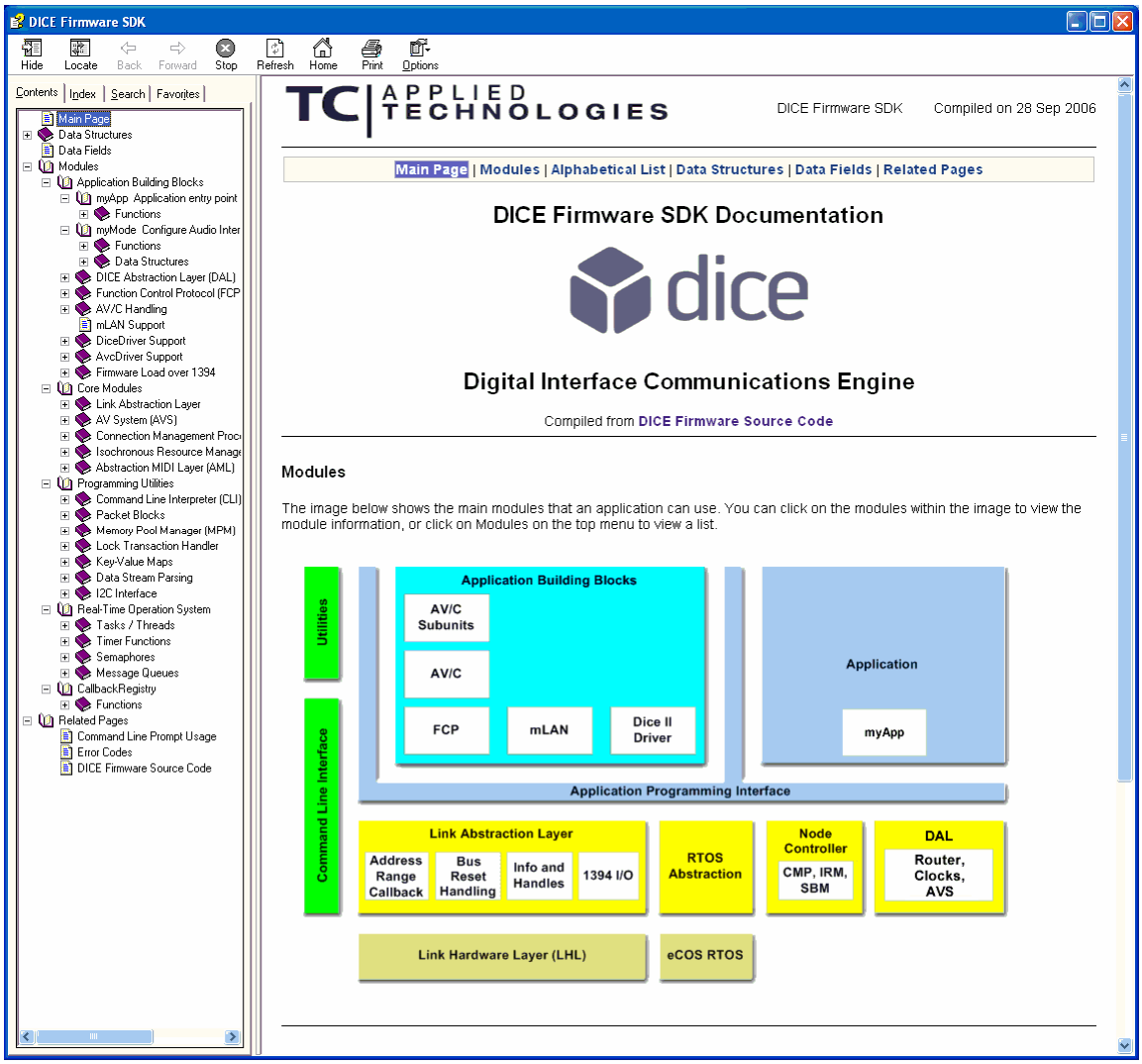


Figure 23.1: Searchable Help generated from the Source Code using HTML Help Workshop

Help2

For those who use Visual Studio .NET for editing, the HTML can be converted to Help2 format and integrated in to VS .NET 7.x using the Help Integration Wizard which can be found online:

http://msdn.microsoft.com/vstudio/extend/default.aspx?pull=/library/en-us/dv_vstechart/html/integration_wizard.asp

24. Resources

Support Contact

TC Applied Technologies
http://www.tctechnologies.tc

References

DICEII Supporting Documents, TC Applied Technologies

Learning the bash Shell 2nd Ed.
By Newham and Rosenblatt, O'Reilly

Embedded Software Development with eCos
By Anthony J. Massa, Prentice Hall

The Linux Development Platform
By Rehman and Paul, Prentice Hall

FireWire(R) System Architecture: IEEE 1394A (2nd Edition)
by Mindshare Inc, Don Anderson, Addison Wesley

ARM Systems Developer's Guide
By Sloss, Symes and Wright, Morgan Kauffman

ARM Architecture Reference Manual
By ARM Ltd., David Seal, Addison Wesley

Tools

MAJIC LT JTAG Probe http://www.epitools.com

FireSpy800 1394 Bus Analyzer http://www.dapdesign.com

1394 Trade Association

http://www.1394ta.org

Protocol Specifications are found here, including AV/C.

AV/C – Audio Video Control

http://www.1394ta.org

OGT - Open Generic Transporter

http://www.aes.org/e-lib/browse.cfm?elib=13191

Obtaining an OUI

http://standards.ieee.org/regauth/oui/forms/

1394 Standards

http://shop.ieee.org/ieeestore/

1394-1995 IEEE Standard for a High Peformance Serial Bus - Firewire Product Type: Standard IEEE Product No:SH94364 IEEE Standard No:1394-1995 ISBN:1-5593-7583-3 Format:Softcover Copyright:1995

1394A-2000 IEEE Standard for a High Performance Serial Bus- Amendment 1 Product Type: Standard IEEE Product No:SS94821 IEEE Standard No:1394A-2000 ISBN:0-7381-1959-8 Format:PDF Copyright:2000

1394B-2002 IEEE Standard for a Higher Performance Serial Bus-Amendment 2 Product Type: Standard IEEE Product No:SS94986 IEEE Standard No:1394B-2002 ISBN:0-7381-3254-3 Format:PDF Copyright:2002

IEEE 1212 Standard

13213: 1994/1212, 1994 ISO/IEC 13213:1994, [ANSI/IEEE Std 1212, 1994 Edition] Information technology - Microprocessor systems - Control and Status Registers (CSR) Architecture for microcomputer buses Product Type: Standard IEEE Product No:SS94220 IEEE Standard No:13213: 1994/1212, 1994 ISBN:0-7381-1214-3 Format:PDF Copyright:1994